Proving the Correctness of Program Transformations

Victor L. Winter
9.2 Future Work

A Proof of Transformation 7a .................................................. 108
B Syntax for a subset of Poly .................................................. 116
C Eight Queens Specification .................................................. 122
D Lambda Form of Eight Queens ........................................... 124
E Recursive Fortran form of Eight Queens ............................... 128
F Otter Representation .......................................................... 134

Bibliography ........................................................................ 137
Abstract

Given a formal specification $S$ of a problem, we wish to obtain a computer program that solves this problem. The traditional approach used to obtain such a program has been to give the formal specification to a programmer. The programmer's task is to study the specification, and after he fully understands it, to write a computer program satisfying the specification. For critical applications, formally proving that the program satisfies the specification is highly desirable. Unfortunately, proving that a program satisfies a formal specification is very difficult, especially when the program is obtained from the formal specification in the traditional manner. In this thesis we explore a paradigm that allows one to obtain programs from formal specifications through automatable transformations. In this paradigm, if the transformations can be shown to be correctness preserving then it follows that the program satisfies the specification.
Chapter 1
Introduction

1.1 Program Verification

Program verification consists of using the formal semantics of a programming language to reason about the behavior of a specific program. This is accomplished by choosing some formal language such as first order logic, stating (in this language) what property is desired of the given program, and using the formal reasoning mechanism provided by the language (e.g., first order logic) together with the program and the semantics of the programming language to prove that the program satisfies the desired property. In this manner a program can be formally verified.

The ultimate goal of program verification is to increase one's confidence in the correctness of a program. Informally, when we say a program is correct, we mean that the program solves the problem we want it to solve. In a later section we will give a formal definition of correctness, but for now the above informal definition should suffice.

Since there may be mistakes made when reasoning about a program, one can never be certain that the formal proof of a program's correctness implies the correctness of the program itself. There are two kinds of errors that can occur in the proof of a program's correctness. The first type of error arises when either the semantics of the program or the semantics of the programming language are misrepresented. In this case one is formally reasoning about either 1) a program that is different from the one we originally intended to verify (i.e., the semantics of the program were misrepresented), or 2) we are verifying the correctness of the program with respect to a programming language that is different (i.e., has different semantics) from the language in which the program was actually written. Empirical evidence suggests that such errors occur more frequently if the notation which is used for the programming language and the notation which is used to reason about the semantics of the program are very similar. For this reason, it is desirable to have a formal notation which is orthogonal to the syntax of the programming language. The second type of error arises if mistakes are made in the reasoning process. This type of error can be greatly minimized through the automation of the reasoning process. A problem that arises due to the automation of program verification is that theorem provers often have difficulty verifying programs. Generally this difficulty can be diminished through the use of effective search strategies and inference rules. However, because the amount of work required to verify a program can grow exponentially with the size of the program, there seems to be little hope of verifying large programs either by hand or automatically.

In an effort to facilitate the construction of correct programs, specification languages are being developed. The motivation behind these languages is that solutions to problems can be stated and reasoned about at a level of abstraction which is divorced from the implementation details of the program realizing the solution. The impact that this has had on formal verification is that it has often been observed that proving the correctness of a formal specification is almost trivial in comparison to proving the correctness of a program which implements the specification. The price that is paid for this relative ease of verification is that specification languages, by definition, create a gap between the high-level realm of the solution and the concrete realm of the implementation of that solution.

Traditionally, the gap between formal specifications and executable programs is bridged by programmers whose job it is to study the formal specification and once they fully understand the specification, to write a program satisfying (i.e., realizing) the specification. In this approach, the transformation from formal specification to program is essentially accomplished in a single step that is carried out by the programmer. After a program has been produced in this fashion, an important question that needs to be answered is “Does the program satisfy the formal specification?” The answer to this question can essentially be obtained only by verifying that the program solves the problem
defined by the formal specification. Empirical evidence suggests that this kind of verification is not much different than an outright verification that the program is correct (i.e., a verification that makes no use of the formal specification). The reason for this is due because it has generally been observed that a large part of the formal program verification process is concerned with implementation details and how they interact to solve the desired problem.

Due to the difficulties encountered in verifying that a program satisfies a formal specification, another paradigm for obtaining programs from formal specifications is being explored. In this paradigm, the gap between formal specifications and programs is bridged by formal means. The idea is that if one could somehow correctly transform a formal specification into a program which implements it then, by definition, the resulting program would satisfy the formal specification from which it was derived. If such a transformation can be accomplished, then formally verifying that a program solves a desired problem would amount to 1) verification of the formal specification, and 2) verification of the correctness of the transformation process.

In this research, a paradigm is investigated in which the gap between formal specifications and programs is bridged through transformations. The idea is to transform a formal specification into an executable program by a sequence of correctness preserving transformations. The hope is that the sequence of transformations will be made up of individual transformations that are simple enough to be easily understood and easily verified by formal means. The goal of this thesis is to construct a methodology that will permit formal verification of general program transformations. In particular we are interested in proving the correctness of transformation sequences that are capable of transforming formal specifications into efficient executable programs.

1.2 The Transformational Approach

1.2.1 Overview

In mathematical terms a transformation, \( T \), can be viewed as a function that takes a program \( x \) as its input and produces a program \( T(x) \) as its output. We will often refer to the program \( x \) as the input program of a transformation and \( T(x) \) as the output program. In this research, we investigate the class of transformations that are based on rewrite rules. A rewrite rule is simply an expression stating that one syntactic schema (the input schema) can be rewritten into a different syntactic schema (the output schema). Rewrite rule transformations have an internal structure that is something like:

\[
T_1 \overset{\text{def}}{=} \text{schema } a \Rightarrow \text{schema } b
\]

The above rewrite rule (i.e., transformation \( T_1 \)) states that “schema \( a \)” should be rewritten to “schema \( b \)”. The application of \( T_1 \) to an input program \( x \), containing one or more occurrences of “schema \( a \)”, will involve rewriting some or all of the occurrences of “schema \( a \)” into “schema \( b \)”. How many, which, and in what order occurrences of the input schema are rewritten depends on the transformation system that one is using.

In practice, transformations producing small syntactic changes are composed into transformation sequences whose overall effect can be quite dramatic. Let \( T_{1,n} \overset{\text{def}}{=} T_1; T_2; \ldots; T_n \), where \( T_i; T_{i+1} \) denotes the sequential composition of transformation \( T_i \) and \( T_{i+1} \). Application of \( T_{1,n} \) to a program \( x \) is denoted \( T_{1,n}(x) \), and is accomplished by first applying \( T_1 \) to \( x \) and then applying \( T_2 \) to the program \( T_1(x) \), and then applying \( T_3 \) to \( T_2(T_1(x)) \) and so on.

In Section 1.1 we mentioned that the goal of this research is to construct a methodology capable of verifying the correctness of transformation sequences capable of transforming formal specifications into executable programs. This raises the question of the domain and range of transformations. Should the domain and range of a transformation be the same? Clearly, in the transformation sequence \( T_{1,n} \), the range of \( T_1 \) must be a subset of the domain of \( T_2 \). In general, the range of \( T_{i-1} \) must be a subset of the domain of \( T_i \) whenever \( T_{i-1} \) and \( T_i \) belong to the same transformation sequence. In this dissertation we restrict our attention to transformations whose domain and range are the same (i.e., transformations whose input and output sets are the same.) This being the case, we can talk about a transformation, \( T \), and the set \( D \) on which it is defined. In this model, \( D \) is a set that is defined in terms of a context-free grammar (e.g., a programming language). This means that transformations can be viewed as functions that map programs to other programs.

How can this model be used when transforming formal specifications into programs? The solution is to view both the specification language and the implementation language as part of a wide spectrum language. Thus specifications and implementations both belong to the same domain (i.e., they belong to the same language). Using this approach, transformation sequences can be constructed that transform programs consisting entirely of high-level constructs.
(i.e., specifications) into programs consisting entirely of low-level programming constructs (i.e., implementations). In this research the wide spectrum language we will study is Poly. Poly is a language consisting of functional and imperative constructs. This language is discussed in greater detail in Chapter 6.

1.2.2 A Formal Notation

When applying transformations to programs to obtain other programs, one often is interested in properties that either the input program or the output program possess. In this section we do not give any examples of specific properties; rather, we simply introduce a notation that will allow us to discuss such properties in abstract terms.

Rather than creating our own terminology we will adapt the standard notation used in the field of program correctness [30]. Program correctness is a field of computer science that is concerned with formally reasoning about the semantics of programs in terms of preconditions, postconditions, weakest preconditions, and program states. In order to give the reader a better understanding of how these terms will be used in our work, we briefly define each of these terms as they are traditionally used.

state – A code segment begins its execution in an initial state, and (if it terminates) will end its execution in a final state. We think of a state as a set of tuples where each tuple is made up of a variable and the value it denotes. For the purposes of this discussion we will use \( x \) to denote an arbitrary state. Using this model of a state, a code segment can be described as a function that accepts a state as its input (i.e., the initial state) and produces a state as its output (i.e., the final state). For example, if \( x \) denotes the initial state in which the code segment \( S \) will begin its execution, then the final state produced by the execution of \( S \) can be denoted \( S(x) \). Note that if the execution of \( S(x) \) never terminates then \( S(x) \) is an infinite state.

properties – Let \( U \) denote the set of all states. A property \( P_i \) is an \( n \)-place predicate whose variables belong to the universe \( U \). We will denote predicates (i.e., properties) by the symbols \( P \), \( Q \), and \( R \). In relation to a given code segment \( S \), predicates of the form \( P_i(x) \) are considered to be preconditions, and predicates of the form \( P_j(x, S(x)) \) are considered to be postconditions.

To further highlight the distinction between preconditions and postconditions, one generally uses the symbol \( Q \) to denote preconditions and the symbol \( R \) to denote postconditions. For example, if we wished to assert that a code segment \( S \) beginning its execution in any initial state satisfying the precondition \( Q \), will terminate its execution in a state satisfying the postcondition \( R \), we would write this as:

\[
\{Q(X)\} S \{R(X, S(X))\}.
\]

wp – The term \( wp \) denotes the weakest precondition which for a given code segment \( S \) and a given postcondition \( R \), is defined as \( X \subseteq U \) such that \( \forall x \in X : R(x, S(x)) \) holds.

In the field of Program Correctness one is interested in formally reasoning about program assertions (i.e., about predicates having the following forms):

1. \( Q_i(X) \) \( \{ \) \( R_j(X, S(X)) \) \( \}\)\
2. \( \{Q_i(X)\} S \{R_j(X, S(X))\} \)

The difference between the first and second predicate is that the first one denotes partial correctness while the second denotes total correctness. The first predicate is to be read: “If the execution of the code segment \( S \) is begun in an initial state satisfying the precondition \( Q_i \) then \( S \) terminates, it will terminate in a final state satisfying the postcondition \( R_j \).” The second predicate is to be read: “If the execution of the code segment \( S \) is begun in an initial state satisfying the precondition \( Q_i \) then \( S \) will terminate in a final state satisfying the postcondition \( R_j \).” Note that the objects manipulated here are 1) programs, and 2) variables and together with their associated values.

In this research we are interested in reasoning about the properties a given transformation might establish. To enable discussion of this topic in familiar terms we will adopt, with slight modification, the terms used in the field of program correctness. We stress that the definitional changes we will make are minimal. To make a distinction between the terms used in program correctness and the terms we use, we will initially subscript all our terms with the symbol \( T \). The symbol \( T \) is intended to point out that the term is one that will be used in discussions involving transformations. Before we give the definitions of \( \text{state}_T \), \( \text{precondition}_T \), \( \text{postcondition}_T \), and \( \text{weakest}_T \), bear in mind that transformations are objects that modify programs. This means that in our paradigm a program plays the role that a data element played in the program correctness paradigm. In summary then, where the program correctness
Section 1.2 The Transformational Approach

paradigm deals with programs and the data elements they manipulate, our paradigm is concerned with transformations and the programs they manipulate.

\textit{state} \_T - In a manner similar to the execution of a code segment, a transformation, \textit{T}, begins its execution in an initial \textit{state} \_T, and (if it terminates) will end its execution in a final \textit{state} \_T. The difference between a code segment and a transformation is that the initial state of a code segment consists of a set of tuples while the initial state of a transformation is a single tuple consisting of a variable and the program (i.e., the value) it denotes. What this means is that one can think of the initial \textit{state} \_T of a transformation simply as being a program. In this thesis we will use the symbol \( x \) to denote an arbitrary \textit{state} \_T (e.g., the initial state (program) of a transformation) and the expression \( T(x) \) to denote the final \textit{state} \_T (i.e., the output program) that is produced when the transformation \( T \) is executed with respect to (i.e., applied to) the initial \textit{state} \_T \( x \). Note that if the execution of \( T(x) \) never terminates then \( T(x) \) \( \equiv \perp \).

\textbf{properties} - Let \( \mathcal{U}_T \) denote the set of all \textit{states} \_T. A property \( P_i \) is an \textit{n-place} predicate whose variables belong to the universe \( \mathcal{U}_T \). We will denote predicates (i.e., properties) by the symbols \( P, Q \), and \( R \). In relation to a given transformation \( T \), predicates of the form \( P_i(x) \) are considered to be preconditions, and predicates of the form \( P_j(x, S(x)) \) are considered to be postconditions.

Again, to further highlight the distinction between preconditions and postconditions, we will generally use the symbol \( Q \) to denote preconditions and the symbol \( R \) to denote postconditions.

In this dissertation we are interested in formally reasoning about predicates having the following forms:

- \( Q(x) \{ T \} R(x, T(x)) \)
- \( Q(x) \{ T_{1,n} \} R(x, T_{1,n}(x)) \)

Note that in contrast to program correctness which is interested in both partial and total correctness, we are only interested in partial correctness. Our desire is to restate the above predicates in terms of theorems and to then formally prove these theorems. Towards this end we rewrite the first predicate as the following (equivalent) theorem:

\textbf{Theorem 1} \( Q(x) \Rightarrow R(x; T(x)) \)

If, for a particular \( T \), the above theorem is proven to hold, then we can conclude that if the property \( Q \) is satisfied by an input program \( x \) then \( R(x, T(x)) \) will hold.

Our intention is to prove theorems about individual transformations and to then use these results to prove theorems about transformation sequences. This will enable us to prove theorems of the form:

\textbf{Theorem 2} \( Q(x) \Rightarrow R(x; T_{1,n}(x)) \)

In this thesis we refer to the precondition associated with a transformation \( T \) as the \textit{context} of \( T \). The \textit{context} of transformation \( T \) defines the set of elements in \( D \) (i.e., the programming language in which we wish to carry out our transformations) to which we are interested in applying the transformation \( T \). Having said this, we will often use the symbol \( Q \) to denote the context of a transformation.

During the course of this research two particular instances of the above theorems have occurred so frequently that they deserve special mention. The two theorems are:

\textbf{Theorem 3} \( \text{true}(x) \Rightarrow \text{Correct}(x, T_{1,n}(x)) \) where \( \text{true}(x) \equiv x \in D \).

\textbf{Theorem 4} \( \text{true}(x) \Rightarrow \text{Correct}(x, T_i(x)) \) where \( \text{true}(x) \equiv x \in D \).

The first theorem states that all input programs \( x \) will be correctly transformed by the transformation sequence \( T_{1,n} \). That is, the output program \( T_{1,n}(x) \) is correct with respect to the input program \( x \). The second theorem makes a similar statement about the individual transformation \( T_i \).

\textbf{1.2.3 Properties of Programs}

Given a transformation sequence \( T_{1,n} \) and an arbitrary context \( Q \) one can investigate what properties are possessed by the output programs \( T_{1,n}(x) \), for which \( Q(x) \) holds. In this section we briefly discuss some of the properties that might be of interest to people writing program transformations.
Properties of programs can be viewed as being syntactic or semantic in nature. An example of a syntactic property is:

\[ P_1(y) \overset{\text{def}}{=} \text{the bodies of lambda expressions consist of single variables} \]

An example of a semantic property is:

\[ \text{Correct}(y, y') \overset{\text{def}}{=} \text{program } y' \text{ is correct with respect to program } y. \]

Among the various properties that an arbitrary transformation might possess, perhaps the most important property is that of correctness. Traditionally, a transformation possessing the property of correctness is referred to as correctness preserving, the implication being that correct with respect to the input program is meant. In the literature one often sees phrases like: transformation \( T \) is correct or transformation \( T \) is correctness preserving. These are accepted ways of stating that \( T \) establishes correctness. If a transformation \( T \) is correctness preserving, then \( T \) may be applied to any program, \( x \), producing an output program \( T(x) \) such that \( T(x) \) is an acceptable substitute for \( x \). In a similar fashion a computer program, \( x' \), that is obtained from a high-level formal specification through a sequence of correctness preserving transformations is guaranteed to be at least as good as the specification from which it was obtained. In other words, \( x' \) is an acceptable implementation of the formal specification.

In this dissertation, phrases like the correctness of a transformation refer to the mathematical concept of less definedness as it is used in the field of program transformation and refinement calculus [58], [50], [12], [34], [63], [31], [46], [56]. It has been shown that a program transformation, \( T \), is correct if the result of applying \( T \) to any program, \( x \), yields a program \( x' \) such that \( x \sqsubseteq x' \). In Chapter 3 we will formally discuss the \( \sqsubseteq \) relation.

1.2.4 The TAMPR Transformation System

TAMPR [8] is a transformation system that views programs in terms of their syntax derivation trees. A TAMPR transformation can be abstractly viewed as a function that when given an input program \( x \), searches the syntax derivation tree of \( x \) for a specific tree structure (schema). If such a schema is found it is rewritten in the manner defined by the transformation. The result is a syntax derivation tree (i.e., output program) that is a modification of the input tree (i.e., the input program). Transformations may be applied to a program individually or they may be composed with other transformations and the resulting transformation sequence may be applied to a program.

TAMPR offers a powerful set of pattern matching constructs that permit the construction of a rich set of transformations.

1.2.5 Goal of This Research

The goal of this research is quite simply stated. We plan to construct an automateable methodology that will allow us to formally prove the correctness of TAMPR transformation sequences. That is, we are interested in proving theorems of the form:

\[ \text{true}(x) \Rightarrow \text{Correct}(x; T_{1,n}(x)) \]

Because the correctness of a transformation sequence depends on the correctness of the individual transformations that the sequence is made up of, our research is broken down into two sections.

The first section is discussed in Chapter 6 and focuses on the construction of an automateable methodology that will allow formal verification of the correctness of individual TAMPR transformations. Using this methodology we want to be able to prove theorems of the form:

\[ Q(x) \Rightarrow \text{Correct}(x; T(x)) \]

where \( Q \) is an arbitrary precondition, and the domain/range of \( T \) is the wide spectrum language Poly. Recall, we mentioned earlier that Poly is the wide spectrum language in which we wish to study the problem of transformation verification. After a methodology has been constructed in which correctness proofs of individual transformations can be achieved, we then automate this methodology using the automated reasoning system OTTER [49].

The second section focuses on the additional problems that are encountered in proving the correctness of transformation sequences. As we will see in Chapter 7, proving the correctness of a transformation sequence can involve more than just proving the correctness of individual transformations.

In summary then, the goal of this research is to 1) construct a framework that will allow formal proofs of the
Section 1.2 The Transformational Approach

correctness of individual transformations, 2) construct a framework that will allow proofs of the correctness of transformation sequences, and 3) demonstrate how correctness proofs of individual transformations can be obtained with the help of OTTER.
Chapter 2
Formal Semantics

As was mentioned in the previous chapter, one of the things needed in order to formally reason about programs is a formal semantics for the programming language one is dealing with. In this chapter we discuss how one can formally define the semantics of a language whose syntax is defined in terms of a context free grammar. In the following section, we give a brief overview of some of the methods available for formally defining the semantics of programming languages. Then in the following section we discuss in greater detail the method we have chosen to formally define the semantics of the wide spectrum language, Poly, that we are working with. In addition we will also discuss some of the theoretical restrictions that must be placed on the semantics of programming language constructs in order for our proof methodology to work.

2.1 Survey of Formal Semantics

2.1.1 Background

The goal of formal semantics is to give meaning to syntactic objects (i.e., symbols and groups of symbols). However, since the formal meaning that is assigned to a syntactic object is itself a syntactic object, this approach seems somewhat circular. This “chicken and egg” problem can be resolved in three ways.

In the first approach, one assumes that there exists a set of syntactic objects, $M_f$ that have a formal semantics which is already known. We will refer to such objects as semantic objects. In this paradigm, the objects in $M_f$ form what we will call the semantic foundation. If the set $M_f$ is rich enough, then all syntactic objects that interest us (i.e., all syntactic objects in our programming language) will have a semantics that can be expressed by an object in $M_f$. Thus the semantics of a set of syntactic objects can then be formally defined by defining the correspondences between syntactic objects and semantic objects. Denotational semantics uses this approach.

In the second approach, one assumes that there exists a set of abstract semantic objects whose semantics is known. The difference between an abstract semantic object and a semantic object is that an abstract semantic object does not have a unique symbolic (physical) representation whereas a semantic object has a symbolic representation. Any symbol can be used to denote an abstract semantic object. Because many different symbols can be used to denote the same abstract semantic object, in this paradigm symbols behave like variables that are quantified over the domain of abstract semantic objects. This being the case, the question arises whether a symbol denotes a unique abstract semantic object. Traditionally this problem is overcome by defining enough properties that the abstract semantic object corresponding to a symbol must possess in order to ensure that the symbol indeed denotes a unique abstract semantic object. Algebraic specifications is an example of this approach.

In the third approach, syntactic objects are given a formal semantics by defining all the relationships between the syntactic objects themselves. In this approach the syntactic objects acquire a meaning unto themselves. Axiomatic semantics is an example of this approach.

2.1.2 Axiomatic Semantics

In axiomatic semantics, syntactic objects are given meaning through axioms defining how syntactic objects relate to one another. This is perhaps best seen through an example. For this discussion, pretend that you know nothing
about binary numbers. That is, you don’t know what a binary number looks like, and you don’t know how to add or
subtract binary numbers. This being the case, you would view binary numbers and binary operations (e.g., addition
and subtraction) as purely syntactic objects that are devoid of meaning. At this point, suppose you are given a
complete set of axioms defining addition and subtraction for binary numbers. You would then be able to use these
axioms to determine if certain equalities hold between these syntactic binary objects. For example, using the axioms
given to you, you could formally prove that the binary expression \(101 + 1\) is in fact equal to the binary number \(110\).
In other words, with the aid of the axioms you could formally reason about binary numbers. Note, one thing that
these axioms do not provide is how binary numbers relate to, say, base 10 numbers. Thus all reasoning concerning
binary numbers must be carried out in terms of binary numbers themselves.

The axiomatic approach to semantics was developed primarily for the purpose of program verification [25] Hoare
[32] has developed an axiomatic method that has been used to define the semantics of a subset of Pascal [33]. One of
the drawbacks of this method is its inability to deal with common language features such as side effects and scoping.

2.1.3 Algebraic Specifications

2.1.3.1 Introduction This section gives an overview of abstract algebras as presented in [36]. One of the reasons
for further investigating this method of specification is due to the fact that it is extensively used in [59], [58], and
[60].

The principle advantage, that is offered by an algebraic specification of a language or a data type, is the fact that
equational and inductive reasoning can be used to formally verify properties of objects which are defined in such a
manner.

2.1.3.2 Signatures An abstract algebra can be viewed as an entity that consists of a family of sets of objects,
and a number of functions whose inputs and outputs are elements of these sets. Such an abstract algebra (or algebra
for short) can define several abstract data types and is therefore often referred to in the literature as a many-sorted
algebra.

In order to be able to talk about an algebra one needs some concrete manifestation of the algebra. To serve this
purpose we introduce the concept of a signature. A signature is a system of notation which allows us to manipulate a
concrete manifestation of the algebra. In other words, a signature gives substance to an otherwise intangible algebra.

A signature is a system of notation, where names are given to the sets, functions, and objects of the algebra. The
names of sets are called sorts (or types). The names of functions are called nonnullary operations (or operations),
and the names of objects are called nullary operations (or constants). For example, one can define the signature of
an abstract Boolean algebra as follows:

1. **Sorts**: \{ boolean \}
2. **Constants**: \{ true, false \}
3. **Operations**:
   (a) **And**: boolean \( \times \) boolean \( \rightarrow \) boolean
   (b) **Or**: boolean \( \times \) boolean \( \rightarrow \) boolean
   (c) **Not**: boolean \( \rightarrow \) boolean

Currently what is missing from the signature is a mapping from the various symbols to abstract entities in the
algebra. Because the signature fails to supply such a mapping, it is possible for a single signature to denote a specific
algebra in more than one way. For example, one mapping might map the symbol \(true\) to the abstract object \texttt{true},
whereas another mapping might map the symbol \(true\) to the abstract object \texttt{false}. In order to precisely define how we
want our signature to denote the boolean algebra we will supply a mapping \(b\) from concrete members of the signature
to abstract members of the algebra. This makes a distinction between the signature object \texttt{true} and the abstract object
\texttt{b.true} which it denotes.

A signature together with a mapping guarantees us that if we consider a specific algebra, then we have a model
for unambiguously denoting objects of the algebra. One problem that still remains, however is that a signature can
denote several algebras.

All that the signature and mapping model has given us is a symbolic environment in which we can discuss (and
reason) about abstract algebras. Our goal is to somehow extend the model of a signature to allow us to precisely
define a unique algebra. Unfortunately, this cannot be done so we will have to settle for the next best thing which is
to extend the signature model to uniquely define an isomorphic class of algebras.
2.1.3.3 Homomorphism and Isomorphism In a nutshell, a homomorphism is a mapping from a more complex algebra to a consistent algebra which has a simpler structure. As a special case, a mapping can be made between two algebras which are consistent and which have the same structural complexity. In this case, the mapping is called an isomorphism. Formally a homomorphism from $A$ to $B$ is given by a family of mappings $\{f_1, f_2, \ldots, f_m\}$ which map objects in algebra $A$ to objects in algebra $B$ so that the behavior of the operations is preserved. This gives us the following rules for mapping objects in algebra $A$ to objects in algebra $B$:

1. **Constants.** For each constant object $s_i$ declared in the signature of $A$, we map this constant (abstract) object to an appropriate constant $s_j$ in the signature of $B$.

   $$f_i(\delta_{A,s_i}) = \delta_{B,s_j}$$

2. **Functions.** For each operation $g_j$ declared in the signature of $A$, we map this function to a function $g_j$ in $B$ so that the following is preserved:

   $$f_i((\delta_{A,g_j})(\delta_{A,t_1}, \delta_{A,t_2}, \ldots, \delta_{A,t_k})) = (\delta_{B,g_j})(f_i(\delta_{A,t_1}), f_i(\delta_{A,t_2}), \ldots, f_i(\delta_{A,t_k}))$$

In the above expression, it is assumed that the function in $A$ which is denoted by the symbol $g_j$ has $k$ arguments, and that the objects $t_1 \ldots t_k$ are arbitrary objects in $A$ having the correct type. In English then, what the above expression is saying is that if you compute a result in $A$ and then map this result to $B$ you will get the same answer as if you mapped the function and all its arguments to $B$ and then computed the result in $B$.

An example of a homomorphism can be seen by considering the abstract algebra, $A$, that is syntactically represented by the natural numbers $\text{Nat}$, the successor function $\text{succ}$, and the constant symbol $\text{zero}_A$ and the abstract algebra, $B$, consisting of the numbers $\text{zero}_B$ and $\text{one}_B$ and the $\text{add-one-mod-two}$ function. The homomorphism from $A$ to $B$ is achieved by mapping all even natural numbers in $A$ to the element $\text{zero}_B$, by mapping all odd natural numbers in $A$ to the element $\text{one}_B$, and by mapping the $\text{succ}$ function to $\text{add-one-mod-two}$. Note that the algebra $A$ can be viewed as a refinement of the algebra $B$.

Two algebras, $A$ and $B$ are considered isomorphic iff there exists a homomorphism from $A$ to $B$ and there exists a homomorphism from $B$ to $A$.

2.1.3.4 Variable-free Term Language, Word Algebra, Variety and Term Algebra. A variable-free term is a term which can be constructed by any finite legal combination of constants and operations which are given in the signature. The set consisting of all such terms is called the variable-free term language.

A variable-free term language may or may not be able to denote every element in an algebra. If such is the case, then there exists a discrepancy between the variable-free term language that can be generated from the signature, and the objects of the abstract algebra. Because of this it is often interesting to consider an abstract algebra where all the objects can be expressed by elements of the variable-free term language. Such algebras are called term algebras. A special kind of term algebra is called the word algebra. The word algebra can be viewed as the algebra where the tangible and the abstract meet. That is, in a word algebra, the variable-free terms do not denote abstract objects. In a word algebra, the variable-free terms are the objects.

A variety over a signature is the set of all possible algebras denoted by that given signature.

2.1.3.5 Categories and Initial Algebras A category of algebras over a signature is a set of algebras denoted by the signature, together with a number of homomorphisms between these algebras, including the identity homomorphism.

An algebra $\mathcal{I}$ is initial in a category $\mathcal{C}$ iff:

$$\mathcal{I} \in \mathcal{C} \land (\forall \mathcal{A} \in \mathcal{C} : (\exists H : H(\mathcal{I}, \mathcal{A})))$$

where $H$ denotes a unique homomorphism. That is, $\mathcal{I}$ is initial iff $\mathcal{I}$ is in the category, and iff a unique homomorphism exists between $\mathcal{I}$ and all other algebras in the category.

As a notational convention, when we refer to a category over a signature without giving the set of homomorphisms of the category, we are referring to the category defined by all possible homomorphisms.

It has been shown that the initial algebra of a category always exists. In fact, the word algebra is the initial algebra of its category.

2.1.3.6 Variables, Substitution, and Interpretation. In this section, we introduce terms of the signature which contain variables. Variables are used to represent a particular class of objects. With the introduction of variables
Section 2.1 Survey of Formal Semantics

comes the notion of substitution, which is basically a replacement of variables by terms, and interpretation which is an assignment of variables to a specific term. All of these concepts have direct analogies in theorem proving and will not be discussed further in this paper, with the exception of pointing out that a term language of a signature is a language containing words which may or may not have variables in them.

2.1.3.7 Axioms, Presentations, and Theories. So far we have defined the concept of signatures which can be used to denote abstract algebras. In trying to denote a unique algebra by a signature, we introduced the concept of a mapping which associates concrete symbols with abstract objects. Unfortunately, a signature together with a mapping is not strong enough to define a unique algebra. The concepts of homomorphism, isomorphism, and category were introduced in order to allow us to concisely discuss the abstract algebras which are denotable by a given signature. In this context we defined an initial algebra of a signature which is unique up to isomorphism. What we have achieved so far is a means for associating signatures with abstract algebras in a unique (up to isomorphism) way. The problem that we have now is that the algebras which are definable by signatures are not very interesting. (They have no structure.) The reason for this is that each variable-free term denotes a different object. We can impose structure on such an algebra by letting several terms denote the same object. This can be achieved through the use of axioms which relate terms to one another.

A signature which is extended through the addition of axioms is called a theory or a presentation. The question can be asked whether or not an algebra satisfies a given theory. We also extend the notion of variety and termalgebra to theories. A variety of a theory is the set of all algebras which satisfy the axioms of the theory. A termalgebra is an algebra denoted by the theory in which each object can be denoted by a variable-free term of the theory.

2.1.3.8 Theories, Categories, and Initial Algebras. In this section we extend the notion of category, and initial algebra to theories. The definitions here are the same as the ones given for signatures with the exception that the word signature is replaced by the word theory. If the initial algebra of a theory exists, it is unique up to isomorphism.

An interesting question is: How can the initial algebra of the category of a theory be found? To find such an initial algebra consider the word algebra of the signature of the theory. Terms (i.e., variable-free terms) of this algebra can be grouped into equivalence classes according to the axioms of the theory. The initial algebra of the theory is obtained from this model by viewing the equivalence classes created in the word algebra as single entities, and by creating a new algebra containing only these entities. The resulting algebra is called the quotient algebra, and it is the initial algebra of the category of the theory.

2.1.3.9 Reasoning Methods. Theories give us a framework for formally describing interesting abstract algebras. We have two tools at our disposal which allow us to reason about these algebras. Equational reasoning can be used to prove theorems about the relationships of objects in the algebra. This form of reasoning is sound and complete. Theorems derivable through this form of reasoning are satisfied by every algebra in the variety over the theory.

Another form of reasoning is called inductive reasoning. As the name states, this form of reasoning involves induction. Since induction requires enumeration of the elements inducted on inductive theorems are only satisfied by termalgebras, of which the initial algebra is one. Proofs in this framework involve proving the desired induction as well as showing some kind of enumerability. This generally involves some kind of canonicalization of terms in the algebras.

2.1.3.10 Initial versus Final Algebras Often times in the literature, initial or final algebras are mentioned. The difference between an initial algebra and a final algebra is as follows:

1. In an initial algebra, two variable-free terms are different unless it can be proven from the axioms that they are the same.
2. In a final algebra, two variable-free terms are the same unless it can be proven from the axioms that they are different.

In a nutshell then, the initial algebra defined by a theory is the termalgebra satisfying the axioms of the theory that has the greatest possible number of objects, whereas the final algebra is the termalgebra satisfying the given axioms that has the smallest possible number of objects. We clarify this statement with the following example:

In this example we are assuming that natural numbers and a boolean algebra are appropriately defined.

1. Sorts : \{ X \}
2. Constants :
   (a) empty : X;
3. **Operations**
   
   (a) \( \text{insert} : \text{nat} \rightarrow X \rightarrow X \)
   
   (b) \( \text{isin} : \text{nat} \rightarrow X \rightarrow \text{boolean} \)

4. **Axioms.**
   
   (a) Declare \( n, n_1, n_2 : \text{nat}; x : X \);
   
   (b) \( \text{insert}( n_1, \text{insert}( n_2, x )) = \text{insert}( n_2, \text{insert}( n_1, x )) \)
   
   (c) \( \text{isin}( n, \text{empty} ) = \text{false} \)
   
   (d) \( \text{isin}( n_1, \text{insert}( n_2, x )) = \)

   \[
   \begin{array}{l}
   \text{ifthenelse}( \text{eq}( n_1, n_2 )) \\
   \text{then true} \\
   \text{else isin}( n_1, x )
   \end{array}
   \]

   The initial algebra defined by this theory is a \textit{bag}. The first axiom states that the order of insertion is irrelevant; however, the number of times an item has been inserted is relevant. This is true, because in initial algebras two variable-free terms are different unless they can be proven to be the same.

   The final algebra defined by this theory is a \textit{set}. This is because two variable-free terms are the same unless they can be proven to be different.

### 2.1.4 Denotational Semantics

Denotational Semantics is an instance of the first approach that was discussed in 2.1.1, and it is the method we have chosen to define the semantics of the language that we will be dealing with. Given a language, \( L \), whose syntax is defined in terms of a context free grammar \( G \), sentences in \( L \) can be viewed in terms of the sequence of productions in \( G \) that were used to create them. If, for a given sentence \( s \in L \), the sequence of productions that was used to generate \( s \) is depicted in tree form, we have what is commonly called the syntax derivation tree (or SDT) for \( s \). In denotational semantics, members of a language (i.e., sentences) are viewed in terms of their SDT’s. The semantics of a node, \( n \), occurring in an arbitrary SDT is defined in terms of an expression constructed from the composition of elements belonging to the set \( M_f \) and the \textit{meaning} (i.e., semantics) of the nodes that are the immediate descendents of \( n \) in the particular SDT. Note, a node in an SDT denotes either a terminal or nonterminal symbol in the grammar of \( L \); thus we will use node and nonterminal/terminal interchangeably.

When defining the semantics of a language \( L \) using denotational semantics, one begins with a mathematical foundation \( M_f \) and one constructs a set of valuation functions \( V \) whose elements are functions that assign various \textit{meanings} to the nonterminal/terminal symbols in the grammar of \( L \). In general, a valuation function, \( B \), will have the form:

\[
B[[T]] \overset{\text{def}}{=} e
\]

where \( e \) is an expression consisting of objects (e.g., constants and operations) belonging to our mathematical foundation, \( M_f \), and valuation functions whose input SDT’s are subtrees of \( T \). (Recall, the mathematical foundation \( M_f \) is simply a set of syntactic objects whose semantics are assumed to be known.) The goal is to construct a valuation function, \( C_{\text{PROG}} \), that can be used to determine the \textit{meaning} of entire programs. Informally stated, the \textit{meaning} of a program is the effect produced by the program when it is executed. In order to capture this notion of \textit{meaning}, we essentially construct a mathematical model of computation, \( M \), that can be used by the denotational semantic definitions to define the effects of program execution in abstract mathematical terms.

Generally, for real programming languages, one cannot directly write \( C_{\text{PROG}} \). Instead, one must break down the problem into a group of smaller problems. For example the problem of defining the meaning of a Pascal program, \( p \), might be broken down into defining the meaning of the functions and procedures of \( p \) followed by defining the meaning of the \textit{main} portion of \( p \). This breakdown will often result in the creation of other valuation functions. Valuation functions are broken down until one obtains a set of valuation functions for which it is possible to define their semantics directly (i.e., without having to break them down further). At this point, one has constructed a formal denotational semantics for the language \( L \).

#### 2.1.4.1 Example: Binary Expressions

Suppose the language of binary expressions is something that we wish to create. That is, suppose nobody in our circle of peers has ever heard of a binary number or binary expression. (Note, the notion of a circle of peers is central to the whole denotational semantic approach because, implicitly, it is basis
Section 2.1 Survey of Formal Semantics

for the construction of the mathematical foundation $M_f$.) We can create the language of binary expressions by simply giving a formal grammar for it. By constructing such a grammar what we are doing is providing a mechanism that will allow our peers (who know nothing about binary expressions) to construct syntactically correct binary expressions and also to recognize binary expressions when they see them. However, just because our peers can construct and recognize binary expressions does not mean that they understand the semantics of the expressions they have constructed. Furthermore, if they don’t understand the semantics of a single binary expression they certainly cannot be expected to reason about semantic relationships and properties that binary expressions might possess. In order to give our peers the ability to reason (on the semantic level) about binary expressions we need to formally define the semantics of the language of binary expressions that we have created.

Denotational semantics is a way to formally define the semantics of a language in terms of concepts that are understood by one’s peers. For example, suppose that everyone in our circle of peers understands the semantics of base 10 numbers and expressions. The denotational semantic definitions that are used to define the language of binary expressions can avail themselves of this knowledge. In other words, denotational semantics can take advantage of the fact that our peers understand the semantics of base 10 numbers and expressions in order to define the semantics of binary expressions. This is accomplished by letting base 10 numbers and expressions belong to our mathematical foundation $M_f$. In this case, all that the denotational definitions provide is a mapping from binary expressions (which are syntactic objects) to base 10 expressions (which are semantic objects). Such a mapping will map binary numbers to base 10 numbers and binary operations to base 10 operations. In this fashion binary expressions are understood (i.e., given a semantics) in terms of base 10 expressions.

Generally, defining the semantics of a language using denotational semantics is more complicated than the binary expressions example we have just discussed. For example in an imperative programming language like Pascal, scoping, aliasing, and parameter passing are just some of the aspects of the language that need to be captured by the denotational semantics. Such language properties are generally defined by using the primitive constants and operations in $M_f$ to construct a mathematical model of computation and then defining the execution of a program in terms of this model.
Chapter 3
Proving the Correctness of Program Transformations

In the previous chapter we surveyed three approaches that can be taken when formally defining the semantics of a programming language. As we mentioned before, one goal of our work is to be able to formally prove the correctness of program transformations. Towards this end, it is necessary to formally define the semantics of the programming language we wish to study. In this chapter, we formally define the notion of correctness as well as investigate the consequences of this definition.

3.1 The Formal Definition of Correctness.

In this thesis, we will use the symbol (i.e., the monotonically less-defined-than symbol) to denote correctness. For those not familiar with monotonicity it is often difficult to see how captures correctness. Therefore in this section, as an aid to the reader, we give a formal definition of correctness in terms of weakest preconditions. We hope that this definition will provide the reader with an intuitive understanding of correctness as well as its implications in the field of program transformation. We then go on to show the relationship between the weakest precondition definition of correctness and the definition of correctness that is expressed in terms of the symbol. Essentially both definitions are the same. However, the version of correctness is broader because it applies to a larger set of syntactic objects than the definition of correctness based on weakest precondition semantics. An example of this is given below. In addition, the relation is discussed in greater detail later in this section as well as in the following sections.

Given two programs , we say that is correct with respect to iff is a refinement of . I n other words correctness and refinement are synonymous. Now the program is a refinement of if for every postcondition R,

\[ wp(p_1, R) \Rightarrow wp(p_2, R) \]

where denotes the weakest precondition function. Another way to express the notion of correctness (i.e., refinement) is through the symbol . If we write

\[ \text{meaning}(p_1) \sqsubseteq \text{meaning}(p_2) \]

then we are saying that the meaning of is a refinement of the meaning of . One can also say that the meaning of is less-defined-than the meaning of . It should be noted that the set of syntactic objects for which it is possible to express, through weakest precondition semantics, the refinement relation is a proper subset of the set of syntactic objects for which it is possible to express the refinement relation using the symbol. For example, weakest precondition semantics are generally used to define composite syntactic objects like assignment statements, conditional statements, blocks of code, and so on. Weakest precondition semantics are generally not used to define the semantics of nondecomposable syntactic objects like integers and identifiers. In the weakest precondition paradigm the semantics of integers and identifiers are assumed to be known. In contrast, the symbol can be used to state refinement relationships among a wide spectrum of objects ranging from assignment statements and conditional statements all the way down to integers and identifiers. In practice, this means that the symbol can be used to state a refinement relationship like:

\[ \text{meaning}(3) \sqsubseteq \text{meaning}(3) \]
while the weakest precondition paradigm (in its traditional form) has difficulty with such low level objects.

A transformation $T$ is correctness preserving if

$$\forall x : \text{meaning}(x) \subseteq \text{meaning}(T(x)).$$

The transformations $T$ that we will consider in this thesis have the form: $t_1 \Rightarrow t_2$. Such a transformation defines a schema and its replacement. In the above case $t_1$ is the schema which is to be replaced by $t_2$. As we have already mentioned in Section 1.2, we will often refer to $t_1$ as the input schema and $t_2$ as the output schema. A single application of the transformation $T$ to a program $x$ amounts to searching $x$ for an (arbitrary) occurrence of an instance of the input schema of $T$ (i.e., in this case $t_1$) and replacing this instance of the input schema with a corresponding instance of the output schema (i.e., $t_2$).

Let $s_1$ be a specific instance (code segment) of the schema $t_1$ occurring in the program $x$. Traditionally, the expression $x[s_1]$ is used to denote that the program $x$ contains the particular code segment $s_1$. If a program $x[s_2]$ is obtained (derived) from a program $x[s_1]$ by simply substituting $s_2$ for $s_1$, this is traditionally written as:

$$x[s_1] \Rightarrow x[s_2].$$

Using this notation, another way to state that a transformation $T$ is correctness preserving is:

$$\forall x, s_1, s_2 : ((s_1 \text{ is an instance of } t_1) \land (s_2 \text{ is a corresponding instance of } t_2) \land (x \text{ contains } s_1))$$

$$\Rightarrow \text{meaning}(x[s_1]) \subseteq \text{meaning}(x[s_2]).$$

From our present definition of what it means for a transformation to be correctness preserving, it appears to be the case that in order to show a transformation is correctness preserving one would need to prove that the transformation is a refinement with respect to all programs and for all instances to which it can be applied. Generally the set of programs and instances to which a transformation can be applied will be infinite. This makes a brute force approach to proving the correctness of a transformation impossible. Fortunately, it turns out that if the semantics of the language constructs is monotonic with respect to the $\subseteq$ relation, then it is sufficient to show that

$$\text{meaning}(t_1) \subseteq \text{meaning}(t_2)$$

in order to conclude that the transformation $T$ is correctness preserving. The definition of monotonicity is as follows:

$$(\text{meaning}(t_1) \subseteq \text{meaning}(t_2)) \Rightarrow (\forall x : \text{meaning}(x[t_1]) \subseteq \text{meaning}(x[t_2])).$$

From this definition we see that in order to prove

$$\forall x : \text{meaning}(x[t_1]) \subseteq \text{meaning}(x[t_2])$$

it is sufficient to prove

$$\text{meaning}(t_1) \subseteq \text{meaning}(t_2)$$

From the definitions of refinement, we see that the refinement relation is reflexive and transitive, i.e., it is a preorder. From a practical point of view, it is essential that the $\subseteq$ relation have these properties if we intend to derive programs from formal specifications through a sequence of refinements. The importance of transitivity and reflexivity can be seen if one considers the sequence of intermediate programs that are produced in the transformation process

$$\text{meaning}(x_0) \subseteq \text{meaning}(x_1) \subseteq \ldots \subseteq \text{meaning}(x_n).$$

In this example, $x_0$ corresponds to the initial specification and $x_n$ corresponds to the program. A single step from $x_i$ to $x_{i+1}$ is accomplished by application of a transformation $T_{i+1}$ to the input program $x_i$. As we have already mentioned, proving that $T_{i+1}$ preserves correctness amounts to showing that the output schema of $T_{i+1}$ is a refinement of its input schema. In addition, because the $\subseteq$ relation is transitive it follows that $\text{meaning}(x_0) \subseteq \text{meaning}(x_2)$ if $\text{meaning}(x_0) \subseteq \text{meaning}(x_1)$ and $\text{meaning}(x_1) \subseteq \text{meaning}(x_2)$. As can be seen from the the last few paragraphs, it can become quite cumbersome to continuously encapsulate syntactic objects like $x_0$ within the “meaning” function.
We therefore drop the “meaning” function and assume that the reader will understand that when we write \( x_0 \subseteq x_1 \) we really mean \( \text{meaning}(x_0) \sqsubseteq \text{meaning}(x_1) \). However, we will occasionally include the “meaning” function when we wish to draw attention to the fact that the objects, like \( x_0 \), that we are dealing with are syntactic and not semantic in nature.

3.2 Determining Monotonicity

In section 3.1 we pointed out the advantages of having the semantics of the program language constructs be monotonic with respect to the \( \subseteq \) relation. In this section we will informally describe how one can go about determining whether a construct (e.g., a mathematical function) is monotonic with respect to the \( \subseteq \) relation. The purpose of this is to give the reader a feeling for how monotonicity is defined as well as how one can go about determining whether an arbitrary construct is monotonic. Because this is our objective, the notation we present is a simplified version of the more general notation that is used when discussing monotonicity in more general theoretical settings. The reader who is interested in these details should consult [46] and [63] for an excellent and more detailed treatment of the subject.

3.2.1 Monotonicity for Elements

Without loss of generality we begin by considering the domain of integers extended with the undefined element \( \perp \) and functions on these elements. Let \( D^\perp \) denote the domain of integers extended with the element \( \perp \). This domain has two levels of refinement. The first level consists of the element \( \perp \) and the second level contains all of the integers. Thus we essentially have a flat domain as far as the \( \subseteq \) relation is concerned. The reason for the flatness of this domain results from the fact that the \( \subseteq \) relation is a measure of information content. For example, the element \( \perp \) has less information than say the integer 5. We say that 5 has more information than \( \perp \) because determining whether a computation is equal to \( \perp \) is undecidable (i.e., there exists no algorithm that can in general determine whether the result of a computation is undefined). On the other hand, 5 does not contain more information or even the same information as, say, 3 or any other integer for that matter (except 5, of course). Therefore neither 5 \( \subseteq \) 3 nor 3 \( \subseteq \) 5 hold. However, 5 \( \subseteq \) 5, and 3 \( \subseteq \) 3 do hold. Hence the flatness of the domain \( D^\perp \).

The definitions of monotonicity for single elements can be used as the basis for defining the \( \subseteq \) relation on tuples. A tuple of the form \( \{x, y\} \) is a refinement of a tuple \( \{v, w\} \), iff \( (v \subseteq x) \land (w \subseteq y) \). So for example, \( \{5, \perp\} \sqsubseteq \{5, 3\} \) and \( \{5, \perp\} \sqsubseteq \{5, 5\} \). Tuples ordered by the \( \subseteq \) relation form a lattice. The reason for considering tuples is because many mathematical functions are viewed as objects that accept a tuple as input and produce a value as their output. In closing it should be mentioned that tuples may have more than two elements, and that tuples of different sizes are not comparable.

3.2.2 Monotonicity for Functions

Having defined monotonicity for the elements and tuples in \( D^\perp \) we are now ready to consider the definition of monotonicity for functions on the elements of \( D^\perp \). We define a function as a mapping from (a tuple of) \( D^\perp \) to \( D^\perp \). Now a function \( f \) is said to be monotonic with respect to the relation \( \subseteq \) if for all tuples \( x, y \in D^\perp \):

\[
x \subseteq y \Rightarrow f(x) \subseteq f(y).
\]

Generally, determining whether a simple function, like addition, is monotonic can be done by exhaustive examination. For example, suppose \( \text{add} \) is a function that takes two elements from \( D^\perp \) as input and adds them together. To show that \( \text{add} \) is monotonic we need to consider the monotonic relationship between the four possible types of tuples which could be inputs to the function \( \text{add} \). For this example, let \( x \) and \( y \) denote integers. That is, neither \( x \) nor 5 is the undefined element. We will show, by enumeration, that the function \( \text{add} \) is monotonic for the various types of tuples which \( \text{add} \) could receive as input. This will allow us to conclude that \( \text{add} \) satisfies the definition of monotonicity stated above (i.e., \( \text{add} \) is monotonic for all inputs).

- **Case 1.** \( (\perp, \perp) \subseteq (\perp, \perp) \Rightarrow add(\perp, \perp) \subseteq add(\perp, \perp) \). This holds because \( add(\perp, \perp) = \perp \), and \( \perp \subseteq \perp \).
- **Case 2.** \( (\perp, \perp) \subseteq (x, \perp) \Rightarrow add(\perp, \perp) \subseteq add(x, \perp) \). This holds because \( add(\perp, \perp) = \perp \), \( add(x, \perp) = \perp \), and \( \perp \subseteq \perp \).
- **Case 2.** \( (\perp, \perp) \subseteq (\perp, y) \Rightarrow add(\perp, \perp) \subseteq add(\perp, y) \). This holds for essentially the same reason given for case 2.
Section 3.2 Determining Monotonicity

Case 3. \((x, \bot) \subseteq (x, y) \Rightarrow \text{add}(x, \bot) \subseteq \text{add}(x, y)\). This holds because \(\text{add}(x, \bot) = \bot\), \(\text{add}(x, y) = y\) is an integer, say \(z\), and \(\bot \subseteq z\).

Case 4. \((\bot, y) \subseteq (x, y) \Rightarrow \text{add}(\bot, y) \subseteq \text{add}(x, y)\). This holds for essentially the same reason given for case 3.

Case 5. \((\bot, \bot) \subseteq (x, y) \Rightarrow \text{add}(\bot, \bot) \subseteq \text{add}(x, y)\). This holds for essentially the same reason given for case 3.

Case 6. \((x, y) \subseteq (x', y')\). Since \(x, x', y, y'\) are all integers (i.e., not the undefined element) \((x, y) \subseteq (x', y')\) only holds when \((x = x') \land (y = y')\). That is when \(x\) and \(x'\) denote the same element, and \(y\) and \(y'\) denote the same element. Because add is a function (i.e., single output) it is monotonic in this case also.

One can use the above approach to determine whether a wide variety of simple functions are monotonic. In addition, there exists a theorem stating that composition of monotonic functions yields a function that is itself monotonic. This theorem and its proof can be found in [46]. With these pieces of information one has the tools to determine whether a language like Pascal minus \{goto’s, recursion, and indefinite iteration\} has constructs that are monotonic. In order to be able to consider languages containing goto’s, recursion and indefinite iteration we avail ourselves of fixed point theory.

3.2.3 Functionals and Fixed-Point Theory

The purpose of fixed-point theory is to provide a unique solution to recursive equations. Any programming language construct that allows indefinite iteration can be defined by a recursive equation. For this reason, fixed point theory can be used to define the semantics of important programming language constructs like goto’s, while-loops, and recursive functions and procedures. A problem with recursive equations is that they generally have more than one solution. Because recursive equations generally have more than one solution, it is important for us to precisely specify what particular solution to a recursive equation we are interested in. If recursive equations had unique solutions, then we could use recursive equations as a method for defining the semantics of programming constructs such as goto’s and while-loops. Consider the following recursive equation:

\[
f(x) = \begin{cases} 
0 & \text{if } x = 0 \\
 f(x + 1) & \text{otherwise}
\end{cases}
\]

This recursive equation defines a mapping between its input and its output. If the input value is 0, then the output value is also 0. This gives us \((0, 0)\) as an input-output pair that is defined by this recursive equation. From the equation we also see that all negative integers will have 0 as their outputs. So \((-1, 0)\) is an input-output pair defined by this recursive equation. So is \((-2, 0), (-3, 0), \ldots\). What happens when the input value is a positive integer? The only requirement the above equation places on the output corresponding to a positive input is that the output be the same as the output corresponding to the input plus 1. In other words, all positive inputs must map to the same output value. What is this value? Well, it turns out that any value will do so long as we are consistent. For example, let \(y\) denote an arbitrary value in the output domain. Then the mapping \((1, y), (2, y), (3, y), \ldots\) is a solution to the recursive equation defined above. What this means is that the above recursive equation has many (an infinite number) of solutions. A solution to a recursive equation is called a fixed point of the equation. The least fixed point of a recursive equation is the fixed point that is least from a monotonic standpoint (i.e., minimal with respect to the relation). If we let \(y = \bot\) then we have the least fixed point of the recursive equation given above. The reason why the least fixed point of a recursive equation is of interest is because it has the property that it is unique (when it exists). Because of its uniqueness, the least fixed point is the solution that is implied when one talks about the solution to a recursive equation.

In theory, functions can be constructed that do not have any fixed points. Also functions can be constructed that have infinitely many fixed points but no least fixed point. Fortunately, it turns out that recursive equations that are monotonic and continuous will always have a unique least fixed point. This result is often referred to as the first recursion theorem of Kleene. Having said this, we now demonstrate how one can go about determining whether the recursive equations (e.g., recursive functions, while-loops, and goto’s) constructible in a programming language will be monotonic and continuous.

Up to this point, we have discussed the \(\subseteq\) relation as it pertains to elements in the set \(D^\bot\). We have also given a definition for what it means for a function to be monotonic with respect to the \(\subseteq\) relation. We now discuss the relation \(\subseteq\) as it pertains to monotonic functions. Let \([D^\bot \rightarrow D^\bot]\) denote the set of all monotonic functions of the form: \(f : D^\bot \rightarrow D^\bot\).
Chapter 3  Proving the Correctness of Program Transformations

**Definition**  For any \( i > j \), we say \( f \sqsubseteq g \) if for all \( x \in D^\perp \):

\[
f(x) \sqsubseteq g(x).
\]

**Definition**  Let \( \{ f_i \} \) denote a sequence of functions \( f_0, f_1, f_2, \ldots \) in \([D^\perp \rightarrow D^\perp]\). If \( f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots \) then \( \{ f_i \} \) is called a chain.

**Definition**  Given a sequence of functions \( \{ f_i \} \), and a function \( f \in [D^\perp \rightarrow D^\perp] \), \( f \) is called an upper bound of the sequence \( \{ f_i \} \) if:

\[
\forall i : i \geq 0 : f_i \sqsubseteq f. 
\]

If in addition, \( f \) is such that \( f \sqsubseteq g \) for every upper bound \( g \) of \( \{ f_i \} \), then \( f \) is called the least upper bound of \( \{ f_i \} \). The least upper bound of \( \{ f_i \} \) is denoted \( \text{lub} \{ f_i \} \).

A functional \( \tau \) is a mapping from \([D^\perp \rightarrow D^\perp]\) into itself. In other words, \( \tau \) takes a monotonic function as input and produces a monotonic function as its output. The concept of a functional is one order higher than a function \( f \), which is a mathematical object that takes an element of \( D^\perp \) as its input and produces an element of \( D^\perp \) as its output. Essentially, what a functional is, is a recursive equation. As we mentioned earlier, two desirable properties of functionals (i.e., recursive equations) are monotonicity and continuity. These are defined as follows:

- A functional \( \tau \) over \([D^\perp \rightarrow D^\perp]\) is said to be monotonic if for all \( f, g \in [D^\perp \rightarrow D^\perp] \):

\[
f \sqsubseteq g \Rightarrow \tau(f) \sqsubseteq \tau(g)
\]

A monotonic functional \( \tau \) over \([D^\perp \rightarrow D^\perp]\) is said to be continuous if for any chain of functions \( \{ f_i \} \), we have:

\[
\tau(\text{lub}\{ f_i \}) = \text{lub}\{ \tau(f_i) \}.
\]

Usually, a functional is defined in terms of a composition of known monotonic functions and a function variable \( F \). Clearly, if the function variable \( F \) (i.e., the input to the functional) is replaced with a monotonic function, then the resulting expression will be monotonic. This follows from the fact that composition of monotonic functions yields a monotonic function. Thus if the semantics of the constructs in our programming language are monotonic, then our functionals (i.e., recursive expressions) will be monotonic. In addition, a theorem exists stating that any functional that is defined by composition of monotonic functions and the function variable \( F \) is continuous. See [46] for proofs of the theorems we mentioned in this section.

In summary, we have demonstrated how one can determine whether the base functions of an arbitrary programming language are monotonic. We have also discussed how one can use the fact that the base functions are monotonic to conclude that the recursive constructs of the language (e.g., recursive functions, while-loops, goto’s, etc.) are both monotonic and continuous. This information allows us to define the semantics of recursive constructs in terms of their least fixed-point semantics. In addition, the knowledge that all the constructs of the language (i.e., both recursive constructs and base functions) are monotonic enables transformations to replace a specific construct in an arbitrary program with a construct that is more refined, thereby obtaining a transformed program that is more refined than the original.
Chapter 4
Related Work

4.1 Transformation Systems

A transformation system can be viewed as a system which transforms elements (e.g., programs, specifications, etc.) of a source or subject language to elements of a target or object language. The subject language and the target language can be very different from one another (e.g., a specification language and a procedural programming language), or they can be very similar (e.g., Pascal and FORTRAN), or they can even be the same language. A Pascal compiler is an example of a transformation system whose source language is Pascal, and whose target language is assembly language. In [34], a formalism is given for proving the correctness of a very simple Pascal-like compiler.

Oftentimes the choice of the subject and target language directly reflect the goal of a transformation system. For example, if the subject language and the target language are the same language, then one can usually assume that the goal of the transformation system is modification or optimization. Program modification involves making changes to data structures, other related program changes or even modifying the semantics of the program. An example of a transformation system which modifies the semantics of programs can be found in [20]. Program optimization, on the other hand, is concerned with improving the efficiency of a program while preserving its meaning (i.e., producing a more efficient program having a semantics equivalent to the original program). In [26] a methodology is investigated for limiting the amount of verification that is necessary in order to verify that the application of a transformation preserves the correctness of the program to which the transformation is applied.

If the subject language is a (descriptive) specification language and the target language is a procedural or applicative programming language, then one can generally assume that the goal of the transformation system is program synthesis. In synthesis the goal is to obtain, through transformations, an executable program from a non-executable specification. Some people view the process of obtaining an executable program, \( P \), from a formal specification, \( S \), as a higher form of programming. This view is not unreasonable when one considers that in the traditional form of programming, a program is written (by hand) according to some specification. In such a paradigm, the programmer is expected to digest the specification and then to produce code that satisfies the specification. In transformational programming, the programmer is expected to transform a specification into a program. The end result of both methodologies is the same (i.e., an executable program is obtained). The only difference between the two is the means which are used to achieve that goal.

If the specification language contains non-computable objects (e.g., non-deterministically defined objects or operations), then the transformation system, with the help of the user, must remove these constructs replacing them with appropriate computable (i.e., deterministic) constructs. In such cases, the final program can never exactly match the specification. One is nondeterministic and the other is not. One generally requires that transformations of this type produce a program which satisfies the specification, but that is not equivalent to it. Transformations in this class therefore are generally not equivalence preserving. However, these transformations must satisfy some notion of correctness preservation. The symbol which is generally associated with this notion is the \( \sqsubseteq \) symbol which was defined in Chapter 3. This symbol is used to indicate that a transformation produces a (transformed) program which is no-worse-than or is an acceptable-substitute-for the original program. Another way to read the \( \sqsubseteq \) symbol is that the program appearing on the left side of \( \sqsubseteq \) symbol is less-defined-than the program appearing on the right side of the \( \sqsubseteq \) symbol. Examples of this kind of transformation system can be found in [34], [58], [46], and [56].

An extensive survey of transformation systems can be found in [65]. In that survey, transformation systems are
global cleanup operations. One way to view global rules is that they apply to aspects of a programming language. These rules include flow analysis, consistency checks, and efficiency information, or codified knowledge for automatic data structure selection. Catalog systems are often referred to as knowledge-based systems. The main problem with catalog systems lies in providing rapid access to individual transformations, and the completeness of the set of transformations.

Another approach to storing transformations is called the generative set approach. In this approach, a small but powerful set of transformations is used to generate new transformations. Generally, additional transformations are needed to fill in the gaps which are not covered by the generative set. A problem with this system is how to compose the base set of transformations to obtain more powerful ones.

Lastly, we mention that there are two types of transformations, procedural and schematic. Procedural transformations are algorithms which operate on entire programs and produce other programs. Procedural transformations are often referred to as global rules or semantic rules. These rules include flow analysis, consistency checks, and global cleanup operations. One way to view global rules is that they apply to aspects of a programming language.
which are not context-free. A difficulty associated with these kinds of rules is that procedural transformations are oftentimes very difficult to reason about (i.e., prove correct).

Schematic transformations, on the other hand, are syntax-oriented transformations which are used to make local changes to programs. These rules generally are more amenable to human perception, but have the disadvantage that they are not ideally suited for expressing different kinds of global information. It should be noted that any kind of global transformation can be carried out by a schematic transformation system (though the required transformations can be quite complex). Conceptually, one of the major advantages offered by the schematic transformation approach, is the ability to write many trivial transformations which are easy to understand, but which have the property that, when composed, they bring about a significant change in the program to which they are applied.

TAMPR, the transformation system on which this research is based, is a schematic transformation system. Because of this, and because correctness proofs for procedural transformations are, for all practical purposes, beyond the abilities of current verification methodologies [65], we limit our attention, in the following survey, to work that has been done in proving the correctness of schematic transformations.

4.2 Correctness Preserving Schematic Transformation Systems

Several people have investigated the idea of transforming programs through correctness preserving schematic transformations. In this chapter we present a survey of some of the work that has been done in this field. The purpose of this survey is to 1) give the reader an idea of the various approaches that can be taken to solve the problem of proving the correctness of schematic program transformations, as well as 2) comparing and contrasting the benefits and drawbacks of the various approaches. Each approach addresses three areas of interest.

1. Between what two levels of abstraction does the transformation take place?
2. How is correctness shown?
3. What are the capabilities of the transformation system?

4.3 Morris

In [56], a methodology is presented for transforming a formal specification \(s\) into a program \(p\). The specification \(s\) is an element of the language \(L_s\) and the program \(p\) is an element of the language \(L_p\). The view taken by Morris is that the language \(L_s\) is a language that is better suited for stating abstract and elegant solutions to problems than the language \(L_p\). The difference between \(L_s\) and \(L_p\) is that \(L_s\) may contain high-level data types, control constructs, and even non-deterministic objects (i.e., non-computable objects) while \(L_p\) must consist only of objects that are executable by a computer.

A question that must be addressed by any system that transforms programs from one language into another language is “How are the semantics of both languages defined?” In [56], Morris merges the specification language and the programming language into a single wide-spectrum language. This wide-spectrum language contains programming constructs as well as specification constructs. The goal of the transformation system then is to take a specification \(s\) consisting of high-level, possibly non-deterministic constructs and to transform this specification into an acceptable program program \(p\) such that \(p\) consists only of constructs that are executable by a computer.

The transformations accomplish this by essentially replacing each high-level construct appearing in \(s\) with a programming construct. Once all the high-level constructs in \(s\) have been replaced we are left with the program \(p\). Proving that the transformations, used to derive \(p\) from \(s\), are correctness preserving is achieved by first defining the semantics of the wide spectrum language in terms of weakest precondition semantics, and then showing that the programming constructs are refinements of the high-level constructs that they replace.

4.4 The Munich Project: CIP

The project CIP (Computer-aided Intuition-guided Programming) was proposed in the first half of the seventies by F. L. Bauer and K. Samelson [3] [4] [5]. CIP is a massive attempt to formalize as much of the software development process as possible. The language in which they are performing transformations is a wide-spectrum language called CIP-L. CIP-L contains high-level nondeterministic constructs, ideally suited for the construction of formal specifications, as well as low level constructs, ideally suited for execution by a computer. The goal of the Munich project
Chapter 4 Related Work

is to provide a method that would allow one to obtain a machine implementation from a high level specification in a formal manner. With this in mind, Bauer et al redefined the traditional software engineering program development process in terms of formal transformations. The idea is that if each step in the process is carried out formally (and correctly), then it follows that the implementation obtained as the end product of this process is correct with respect to the original specification. Given this philosophy, it is reasonable to expect human assistance throughout most of the software development process. This means that the determination of which transformation should be applied to what section of the program is made by the user. Thus the transformation system is manual in nature.

In the semantics of CIP-L, algebraic theories (i.e., algebraic specifications) as specified by abstract algebras play a key role in their approach. This is evident because the base types of CIP-L are constructed as hierarchies of abstract algebras. An extraordinary amount of effort is devoted to the construction of these hierarchies as well as demonstrating that they are correctly defined.

Transformations are viewed as term rewriting rules consisting of a pair of program schemes called input and output templates [19]. In addition, transformations are generally accompanied by an applicability condition defining under what conditions the transformation may be applied.

In [60], Broy states that in program construction by transformation, the development process should go through the following phases:

- Requirements engineering: In this stage, an informally posed problem is analyzed and formalized until a formal specification is obtained.
- Transformation of the specification: The formal specification that is obtained from the previous step is generalized and/or completed. This may involve breaking down the specification into subproblems. Also in this phase, the transition to algorithmic versions is prepared.
- From specification to algorithms: Conditional equations are derived for the specified functions that can be used as recursive equations. In general, termination properties of the resulting recursive programs have to be shown separately.
- Transformation of recursive, functional programs: In this phase, recursive and functional programs are transformed in order to make them more efficient, and to put them in a form so that they can more easily be transformed into procedural programs. The transformations at this stage are usually based on induction principles.
- Transition to procedural programs: In this phase, functional programs are transformed into the classical sequential stored program machine type. At this stage, program variables, iteration, and assignment is introduced.
- Transformation of procedural programs: In this phase, procedural programs are optimized and possibly put into a particular (machine-oriented) form.

Of all of the above steps, only the requirements engineering step cannot be performed in a formal framework. This step, though acknowledged as one of the most important parts of program design, does not fall under the formal framework of the CIP project.

Transformations are divided into three classes depending on the problems that arise when trying to determine their applicability condition. The three classes are as follows:

1. Syntactic transformations. Transformations belonging to this class have applicability conditions that can be decided by inspecting the syntax of the portion of the program to which they will be applied. An output template in this class generally consists of a rearrangement or slight modification of the input template.
2. Semantic transformations. Transformations belonging to this class have applicability conditions that contain semantic conditions that are generally not decidable. An example of a semantic condition that is not decidable in general is if a value has been assigned to particular program variable. This “assignment problem” is simply an instance of the well known (undecidable) problem of whether an if-branch is taken during the execution of a program. An output template in this class generally consists of a rearrangement or slight modification of the input template.
3. Design decisions. Transformations belonging to this class can have output templates that can be totally different from their respective input templates. In addition, applicability conditions can be either syntactic or semantic in nature.

One of the difficulties that is encountered in the process of transforming a specification into a machine executable program is the replacement of nondeterministic (i.e., non-computable) constructs with deterministic constructs. This
is a recurring problem when one considers the transformation of (nondeterministic) specifications to (deterministic) programs. Another problem encountered is how one can transform quantifiers (\(v\) and \(\exists\)) into constructs amenable to machine execution. Once this has been accomplished, one is faced with more common problems like recursion removal, and various optimizations.

### 4.5 Hoare and Jifeng

In [34], Hoare and Jifeng present a methodology that can be used to prove the correctness of a compiler. This work focuses on the transformation of a program belonging to a Pascal-like language into an assembly language program. In [34], a compiler is viewed as a transformation system that transforms elements (i.e., programs) from one model of computation (human oriented) into another model of computation (machine oriented). A key concept here is how Hoare and Jifeng relate the two models to one another. Once this has been done, a structural induction proof method can be used to prove the compiler correct.

The idea is to define the semantics of the source language, define the semantics of the object language and then to show that when a source program \(p\) is compiled, producing an object program \(c\), where the semantics of \(c\) is no worse than (i.e., \(\leq\)) the semantics of \(p\). For the purposes of this paper, the program \(p\) can be thought of as a sequence of instructions that can be comprehended and carried out by a human. The execution of a program \(p\) then amounts to a human sequentially carrying out each of the instructions in \(p\) in the order in which they occur. An assembly language program, on the other hand, is a code segment consisting of a sequence of instructions which are ideally suited to execution by a machine. These two methods of computation (human vs. machine) are at different levels of abstraction. For example, when discussing the execution of a high-level program \(p\), one talks about variables and their values. When discussing the execution of an assembly language code segment, one talks about storage locations and their contents. Clearly any proof attempting to show that \(c\) is correct with respect to \(p\) must be able show some sort of correspondence between high-level program instructions and low-level assembly language instructions in addition to showing a correspondence between specific variables and storage locations as well as between specific values and storage location contexts. Hoare and Jifeng accomplish this by first assuming that the formal semantics of the high-level language is already known, then by defining the semantics of the assembly language in terms of an interpreter that is itself written in the high-level language, and lastly by using the symbol table created by the compilation process in order to determine the correspondences between variables and their values and storage locations and their contents.

### 4.6 Dershowitz and Manna

In [20], program transformations are viewed as a vehicle that allow one to transform an existing program \(p_1\) satisfying a specification \(s_1\) into a program \(p_2\) satisfying a specification \(s_2\). The idea is that since it has already been formally proven that \(p_1\) satisfies \(s_1\), a slight modification of \(p_1\) via a transformation will produce a program \(p_2\) for which it will be relatively easy to show that it satisfies \(s_2\). The hope is that much of the proof that \(p_1\) satisfies \(s_1\) can be reused in the proof showing that \(p_2\) satisfies \(s_2\). This being the case, new programs and their correctness proofs can be obtained from existing programs, which have been formally proven correct, with very little effort.

The model of the transformation system in [20] is as follows:

1. We are given an annotated program \(p_1\) and a specification \(s_1\). The annotations in \(p_1\) take the form of loop assertions, output assertions, and anything else that is required for formal verification of the program.
2. We assume that \(p_1\) satisfies \(s_1\). Furthermore, we assume that the annotations present in \(p_1\) are sufficient in order to prove the correctness of \(p_1\).
3. We are given a set of transformations in ground form (i.e., containing no schema variables) or in the form of schemas.
4. Transformations are applied to the annotated program \(p_1\) globally. That is, transformations are applied to all occurrences of the patterns which they match, whether these patterns occur inside the assertions or inside the code. The transformed assertions are now used to obtain verification conditions for the transformed program.
5. The transformed program \(p_2\) satisfies the specification \(s_2\) if there exists an output assertion that implies the output assertion stated in the specification \(s_2\).

Dershowitz and Manna view transformations as analogies. This stems from the fact that they want to transform a
program $p_1$ into a similar, but different, program $p_2$. The idea is that if it is possible to efficiently transform a program $p_1$ into a program $p_2$, then $p_1$ and $p_2$ must somehow be related. Hence the notion of analogy. A fundamental question concerning analogies is what are their limitations? That is, just how different can the transformed program be from the original?

Consider the following example:

$$(y, z) \leftarrow (0, A[0])$$

loop until $y = n$

$\quad y \leftarrow y + 1$

$\quad z \leftarrow \min(z, A[y])$

repeat

assert $z = \min(A[0 : n])$

The above code segment finds the minimum value of an array indexed from 0 to $n$. If we wanted to transform this program into an analogous code segment which finds the maximum of an array indexed from 1 to $n$, then the transformations $\min \rightarrow \max$ and $3 \rightarrow 4$ come to mind. However, the problem here is in determining which occurrences of 3 should be replaced by 4. Clearly, if the above code segment is part of a larger program one would need to be careful about which occurrences of 0 should be replaced by 1.

### 4.7 Gerhart

#### 4.7.1 Semantics of Languages

In [26], the semantics of programming languages are defined in terms of attributes that are associated with the nodes of syntax derivation trees of programs in the language. This semantic method was introduced by Knuth [38] and is somewhat similar to the semantic model of Hehner [31] which views programming constructs as predicate transformers.

The basic idea in Gerhart’s work is to tie in the notion of pre-, post- and verification condition to the actual production rules that generate the syntax of a language. When doing this, there are two things that are of interest. The first is how a given production breaks up or otherwise modifies its pre- or postcondition. The second is what verification condition is associated with a given production? An example of the first type can be seen by a production that generates a statement and a compound statement from a compound statement. Consider the grammar production:

$$CS_1 \rightarrow S;CS_2$$

Note that the subscripts on the non-terminal symbol CS are only added for the benefit of this discussion. Let $pre_1$ and $post_1$ be the pre- and postconditions associated with $CS_1$. If the above is performed then $pre_1$ will be the precondition associated with the nonterminal symbol $S$, $post_1$ will be the postcondition associated with the nonterminal symbol $CS_2$, and the postcondition of $S$ will be equivalent to the precondition of $CS_2$. The verification condition for $CS_1$ is equal to (the verification condition of $S \land$ the verification condition of $CS_2$).

An example of a verification condition associated with a given production is as follows. Consider the production:

$$S \rightarrow \text{while } B \{ \text{ assert } A \}(CS)$$

In this example, the precondition of $S$ is the assertion $A$ (i.e., the loop invariant). The verification condition of $S$ is:

1. The verification condition of $CS$, and
2. $A \land \neg B \rightarrow \text{postcondition}(S)$, and
3. $A \land B \rightarrow \text{precondition}(CS)$

The conditions that need to be verified are the standard conditions that need to be verified in order to prove the partial correctness of a loop construct.

Summarizing then, in this approach preconditions, postconditions, and verification conditions are associated with each node of the parse tree corresponding to a program, and are referred to as the attributes of the node. The rules
Section 4.8  Sere and von Wright

defining how grammar productions and pre-, post-, and verification conditions correspond define the semantics of the programming language. The semantics of the program can be used to prove the partial correctness of the program. That is, the semantics of the program can be used to determine whether a program establishes a desired postcondition (or precondition as the case may be).

4.7.2 Transformations

Given a language whose semantics are defined as described in the previous section, we now discuss the role played by transformations applied to programs written in such a language. Gerhart is interested in correctness preserving transformations. In [26], transformations of this type are used to transform an inefficient easy-to-understand program into an efficient harder-to-understand program. Gerhart points out that correctness preserving transformations are different from compiler optimizations because optimizations are equivalence preserving while transformations are correctness preserving. The difference between compiler optimizations and program transformations can be seen in the following example:

\[
\begin{align*}
\{ \text{Assert: } 1 \leq N \} \\
i := 1; \\
\{ \text{Assert: } \forall j : 1 \leq j < i : V[j] = 0 \}
\end{align*}
\]

This segment of code can be transformed to:

\[
\begin{align*}
\{ \text{Assert: } 1 \leq N \} \\
i := 1; \\
V[i] := 0; \\
\{ \text{Assert: } \forall j : 1 \leq j < i : V[j] = 0 \}
\end{align*}
\]

The idea here is that transformations can make use of assertions which are generally not available to optimizing compilers. Transformations can then alter a program in a way which is consistent with the assertions of the program. These alterations can then be used to alter the assertions themselves, and in this manner a program can be transformed.

The transformation process goes as follows. We begin with a program \( p \) which we have proven correct. Applying a transformation alters \( p \) to produce \( p' \). The programs \( p \) and \( p' \) have parse trees \( t \) and \( t' \) respectively. Furthermore, \( t \) and \( t' \) are the same except for the subtree where the transformation took place. Let \( r \) denote the subtree in \( t \) which is effected by the transformation. Similarly, let \( r' \) denote the subtree in \( t' \) corresponding to \( r \). Let \( t - r \) denote the tree which is obtained from \( t \) by removing the subtree \( r \). From these definitions, we can conclude that \( t - r \) is the same as \( t' - r' \).

The key to successfully applying correctness preserving transformations to programs (parse trees) is to be able to determine the area of the tree which is effected by the transformation (i.e., \( r \)). When a transformation is applied, the effected area (e.g., the context) of the tree needs to be re-proven. This is necessary because, in general, the verification conditions for nodes in the effected area will have changed. In [26], several lemmas and theorems are given to justify that this kind of local re-proving is sufficient to allow one to conclude that the entire program is still correct. The process for proving a program correct after a transformation has been applied is as follows:

1. Find the effected subtree. That is, find \( r \). The subtree \( r \) is called the syntactic range of the replacement (i.e., the transformation).
2. Compute the attributes of the subtree \( r' \) using the semantic context of \( r \).
3. Prove that the meaning of \( r' \) is correct. This is essentially accomplished by showing that \( r' \) satisfies the pre- and postconditions of \( r \) with respect to the program \( t \).
4. Copy the attributes to \( t' \) which carry over from \( t \) and compute the attributes of \( t' \) which do not carry over from \( t \).

4.8 Sere and von Wright

In [64], a refinement calculus is presented in which transformations are formalized and proved with the assistance of the HOL (Higher Order Logic [29]) theorem prover. The main contribution in [64] is the formalization and proofs of a collection of program transformation rules using HOL. The view presented here is that programs can be obtained from formal specifications through stepwise refinement. The refinement calculus used by Sere and von Wright is
Chapter 4 Related Work

based on the weakest precondition calculus of Dijkstra [22]. This requires that the semantics of their language be formally defined in terms of weakest precondition semantics. The definition of refinement presented in [64] is what we used when we first defined correctness in Chapter 3. In the interests of presenting a coherent summary of the work in [64] we repeat some of the definitions of refinement that we presented in Chapter 3.

In [64], a statement $s$ is said to be a refinement of a statement $s'$ if for every postcondition $R$,

$$wp(s, R) \Rightarrow wp(s', R).$$

If such is the case then we write $s \subseteq s'$. If $s'$ is a refinement of $s$, then $s$ is said to preserve the correctness of $s$.

Another way to say this is as follows:

$$(s \subseteq s') \Leftrightarrow \forall P, Q : \{P\}s\{Q\} \Rightarrow \{P\}s'\{Q\}.$$

As is the case in our work, Sere and von Wright require that statement constructors be monotonic with respect to the refinement relation. That is,

$$T \subseteq T' \Rightarrow S[T] \subseteq S[T'].$$

In the above formula, the term $S[T]$ denotes the statement $S$ in which $T$ occurs as a substatement. Often one encounters a situation in which a statement $S$ is such that $S[T] \subseteq S[T']$ holds but $T \subseteq T'$ does not hold. Such transformations (i.e., the replacement of $T$ by $T'$ in the statement $S$) can be proven correct with the assistance of program assertions with the technique presented in [2]. To assert $Q$ at some point in a program all one needs to do is insert the statement $\{Q\}$ in the desired location. The assertion statement $\{Q\}$ is a Boolean expression involving the program state that exists at the point in the program where the assertion is placed. If during the execution of the program the Boolean expression, $Q$, in the assertion statement evaluates to true, then the assert statement behaves like a skip statement (i.e., an empty statement). On the other hand, if $Q$ evaluates to false, then the assert statements behaves like an abort statement.

In our example, proving the correctness of a transformation that is conditionally correct (i.e., in general $T'$ is not a refinement of $T$, but $T'$ is a refinement of $T$ in execution states satisfying $Q$) would be accomplished as follows:

1. If we can prove that $S[T] \subseteq S[\{Q\}; T]$, where $Q$ is a program assertion, and
2. If we can prove that $\{Q\}; T \subseteq T'$,

then by monotonicity and transitivity we can conclude that $S[T] \subseteq S[T']$. This example illustrates the importance of the assertion statement in the refinement calculus.

In this system then, one begins with a formal specification $s_0$ which is transformed by program refinements until a program $s_n$ is obtained. This gives us a sequence of refinements having the form:

$$s_0 \subseteq s_1 \subseteq \cdots \subseteq s_{n-1} \subseteq s_n$$

which by transitivity implies that the program $s_n$ is a refinement of the initial specification $s_0$. In this paradigm an individual refinement step consists of the application of a transformation rule. The correctness of the transformation rule itself is verified by showing that the application conditions of the rule are satisfied.

In this framework, Sere and von Wright go on to define a set of basic transformation rules. These basic transformation rules are so simple that their proof essentially follows from the weakest precondition semantics of the statements in the language. Once these rules have been formalized Sere and von Wright present a set of slightly more complex transformation rules. The correctness of these transformations is based on the correctness of the basic transformation rules. In this fashion, a hierarchy of ever-increasing complex transformation rules can be constructed whose proof is based on the correctness of the transformation rules which precede it in the hierarchy.

4.9 Relation to Our Work

This section compares and contrasts TAMPR and our research with the work of [56][3], [4][5][34], [20], [26], and [64] that was presented in this survey.

Perhaps the most profound difference between our research and the work presented in our survey is the approach that we have taken to modeling the pre- or enabling conditions of transformations. The traditional approach is to express preconditions of transformations in terms of semantic properties that must be verified with respect to the specific program under consideration, before the transformation may be applied.
In this research we take advantage of the fact that TAMPR transformations are (generally) applied exhaustively in order to capture the effects of transformations at the program level (see Chapter 7). Essentially the preconditions of a TAMPR transformation $T_i$ is stated in terms of a syntactic property $Q_i'$ that is expressed in terms of entire programs. The idea is to show that a precondition $Q_i'$ is established by the transformations preceding $T_i$, in this manner we are reasoning about whether $T_i$ may be applied to a potentially infinite class of programs. In other words, we do not consider program-specific preconditions. The semantic effects of $Q_i'$ is incorporated into our formal reasoning through a process we call generalization (see Chapter 7).

A difference between TAMPR and [34] is the fact that, in general TAMPR transformation sequences are composed of transformations that build on one another. Thus, while each transformation in a TAMPR transformation sequence might have a small effect when considered in isolation, the cumulative effect produced by a TAMPR transformation sequence when taken as a whole can be quite dramatic. In contrast, the transformational process in [34] essentially appears to be a one step process. That is, programming constructs are transformed into assembly code in a single transformational step. In [20], it also appears that this single transformational step approach was taken.

An important difference between our work and the work of [56] and [64] is the size and scope of the language considered. Both Morris and Sere and von Wright present a very small language whose main purpose is to provide an arena in which ideas and results can be presented and discussed. The most notable omission from the languages of [56] and [64] are subprograms. Poly, on the other hand, is a real specification-programming language containing subprograms as well as many other constructs that are missing from the languages presented in [56] and [64]. We would like to mention that in the initial stages of our research we also used a small well-behaved language, like the ones presented in [56] and [64], to try out our ideas. However, one of the things that we learned during the course of this research is that one should not underestimate the difficulties involved with scaling up results from a small ideal language to a large real language. Another difference between our work and the work in [56] is that Morris considers transformations involving nondeterministic (i.e., non-computable) constructs. In contrast, the language Poly that we consider in our work does not contain any nondeterministic constructs (at present).

The work in [26] is quite similar to our own. However, Gerhart uses a different formal semantic model to define the semantics of her programming language. As it turns out, a drawback associated with the combination of this semantic model and her approach is that determining the formal semantics of the effect of a transformation could (in the worst case) involve recomputing the semantics of virtually the entire program in which the transformation takes place. In addition, Gerhart points out that her methodology cannot handle schema variables. It is the interaction of the inability to handle schema variables with the formal semantic model which causes the problem we mentioned in the previous sentence. In contrast, our methodology can, through delta-functions, handle schema variables and therefore we are able to prove the correctness of transformations that may be applied to a large class of programs. Gerhart, on the other hand, is interested in reasoning about the application of program transformations to specific programs. Furthermore, Gerhart requires that the initial program that one desires to transform be proven correct (a potentially difficult task). This proof can then be reused (to a certain extent) when proving the correctness of specific program transformations. Our approach does not require that the initial specification (i.e., program) be proven correct. In our approach we are only interested in proving that a transformed program is correct with respect to its initial specification.
Chapter 5
The TAMPR Transformation System

5.1 Introduction

In this chapter we give a high level overview of the TAMPR transformation system. The goal of this overview is to give the reader a rudimentary understanding of:

1. what a TAMPR transformation is,
2. how one goes about informally proving the correctness of TAMPR transformations,
3. how TAMPR uses transformations to transform programs,
4. how a set of TAMPR transformations can be composed into a transformation sequence as well as the semantics of such transformation sequences, and
5. the various types of transformation constructs that are available in TAMPR.

In the following chapter, we then describe how one formally proves the correctness of individual TAMPR transformations.

5.2 Overview

TAMPR is a transformation system created by Dr. James M. Boyle at Argonne National Laboratory. TAMPR is an acronym standing for Transformation Assisted Multiple Program Realization (system). In TAMPR programs and program transformations are viewed in terms of their syntax derivation trees (SDT’s). That is, the objects that TAMPR manipulates are SDT’s. Essentially, a TAMPR transformation is a rule stating that one kind of SDT pattern, an input pattern, should be rewritten into a different kind of (SDT) pattern, an output pattern. Abstractly, one can simply think of these transformations as having the form: \( w_{\text{in}} \Rightarrow w_{\text{out}} \) where \( w_{\text{in}} \) and \( w_{\text{out}} \) are SDT patterns. When discussing transformations at this level of abstraction, we will denote the input and output patterns of an arbitrary transformation, \( T_i \), by \( t_{i,\text{in}} \) and \( t_{i,\text{out}} \) respectively.

Oftentimes we wish to discuss specific aspects of a transformation pattern. In this case, the notation \( t_{i,\text{in}} \) and \( t_{i,\text{out}} \) provides us with no insight into the internal structure of the patterns. On the other hand, it will usually be the case that writing out the entire SDT for \( t_{i,\text{in}} \) or \( t_{i,\text{out}} \) will be too cumbersome. To deal with this problem, we adopt a representation of SDT’s that is both concise yet expressive enough to allow us to discuss various aspects of transformation patterns that interest us.

Consider a grammar \( G \) and the language \( L(G) \) that is defined by \( G \). In formal language theory, elements of a language (i.e., in this case programs) are referred to as sentences. We extend this terminology by introducing the idea of a phrase. Given an arbitrary nonterminal symbol, \( d \), belonging to the grammar \( G \), \( \alpha \) is a phrase of \( d \) iff \( d \Rightarrow^* \alpha \). Note, a phrase may contain nonterminal symbols. We use the term completed_phrase to denote a phrase that consists solely of terminal symbols. Utilizing the idea of a phrase, a static transformation schema can be defined as a nonterminal grammar symbol, which is referred to as a dominating symbol, and a phrase that is derivable from that dominating symbol. For example, if \( d \) is a nonterminal symbol in \( G \), and if \( d \Rightarrow \alpha \), then \( d\{\alpha\} \) is a transformation pattern denoting an SDT whose dominating symbol is \( d \) and whose leaf elements form the phrase \( \alpha \). If \( d\{\alpha\} \) is an
Section 5.2 Overview

input schema, then we refer to $\alpha$ as the input phrase. Similarly, if $d\{\alpha\}$ is an output schema, then we refer to $\alpha$ as the output phrase. In general, when the distinction between input and output is not important, we will refer to the expression $d\{\alpha\}$ as a transformation schema or simply a schema. This notation can be used is a recursive fashion to give structure to phrases.

For example, suppose $d_1 \Rightarrow \gamma d \beta$ and $d \Rightarrow \alpha$. If this is true, then the derivation $d_1 \Rightarrow \gamma d \beta \Rightarrow \gamma \alpha \beta$ must be possible. Consider the derivation sequence $d_1 \Rightarrow \gamma d \beta \Rightarrow \gamma \alpha \beta$ and the schema $d_1\{\gamma \alpha \beta\}$. Since we know some of the internal structure of this pattern (namely that $\gamma d \beta \Rightarrow \gamma \alpha \beta$) we may express this information by writing $d_1\{\gamma d\{\alpha\}\}$ in place of $d_1\{\gamma \alpha \beta\}$. In this fashion we can include as little or as much of the internal structure of a phrase as suits our purposes.

In the above example $d_1\{\gamma d \beta\}$ is a schema whose phrase contains a nonterminal symbol (i.e., $\gamma d \beta$ contains the nonterminal symbol $d$). When a nonterminal symbol occurs in a phrase, it is called a schema variable. Schema variables denote the set of all completed_phrases (usually this set is infinite) that can be derived from the schema variable according to the production rules of the grammar under consideration.

In addition, it should be noted that from the point of view of schema variables, the output phrase of a transformation stands in an intimate relationship to its input phrase. Essentially, the output phrase of a transformation is a rearrangement or possibly a reduction of the schema variables occurring in the input phrase. While the output phrase of a TAMPR transformation may contain terminal symbols that do not occur in the input phrase, TAMPR does not permit the output phrase of a transformation to introduce a schema variable that does not occur in the corresponding input phrase. Clearly, when one applies a transformation to a concrete program any (new) schema variable that is introduced in the output pattern would not have any concrete code segment corresponding to it. This would result in a syntactically illegal program.

Lastly, TAMPR requires that the dominating symbol of the output pattern of a transformation be the same as the dominating symbol of its corresponding input pattern. This requirement ensures that transformations produce syntactically legal programs.

5.2.1 A Simple Example

We begin by illustrating how TAMPR works with a simple example. Consider the following grammar:

```
expr $\rightarrow$ expr + term
expr $\rightarrow$ expr - term
expr $\rightarrow$ term

term $\rightarrow$ term * factor
term $\rightarrow$ term / factor

factor $\rightarrow$ digit | (expr)
```

In this grammar, the SDT for the expression “7 * (5 + 3)” is:
And the SDT for the expression “$7 \times 5 + 7 \times 3$” is:

Figure 2

The schema $expr\{7 \times (5 + 3)\}$ in our notation denotes the SDT in Figure 1. Similarly, $expr\{7 \times 5 + 7 \times 3\}$ is our notation for the SDT in Figure 2. A TAMPR transformation can be constructed that rewrites the schema $expr\{7 \times (5 + 3)\}$ into $expr\{7 \times 5 + 7 \times 3\}$ as follows:
Section 5.2  Overview

\[ T_1 \overset{\text{def}}{=} expr \{
    \begin{align*}
    &.sd. \\
    &expr \{ 7 * (5 + 3) \} \\
    \Rightarrow \\
    &expr \{ 7 * 5 + 7 * 3 \} \\
    &.sc.
    \end{align*}
\}

The heart of this transformation is:

\[
expr \{ 7 * (5 + 3) \} \\
\Rightarrow \\
expr \{ 7 * 5 + 7 * 3 \}
\]

which states that the (input) schema \( expr \{ 7 * (5 + 3) \} \) is to be replaced with the (output) schema \( expr \{ 7 * 5 + 7 * 3 \} \). The rest of the symbols occurring in the transformation can be viewed as syntactic “sugar” that has been added for the benefit of TAMP.

To prove the above transformation correct, one would need to prove the following theorem:

**Theorem 5**  \( \text{meaning} (expr \{ 7 * (5 + 3) \}) \sqsubseteq \text{meaning}(expr \{ 7 * 5 + 7 * 3 \}) \)

The sufficiency of this theorem for proving correctness is discussed in Chapter 3. The proof of this theorem amounts to formally verifying that the meaning of \( expr \{ 7 * (5 + 3) \} \) is less-defined than the meaning of \( expr \{ 7 * 5 + 7 * 3 \} \) (i.e., \( expr \{ 7 * 5 + 7 * 3 \} \) is a refinement of \( expr \{ 7 * (5 + 3) \} \)). For the purposes of this example we will assume that the reader knows the formal semantics of the grammar given above. Using the formal semantics of our expression grammar we conclude that the meaning of \( expr \{ 7 * (5 + 3) \} \) is 56 and that the meaning of \( expr \{ 7 * 5 + 7 * 3 \} \) is also 56. From Chapter 3 we know that 56 \( \sqsubseteq \) 56. Recall, the equivalence relation \( \equiv \) implies the less defined relation \( \sqsubseteq \). Bear in mind that in a formal proof of the above theorem one would need to formally and explicitly define the semantics of the expression grammar. This would allow us to formally deduce that “meaning(expr \{ 7 * (5 + 3) \})” and “meaning(expr \{ 7 * 5 + 7 * 3 \})” are both equal to the mathematical object 56.

5.2.2  A Second Example

If the only transformations that one could write in TAMP had a form similar to that of the transformation given in the previous example, then writing TAMP transformations would not be very interesting, and proving their correctness would be relatively straightforward. Proving the correctness of a transformation would amount to using the formal semantics of the language to show that the meaning of the input schema is less defined than the meaning of the output schema. However, as was mentioned earlier, TAMP transformations can be written in terms of phrases containing schema variables. The problem here is “What is the meaning of a schema variable?” Intuitively, a transformation whose input phrase contains one or more schema variables will match (i.e., can denote) a very large (potentially infinite) class of SDT’s. One of the contributions of this research has been to extend the traditional denotational semantics in order to allow a formal semantics to be assigned to schema variables. How this is accomplished will be discussed in Chapter 6.

We now consider a transformation that is a more general version of the transformation presented in our first example.

\[ T_2 \overset{\text{def}}{=} expr \{
    \begin{align*}
    &.sd. \\
    &expr \{ \text{term} \ “1”*\{expr \ “1”+\text{factor} \ “1”\} \} \\
    \Rightarrow \\
    &expr \{ \text{term} \ “1”*\{expr \ “1”\}+\text{term} \ “1”*\text{factor} \ “1”\} \\
    &.sc.
    \end{align*}
\}

Note that the quoted integers here are used as a means to distinguish various instances of schema variables from
one another, and to describe relationships of schema variable occurring in the input pattern with those occurring in
the output pattern. Quoted integers are not part of the expression grammar, but rather they are part of our transfor-
mation notation. For example, term “1” denotes a particular (specific) instance of the schema variable term. In the
transformation T2 the instance of the schema variable denoted by term “1” occurs once in the input schema and twice
in the output schema.

Applying T2 once to the SDT expr {7 * (5 + 4 + 3)} yields the SDT expr {7 * (5 + 4) + 7 * 3}. Applying T2 to
the SDT expr {7 * (5 + 4)} yields the SDT expr {7 * (5) + 7 * 4}. Thus the overall effect of exhaustively applying
T2 to the SDT expr {7 * (5 + 4 + 3)} yields the SDT expr {7 * (5) + 7 * 4 + 7 * 3}).

As was the case in example 1, proving the correctness of transformation T2 amounts to proving the following:

\[ \text{Theorem 6} \quad \text{meaning}(\text{expr}\{\text{term}\cdot1\cdot(\text{expr}\cdot1\cdot) + \text{factor}\cdot1\cdot}) \supseteq \text{meaning}(\text{expr}\{\text{term}\cdot1\cdot\text{expr}\cdot1\cdot + \text{term}\cdot1\cdot\text{factor}\cdot1\cdot}) \]

In order to prove the correctness of this theorem, one needs to make use of the property that multiplication
distributes over addition. If distributivity of multiplication over addition is an axiom or a known theorem in our
formal semantic domain, this is not a problem. However, if this is not the case, then such a theorem would need to
be proven as a prerequisite for proving the correctness of T2.

This example illustrates that even the correctness proofs of very simple TAMPR transformations can possibly
require proofs of nontrivial theorems in the semantic domain.

5.3 Application of TAMPR Transformations

The semantics of the application of a TAMPR transformation \( T \overset{\text{def}}{=} t_\text{in} \Rightarrow t_\text{out} \) to a program \( x \) is denoted \( T(x) \) and is informally described by the following pseudo-code:

\[
\begin{align*}
current\_program & := x \\
\text{while} \ (\text{the SDT of the current\_program contains an instance of the schema} \ t_\text{in}) \ \text{do} \\
\text{begin} \{\text{while loop}\} \\
\text{find the first instance of} \ t_\text{in} \ \text{in the current\_program. This is accomplished by} \\
\text{searching the SDT of the current\_program from bottom to top and left to right} \\
\text{(i.e., from leaves to the root) for the first occurrence of a schema matching} \ t_\text{in}. \\
temp\_program & := \text{the current\_program with the first instance of} \ t_\text{in} \ \text{transformed into} \\
\text{t_out}. \\
current\_program & := temp\_program \\
\text{end} \{\text{while loop}\}
\end{align*}
\]

It should be noted that it is possible to write a TAMPR transformation \( T \) for which there exists a program \( x \)
such that \( T(x) \) never terminates. However, this poses no problem for our methodology because we are interested
only in formally proving that if the application of a transformation halts, then the output program produced will
possess some desired property (e.g., correctness with respect to the input program). Clearly a transformation whose
application never halts will never produce an output program. Whether or not the application of a transformation
terminates is not an issue with respect to this research.

5.3.1 Transformation Sequences

Up to this point we have been considering individual TAMPR transformations. We have given some simple examples
of TAMPR transformations and we have discussed how one could go about proving the correctness of such transfor-
mations. In addition we also informally defined the semantics of a TAMPR transformation (i.e., how TAMPR
Section 5.4 Various Types of TAMPR Transformations.

TAMPR has four constructs that allow the description of various aspects of SDT’s. By combining these constructs it is possible to define very complex and expressive transformation patterns. The goal of this research is to construct a methodology that will allow proofs of correctness theorems involving some of the more common kinds of transformation patterns. An area of future research will be to extend the current methodology in order to allow reasoning about the entire class of transformations that are available in TAMPR.

TAMPR transformations can be constructed from four basic types of constructs.

1. Static Constructs - these are simply expressions of the form: \( \text{dominating\_symbol \{phrase\} } \). Figures 1 and 2 are examples of static constructs.

2. Annotation Constructs - In TAMPR, schema variables may themselves be annotated by qualification expressions and by dynamic patterns.
   
   (a) Qualification Expressions - Qualification expressions (or qualifications for short) allow one to define input schemas that match particular contexts. Essentially, qualifications are Boolean-like expressions containing schemas that can be associated with schema variables. The purpose of qualification expressions is to enable the specification of a restricted set of SDT’s. An SDT, \( t \), belongs to the set defined by a qualification only if \( t \) satisfies the qualification (i.e., only if the qualification evaluates to true with respect to \( t \)). The schema variables of input schemas may be annotated by qualifications. For example, consider the input schema

   3. If \( W \)

   4. If \( W \)

   5. If \( W \)

   6. If \( W \)

   A set of TAMPR transformations can be composed into a transformation-set-sequence according to the following rules:

   1. A single program transformation \( T_i \) is a transformation-set-sequence, and is denoted by \( T_{i,1} \). The application of \( T_{i,j} \) to an input program \( x \) results in the (exhaustive) application of \( T_i \) to \( x \) and is denoted by \( T_{i,j}(x) \). It should be noted that the only purpose served by the subscripts of transformations is to provide us with a notation that allows us to distinguish transformations from one another. Having said this, it should be clear that it is possible to rename the subscripts of transformations without affecting their semantics.

   2. If \( T_{i,j} \) and \( T_{m,n} \) are program transformation-set-sequences, then \( T_{i,j} \circ T_{m,n} \) is a transformation-set-sequence. The application of \( T_{i,j} \circ T_{m,n} \) to a program \( x \) is accomplished by first applying \( T_{m,n} \) to \( x \) (yielding \( T_{i,j}(x) \) as an output program) and then applying \( T_{i,j} \) to the program \( T_{m,n}(x) \). As we mentioned earlier, we can renumber of subscripts of \( T_{m,n} \) so that they range from \((j + 1) \) to \((n - m) \) \((j + 1) \). This gives us \( T_{i,j} \circ T_{(j+1),(n-m)+(j+1)} \) which we denote by \( T_{(n-m)+j+1} \).

   3. If \( T_{i,j} \) is a transformation-set-sequence, then \( (T_{i,j})^* \) is a transformation-set-sequence. The application of \( (T_{i,j})^* \) to an input program \( x \) results in an exhaustive application of \( T_{i,j} \) to \( x \). This exhaustive application is accomplished by first applying \( T_{i,j} \) to \( x \), which results in the output program \( T_{i,j}(x) \). The transformation-set-sequence \( T_{i,j} \) is then applied to \( T_{i,j}(x) \), yielding \( T_{i,j}(T_{i,j}(x)) \). This continues until an output program, \( y \), is obtained such that none of the individual transformations in \( T_{i,j} \) apply to \( y \). If such a program is not obtained, then the application of \( T_{i,j} \) is continued indefinitely.

   As was the case with individual transformations, it is possible to construct transformation-set-sequences that will never terminate when they are applied to a particular program, even in the case when the application of each individual transformation in the sequence, when considered in isolation, halts. In this research, we do not consider nonterminating transformation-set-sequences. Instead, we focus our attention on transformation-set-sequences that halt. Again, this poses no problems from a correctness standpoint, because a transformation-set-sequence whose application never terminates will never produce an output program. Whether or not a transformation-set-sequence terminates is not an issue with respect to this research.
where $b$ is a schema variable. In this context, the schema variable $b$ can be annotated with the qualification $\text{qual}$ by writing $b\{\text{qual}\}$. Thus the annotated input schema becomes $d\{ab\{\text{qual}\}\beta\}$. Now, the schema $d\{ab\{\text{qual}\}\beta\}$ is considered to match another schema having the form $d\{ab\{\gamma\}\beta\}$ only if $b\{\gamma\}$ is an SDT belonging to the set of SDT’s defined by the expression $\text{qual}$.

(b) Dynamic Patterns - Dynamic patterns can be used to define a somewhat more arbitrary set of SDT’s than those that can be defined using qualifications. As with qualifications, dynamic patterns are expressions containing schemas that can be used to annotate schema variables. In TAMPR, the wild card symbol is denoted by a question mark “?”. This symbol plays a role similar to the “*” symbol in UNIX. For example, if the schema variable $e$ is annotated with the dynamic pattern $i?Bj$ it denotes the set of all phrases beginning with $i$ and ending with $j$ that can be derived from the dominating (i.e., nonterminal) symbol $b$ with respect to the grammar $G$ under consideration. Thus any program whose SDT contains a subtree having $b$ as its root symbol such that $b$ derives a phrase of the form $\alpha\gamma\beta$ will match the schema $b\{\alpha?\beta\}$.

3. Subtransformation Pattern Constructs - These are the most complex transformations available in TAMPR. A subtransformation is an entire transformation sequence that can be associated with a schema that occurs in the output schema of another transformation. In other words, a subtransformation is a transformation sequence that occurs within the context of another transformation. Subtransformations are useful when a set of transformations needs to be applied in a limited context, for example when it is necessary to substitute the value of a variable only within the scope of its declaration, or to rename an identifier within its scope.

In practice, qualification expressions, dynamic patterns, and subtransformation patterns can occur as part of a single TAMPR transformation, and it is through their interaction that the expressive capabilities of TAMPR transformations are realized. For a formal treatment of the syntax and semantics of these constructs see [8].

In this work, we construct a methodology that will allow us to formally reason about:

1. Transformations composed entirely of static schemas.
2. Transformations composed of static patterns and “common dynamic patterns”. The difference between a “common dynamic pattern” and a dynamic pattern will be formally discussed in Chapter 6. It so happens that every dynamic pattern that has been written for TAMPR belongs to the class of “common dynamic patterns”. This is not surprising since dynamic patterns that do not belong to the class of “common dynamic patterns” would in practice be either 1) very difficult to understand, or 2) very difficult to apply in a useful manner.
3. A large class of transformation sequences composed from any transformation that we can individually reason about. Again, this will be discussed in Chapters 6 and 7.

As for qualification patterns and subtransformation patterns, we leave the incorporation of these constructs into our methodology to future research.
Chapter 6
The Formal Semantics of Individual Transformations

6.1 Overview
In this chapter we focus our attention on formalizing the semantics of individual TAMPR transformations. This formalization consists of four parts. First we must formally define the syntax and semantics of the wide spectrum specification-programming language in which we wish to transform high-level specifications into executable programs. Towards this end, we introduce the language Poly which we have chosen to demonstrate the practical value of our methodology. We begin with a high-level discussion of some of the features of Poly that make it a language well suited to the transformational programming paradigm. Following this discussion, we give a denotational semantics for Poly. This denotational description includes 1) an informal description of the semantic foundation, $M$, that our denotational semantics is based on, 2) an informal description of the mathematical model of computation used to define the semantics of Poly, 3) formal definitions of the auxiliary functions that are used, and 4) the denotational semantic definitions of a substantial subset of Poly.

Second, we describe how we have been able to extend through $\text{delta}$-functions, the traditional denotational semantics of Poly. $\text{Delta}$-functions are a contribution of this research and essentially enable formal semantics to be assigned to schema variables. This allows us to use denotational semantics, extended with $\text{delta}$-functions, to formally define the semantics of TAMPR (static) schemas.

In the third step, we describe how one can define the semantics of qualification expressions, and finally we describe how one can define the semantics of a specific class of dynamic patterns. Having accomplished this, we are in a position to consider reasoning about individual TAMPR transformations that are composed from static schemas, qualification expressions, and specific dynamic patterns.

6.2 The Language Poly
Poly is a wide spectrum language that is essentially composed of ML, Lisp, and FORTRAN 66 constructs. A partial syntax of Poly is given in appendix B. Poly is a language that was specifically created for use with TAMPR by Dr. James M. Boyle of Argonne National Laboratory, Dr. Terence J. Harmer of The Queen’s University of Belfast, and Mr. Steven Fitzpatrick. Poly has been used to specify and implement solutions to a wide range of problems, including the programs in the LINPACK package for solving systems of linear equations, cellular-automaton and multigrid partial differential equation solvers, assignment (non-linear programming) algorithms, and the specification and implementation of the TAMPR transformer itself.

In Poly, a formal specification can be written in a functional form that can then be transformed into a FORTRAN-like program with the assistance of TAMPR. Note, we say a FORTRAN-like program, not a FORTRAN 66 program. The reason we make this distinction is because the FORTRAN-subset of Poly in some cases has a different semantics than FORTRAN 66. For example, the FORTRAN-subset of Poly, that we will from here on out refer to as Poly-Fortran, allows subroutines to call one another in a recursive fashion; this is not allowed in FORTRAN 66. Another difference is the semantics of parameter passing. In Poly, all subroutine parameters are passed by value. This is significantly different from the parameter passing scheme of FORTRAN 66.

The fact that Poly-Fortran differs from FORTRAN 66 is an important point because in our current programming
paradigm (i.e., programming through correctness preserving transformations), we obtain machine executable assembly code from a high-level functional specification in a two step process. In the first step, the functional specification is transformed into a Poly-Fortran program, \( p \), through correctness preserving transformations. In the second step, the program \( p \) is compiled using an existing FORTRAN 66 compiler. For this paradigm to work, the semantics assigned to \( p \) by the FORTRAN 66 compiler need to be a refinement of the semantics assigned to \( p \) in Poly. If this refinement condition is satisfied, then the assembly code that is produced by the compilation of \( p \) will be correct with respect to the original function specification (provided the FORTRAN 66 compiler is correct).

The problem now is how can one bridge the gap between Poly-Fortran and FORTRAN 66? Fortunately, there exists a subset \( \mathcal{P} \) of Poly-Fortran such that for any program \( p \in \mathcal{P} \) the semantics of \( p \) with respect to the denotational definitions of Poly will be equivalent to or less-defined-than the semantics of \( p \) with respect to the formal semantics of FORTRAN 66. At present, the subset \( \mathcal{P} \) has only been informally defined. We leave the formal definition of \( \mathcal{P} \) to future work. A task we believe to be time consuming but relatively straightforward. Once \( \mathcal{P} \) has been formally defined, all that remains to be done is to require that our transformation sequence produce an output program belonging to \( \mathcal{P} \).

In this dissertation, therefore, we are solely concerned with constructing a methodology that will allow us to formally prove the correctness of program transformations that transform one Poly program (e.g., a specification) into another Poly program (e.g., a Poly-Fortran implementation).

In closing, it should be noted that one could do away with the need for obtaining machine executable code from a FORTRAN 66 compiler altogether by simply continuing the transformation process. In such an approach a specification would be transformed into machine executable code rather than a Poly-Fortran implementation. One reason we do not take this approach is this work to minimize the number of transformations that need to be written and whose correctness needs to be proven. Another reason is for portability. For example, a number of FORTRAN 66 compilers exist that will produce machine code for various machine instruction sets. This means that our Poly-Fortran implementation is relatively portable and can be compiled and executed on a number of different machines. However, if we had decided to transform specifications into machine code then our transformation sequence would need to be changed whenever we wished to obtain machine code for a different instruction set.

### 6.3 The Semantics of Poly

In order to formally reason about the correctness of Poly program transformations, we need a formal definition of the semantics of Poly. As was mentioned in Section 2.1.4, we will formally define the semantics of Poly using denotational semantics. Towards this end, our first step is to construct a set of mathematical objects \( M_f \) whose formal semantics we will assume are known. The set \( M_f \) is essentially made up of 1) constants of various types (e.g., integer constants), 2) functions, and 3) various operations on these types (e.g., the addition operation, the composition operation, etc.). We would like to stress the fact that we are assuming that the objects in \( M_f \) have a semantics that is operation, etc.). We would like to stress the fact that we are assuming that the objects in \( M_f \) have a semantics that is known to the reader. Because the objects in \( M_f \) are known, they do not need to be explicitly given a formal semantics. However, for the sake of readability, we will provide informal semantics for these objects. These informal semantics will be in the form of English descriptions and examples.

#### 6.3.1 The Mathematical Foundation

1. **function access**: \( g(x) \) — Let \( g \) denote an arbitrary function and let \( x \) denote an arbitrary input. The expression \( g(x) \) denotes the value that is produced when the function \( g \) is evaluated with respect to the input \( x \). For example, let \( g \) be a function mapping integers to reals. Furthermore, let \( g \) be a function which maps the integer 2 to the real number 3.5. The expression \( g(2) \) denotes the real number corresponding to the input integer 2, which in this case would be 3.5.

2. **function alteration** — Let \( g \) be a function mapping integers to reals. The expression \( [2 \mapsto 1.5]g \) denotes a function that maps the integer 2 to the real number 1.5. In the case for integers other than 2, the function \( [2 \mapsto 1.5]g \) has the same output value (i.e., mapping) as \( g \).

3. the set of identifiers \( \mathcal{ID} \equiv \{ x \mid x \text{ is an identifier} \} \). The identifiers that belong to the set \( \mathcal{ID} \) are the traditional program identifiers that are available in imperative programming languages like Pascal. For example, \( x_1, x_2, \) and \( x_{yz} \) are all identifiers that belong to the set \( \mathcal{ID} \).

4. the sets integer, real, logical, character, double precision, and complex. At first glance one might get the
impression that double precision should be a term denoting a set of syntactic elements that exist inside a programming language like FORTRAN or Poly. The inclusion of sets like double precision in our mathematical foundation raises the philosophical question of what objects can be “reasonably” included in Mf? Can all of the syntactic objects in Poly be included in Mf? If not, which syntactic objects cannot be included and why? Essentially, by including more syntactic objects from Poly in Mf, our denotational definitions become more axiomatic in nature. In the limit, when all the syntactic objects in Poly are part of Mf, what we are left with is an axiomatic definition of Poly.

In this research we are not interested in working out the details of a numerical model that will permit reasoning about the precision of numerical computations. We acknowledge that numerical models are of major importance to computational science, but we leave such research to those working in the field of numerical analysis. Our interest lies in the construction of a general methodology that will allow correctness proofs of TAMPR transformations. We therefore feel justified in simplifying our problem by including sets like double precision in our mathematical foundation.

- We will use this symbol to denote general (undecidable) nontermination. This symbol is discussed further in section 6.3.5.
- We will use this symbol to denote semantic typos. This is discussed further in section 6.3.5.

7. mathematical operations:

(a) We assume that these operations have “built in” type checking. For example, suppose we tried to add the integer 5 to the character b. In this case the result of the addition operation would be \( \text{⊥}_{\text{error}} \). We realize that simply assuming that arithmetic operations in our mathematical foundation will behave in such a “well defined” manner is perhaps overstepping what can “reasonably” be included in a mathematical foundation. However, we feel that the extra complexity introduced by supplying such type checking information within the denotational semantics serves no useful purpose with respect to the objectives of this research.

(b) the choice operation: \( a \triangleright b; \square c \). If the expression \( a \) evaluates to true, then the value of the expression \( a \triangleright b; \square c \) is \( b \). If \( a \) evaluates to false, then the value of the expression \( a \triangleright b; \square c \) is \( c \), and if \( a \) is undefined, then the value of \( a \triangleright b; \square c \) is undefined. In addition, we permit the choice notation to be extended to permit a case statement. For example, the evaluation of the expression

\[
\begin{align*}
& a_1 \triangleright b_1; \\
& a_2 \triangleright b_2; \\
& \square c
\end{align*}
\]

is \( b_1 \) if \( a_1 \) is true, is \( b_2 \) if \( a_1 \) evaluates to false and \( a_2 \) evaluates to true, and is \( c \) in the case where both \( a_1 \) and \( a_2 \) evaluate to false.

The notation we use for the choice operation is similar to the notation that is used in [63].

(c) the fixed point operator. Fix is an operation that accepts a functional (i.e., a recursive equation), \( f \), as its input and that produces the least fixed point of \( f \) as its output. Essentially, fix is nothing more than a notation that allows us to denote the least fixed point of a functional. For example, consider a Pascal-like language having a while-loop construct whose syntax has the form while \( B \) do \( C \), where \( B \) is a boolean expression and \( C \) is a compound statement. Furthermore, let \( s \) denote a (traditional) store function, and let \( B_{\text{BOOL}} \) and \( C_{\text{STMT}} \) denote valuation functions in a direct denotational semantics (i.e., a denotational semantics without continuations) of this Pascal-like language. One way the while-loop construct (given above) can be defined in this framework is as follows:

\[
C_{\text{STMT}}[[\text{while } B \text{ do } C]] \triangleq \lambda s. B_{\text{BOOL}}[[B]] \triangleright C_{\text{STMT}}[[\text{while } B \text{ do } C]](C_{\text{STMT}}[[C]]s) \square s.
\]

This definition is somewhat unsatisfactory since the semantics of a while-loop is defined in terms of the semantics of a while-loop (i.e., a while-loop is defined in terms of itself). This stumbling block is removed with the addition of the fix operation to our mathematical foundation. Using fix the semantics of the
6.3.2 The Environment

The environment is a function that maps identifiers to environment_objects and is denoted by the symbol $\varepsilon$. Any form of declaration of an identifier in a Poly program will cause an update to the environment function to occur. This update consists of binding the identifier to an environment_object. The environment_object is a tuple that contains enough information to allow us to define the proper uses of the identifier being declared (e.g., when the identifier can be assigned to, when it cannot be assigned to, etc.). To the reader, it may seem that environment_objects that we are...
about to describe are unnecessarily complex. One of the reasons for this complexity is to facilitate the correctness proofs of transformations that transform Poly-Lisp and Poly-ML constructs into Poly-Fortran constructs.

### 6.3.2.1 The Environment Object

An environment object consists of a type information tuple and a storage information tuple, and has the form:

\[(\text{type_information}, \text{storage_information})\]

The type information tuple consists of an id_type and an evaluation_type. The id_type and evaluation_type provide information concerning what operations an identifier may participate in (e.g., assignment and lookup operations) as well as providing type checking information. For example, in the Poly language, identifiers having \(\text{id}_\text{type} = \text{procedure}\) may not occur on the lefthand side of an assignment statement. On the other hand, identifiers having \(\text{id}_\text{type} = \text{function}\), may in certain contexts (e.g., in an assignment statement occurring within the body of the function) occur on the lefthand side of assignment statements. For example, the return value (i.e., output) of a Poly-Fortran function is stored in this manner.

The evaluation_type information is used to enable type checking whenever an identifier is assigned a value. For example, let \(f\) be a function identifier whose output is of type real. In this case the identifier \(f\) will have \(\text{evaluation}_\text{type} = \text{real}\). Suppose that the assignment statement \(f = e\) occurs at some point in the definition (i.e., in the body) of \(f\). In order for this assignment to be “legal” the value resulting from the evaluation of \(e\) must be of a type that is compatible with the type real. If the type of \(e\) is compatible with the type real, then \(e\) can be cast into a real value.

In Poly, type checking between formal and actual parameters is performed using only the evaluation_type information.

In an environment object, the storage information tuple consists of an l-value_location and an r-value_location. The l-value_location of an identifier is the address in the store that is used when the identifier occurs on the lefthand side of an assignment statement. The r-value_location of an identifier is the location in the store that is used when the identifier is dereferenced during the evaluation of an expression. For many identifiers, their l-value_location will be identical to their r-value_location. However, in Poly a distinction needs to be made between these two uses of an identifier because Poly-Fortran identifiers may not be assigned to while Poly-Fortran identifiers may be assigned to. In our model this can be expressed by defining the l-value_location of a Poly-Lisp identifier to be undefined.

### 6.3.2.2 Identifiers and their Corresponding Environment Objects

Identifiers in Poly programs are divided into four id_types and nine evaluation_types. The id_types are 1) variable, 2) constant, 3) function, and 4) subroutine. The evaluation_types are 1) cell, 2) integer, 3) real, 4) logical, 5) double precision, 6) complex, 7) character, 8) function, and 9) subroutine. All identifiers occurring in a Poly program have an id_type and an evaluation_type associated with them. For example, identifiers occurring in declaration statements and as formal parameters of lambda expressions are of type variable. Variables that are created in declaration statements (i.e., Poly-Fortran identifiers) can be dereferenced with respect to their r-value_location in expressions and can have the results of expressions assigned to them using their l-value_location. Constant identifiers, variables created in formal parameters of “lambda expressions”, and Poly-Lisp and Poly-ML identifiers may only be dereferenced.

Functions are identifiers that are bound to either Poly-Lisp, Poly-ML, or Poly-Fortran functions. A major difference between Poly-Lisp, Poly-ML and Poly-Fortran functions is that an identifier corresponding to a Poly-Fortran function can be assigned to within the body of the function. In addition an identifier corresponding to a Poly-Fortran function may not be assigned to outside of its body. Identifiers corresponding to Poly-Lisp and Poly-ML functions, on the other hand, may not be assigned to at all (other than their initial binding). Lastly, a subroutine is a class of identifiers that may only be dereferenced (i.e., the subroutine may be called). Because a subroutine call does not produce an explicit output, we will associate the evaluation_type none with the evaluation_type of subroutine identifiers.

### 6.3.2.3 The importance of the evaluation_type cell

In the TAMPR transformational approach, the evaluation_type cell plays a crucial role. Initially, Poly specifications may be written in terms of a universal data type. This data type is essentially a Lisp list. As would be expected, these lists can contain any mixture of characters, integers, reals, and even other lists. When transforming a Poly specification into a Poly-Fortran program, this list data type presents a problem. The problem arises because Poly specifications are currently being transformed into imperative Poly-Fortran programs by first applying a sequence of transformations that transforms the functional skeleton of a Poly specification into an imperative Poly-Fortran skeleton, and then by applying a sequence of transformations capable of transforming the list data type that is used in the Poly specification into a representation that is expressible in terms of the data types available in FORTRAN 66.
Chapter 6  The Formal Semantics of Individual Transformations

<table>
<thead>
<tr>
<th>code segment</th>
<th>environment update</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer x;</td>
<td>$[x \mapsto ((\text{variable}, \text{integer}), (\alpha, \alpha))]\varepsilon$</td>
</tr>
<tr>
<td>lambda x @...</td>
<td>$[x \mapsto ((\text{variable}, \text{cell}), (\alpha, \perp_{\text{error}}))]\varepsilon$</td>
</tr>
</tbody>
</table>
| real function f(x) cell x;                | $[f \mapsto ((\text{function}, \text{real}), (\alpha, \perp_{\text{error}}))]$
|                                           | $([x \mapsto ((\text{variable}, \text{cell}), (\alpha', \alpha'))] \varepsilon)$ |

1. Declaration Examples

How functional skeletons can be transformed into imperative Poly-Fortran skeletons is discussed in greater detail in Chapter 8. At this point we are only interested in the “data type” problem that arises from this approach.

As we have already mentioned, the first sequence of transformations that is applied to a Poly specification essentially transforms the functional skeleton into a Poly-Fortran skeleton. This transformation sequence produces a Poly-Fortran skeleton that essentially defines a computation using the variables that occurred in the original specification (i.e., the computation is defined in terms of variables of type list.) Certainly, the list data type is not part of FORTRAN 66. However, we do define a list data type for Poly-Fortran. The data type cell serves this purpose. An identifier of type cell may be bound to a list, and cell is a data type that is part of Poly-Fortran. In addition to being part of Poly-Fortran, cell is also the universal data type that is used in Poly specifications. It is through cell that we are able to “transformationally” bridge the gap that exists between the data type occurring in functional specifications and the various data types available in Poly-Fortran and ultimately those in FORTRAN 66.

6.3.2.4 Examples We now give a few examples of various Poly identifiers and their corresponding environment objects. As we have already mentioned, in our model of computation an environment object has the form:

$(((\text{id\_type}, \text{evaluation\_type}) , (r\text{-value\_location}, l\text{-value\_location}))$.

For an identifier $i$ that is declared in an integer declaration statement, its id_type will be variable and its evaluation_type will be integer. For a Poly-Fortran integer function, its id_type will be function and its evaluation_type will be integer. The r-value_location denotes the location in the store where the value corresponding to $i$ will be stored. Similarly the l-value_location denotes the location in the store that is used when $i$ occurs on the left-hand side of an assignment statement. For Poly-Fortran identifiers of type variable, there is no difference between the r-value_location and the l-value_location. An identifier of type constant has an r-value_location but has an l-value_location that is undefined (i.e., $\perp_{\text{error}}$). An identifier corresponding to a Poly-Fortran function will have a l-value_location that is undefined when the identifier is assigned-to outside of its body and it will have a defined l-value_location when it is assigned-to within its body. It is the job of our denotational semantic definitions to ensure that all of this happens correctly. Bear in mind, the informal description we are presenting is simply a high-level summary of what is going on in the denotational semantic definitions of Poly. The informal descriptions we are now giving are for the benefit of the reader and play no part in the formal semantics of Poly.

We conclude this section with a table containing various Poly code segments that are responsible for environment updates along with the actual environment update they produce. In this table the notational conventions are as follows:

- The symbol $\varepsilon$ denotes the environment that exists before the particular code segment is encountered. The updates that are caused by the various code segments appearing in the table will be defined in terms of the initial environment $\varepsilon$.
- The symbols $\alpha$, and $\alpha'$ denote new (i.e., fresh) storage locations.
- The symbol $\perp_{\text{error}}$ denotes the undefined value.
- The symbol $\perp'_{\text{error}}$ denotes an undefined value that will be replaced by a defined value (i.e., an $\alpha$) at some point during the execution of the program. Theoretically, there is no difference between $\perp_{\text{error}}$ and $\perp'_{\text{error}}$. The only reason why $\perp'_{\text{error}}$ is used is to remind the reader that this particular undefined value can become defined at certain points during the execution of a program. For example, the l-value_location address of a function identifier becomes defined in the body of the function and becomes undefined after the body of the function has been exited.
6.3.3 The Location Counters $\alpha$ and $\beta$

In addition to an environment function, our model of computation contains a primary location counter, $\alpha$, and a secondary location counter, $\beta$. The function of the location counters is to provide new locations in the store function. This will enable the store function to create mappings between locations and the (denotable) values that are associated with identifiers in Poly programs. For this kind of bookkeeping to work, each identifier in a Poly program needs to be associated with a unique location in the store. (For the sake of simplicity, at this point in the discussion we are not considering the case of aliasing and parameter passing. It is the job of $\alpha$ and $\beta$ to ensure that this is done correctly.) One might ask why two counters are needed. Shouldn’t one be enough? It turns out that having two counters is very convenient (almost necessary) when one considers the semantics of Lisp and ML, together with Poly_Fortran functions. Recall that our objective is to transform a functional specification into a Poly-Fortran program using correctness-preserving transformations. The addition of the $\beta$ counter to our model of computation makes these proofs much easier. Essentially the purpose served by $\beta$ is to provide a means to “make room” for Poly_Fortran function identifiers without disturbing the correspondences between program identifiers and $\alpha$ storage locations.

6.3.4 The Store

The last part of our model of computation is the store function. The store is a function that maps (binds) locations to denotable values. We make a distinction between denotable values and values because we wish to emphasize the fact that denotable values are values that we expect can be stored in the memory of a real (physical) machine in some form. Having said this, from now on we will use the terms denotable values and values interchangeably.

In the store function then, are “placed” (i.e., an initial store function is updated through the function alteration operation) all values that are associated with Poly identifiers. For example, if the variable $x$ is assigned the value 5 in a Poly program, then 5 will be placed in the store in the location corresponding to the variable $x$. The values (i.e., meanings) of identifiers corresponding to functions and subroutines are also placed in the store. Recall the objective of denotational semantics is to map a syntactic construct, $s$, (e.g., an assignment statement, a block of code, etc.) occurring in a Poly program into a semantic expression, $e$, constructed solely from the composition of objects belonging to the mathematical foundation $M_t$. Therefore, when we refer to the meaning, or value of a syntactic object, $s$, what we are referring to is the expression $e$ that corresponds to $s$ with respect to our denotational semantic definitions. In the above example involving the assignment of 5 to the variable $x$, the value 5 is placed in the store because 5 is an object that in addition to being a syntactic object in Poly also belongs to our mathematical foundation $M_t$.

6.3.5 Computation, Nontermination, and Undefined Operations

In this section, we present a paradigm of the general computing process and how it relates to nontermination and undefined operations. The paradigm presented here is a contribution of this research. We believe that this paradigm resolves much of the vagueness that surrounds the notions of computation, nontermination, and undefined operations. We begin by giving a brief background of nontermination and undefined operations.

6.3.5.1 Background

Formal language theory has shown that (programming) languages that are defined in terms of context-free grammars can be easily recognized. Because of this, modern day programming languages are defined in terms of context-free grammars. Ideally, any sentence derived from the start symbol of a grammar should be part of the language defined by that grammar. Unfortunately, context-free grammars are not quite powerful enough to capture some of the syntax requirements of modern day programming languages (e.g., Pascal). For example, in a context-free grammar, it is not possible to write productions requiring that variables be declared before they are used. This means that a sentence $s$ can be derived from the start symbol of the context-free grammar defining Pascal such that $s$ is not considered to be part of the Pascal language. This seems to undermine the whole idea of using a grammar to define the syntax of a language. One way to resolve this dilemma is to consider sentences like $s$ to be syntactically part of Pascal, but that the meaning of $s$ will be some error value, where it is agreed that no one wants to write a program whose meaning is equal to this error value.

Early programming languages like FORTRAN did not require variables to be explicitly declared. This lead to a large class of program bugs. For example, any misspelling of an identifier would result in the creation of a new variable rather than raising some sort of error flag. People in the programming community soon realized that program debugging would be much easier if the programmer did not have to worry quite so much about misspelled-variable
bugs. Therefore, newer languages and newer versions of existing languages required variables to be explicitly declared before their use. Not doing so would cause the compiler to map the meaning of the program to the error value. This language improvement greatly reduced the probability that a misspelled identifier would go undetected. Note that if an identifier \( x \) is misspelled as \( y \) then if \( y \) also happens to be a declared variable we still have a problem. Typing has further reduced the likelihood that an identifier misspelling will go undetected. However, bugs arising from misspelled variables can still occur, although their likelihood is greatly reduced.

The purpose of requiring variables to be typed and declared before their use is to enable the compiler to assist the programmer in detecting semantic typos. Informally, a semantic typo can be thought of as a semantic error that arises more from a “slip of the finger” rather than a “conceptual error”. As programming languages evolved, compilers began to check for more and more kinds of semantic typos. For example, many compilers require that variables must be assigned-to before they are dereferenced. The end result of all of this is that in most modern programming languages, there are a great many sentences that are syntactically correct as far as the context-free grammar is concerned, but whose meaning will be mapped to the error value by a compiler for that language.

6.3.5.2 Nontermination If semantic typos are at one end of the “semantic error” spectrum, then nontermination is at the other. For a specific set of inputs, deciding whether a program will halt is, in general, undecidable. Mathematicians usually introduce a symbol like \( \downarrow \) to denote nontermination. This allows them to formally consider the consequences of nontermination within their field of study. However, one must keep in mind that one cannot test for nontermination (e.g., if \( f(x) = \downarrow \) then “take some action”). Thus \( \downarrow \) is significantly different from the error value we assign to programs containing semantic typos. In the next section we present a view which essentially unifies semantic typos and nontermination with respect to computation.

6.3.5.3 A Unifying View We begin by viewing the objects in our semantic domain, \( M_f \), as in terms of a hierarchy of sets. The first set (i.e., the lowest set in our hierarchy) we call constant objects. The elements of the sets \( I, D, integer, real, logical, character, double precision, \) and complex are all constant objects. Let \( b \) and \( c \) denote two arbitrary constant objects. The tuple \((b, c)\) is also considered to be a constant object.

The second set we will call function objects. We view constant objects to be a proper subset of function objects. In \( M_f \), there are two ways in which a function object can be constructed. Using lambda calculus, we can construct functions of the form \((\lambda x. e)\), where \( e \) is a semantic expression which will be defined below. In addition to lambda calculus, the function alteration mechanism also provides us with a means to construct function objects. For example, let \( f \) denote an arbitrary function, let \( x \) denote an arbitrary identifier and let \( e' \) denote an arbitrary function object. The function \([x \mapsto e']f\) is also a function object. Let \( f \) and \( g \) denote two arbitrary function objects. The tuple \((f, g)\) is also considered to be a function object. In general, the highest element in a tuple will determine what set the tuple belongs in. For example, if \( f \) is a function object that is not a constant object, and if \( c \) is a constant object, then the tuple \((f, c)\) will be a function object.

The third set (and highest set in our hierarchy) we will call semantic expressions. Function objects are a proper subset of semantic expressions. Semantic expressions are created by passing function objects as arguments to operations. In \( M_f \), we consider function access, expression composition, \(+, -, *, /, choice, and fix\) to be operations. For notational convenience we will assume for this discussion that all operations in \( M_f \) are in prefix form. For example, if the addition operation is in prefix form we would write \(+ (3, 4)\) instead of \(3 + 4\). By pretending that all operations are in prefix form we are able to easily discuss an arbitrary operation \( op \) without having to concern ourselves with the fact that different operation symbols occur syntactically at different places with respect to their operands.

Let \( a' \) denote an arbitrary n-ary vector of semantic expressions, and let \( op \) denote an arbitrary n-ary operation in \( M_f \) in prefix form. Then \( op(a') \) is also a semantic expression. For example, \(+ (3, 4)\). The expression \(+ (3, 4)\) is a semantic expression. It should be noted that \(+ (3, 4)\) is not a function object.

In \( M_f \), axioms exist that define the behavior of operations. Note that we have not given these axioms in this dissertation. The reason for this is that we assume the semantics of \( M_f \) to be known, which in turn implies that a statement of the axioms is unnecessary. Nevertheless, axioms that define operations can be viewed as rewrite rules that allow semantic expressions to be simplified. For example, by applying the axioms of addition to the semantic expression \(+ (3, 4)\) we are able to rewrite this semantic expression into the constant object \( 7 \). In a similar fashion the axioms defining function access allow us to rewrite the semantic expression \( g(x) \) into a simpler form. In general, one can define the simplification of a semantic expression as the removal of an operation through rewrite rules (i.e., axioms) defining that operation. Note, a simplification produces a semantic expression having fewer operations than the original semantic expression.

At this point we introduce the semantic expression \( \downarrow_{error} \). We define \( \downarrow_{error} \) to be a semantic expression that is not a member of the function object set. In addition, \( \downarrow_{error} \) is a semantic expression to which no simplification axioms
can be applied. In other words, it is not possible to rewrite $\bot_{\text{error}}$ to a function object.

The execution of syntactic objects called programs can now be defined in terms of the simplification of semantic expressions. To execute a program it must first be converted into a semantic expression. In our case, this conversion will be accomplished through denotational semantics. The execution of a program is now defined as the simplification of the semantic expression corresponding to the program. Execution of a program is considered to be completed when the expression object corresponding to the program has been simplified to a function object. If such a simplification is not possible then execution of the program is considered to be undefined (i.e., $\bot$).

The simplification of a semantic expression to a function object will not be possible if 1) the semantic expression is simplified to the value $\bot_{\text{error}}$, or 2) an infinite sequence of simplifications exist.

Programs containing semantic typos will have their corresponding semantic expressions simplified to $\bot_{\text{error}}$. Because, by definition, $\bot_{\text{error}}$ cannot be simplified to a function object, the execution of a program having the meaning $\bot_{\text{error}}$ will be undefined. Programs containing infinite loops will be converted to semantic expressions having infinite sequences of simplifications and thus will also be undefined.

The view we have given in this section hopefully gives the reader a reasonable understanding of what we mean when we say “a program is undefined” or “a program is in error (e.g., $\bot_{\text{error}}$)”. The model we have given here will be especially important in section 6.7 when we discuss various aspects of correctness proofs.

### 6.3.6 Auxiliary Function Definitions

Having informally described our model of computation $M$, as well as what it means to execute a Poly program, we are now ready to formally define the semantics of Poly.

Let $M' \overset{\text{def}}{=} \text{Model}(\varepsilon, \alpha, \beta, s)$ denote a mathematical computer (i.e., a model of computation). While it is possible to define the effect the execution of a Poly program will have on $M'$ strictly in terms of expressions constructed from the composition of objects in our mathematical foundation $M_0$, this is generally not desirable. The reason for this is that this approach can lead to semantic definitions that are unnecessarily complex and difficult to read. Therefore, we construct a set of definitions whose sole purpose is to make our denotational semantics definitions more readable. We call these definitions auxiliary definitions because their only purpose is to help us understand the denotational semantic definitions of Poly by abstracting away some of the mechanical complexities.

An auxiliary function is defined by 1) giving the domains of its arguments and its result, and 2) giving a lambda expression that defines the function.

1. update_env: identifier $\times$ environment_object $\times$ model $\rightarrow$ model
   
   $\text{update\_env} \overset{\text{def}}{=} \lambda \text{id.} \lambda \text{o.} \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \text{Model}(\text{id} \mapsto \alpha) \varepsilon, \alpha, \beta, s$

2. update_store: alpha_value $\times$ storeable_value $\times$ model $\rightarrow$ model
   
   $\text{update\_store} \overset{\text{def}}{=} \lambda \text{loc.} \lambda \text{v.} \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \text{Model}(\varepsilon, \alpha, \beta, \text{loc} \mapsto \text{v}) s$

3. update_model_env: environment $\times$ model $\rightarrow$ model
   
   $\text{update\_model\_env} \overset{\text{def}}{=} \lambda \varepsilon'. \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \text{Model}(\varepsilon', \alpha, \beta, s)$

4. update_model_alpha: alpha_value $\times$ model $\rightarrow$ model
   
   $\text{update\_model\_alpha} \overset{\text{def}}{=} \lambda \alpha'. \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \text{Model}(\varepsilon, \alpha', \beta, s)$

5. update_model_beta: beta_value $\times$ model $\rightarrow$ model
   
   $\text{update\_model\_beta} \overset{\text{def}}{=} \lambda \beta'. \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \text{Model}(\varepsilon, \alpha, \beta', s)$

6. update_model_store: store $\times$ model $\rightarrow$ model
   
   $\text{update\_model\_store} \overset{\text{def}}{=} \lambda s'. \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \text{Model}(\varepsilon, \alpha, \beta, s')$

7. env: model $\rightarrow$ environment
   
   $\text{env} \overset{\text{def}}{=} \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \varepsilon$

8. alpha: model $\rightarrow$ alpha_value
   
   $\text{alpha} \overset{\text{def}}{=} \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \alpha$

9. beta: model $\rightarrow$ beta_value
   
   $\text{beta} \overset{\text{def}}{=} \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \beta$

10. store: model $\rightarrow$ store
    
    $\text{store} \overset{\text{def}}{=} \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \text{store}$

11. access_store: alpha_value $\times$ model $\rightarrow$ storeable_value
access_store def \( \lambda \text{loc}. \lambda \text{Model}(\varepsilon, \alpha, \beta, s). s(\text{loc}) \)

access_env def \( \lambda \text{id}. \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \varepsilon(\text{id}) \)

r_value_location def \( \lambda \text{id}. \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \varepsilon(\text{id}) \)

12. access_env: identifier \times \text{model} \rightarrow \text{environment_object}

13. r_value_location: identifier \times \text{model} \rightarrow \text{alpha_value}

l_value_location def \( \lambda \text{id}. \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \varepsilon(\text{id}) \)

14. l_value_location: identifier \times \text{model} \rightarrow \text{alpha_value}

define_l_value_location def \( \lambda \text{id}. \lambda \text{Model}(\varepsilon, \alpha, \beta, s). \varepsilon(\text{id}) \)

15. define_l_value_location: identifier \times \text{model} \rightarrow \text{model}

16. type_of: \text{type} \times \text{denotable_value} \rightarrow \text{type}

type_of def \( \lambda v. \)

(a) inInteger: denotable_value \rightarrow \text{type}

(b) inReal: denotable_value \rightarrow \text{type}

(c) inIdentifier: denotable_value \rightarrow \text{type}

17. get_evaluation_type: identifier \times \text{model} \rightarrow \text{type}

get_evaluation_type def \( \lambda \text{id}. \lambda m. \)

(a) isInteger: \text{type} \times \text{denotable_value} \rightarrow \text{boolean}

(b) isReal: \text{type} \times \text{denotable_value} \rightarrow \text{boolean}

(c) isIdentifier: \text{type} \times \text{denotable_value} \rightarrow \text{boolean}

19. Typing functions. In the formal semantics of Poly, a lot of type checking is performed. In order to permit the kinds of type checking we want to carry out, it is necessary to be able to determine the type of every value that is placed in the store. Through the typing functions below, this is realized.

(a) inInteger: denotable_value \rightarrow \text{type}

(b) inReal: denotable_value \rightarrow \text{type}

(c) inIdentifier: denotable_value \rightarrow \text{type}

20. Type checking functions.

(a) isInteger: \text{type} \times \text{denotable_value} \rightarrow \text{boolean}

(b) isReal: \text{type} \times \text{denotable_value} \rightarrow \text{boolean}
Section 6.3  The Semantics of Poly

isReal def $\lambda (t, v)$.
\( (t = \text{real}) \uparrow \text{true}; \)
\( \Box \text{false} \)

(c) isIdentifier: (type,denotable_value) $\rightarrow$ boolean
isIdentifier def $\lambda (t, v)$.
\( (t = \text{identifier}) \uparrow \text{true}; \)
\( \Box \text{false} \)

21. value_of: (type,denotable_value) $\rightarrow$ denotable_value
value_of def $\lambda (t, v)$.

22. The auxiliary function compatible_types expects to receive the type of an “assigned to” or formal parameter as its first argument and expects the type of an “assignable value” or actual parameter as its second argument. What types are considered “compatible” is determined by the language.

compatible_types:type $\times$ type $\rightarrow$ boolean
compatible_types def $\lambda t_1. \lambda t_2$.
\( (t_1 = \text{cell}) \uparrow \text{true}; \)
\( (t_1 = \text{integer}) \land (t_2 = \text{integer}) \uparrow \text{true}; \)
\( (t_1 = \text{real}) \land (t_2 = \text{real}) \uparrow \text{true}; \)
\( (t_1 = \text{double precision}) \land (t_2 = \text{double precision}) \uparrow \text{true}; \)
\( (t_1 = \text{complex}) \land (t_2 = \text{complex}) \uparrow \text{true}; \)
\( (t_1 = \text{logical}) \land (t_2 = \text{logical}) \uparrow \text{true}; \)
\( (t_1 = \text{character}) \land (t_2 = \text{character}) \uparrow \text{true}; \)
\( \Box \text{false}; \)

23. update_type_function: identifier $\times$ type $\times$ type_function $\rightarrow$ type_function
update_type_function def $\lambda \text{id}. \lambda t. \lambda t_f. [\text{id} \mapsto t]t_f$

24. create_env_object1: type $\times$ alpha_value $\rightarrow$ environment_object
create_env_object1 def $\lambda t. \lambda \alpha. ((\text{variable}, t), (\alpha, \alpha))$

25. create_env_object2: type $\times$ alpha_value $\rightarrow$ environment_object
create_env_object2 def $\lambda t. \lambda \alpha. ((\text{variable}, t), (\alpha, \perp_{\text{error}}))$

26. type_function: type $\rightarrow$ identifier $\rightarrow$ type
    type_function def $\lambda t. \lambda \text{id}. t$

27. update_type_function: identifier $\rightarrow$ type $\rightarrow$ type_function $\rightarrow$ type_function
    update_type_function def $\lambda \text{id}. \lambda t. \lambda t_f. [\text{id} \mapsto t]t_f$

Having formally defined our auxiliary functions, we may now make use of them in a formal setting. In other words, we are finally in a position where we can formally define the semantics of Poly. While the definitions that we give for Poly are formal and do not require any informal explanations, we will nevertheless give informal (English) explanations of various valuation functions that occur in our denotational definition of Poly. These informal explanations are operational in nature. That is, valuation functions will be informally described in an operational manner similar to the approach that one might use to informally describe an algorithm. It should be noted that these informal explanations are included for the sake of the reader and have no effect on the formal definitions themselves.

6.3.7  Denotational Semantic Definitions of Poly

Our denotational semantic definitions for Poly will proceed in a top down fashion (more or less). Initially the denotational definitions performs two passes over the subroutine declaration portion of a program. These two passes are performed by the valuation functions $Z$ and $Y$. This two-pass approach eliminates the need for the complex fixed point definition that is traditionally used to define the appropriate environment for statically scoped recursive subprograms. In the two-pass approach presented in this paper, the valuation function $Z$ essentially binds identifiers of subprograms (including the main portion of the program) to the appropriate memory locations within the environment function $\varepsilon$. The valuation function $Y$ then translates the bodies of the subprograms (including the main) into
the appropriate semantic expressions and places these expressions in the proper locations in the store functions.

1. The meaning of a Poly program is:

\[
Z_{\text{PROG\_UNIT\_CFLIST}}[[\text{prog\_unit}]] \triangleq Z_{\text{PROG\_UNIT}}[[\text{prog\_unit}]]
\]

\[
Z_{\text{PROG\_UNIT\_CFLIST}}[[\text{prog\_unit\_cflist prog\_unit}]] \triangleq \lambda m. \lambda c_m.
\]

\[
\begin{align*}
Z_{\text{PROG\_UNIT\_CFLIST}}[[\text{prog\_unit}]] & \triangleq \lambda m. \lambda c_m. \\
\end{align*}
\]

\[
\begin{align*}
\text{where} \\
Z_{\text{PROG\_UNIT\_CFLIST}}[[\text{prog\_part}]] & \triangleq Z_{\text{PROG\_PART}}[[\text{prog\_part}]] \\
Z_{\text{PROG\_PART}}[[\text{prog\_body}]] & \triangleq Z_{\text{PROG\_BODY}}[[\text{prog\_body}]] \\
Z_{\text{PROG\_BODY}}[[\text{exec\_part}]] & \triangleq Z_{\text{EXEC\_PART}}[[\text{exec\_part}]]
\end{align*}
\]

2. In the following discussion, we use the term subprogram when referring to a Poly function or subroutine. Traditionally, denotational semantics solves the problem of defining the values of functions and subroutines in statically scoped languages with a horrendously complex fixed point equation. For an example of such an equation see [28]. The reason for the fixed point is that a subprogram can, in its body, call another subprogram whose definition occurs (syntactically) after the current subprogram. Such a forward reference causes a problem if the denotational semantic definitions perform only a single pass at this level. The problem is that the identifier which is dereferenced has not been defined yet, and as we have already mentioned the problem has been traditionally solved through the use of the fixed point operator. In our denotational semantics of Poly we take a different approach. We essentially perform two passes over the subprogram definitions. The first pass is performed by the valuation function \( Z \). The valuation function \( Z \) is responsible for constructing an environment with entries for all of the subprograms occurring in the program. Note that because subprogram declarations only occur at a very high level in the grammar, we only need to scan the portion of the grammar capable of declaring subprograms (i.e., we do not need to scan conditional statements, assignment statement, etc.).
Section 6.3 The Semantics of Poly

\[ Z_{\text{EXEC\_PART}} \vert \text{range} \rangle \triangleq Z_{\text{RANGE}} \vert \text{range} \rangle \]
\[ Z_{\text{RANGE}} \vert \text{stmt\_tail} \rangle \triangleq Z_{\text{STMT\_TAIL}} \vert \text{stmt\_tail} \rangle \]
\[ Z_{\text{STMT\_TAIL}} \vert \text{p\_end\_stmt} \rangle \triangleq \lambda m. \lambda c_m. \\
c_m(\text{update\_env}(\text{main}) \\
(\text{subroutine}, \text{none}), (\alpha(m), \bot_{\text{error}})) \\
(\text{update\_model\_alpha}(\alpha(m)+1, m)) \\
) \]
\[ Z_{\text{STMT\_TAIL}} \vert \text{p\_spec\_stmt stmt\_tail} \rangle \triangleq Z_{\text{P\_STMT}} \vert \text{p\_stmt} \rangle \\
Z_{\text{P\_STMT}} \vert \text{prefix stmt ;} \rangle \triangleq Z_{\text{STMT}} \vert \text{stmt} \rangle \]
\[ Z_{\text{STMT}} \vert \text{group\_stmt} \rangle \triangleq \lambda m. \lambda c_m. \\
c_m(\text{update\_env}(\text{main}) \\
(\text{subroutine}, \text{none}), (\alpha(m), \bot_{\text{error}})) \\
(\text{update\_model\_alpha}(\alpha(m)+1, m)) \\
) \]
\[ Z_{\text{STMT}} \vert \text{if\_stmt} \rangle \triangleq \lambda m. \lambda c_m. \\
c_m(\text{update\_env}(\text{main}) \\
(\text{subroutine}, \text{none}), (\alpha(m), \bot_{\text{error}})) \\
(\text{update\_model\_alpha}(\alpha(m)+1, m)) \\
) \]
\[ Z_{\text{STMT}} \vert \text{basic\_stmt} \rangle \triangleq \lambda m. \lambda c_m. \\
c_m(\text{update\_env}(\text{main}) \\
(\text{subroutine}, \text{none}), (\alpha(m), \bot_{\text{error}})) \\
(\text{update\_model\_alpha}(\alpha(m)+1, m)) \\
) \]
\[ Z_{\text{STMT}} \vert \text{convience\_stmt} \rangle \triangleq Z_{\text{CONVIENCE\_STMT}} \vert \text{convience\_stmt} \rangle \\
Z_{\text{CONVIENCE\_STMT}} \vert \text{expressions expr\_list} \rangle \triangleq Z_{\text{EXPR\_LIST}} \vert \text{expr\_list} \rangle \\
Z_{\text{EXPR\_LIST}} \vert \text{expr} \rangle \triangleq Z_{\text{EXPR}} \vert \text{expr} \rangle \\
Z_{\text{EXPR\_LIST}} \vert \text{expr\_list, expr} \rangle \triangleq \lambda m. \lambda c_m. \\
Z_{\text{EXPR}} \vert \text{expr} \rangle \\
m \quad (\lambda m'. Z_{\text{EXPR\_LIST}} \vert \text{expr\_list} \rangle m' c_m) \\
) \]
\[ Z_{\text{EXPR}} \vert \text{op\_expr} \rangle \triangleq Z_{\text{OP\_EXPR}} \vert \text{op\_expr} \rangle \\
Z_{\text{OP\_EXPR}} \vert \text{primary} \rangle \triangleq Z_{\text{PRIMARY}} \vert \text{primary} \rangle \\
Z_{\text{PRIMARY}} \vert \text{entity} \rangle \triangleq Z_{\text{ENTITY}} \vert \text{entity} \rangle \\
Z_{\text{ENTITY}} \vert \text{basic\_entity type\_info} \rangle \triangleq Z_{\text{BASE\_ENTITY}} \vert \text{basic\_entity} \rangle \\
Z_{\text{BASE\_ENTITY}} \vert \text{var} \rangle \triangleq Z_{\text{VAR}} \vert \text{var} \rangle \\
}
Chapter 6  The Formal Semantics of Individual Transformations

\[ Z_{VAR}[\text{function_definition}] \equiv Z_{FUNCTION_DEFINITION}[\text{function_definition}] \]

\[ Z_{FUNCTION_DEFINITION}[\text{functional_declaration end}] \equiv Z_{FUNCTIONAL_DECLARATION}[\text{functional_declaration}] \]

\[ Z_{FUNCTIONAL_DECLARATION}[\text{function_declaration}] \equiv Z_{FUNCTION_DEFINITION}[\text{function_declaration}] \]

\[ Z_{FUNCTIONAL_DECLARATION}[\text{fun rule_list}] \equiv Z_{RULE_LIST}[\text{rule_list}] \]

\[ Z_{RULE_LIST}[\text{rule}] \equiv Z_{RULE}[\text{rule}] \]

\[ Z_{RULE}[\text{expr} = \text{expr}_2] \equiv \lambda m. \lambda c_m. \]

\[ \text{isIdentifier}(F_{EXPR}[\text{expr}_1]) \]

\[ c_m(\text{update_env}(F_{EXPR}[\text{expr}_1]),
\quad ((\text{function}, \bot_{error}), (\alpha(m), \bot_{error})))
\]

\[ (\text{update_model_alpha}(\alpha(m)+1,m)) \]

\[ \Box /* \text{expr}_1 \text{ must be an identifier */} \]

\[ \bot_{error} \]

\[ Z_{PROG_PART}[\text{p_subprog_stmt prog body}] \equiv \lambda m. \lambda c_m. \]

\[ \text{function_type}(F_{\text{p_subprog_stmt}}[\text{p_subprog_stmt}]) \]

\[ c_m(\text{update_env}(F_{\text{p_subprog_stmt}}[\text{p_subprog_stmt}]),
\quad ((\text{function}, R_{\text{p_subprog_stmt}}[\text{p_subprog_stmt}]), (\alpha(m), \bot_{error})),
\quad (\text{update_model_alpha}(\alpha(m)+1,m))) \]

\[ /* \text{Note, initially a function is given an undefined l-value_location. This is done because we want to make assignments to the function id, when they occur outside of the function body, undefined. Only within the body of a function do we want to permit assignments to the function id */} \]

\[ /* \text{else subroutine */} \]

\[ \Box c_m(\text{update_env}(F_{\text{p_subprog_stmt}}[\text{p_subprog_stmt}]),
\quad ((\text{subroutine}, \text{none}), (\alpha(m), \bot_{error})),
\quad (\text{update_model_alpha}(\alpha(m)+1,m))) \]

3. The second pass through the program is performed by the valuation function \( Y \). \( Y \) is a function that is responsible for updating the store with the denotational meanings of subprograms. Initially, \( Y \) is given the environment \( \varepsilon \) that results from the pass performed by the valuation function \( Z \). The environment \( \varepsilon \) associates a location in the store with each subprogram identifier. Thus, whenever \( Y \) encounters a subprogram declaration in the program, it simply needs to store the meaning of the associated subprogram body in the location that has been set aside for it in \( \varepsilon \).

Note that we have written the denotational semantic definitions in such a way that the body of a Poly-Fortran function will be passed an environment in which the Poly-Fortran function identifier will have a defined l-value_location. This means that within the body of the Poly-Fortran function it will be possible to make assignments to the function identifier. However, once the function body is exited, either through a return or through a call to another subprogram, the l-value_location of the function will become undefined. This can be accomplished in a relatively straightforward manner because, in Poly-Fortran, subprograms are statically scoped. Note that a statically scoped function is defined in terms of an environment that is different from environments that

48
exist outside the function body (e.g., in the main portion of the program).

\[ Y_{\text{PROG\_UNIT\_CFLIST}}[[\text{prog\_unit}]] \overset{\text{def}}{=} Y_{\text{PROG\_UNIT}}[[\text{prog\_unit}]] \]
\[ Y_{\text{PROG\_UNIT\_CFLIST}}[[\text{prog\_unit\_cflist prog\_unit}]] \overset{\text{def}}{=} \lambda m. \lambda c_m. \]
\[ (m) \]
\[ \lambda m'. Y_{\text{PROG\_UNIT\_CFLIST}}[[\text{prog\_unit}]] \]
\[ \text{update\_model\_store(store(m'), m)} \]
\[ (c_m) \]
\]
\[ Y_{\text{PROG\_UNIT\_CFLIST}}[[\text{prog\_part ;}]] \overset{\text{def}}{=} Y_{\text{PROG\_PART\_CFLIST}}[[\text{prog\_part}]] \]
\[ Y_{\text{PROG\_PART\_CFLIST}}[[\text{prog\_body}]] \overset{\text{def}}{=} \lambda m. \lambda c_m. \]
\[ \text{is\_main}_F[[\text{prog\_body}]] \]
\[ \triangleright \ c_m \ (\text{update\_store} \ (r\_value\_location(\text{main}, m))) \]
\[ \ (C_{\text{PROG\_BODY}}[[\text{prog\_body}]]) \]
\[ (m) ; \]
\[ \]
\[ /* \text{lisp\ function\ definitions */} \]
\[ Y_{\text{PROG\_BODY\_CFLIST}}[[\text{prog\_body}]] (m) (c_m) \]
\[ Y_{\text{PROG\_BODY\_CFLIST}}[[\text{exec\_part ;}]] \overset{\text{def}}{=} Y_{\text{EXEC\_PART\_CFLIST}}[[\text{exec\_part}]] \]
\[ Y_{\text{EXEC\_PART\_CFLIST}}[[\text{range}]] \overset{\text{def}}{=} Y_{\text{RANGE\_CFLIST}}[[\text{range}]] \]
\[ Y_{\text{RANGE\_CFLIST}}[[\text{stmt\_tail}]] \overset{\text{def}}{=} Y_{\text{STMT\_TAIL\_CFLIST}}[[\text{stmt\_tail}]] \]
\[ Y_{\text{STMT\_TAIL\_CFLIST}}[[\text{p\_stmt stmt\_tail}]] \overset{\text{def}}{=} Y_{\text{STMT\_CFLIST}}[[\text{p\_stmt}]] \]
\[ Y_{\text{STMT\_CFLIST}}[[\text{prefix stmt ;}]] \overset{\text{def}}{=} Y_{\text{STMT\_CFLIST}}[[\text{stmt}]] \]
\[ Y_{\text{STMT\_CFLIST}}[[\text{convience\_stmt}]] \overset{\text{def}}{=} Y_{\text{CONVIENCE\_STMT\_CFLIST}}[[\text{convience\_stmt}]] \]
\[ Y_{\text{CONVIENCE\_STMT\_CFLIST}}[[\text{expressions expr\_list}]] \overset{\text{def}}{=} Y_{\text{EXPR\_LIST\_CFLIST}}[[\text{expr\_list}]] \]
\[ Y_{\text{EXPR\_LIST\_CFLIST}}[[\text{exp}]] \overset{\text{def}}{=} Y_{\text{EXPR\_CFLIST}}[[\text{exp}]] \]
\[ Y_{\text{EXPR\_LIST\_CFLIST}}[[\text{expr\_list, exp}]] \overset{\text{def}}{=} \lambda m. \lambda c_m. \]
\[ \text{Expr\_exp}[[\text{exp}]] (m) \]
\[ \ (\lambda m. Y_{\text{EXPR\_LIST\_CFLIST}}[[\text{expr\_list}]] (m') (c_m) \)
\[
\]
\[ Y_{\text{EXPR\_CFLIST}}[[\text{op\_exp}]] \overset{\text{def}}{=} Y_{\text{OP\_EXPR\_CFLIST}}[[\text{op\_exp}]] \]
\[ Y_{\text{OP\_EXPR\_CFLIST}}[[\text{primary}]] \overset{\text{def}}{=} Y_{\text{PRIMARY\_CFLIST}}[[\text{primary}]] \]
\[ Y_{\text{PRIMARY\_CFLIST}}[[\text{entity}]] \overset{\text{def}}{=} Y_{\text{ENTITY\_CFLIST}}[[\text{entity}]] \]
\[ Y_{\text{ENTITY\_CFLIST}}[[\text{basic\_entity type\_info}]] \overset{\text{def}}{=} Y_{\text{BASIC\_ENTITY\_CFLIST}}[[\text{basic\_entity}]] \]
\[ Y_{\text{BASIC\_ENTITY\_CFLIST}}[[\text{var}]] \overset{\text{def}}{=} Y_{\text{VAR\_CFLIST}}[[\text{var}]] \]
\[ Y_{\text{VAR\_CFLIST}}[[\text{function\_definition}]] \overset{\text{def}}{=} Y_{\text{FUNCTION\_DEFINTION\_CFLIST}}[[\text{function\_definition}]] \]
\[ Y_{\text{FUNCTION\_DEFINTION\_CFLIST}}[[\text{functional\_declaration end}]] \overset{\text{def}}{=} Y_{\text{FUNCTION\_DECLARATION\_CFLIST}}[[\text{functional\_declaration}]] \]
\[ Y_{\text{FUNCTION\_DECLARATION\_CFLIST}}[[\text{function\_declaration}]] \overset{\text{def}}{=} Y_{\text{FUNCTION\_DECLARATION\_CFLIST}}[[\text{function\_declaration}]] \]
$$Y_{\text{FUNCTION\_DECLARATION}}[[\text{fun rule_list}]] \overset{\text{def}}{=} Y_{\text{RULE\_LIST}}[[\text{rule_list}]]$$
$$Y_{\text{RULE\_LIST}}[[\text{rule}]] \overset{\text{def}}{=} Y_{\text{RULE}}[[\text{rule}]]$$

$$Y_{\text{RULE}}[[\text{expr} \equiv \text{expr}_2]] \overset{\text{def}}{=} \lambda \text{m}. \lambda c. m.$$

$$c_m \ (\text{update\_store} \ (\text{r\_value\_location} \ (F_{\text{EXPR}}[[\text{expr}_1]]) \ (m)) \ )$$

\[
\text{inFunction} ((\lambda \text{m}. \ E_{\text{EXPR}}[[\text{expr}_2]]) \ \text{update\_model\_env}(\text{env}(m),m')))
\]

$$Y_{\text{PROG\_PART}}[[\text{p\_subprog\_stmt prog\_body}]] \overset{\text{def}}{=} \lambda \text{m}. \lambda c. m.$$

\[
\text{function\_type}_{\text{p\_SUBPROG_STMT}}[[\text{p\_subprog\_stmt}]] \ \\
\triangleright \ T_{\text{PROG\_BODY}}[[\text{prog\_body}]] \\
\text{type\_function}(\bot_{\text{error}}) \\
(\lambda f. c_m \ (\text{update\_store} \ (\text{r\_value\_location} \ (F_{\text{P\_SUBPROG\_STM}T}[[\text{p\_subprog\_stmt}]])) \ (m)) \\
\text{inFunction} \ (f) \ (m) \\
) \\
) \\
) \\
(\lambda m'. \lambda m''. \lambda k. \\
\text{isModel} \ (m'') \\
\triangleright \ (\text{C1\_PROG\_BODY}[[\text{prog\_body}]) \\
(\text{define\_l\_value\_location} \ (F_{\text{P\_SUBPROG\_STM}T}[[\text{p\_subprog\_stmt}]]) \\
\text{(update\_model\_env}(\text{env}(m'),m'')) \\
) \\
(\lambda m'''. k(\text{access\_store} \ (\text{l\_value\_location} \ (F_{\text{P\_SUBPROG\_STM}T}[[\text{p\_subprog\_stmt}]])) \\
(m) \\
) \\
(m''') \\
) \\
) \\
\square \bot_{\text{error}} /* too many arguments have been passed to this function */ \\
) \\
) \\
) \\
/* else the subprogram is a subroutine */ \\
\square T_{\text{PROG\_BODY}}[[\text{prog\_body}]] \\
\text{type\_function}(\bot_{\text{error}}) \\
(\lambda t. L_{\text{P\_SUBPROG\_STM}T}[[\text{p\_subprog\_stmt}]] \\
) \\
50
Section 6.3 The Semantics of Poly

\( (m) \)

\( (\lambda f. c_m \text{update\_store} \ (r_{\text{value\_location}} \ (F_{P_{\text{SUBPROG\_STMT}}[[p_{\text{subprog\_stmt}}]]}) \text{inSubroutine} (f) \ (m)) \)

\( (t) \)

Note that in the above denotation the function \( l_{\text{value\_location}} \) is given the model \( m \) and not the model \( m'' \). Also, define \( l_{\text{value\_location}} \) associates a location with the function id. This allows assignments to the function id only within the body of a function.

4. The valuation function \( T \) is responsible for creating a type\_function for the formal parameters of a subprogram or lambda expression. The type\_function that is returned is a function from identifiers to types. Formal parameters can be of type 1) cell, 2) integer, 3) real, 4) logical, 5) double precision 6) complex, 7) character and 8) error. A formal parameter introduced in a lambda expression will automatically be of type cell. A formal parameter of a subprogram will receive the type that is assigned to it in the subprogram body. If the formal parameter of the subprogram is not declared in the body of the subprogram, then its type will be \( \bot_{\text{error}} \). A program containing a formal parameter of type \( \bot_{\text{error}} \) is considered to be undefined. When the valuation function \( T \) is called by other valuation functions, it expects to be passed an initial or base type\_function. In the current definitions, the base type\_function that is passed is type\_function(\( \bot_{\text{error}} \)). The definition of type\_function is given in Section 6.3.6.

$$T_{\text{BOUND\_VARS}}[[\text{expr\_list } @]] \overset{\text{def}}{=} \lambda t_f. \lambda c_f. c_t(t_f)$$

/* return a type\_function that causes all formal parameters to be of type cell */

$$T_{\text{BOUND\_VARS}}[[@]] \overset{\text{def}}{=} \lambda t_f. \lambda c_f. c_t(t_f)$$

/* return initial type\_function which is equal to \( \bot_{\text{error}} \) */

$$T_{\text{PROG\_BODY}}[[\text{exec\_part}]] \overset{\text{def}}{=} T_{\text{EXEC\_PART}}[[\text{exec\_part}]]$$

$$T_{\text{EXEC\_PART}}[[\text{range}]] \overset{\text{def}}{=} T_{\text{RANGE}}[[\text{range}]]$$

$$T_{\text{RANGE}}[[\text{stmt\_tail}]] \overset{\text{def}}{=} T_{\text{STMT\_TAIL}}[[\text{stmt\_tail}]]$$

$$T_{\text{STMT\_TAIL}}[[\text{p\_end\_stmt}]] \overset{\text{def}}{=} \lambda t_f. \lambda c_f. c_t(t_f)$$

$$T_{\text{STMT\_TAIL}}[[\text{p\_stmt\_stmt\_tail}]] \overset{\text{def}}{=} T_{\text{P\_STMT}}[[\text{p\_stmt}]]$$

$$T_{\text{P\_STMT}}[[\text{prefix\_stmt} ;]] \overset{\text{def}}{=} T_{\text{STMT}}[[\text{stmt}]]$$

$$T_{\text{STMT}}[[\text{group\_stmt}]] \overset{\text{def}}{=} T_{\text{GROUP\_STMT}}[[\text{group\_stmt}]]$$

$$T_{\text{GROUP\_STMT}}[[\text{block\_clause\_range}]] \overset{\text{def}}{=} \lambda t_f. \lambda c_f. c_t(t_f) \text{ /* no declare clause exists */}$$

$$T_{\text{GROUP\_STMT}}[[\text{declare\_clause\_range}]] \overset{\text{def}}{=} T_{\text{2\_RANGE}}[[\text{range}]]$$

$$T_{\text{2\_RANGE}}[[\text{stmt\_tail}]] \overset{\text{def}}{=} T_{\text{2\_STMT\_TAIL}}[[\text{stmt\_tail}]]$$

$$T_{\text{2\_STMT\_TAIL}}[[\text{p\_end\_stmt}]] \overset{\text{def}}{=} \lambda t_f. \lambda c_f. c_t(t_f)$$

$$T_{\text{2\_STMT\_TAIL}}[[\text{p\_spec\_stmt\_stmt\_tail}]] \overset{\text{def}}{=} \lambda t_f. \lambda c_f.$
5. The valuation function \( L \) is responsible for constructing a mechanism allowing actual parameters to be passed to subprograms and lambda expressions. \( L \) accomplishes this by constructing a \( \lambda \)-arg binding for each of the formal parameters of the subprogram or lambda expression. In this model, an actual parameter is passed to a subprogram or lambda expression in a four-step process. In the first step, a new storage location, \( \alpha \), is passed to the \( \lambda \)-arg definition. Then an environment object corresponding to the type of the formal parameter is created using the information provided in the type function and the storage location \( \alpha \). This environment object is then bound to its corresponding formal parameter. This updated environment is the environment that is passed to the body of the subprogram or lambda expression. Lastly, the actual parameter is evaluated and its value is stored in the location \( \alpha \), and this updated store is passed to the body of the subprogram or lambda expression. The valuation function \( L \) is responsible for constructing the \( \lambda \)-arg bindings, for creating the
appropriate environment_object, and for binding this environment_object to the appropriate formal parameter in the environment. The valuation functions $L F$ and $LL$ were created to distinguish between arguments belonging to Poly-Fortran functions and arguments belonging to lambda abstractions.

$$L_{P \_S U B P R O G \_ S T M T}([\text{prefix} \subprog\_type \subprog\_head]) \overset{\text{def}}{=} L_{\text{SUBPROG\_HEAD}}([\subprog\_head])$$

$$L_{\text{SUBPROG\_HEAD}}[[\text{ident}]] \overset{\text{def}}{=} \lambda m. \lambda c_f. \lambda f. \lambda t. c_f(f(m))$$

$$L_{\text{SUBPROG\_HEAD}}[[\text{ident formals}]] \overset{\text{def}}{=} L_{\text{FORMALS}}[[\text{formals}]]$$

$$L_{\text{FORMALS}}[\langle \lambda \rangle] \overset{\text{def}}{=} \lambda m. \lambda c_f. \lambda f. \lambda t. c_f(f(m))$$

$$L_{\text{FORMALS}}[\langle \text{expr\_list} \rangle] \overset{\text{def}}{=} LF_{\text{EXPR\_LIST}}[[\text{expr\_list}]]$$

$$LF_{\text{EXPR\_LIST}}[[\text{expr}]] \overset{\text{def}}{=} \lambda m. \lambda c_f. \lambda f. \lambda t.$$
isAlpha(arg) \land (compatible_types(t(FEXPR[[expr]],type_of(v)))
\triangleright f(update_env (FEXPR[[expr]]))
\quad create_env_object1(t(FEXPR[[expr]],arg)
\quad m')
);
\triangleright /\* error occurred in function call */
⊥error

/* perform a partial test to see if the number of actual parameters is equal */
/* to the number of formal parameters, and */
/* check if actual parameter has the same type as the formal parameter */

isAlpha(arg) \land (compatible_types(t(FEXPR[[expr]],type_of(v)))
\triangleright (f(update_env (FEXPR[[expr]])
\quad create_env_object2(t(FEXPR[[expr]],arg)
\quad m))
)
);
\triangleright /\* error occurred in function call */
⊥error

LBOUND_VARS[[expr_list @]] \overset{\text{def}}{=} LLEXPR_LIST[[expr_list]]

 LLEXPR_LIST[[expr]] \overset{\text{def}}{=} \lambda m. \lambda c_f. \lambda f. \lambda t.

isIdentifier(FEXPR[[expr]]) /* a formal parameter must be an identifier */
\triangleright c_f (\lambda arg. \lambda v.
/* perform a partial test to see if the number of actual parameters is equal */
/* to the number of formal parameters, and */
/* check if actual parameter has the same type as the formal parameter */

isAlpha(arg) \land (compatible_types(t(FEXPR[[expr]],type_of(v)))
\triangleright (f(update_env (FEXPR[[expr]])
\quad create_env_object2(t(FEXPR[[expr]],arg)
\quad m))
)
);
\triangleright /\* error occurred in function call */
⊥error

LLEXPR_LIST[[expr_list, expr]] \overset{\text{def}}{=} \lambda m. \lambda c_f. \lambda f. \lambda t.

isIdentifier(FEXPR[[expr]]) /* a formal parameter must be an identifier */
\triangleright LLEXPR_LIST[[expr_list]]
\quad c_f
\quad (\lambda m'. \lambda arg. \lambda v.
\quad}
### The Semantics of Poly

/* perform a partial test to see if the number of actual parameters is equal */
/* to the number of formal parameters, and */
/* check if actual parameter has the same type as the formal parameter */

\[
\text{isAlpha}(\text{arg}) \land \text{compatible_types}(\text{type_of}(\text{expr}), \text{type_of}(\nu))
\]

\[
\text{f(update\_env}(\text{expr}), \text{create\_env\_object2}(\text{expr}, \text{arg}))
\]

\[
\text{f(update\_env}(\text{expr}), \text{create\_env\_object2}(\text{expr}, \text{arg}))
\]

\[
\text{\texttt{PENDING\_ARGS} def } \text{\texttt{ARGS}}
\]

**6.** The valuation function \( P \) is responsible for passing actual arguments to the bodies of subprograms and lambda expressions. In Poly, subprograms and lambda expressions have the same parameter passing semantics. Actual parameters are passed by value. Note, this is different from the parameter passing semantics implemented in FORTRAN 66. The valuation function \( P \) is responsible for evaluating the actual parameters that are passed to subprograms and lambda expression. For each actual argument, \( a_i \), to a subprogram or lambda expression the evaluation will produce a corresponding value, \( v_i \). For each value \( v_i \) that is obtained, a new storage location \( \alpha_i \) is generated, and the value \( v_i \) is stored in the location \( \alpha_i \).

It turns out that the semantics of Poly does not distinguish between subprogram calls of the form `foo(x,y)` and `foo(x)(y)`. This causes a slight inconvenience for our denotational semantic definitions. Note, the grammar productions in Poly cause `<pending args>` to be generated from **left to right** and `<expr list>`’s to be generated from **right to left**. This means that our denotational semantic definitions need to process both `<pending args>` and `<expr list>`’s in the proper order. Thus, when processing `<expr list>`’s we need to delay the object \( f \) from immediately receiving the value of the current `<expr>`. The reason for this is that we essentially need to reverse the order of the expressions in an `<expr list>` before we pass those values on to \( f \).

There are four cases we need to take into consideration when we are defining the interaction of `<pending args>` and `<expr list>`’s.

(a) We are evaluating the rightmost (i.e., the last) `<arg>`.
   I. We are evaluating an `<expr>` that is rightmost.
   II. We are evaluating an `<expr>` that is not rightmost.

(b) We are evaluating an `<arg>` that is not rightmost.
   I. We are evaluating an `<expr>` that is rightmost.
   II. We are evaluating an `<expr>` that is not rightmost.

Note that in the rightmost `<expr>` of the last (i.e., the rightmost) `<arg>` we need a slightly different behavior than we do for all of the other cases. To accomplish this we create two valuation functions \( P\text{A1} \) and \( P\text{A2} \) which allow us to distinguish between the two `<arg>` cases, and we also create two valuation functions \( P\text{E1} \) and \( P\text{E2} \) which allow us to distinguish between the case where `<expr list>` occurs in the last (rightmost) `<arg>` and the case where `<expr list>` occurs anywhere else. In addition to these four valuation functions, the valuation function \( P \) acts as a wrapper and sets everything up.

\[
P_{\text{PENDING\_ARGS}}[\text{args}] \triangleq P_{\text{A2\_ARGS}}[\text{args}]
\]

\[
P_{\text{PENDING\_ARGS}}[\text{args pending\_args}] \triangleq \lambda m. \lambda f. \lambda c_m.
\]
\[ PA1_{ARGS}[\text{args}] = \lambda m. \lambda f. p_{a1} (f, m) \]

\[ PE1_{EXPR\_LIST}[[\text{expr}]] = \lambda m. \lambda f. \lambda p_{a1}. \]

\[ P_{PENDING\_ARGS}[[\text{pending_args}]] \]
\[ \text{update_model_env}(\text{env}(m), m') \]
\[ f' \]
\[ c_m \]

\[ \square f' \text{ too many arguments have been passed to this subprogram */} \]

\[ \bot_{error} \]

\[ PA1_{ARGS}[[()]] = \lambda m. \lambda f. \lambda p_{a1}. p_{a1} (f, m) \]

\[ PA1_{ARGS}[[\text{expr_list}]] = \lambda m. \lambda f. \lambda p_{a1}. \]

\[ PE1_{EXPR\_LIST}[[\text{expr_list}]] = \lambda m. \lambda f. \lambda p_{a1}. \]

\[ E_{EXPR}[[\text{expr}]] = \lambda v. p_{a1} \]
\[ (f (\alpha (m)) (v), \text{update_store} (\alpha (m)), (v), (\text{update_model_alpha}(\alpha (m') + 1, m'))) \]

\[ PE1_{EXPR\_LIST}[[\text{expr_list}, \text{expr}]] = \lambda m. \lambda f. \lambda p_{a1}. \]

\[ PE1_{EXPR\_LIST}[[\text{expr_list}]] = \lambda m. \lambda f. \lambda p_{a1}. \]

\[ E_{EXPR}[[\text{expr}]] = \lambda v. p_{a1} \]
\[ (f' (\alpha (m')) (v), \text{update_store} (\alpha (m')), (v), (\text{update_model_alpha}(\alpha (m') + 1, m'))) \]
Section 6.3 The Semantics of Poly

/* too many arguments */

\[ \downarrow_{\text{error}} \]

\[ P_{\text{A1ARGS}}[()] \overset{\text{def}}{=} \lambda m. \lambda f. \lambda c_m. f(m)(c_m) \]

\[ P_{\text{A2ARGS}}[[\text{expr_list}]] \overset{\text{def}}{=} \lambda m. \lambda f. \lambda c_m. \]

\[ P_{\text{E2EXPR.LIST}}[[\text{expr_list}]] \overset{\text{def}}{=} \lambda m. \lambda f. \lambda c_m. \]

\[ E_{\text{EXPR}}[[\text{expr}]] \]

\[ m \]
\[ (\lambda v. f(\alpha(m))(v)) \quad \text{update_store} \quad (\alpha(m)) \]
\[ (v) \quad (\text{update_model_alpha}(\alpha(m) + 1,m)) \]
\[ (c_m) \]

\[ P_{\text{E2EXPR.LIST}}[[\text{expr_list}, \text{expr}]] \overset{\text{def}}{=} \lambda m. \lambda f. \lambda c_m. \]

\[ P_{\text{E1EXPR.LIST}}[[\text{expr_list}]] \]

\[ m \]

\[ f \]
\[ (\lambda (f', m').) \]

\[ f' \neq \downarrow_{\text{error}} \]

\[ E_{\text{EXPR}}[[\text{expr}]] \quad \text{update_model_env(env(m),m')} \]
\[ (\lambda v. f'(\alpha(m'))(v)) \quad \text{update_store} \quad (\alpha(m')) \]
\[ (v) \quad (\text{update_model_alpha}(\alpha(m') + 1,m')) \]
\[ (c_m) \]

/* too many arguments */

\[ \downarrow_{\text{error}} \]

7. The valuation functions \( C1, C2, C3, \) and \( C4 \) are used to define the semantics of subprogram bodies. A subprogram declaration basically consists of three components: 1) the subprogram head which contains the name, type (if applicable), and formal parameter list (if applicable) of the subprogram, 2) the declaration clause (if applicable) which declares the types of the formal parameters in 1), and 3) the body of the subprogram. It is not necessary for parameterless subprograms to have a declaration clause. However because of this it is necessary to delimit the body of subprograms in some fashion. In Poly, the bodies of subprograms are delimited by the block construct. Subprograms whose bodies are not delimited by the block construct, or subprograms that have constructs, other than the declaration clause mentioned in 2), outside of the delimiting block construct
are considered to be in error.

\[
C_1^{\text{PROG BODY}}[[\text{exec part}]] \triangleq C_1^{\text{EXEC PART}}[[\text{exec part}]]
\]
\[
C_1^{\text{EXEC PART}}[[\text{range}]] \triangleq C_1^{\text{RANGE}}[[\text{range}]]
\]
\[
C_1^{\text{RANGE}}[[\text{stmt tail}]] \triangleq C_1^{\text{STMT TAIL}}[[\text{stmt tail}]]
\]
\[
C_1^{\text{STMT TAIL}}[[\text{p stmt stmt tail}]] \triangleq \lambda m. \lambda c_m.
\]
\[
C_2^{\text{p stmt}}[[\text{p stmt}]]
\]
\[
m \quad (\lambda v. \lambda m').
\]
\[
\triangleright /\ast p\_stmt is the body of the subprogram \ast /
\]
\[
C_P^{\text{STMT}}[[\text{p stmt}]]
\]
\[
m' \quad (\lambda m''. C_3^{\text{STMT TAIL}}[[\text{stmt tail}]]
\]
\[
m'' \quad c_m
\)
\]
\[
\square /\ast p\_stmt is a declare clause \ast /
\]
\[
C_4^{\text{STMT TAIL}}[[\text{stmt tail}]] m' c_m
\]
\)
\[
C_2^{\text{p stmt}}[[\text{prefix stmt ;}]] \triangleq C_2^{\text{STMT}}[[\text{stmt}]]
\]
\[
C_2^{\text{STMT}}[[\text{group stmt}]] \triangleq C_2^{\text{GROUP STMT}}[[\text{group stmt}]]
\]
\[
C_2^{\text{GROUP STMT}}[[\text{declare clause range}]] \triangleq \lambda m. \lambda c_m. c_m(\text{false})(m)
\]
\[
C_2^{\text{GROUP STMT}}[[\text{block clause range}]] \triangleq \lambda m. \lambda c_m. c_m(\text{true})(m)
\]
\[
C_3^{\text{STMT TAIL}}[[\text{p end stmt}]] \triangleq \lambda m. \lambda c_m. c_m(m)
\]
\[
C_4^{\text{STMT TAIL}}[[\text{p stmt stmt tail}]] \triangleq \lambda m. \lambda c_m.
\]
\[
C_2^{\text{p stmt}}[[\text{p stmt}]]
\]
\[
m \quad (\lambda v. \lambda m').
\]
\[
\triangleright /\ast p\_stmt is the body of the subprogram \ast /
\]
\[
C_P^{\text{STMT}}[[\text{p stmt}]]
\]
\[
m' \quad (\lambda m''. C_3^{\text{STMT TAIL}}[[\text{stmt tail}]]
\]
\[
m'' \quad c_m
\)
\]
\[
\square /\ast error \ast /
\]
\[
\downarrow \text{error}
\]
8. The valuation function $C$ defines the semantics of statements and compound statements.

$$C_{\text{PROG BODY}}[\text{exec_part}] \equiv C_{\text{EXEC PART}}[\text{exec_part}]$$

$$C_{\text{EXEC PART}}[\text{range}] \equiv C_{\text{RANGE}}[\text{range}]$$

$$C_{\text{RANGE}}[\text{stmt_tail}] \equiv C_{\text{_STMT TAIL}}[\text{stmt_tail}]$$

$$C_{\text{_STMT TAIL}}[\text{p_end_stmt}] \equiv C_{\text{P END_STMT}}[\text{p_end_stmt}]$$

$$C_{\text{_STMT TAIL}}[\text{p_stmt stmt_tail}] \equiv \lambda m. \lambda c_m. \lambda m', C_{\text{_STMT TAIL}}[\text{stmt_tail}] m' c_m$$

$$C_{\text{_STMT TAIL}}[\text{p_spec_stmt stmt_tail}] \equiv \lambda m. \lambda c_m. \lambda m', C_{\text{_STMT TAIL}}[\text{stmt_tail}] m' c_m$$

$$C_{\text{P END_STMT}}[\text{prefix end}] \equiv C_{\text{END}}[\text{end}]$$

$$C_{\text{END}}[\text{end}] \equiv \lambda m. \lambda c_m. c_m(m)$$

$$C_{\text{_STMT}}[\text{prefix stmt ;}] \equiv C_{\text{_STMT}}[\text{stmt}]$$

$$C_{\text{_STMT}}[\text{group_stmt}] \equiv C_{\text{GROUP_STMT}}[\text{group_stmt}]$$

$$C_{\text{_STMT}}[\text{if_stmt}] \equiv C_{\text{IF_STMT}}[\text{if_stmt}]$$

$$C_{\text{_STMT}}[\text{basic stmt}] \equiv C_{\text{BASIC_STMT}}[\text{basic stmt}]$$

$$C_{\text{_STMT}}[\text{if_head basic stmt}] \equiv \lambda m. \lambda c_m. \lambda v. \lambda m'. C_{\text{BASIC_STMT}}[\text{basic stmt}] m c_m;$$

$$\left( \lambda m'. \lambda c_m. C_{\text{_STMT TAIL}}[\text{stmt_tail}] m c_m \right)$$

$$\left( \lambda m'. \lambda c_m. \text{update_model_store}(\text{store}(m'), m) \right)$$

$$C_{\text{GROUP_STMT}}[\text{declare_clause range}] \equiv D_{\text{RANGE}}[\text{range}]$$

$$C_{\text{GROUP_STMT}}[\text{block_clause range}] \equiv \lambda m. \lambda c_m. C_{\text{RANGE}}[\text{range}] m (\lambda m'. c_m(\text{update_model_store}(\text{store}(m'), m)))$$

$$C_{\text{GROUP_STMT}}[\text{loop_clause range}] \equiv \lambda m. \lambda c_m.$$
Chapter 6  The Formal Semantics of Individual Transformations

\[ H_{\text{LOOP_CLAUSE}}[[\text{range}]] \quad C_{\text{RANGE}}[[\text{range}]] \]
\[ m \quad (\lambda m'. c_m(\text{update_model_store}(\text{store}(m'), m))) \]

\[ C_{\text{IF_STMT}}[[\text{if clause range}]] \stackrel{\text{def}}{=} \lambda m. \lambda c_m. \]
\[ E_{\text{IF_CLAUSE}}[[\text{if clause}]] \]
\[ m \quad (\lambda v. \]
\[ \text{isBoolean} \quad v \]
\[ \quad \triangleright v \]
\[ \quad \triangleright C_{\text{RANGE}}[[\text{range}]] \quad m \quad c_m; \]
\[ \quad \Box c_m(m); \]
\[ \quad \Box /\* a conditional branch requires a value of type boolean */ \]
\[ \quad \perp_{\text{error}} \]
\[ ) \]
\[ C_{\text{IF_STMT}}[[\text{if clause range}_1 \text{ else range}_2]] \stackrel{\text{def}}{=} \lambda m. \lambda c_m. \]
\[ E_{\text{IF_CLAUSE}}[[\text{if clause}]] \]
\[ m \quad (\lambda v. \]
\[ \text{isBoolean} \quad v \]
\[ \quad \triangleright v \]
\[ \quad \triangleright C_{\text{RANGE}}[[\text{range}_1]] \quad m \quad c_m; \]
\[ \quad \Box C_{\text{RANGE}}[[\text{range}_2]] \quad m \quad c_m; \]
\[ \quad \Box /\* a conditional branch requires a value of type boolean */ \]
\[ \quad \perp_{\text{error}} \]
\[ ) \]
\[ C_{\text{BASIC_STMT}}[[\text{noref basic stmt}]] \stackrel{\text{def}}{=} C_{\text{NOREF_BASIC_STMT}}[[\text{noref basic stmt}]] \]
\[ C_{\text{NOREF_BASIC_STMT}}[[\text{assignment}]] \stackrel{\text{def}}{=} C_{\text{ASSIGNMENT}}[[\text{assignment}]] \]
\[ C_{\text{ASSIGNMENT}}[[\text{var} = \text{expr}]] \stackrel{\text{def}}{=} \lambda m. \lambda c_m. \]
\[ E_{\text{EXPR}}[[\text{expr}]] \quad m \quad (\lambda v. M_{\text{VAR_MOD}}[[\text{var}]] \quad v \quad m \quad c_m) \]

9. The valuation function \( H \).
\[ H_{\text{LOOP_CLAUSE}}[[\text{dowhile selector}]] \stackrel{\text{def}}{=} \lambda f. \lambda m. \lambda c_m. \]
\[ \text{fix} (\lambda c_m'. E_{\text{SELECTOR}}[[\text{selector}]] \quad m \quad (\lambda v. \]
\[ \text{isBoolean} \quad v \]
\[ \quad \triangleright v \]

60
\[ f(m)(\epsilon_m); \]
\[ \Box c_m(m'); \]
\[ \Box \text{/* the selector of a while loop must be a boolean value */} \]

\[ H_{\text{LOOP_CLAUSE}}[\text{do iter_clause ;}] \triangleq H_{\text{ITER_CLAUSE}}[\text{iter_clause}] \]
\[ H_{\text{ITER_CLAUSE}}[\text{var = iter_spec}] \triangleq \lambda f. \lambda m. \lambda c_m. \]
\[ I_{\text{ITER_SPEC}}[[\text{range}]] \]
\[ f \]
\[ r_{\text{value_location}}(F_{\text{VAR}}[[\text{var}]])(m) \]
\[ m \]
\[ c_m \]

10. The valuation function \( I \).
\[ I_{\text{ITER_SPEC}}[[\text{expr_list}]] \triangleq I_{\text{EXPR_LIST}}[[\text{expr_list}]] \]
\[ I_{\text{EXPR_LIST}}[[\text{expr_list}, \text{expr}]] \triangleq \lambda f. \lambda \alpha_1. \lambda m. \lambda c_m. \]
\[ E_{\text{EXPR_LIST}}[[\text{expr_list}]] \]
\[ m \]
\[ (\lambda v. E_{\text{EXPR}}[[\text{expr}]], \text{update_store}(\alpha_1)(v)(m), \text{fix} (\lambda \epsilon_m', (\lambda m_1. \text{access_store}(\alpha_1)(m_1) \leq v')) \]
\[ \Box f \]
\[ m \]
\[ (\lambda m_2, \epsilon'_m, (\text{update_store}(\alpha_1)(m_2) + 1)) \]
\[ (m_2); \]
\[ \Box c_m(m_1) \]

11. The valuation function \( M \) is responsible for assignments.
\[ M_{\text{VAR_MOD}}[[\text{function_expr}]] \triangleq M_{\text{FUNCTION_EXPR_MOD}}[[\text{function_expr}]] \]
\[ M_{\text{FUNCTION_EXPR_MOD}}[[\text{ident}]] \triangleq \lambda v. \lambda m. \lambda c_m. \]
\[ (l_{\text{value_location}}(F_{\text{IDENT}}[[\text{ident}]])) \neq \bot_{\text{error}} \]
\[ \land (\text{compatible_types}(\text{get_evaluation_type}(F_{\text{IDENT}}[[\text{ident}]], m), \text{type_of}(v)) \]
\[ \Box c_m \]
\[ \text{update_store} (l_{\text{value_location}}(F_{\text{IDENT}}[[\text{ident}]]) \]
\[ (m) \]
\[ v \]
\[ m \]
\[ \Box \text{/* this identifier is 1) not declared, 2) cannot be assigned to, or 3) is of an incompatible type */} \]

\[ \bot_{\text{error}} \]
12. The valuation function \( D \) defines the semantics of declaration statements.

\[
D_{\text{RANGE}}(\text{[stmt\_tail]}]) \equiv D_{\text{STMT\_TAIL}}(\text{[stmt\_tail]}])
\]

\[
D_{\text{STMT\_TAIL}}(\text{[p\_end\_stmt]}]) \equiv \lambda m. \lambda c_m. c_m(m)
\]

\[
D_{\text{STMT\_TAIL}}(\text{[p\_stmt\_stmt\_tail]}]) \equiv \perp_{\text{error}}
\]

/* nondeclaration statements may not occur within a RANGE of declarations. */

\[
D_{\text{STMT\_TAIL}}(\text{[p\_spec\_stmt\_stmt\_tail]}]) \equiv \lambda m. \lambda c_m. (m)
D_{\text{P\_SPEC\_STMT}}(\text{[p\_spec\_stmt]}]) \equiv \lambda m'. D_{\text{STMT\_TAIL}}(\text{[stmt\_tail]}]) (m')
D_{\text{P\_SPEC\_STMT}}(\text{[prefix spec\_stmt ;]}]) \equiv D_{\text{SPEC\_STMT}}(\text{[spec\_stmt]}])
D_{\text{SPEC\_STMT}}(\text{[spec]}]) \equiv D_{\text{SPEC}}(\text{[spec]}])
D_{\text{SPEC}}(\text{[type spec]}]) \equiv D_{\text{TYPE\_SPEC}}(\text{[type spec]}])
D_{\text{TYPE\_SPEC}}(\text{[standard\_type\ expr\_list]}]) \equiv \lambda m. \lambda c_m.
D_{\text{1\_EXPR\_LIST}}(\text{[expr\_list]}]) \equiv
m
F_{\text{STANDARD\_TYPE}}(\text{[standard\_type]}]
D_{\text{1\_EXPR\_LIST}}(\text{[expr]}]) \equiv
m
isIdentifier (F_{\text{EXPR}}(\text{[expr]}]))
\]

\[
D_{\text{1\_EXPR\_LIST}}(\text{[expr\_list, expr]}]) \equiv \lambda m. \lambda t. \lambda c_m.
F_{\text{EXPR}}(\text{[expr]}])
\]

/* Note that the assumption being made here is that all identifiers that arise through declaration statements have their id\_type equal to variable and their evaluation\_type equal to the type specified in the declaration statement (i.e., type = t). Also, these identifiers will have their r\_value\_location equal to their l\_value\_location. This implies that one should not be able to create objects like functions and subroutines in this kind of a declaration statement. */

Section 6.3 The Semantics of Poly

\[ D_{\text{EXPR_LIST}}[\text{expr_list}] = \]
\[
\text{update_env} ( F_{\text{EXPR}}[\text{expr}])
\]
\[
\text{create_env_object1}(t, \alpha(m))
\]
\[
\text{update_model}_\alpha(\alpha(m) + 1, m)
\]

\[ \square \]

13. The valuation function \( E \) defines the semantics of expressions.

\[ E_{\text{IF_CLAUSE}}[[\text{and_if_head then :}]] \overset{\text{def}}{=} E_{\text{AND_IF_HEAD}}[[\text{and_if_head}]] \]
\[ E_{\text{AND_IF_HEAD}}[[\text{if_head}]] \overset{\text{def}}{=} E_{\text{IF_HEAD}}[[\text{if_head}]] \]
\[ E_{\text{IF_HEAD}}[[\text{if selector}]] \overset{\text{def}}{=} E_{\text{SELECTOR}}[[\text{selector}]] \]
\[ E_{\text{SELECTOR}}[[\text{expr_list}]] \overset{\text{def}}{=} E_{\text{EXPR_LIST}}[[\text{expr_list}]] \]
\[ E_{\text{EXPR_LIST}}[[\text{expr}]] \overset{\text{def}}{=} E_{\text{EXPR}}[[\text{expr}]] \]
\[ E_{\text{EXPR_LIST}}[[\text{expr_list}, \text{expr}]] \overset{\text{def}}{=} \text{not defined at present} \]
\[ E_{\text{EXPR}}[[\text{op_expr}]] \overset{\text{def}}{=} E_{\text{OP_EXPR}}[[\text{op_expr}]] \]
\[ E_{\text{OP_EXPR}}[[\text{op_expr op primary}]] \overset{\text{def}}{=} \lambda m. \lambda k. \]
\[ E_{\text{OP_EXPR}}[[\text{op_expr}]] \]
\[ m \]
\[ (\lambda v. O_{\text{OP}}[[\text{op}]] \]
\[ \lambda f_v. E_{\text{PRIMAR}}[[\text{primary}]] \]
\[ m \]
\[ (\lambda v'. (f_v(v') \neq \bot_{\text{error}})) \]
\[ \lambda k(f_v(v')); \]
\[ \square \]

13. The valuation function \( E \) defines the semantics of expressions.

\[ E_{\text{IF_CLAUSE}}[[\text{and_if_head then :}]] \overset{\text{def}}{=} E_{\text{AND_IF_HEAD}}[[\text{and_if_head}]] \]
\[ E_{\text{AND_IF_HEAD}}[[\text{if_head}]] \overset{\text{def}}{=} E_{\text{IF_HEAD}}[[\text{if_head}]] \]
\[ E_{\text{IF_HEAD}}[[\text{if selector}]] \overset{\text{def}}{=} E_{\text{SELECTOR}}[[\text{selector}]] \]
\[ E_{\text{SELECTOR}}[[\text{expr_list}]] \overset{\text{def}}{=} E_{\text{EXPR_LIST}}[[\text{expr_list}]] \]
\[ E_{\text{EXPR_LIST}}[[\text{expr}]] \overset{\text{def}}{=} E_{\text{EXPR}}[[\text{expr}]] \]
\[ E_{\text{EXPR_LIST}}[[\text{expr_list}, \text{expr}]] \overset{\text{def}}{=} \text{not defined at present} \]
\[ E_{\text{EXPR}}[[\text{op_expr}]] \overset{\text{def}}{=} E_{\text{OP_EXPR}}[[\text{op_expr}]] \]
\[ E_{\text{OP_EXPR}}[[\text{op_expr op primary}]] \overset{\text{def}}{=} \lambda m. \lambda k. \]
\[ E_{\text{OP_EXPR}}[[\text{op_expr}]] \]
\[ m \]
\[ (\lambda v. O_{\text{OP}}[[\text{op}]] \]
\[ \lambda f_v. E_{\text{PRIMAR}}[[\text{primary}]] \]
\[ m \]
\[ (\lambda v'. (f_v(v') \neq \bot_{\text{error}})) \]
\[ \lambda k(f_v(v')); \]
\[ \square \]

13. The valuation function \( E \) defines the semantics of expressions.

\[ E_{\text{IF_CLAUSE}}[[\text{and_if_head then :}]] \overset{\text{def}}{=} E_{\text{AND_IF_HEAD}}[[\text{and_if_head}]] \]
\[ E_{\text{AND_IF_HEAD}}[[\text{if_head}]] \overset{\text{def}}{=} E_{\text{IF_HEAD}}[[\text{if_head}]] \]
\[ E_{\text{IF_HEAD}}[[\text{if selector}]] \overset{\text{def}}{=} E_{\text{SELECTOR}}[[\text{selector}]] \]
\[ E_{\text{SELECTOR}}[[\text{expr_list}]] \overset{\text{def}}{=} E_{\text{EXPR_LIST}}[[\text{expr_list}]] \]
\[ E_{\text{EXPR_LIST}}[[\text{expr}]] \overset{\text{def}}{=} E_{\text{EXPR}}[[\text{expr}]] \]
\[ E_{\text{EXPR_LIST}}[[\text{expr_list}, \text{expr}]] \overset{\text{def}}{=} \text{not defined at present} \]
\[ E_{\text{EXPR}}[[\text{op_expr}]] \overset{\text{def}}{=} E_{\text{OP_EXPR}}[[\text{op_expr}]] \]
\[ E_{\text{OP_EXPR}}[[\text{op_expr op primary}]] \overset{\text{def}}{=} \lambda m. \lambda k. \]
\[ E_{\text{OP_EXPR}}[[\text{op_expr}]] \]
\[ m \]
\[ (\lambda v. O_{\text{OP}}[[\text{op}]] \]
\[ \lambda f_v. E_{\text{PRIMAR}}[[\text{primary}]] \]
\[ m \]
\[ (\lambda v'. (f_v(v') \neq \bot_{\text{error}})) \]
\[ \lambda k(f_v(v')); \]
\[ \square \]

13. The valuation function \( E \) defines the semantics of expressions.

\[ E_{\text{IF_CLAUSE}}[[\text{and_if_head then :}]] \overset{\text{def}}{=} E_{\text{AND_IF_HEAD}}[[\text{and_if_head}]] \]
\[ E_{\text{AND_IF_HEAD}}[[\text{if_head}]] \overset{\text{def}}{=} E_{\text{IF_HEAD}}[[\text{if_head}]] \]
\[ E_{\text{IF_HEAD}}[[\text{if selector}]] \overset{\text{def}}{=} E_{\text{SELECTOR}}[[\text{selector}]] \]
\[ E_{\text{SELECTOR}}[[\text{expr_list}]] \overset{\text{def}}{=} E_{\text{EXPR_LIST}}[[\text{expr_list}]] \]
\[ E_{\text{EXPR_LIST}}[[\text{expr}]] \overset{\text{def}}{=} E_{\text{EXPR}}[[\text{expr}]] \]
\[ E_{\text{EXPR_LIST}}[[\text{expr_list}, \text{expr}]] \overset{\text{def}}{=} \text{not defined at present} \]
\[ E_{\text{EXPR}}[[\text{op_expr}]] \overset{\text{def}}{=} E_{\text{OP_EXPR}}[[\text{op_expr}]] \]
\[ E_{\text{OP_EXPR}}[[\text{op_expr op primary}]] \overset{\text{def}}{=} \lambda m. \lambda k. \]
\[ E_{\text{OP_EXPR}}[[\text{op_expr}]] \]
\[ m \]
\[ (\lambda v. O_{\text{OP}}[[\text{op}]] \]
\[ \lambda f_v. E_{\text{PRIMAR}}[[\text{primary}]] \]
\[ m \]
\[ (\lambda v'. (f_v(v') \neq \bot_{\text{error}})) \]
\[ \lambda k(f_v(v')); \]
\[ \square \]
Chapter 6  The Formal Semantics of Individual Transformations

/* a function_definition may occur only as part of a convenience statement, and in which case the valuation functions
   Z and Y will be responsible for defining the semantics of the function_definition. */

\[
E_{\text{VAR}}[[\text{function_expr}]] \defeq E_{\text{FUNCTION_EXPR}}[[\text{function_expr}]]
\]

\[
E_{\text{FUNCTION_APP}}[[\text{function_expr pending_args}]] \defeq \lambda m. \lambda k. E_{\text{FUNCTION_EXPR}}[[\text{function_expr}]]
\]

\[
\begin{align*}
E_{\text{FUNCTION_EXPR}}[[\text{ident}]] & \defeq \lambda m. \lambda c_f. \\
\text{r_value_location}(F_{\text{IDENT}}[[\text{ident}]]){m} & \neq \perp_{\text{error}} \\
\triangleright /* note the object dereferenced may be 1) a function, 2) a subroutine, or 3) a standard type */ \\
\text{isFunction} & (\text{access_store}(\text{r_value_location}(F_{\text{IDENT}}[[\text{ident}]]){m})) \\
\triangleright c_f & (\text{access_store} (\text{r_value_location}(F_{\text{IDENT}}[[\text{ident}]]) (m)) \\
\text{);} & \\
\triangleright /* an identifier of type function was required at this point */ \\
\perp_{\text{error}} \\
\text{;} & \\
\triangleright /* this identifier has an undefined r-value_location (e.g., it has not be declared) */ \\
\perp_{\text{error}} \\
\end{align*}
\]

\[
E_{\text{FUNCTION_EXPR}}[[\text{lambda_abstraction}]] \defeq E_{\text{LAMBDA_ABSTRACTION}}[[\text{lambda_abstraction}]]
\]

\[
E_{\text{LAMBDA_ABSTRACTION}}[[\text{lambda body end}]] \defeq E_{\text{BODY}}[[\text{body}]]
\]

\[
E_{\text{BODY}}[[\text{bound_vars expr_list}]] \defeq \lambda m. \lambda c_f. \\
T_{\text{BOUND_VARS}}[[\text{bound_vars}]] type_function(\perp_{\text{error}}) \\
(\lambda t. T_{\text{BOUND_VARS}}[[\text{bound_vars}]] \\
(\lambda m'. /* supplies the environment */ \\
(\lambda m''. /* supplies the store */ \\
(\lambda k. E_{\text{EXPR_LIST}}[[\text{expr_list}]] update_model_env(env(m'),m'') \\
\text{;} \\
\text{;} \\
\text{;} \\
\text{)} \\
\text{)} \\
\text{)} \\
\text{)} \\
\text{)} \\
\end{align*}
\]

\[
E_{\text{FUNCTION_EXPR}}[[\text{ident}]] \defeq \lambda m. \lambda k. \\
\text{r_value_location}(F_{\text{IDENT}}[[\text{ident}]]){m} & \neq \perp_{\text{error}}
\]
Section 6.3  The Semantics of Poly

/* note the object dereferenced may be 1) a function, 2) a subroutine, or 3) a standard type */

k  (access_store (r_value_location(FIDENT[[ident]]) (m))

);  

/* the identifier cannot be dereferenced */

⊥_error

E_FUNCTION_EXPR[[lambda_abstraction]] def E_LAMBDA_ABSTRACTION[[lambda_abstraction]]
E_LAMBDA_ABSTRACTION[[lambda body end]] def E_BODY[[body]]

E_BODY[[bound_vars expr_list]] def λ m. λ k.
  T_BOUND_VARS[[bound_vars]]
  type_function(⊥_error)
  (λ t. L_BOUND_VARS[[bound_vars]]
   m
   (λ f. k (inFunction (f))) /* the resulting object is of type function */
   (λ m'. /* supplies the environment */
    (λ m''. /* supplies the store */
     (λ k'.
      E_EXPR_LIST[[expr_list]]
      update_model_env(env (m'), m'')
     )
    )
   )
  )
)
)

E_FUNCTION_EXPR[[cond_expr]] def E_COND_EXPR[[cond_expr]]
E_FUNCTION_EXPR[[ ( expr_list )]] def E_EXPR_LIST[[expr_list]]

E_COND_EXPR[[use expr_list1 if_head otherwise expr_list2 end]] def λ m. λ k.

E_IF_HEAD[[if_head]]
  m
  (λ v.
   isBoolean (v)
   ▷
    v
    ▷ E_EXPR_LIST[[expr_list1]] m k;
    □ E_EXPR_LIST[[expr_list2]] m k;
    □ /* a conditional expression requires a value of type boolean */
    ⊥_error
  )

E_IF_HEAD[[if selector]] def E_SELECTOR[[selector]]
E_SELECTOR[[ ( expr )]] def E_EXPR[[expr]]
14. The valuation function \( O \). Note that no type checking between \( v \) and \( v' \) is performed within the denotational semantic definitions. It is assumed that arithmetic operations in the mathematical foundation will produce an \( \bot_{\text{error}} \) value whenever they are given two values having incompatible types. Usually, type checking of this sort is done within the denotational semantic definitions themselves. This makes our approach somewhat of a shortcut. However, since the goal of this research is to construct a methodology capable of proving the correctness of program transformations, and not to construct a production quality denotational semantics of Poly, we feel that this shortcut is justified. Also, from a proof of correctness point of view, if an \( \bot_{\text{error}} \) value is generated, it does not matter where the \( \bot_{\text{error}} \) value is generated; all that matters is whether an \( \bot_{\text{error}} \) value is generated.

\[
O_{\text{OP}}[[\text{add}_\text{op}]] \triangleq O_{\text{ADD}_\text{OP}}[[\text{add}_\text{op}]]
\]

\[
O_{\text{OP}}[[\text{mult}_\text{op}]] \triangleq O_{\text{MULT}_\text{OP}}[[\text{mult}_\text{op}]]
\]

\[
O_{\text{ADD}_\text{OP}}[[\text{+}]] \triangleq \lambda v. \lambda k_{f_x}. k_{f_y}. (\lambda v'. \text{value}_0f(v) + \text{value}_0f(v'))
\]

\[
O_{\text{ADD}_\text{OP}}[[\text{-}]] \triangleq \lambda v. \lambda k_{f_x}. k_{f_y}. (\lambda v'. \text{value}_0f(v) - \text{value}_0f(v'))
\]

\[
O_{\text{MULT}_\text{OP}}[[\text{*}]] \triangleq \lambda v. \lambda k_{f_x}. k_{f_y}. (\lambda v'. \text{value}_0f(v)*\text{value}_0f(v'))
\]

\[
O_{\text{MULT}_\text{OP}}[[\text{/}]] \triangleq \lambda v. \lambda k_{f_x}. k_{f_y}. (\lambda v'. \text{value}_0f(v)/\text{value}_0f(v'))
\]

\[
O_{\text{MULT}_\text{OP}}[[\text{div}]] \triangleq \lambda v. \lambda k_{f_x}. k_{f_y}. (\lambda v'. \text{value}_0f(v) \text{ div} \text{ value}_0f(v'))
\]

\[
O_{\text{MULT}_\text{OP}}[[\text{mod}]] \triangleq \lambda v. \lambda k_{f_x}. k_{f_y}. (\lambda v'. \text{value}_0f(v) \text{ mod} \text{ value}_0f(v'))
\]

15. The valuation function \( F \).

\[
F_{\text{SUBPROG}_\text{STMT}}[[\text{prefix subprog}_\text{type} \text{ subprog}_\text{head} ;]] \triangleq F_{\text{SUBPROG}_\text{HEAD}}[[\text{subprog}_\text{head}]]
\]

\[
F_{\text{SUBPROG}_\text{HEAD}}[[\text{ident}]] \triangleq F_{\text{IDENT}}[[\text{ident}]]
\]

\[
F_{\text{SUBPROG}_\text{HEAD}}[[\text{ident} \text{ formals}]] \triangleq F_{\text{IDENT}}[[\text{ident}]]
\]

\[
F_{\text{EXPR}_\text{LIST}}[[\text{expr}]] \triangleq F_{\text{EXPR}}[[\text{expr}]]
\]

\[
F_{\text{EXPR}}[[\text{op}_\text{expr}]] \triangleq F_{\text{OP}_\text{EXPR}}[[\text{op}_\text{expr}]]
\]

\[
F_{\text{OP}_\text{EXPR}}[[\text{primary}]] \triangleq F_{\text{PRIMARY}}[[\text{primary}]]
\]

\[
F_{\text{PRIM}_{\text{ARY}}}[[\text{entity}]] \triangleq F_{\text{ENTITY}}[[\text{entity}]]
\]

\[
F_{\text{ENTITY}}[[\text{basic} \text{ entity} \text{ type info}]] \triangleq F_{\text{BASIC}_\text{ENTITY}}[[\text{basic} \text{ entity}]]
\]

\[
F_{\text{BASIC}_\text{ENTITY}}[[\text{var}]] \triangleq F_{\text{VAR}}[[\text{var}]]
\]

\[
F_{\text{VAR}}[[\text{function} \text{ expr}]] \triangleq F_{\text{FUNCTION}_\text{EXPR}}[[\text{function} \text{ expr}]]
\]

\[
F_{\text{FUNCTION}_\text{EXPR}}[[\text{ident}]] \triangleq F_{\text{IDENT}}[[\text{ident}]]
\]

\[
F_{\text{IDENT}}[[\text{id}]] \triangleq \text{identifier} (\text{id})
\]

\[
F_{\text{STANDARD}_\text{TYPE}}[[\text{integer}]] \triangleq \text{integer}
\]

\[
F_{\text{FUNCTION}_\text{EXPR}}[[\text{cell}]] \triangleq \text{cell}
\]

\[
F_{\text{CONST}}[[\text{i} \text{ const}]] \triangleq \text{inInteger} (F_{\text{CONST}}[[\text{i} \text{ const}]])
\]

\[
F_{\text{CONST}}[[\text{f} \text{ const}]] \triangleq \text{inReal} (F_{\text{CONST}}[[\text{f} \text{ const}]])
\]

\[
F_{\text{CONST}}[[\text{id}]] \triangleq \text{i rep}
\]

\[
F_{\text{F Const}}[[\text{id}]] \triangleq \text{f rep}
\]

16. The valuation function \( \text{function}_\text{type} \).

\[
\text{function}_\text{type}_{\text{P SUBPROG}_\text{STMT}}[[\text{prefix subprog}_\text{type} \text{ subprog}_\text{head} ;]] \triangleq \text{function}_\text{type}_{\text{SUBPROG}_\text{TYPE}}[[\text{subprog}_\text{type}]]
\]

\[
\text{function}_\text{type}_{\text{SUBPROG}_\text{TYPE}}[[\text{sfb}]] \triangleq \text{function}_\text{type}_{\text{SFB}}[[\text{sfb}]]
\]
In Chapter 5 we defined a completed_phrase as a string of terminal symbols derived from a single nonterminal symbol. Clearly, any Poly program is a completed_phrase derived from the start symbol of the Poly grammar. Similarly, in Poly the expression $y + 5$ is a completed_phrase that can be derived from the nonterminal $<expr>$. (See Appendix B for a subset of the Poly grammar.) We denote such a completed_phrase with the schema $<symbol>$

The denotational semantic definitions in Section 6.3.7 are traditionally used to determine the semantics of completed_phrases that are programs. It is not entirely obvious how the definitions in Section 6.3.7 can be used to determine the semantics of completed_phrases that are not programs. For example, what is the denotational meaning of the pattern $<symbol>$

The point we are making here is that completed_phrases, derived from a nonterminal symbol that is not the start symbol of the grammar, may have more than one meaning with respect to a set of denotational definitions. Because of this, one has to be very careful when determining the semantics of such completed_phrases.

In summary, the semantics of any completed_phrase must be defined as the set of the semantics assigned to it by
all of the valuation functions which are capable of defining it (i.e., all the valuation functions that accept the SDT of the completed_phrase as input). A consequence of this is that in order to prove that a transformation of the form:

\[ T \xrightarrow{\text{def}} <a> \{ \alpha \} \]

\[ \Rightarrow \]

\[ <a> \{ \alpha' \} \]

\[ \text{sc.} \]

is correctness preserving, one needs to show that the meaning of the input schema, \(<a> \{ \alpha \}\), is less-defined than the corresponding meaning of the output schema, \(<a> \{ \alpha' \}\), for all of the valuation functions that are capable of defining a semantics for \(<a>\). In this thesis we define this particular use of the word *capable* as follows:

- **capable** – A valuation function having the form \(V_y\) is *capable* of defining a semantics of a nonterminal symbol \(b\) iff the subscript of the valuation function (i.e., \(b'\)) is similar to \(b\) (e.g., upper case symbols instead of lower case, and underscores instead of blanks). For example, in the denotational definition of Poly, the valuation function \(Y_{\text{EXPR\_LIST}}\) is capable of defining the semantics of the nonterminal symbol \(<expr\ list\>\) because EXPR\_LIST, the subscript of \(Y_{\text{EXPR\_LIST}}\), is similar to the nonterminal symbol \(<expr\ list\>\).

Usually, there will only be a single valuation function capable of defining a schema. However, one needs to be aware that cases can exist where the semantics of several valuation functions need to be considered.

In summary, the denotational definitions in Section 6.3.7 when used with care are able to determine the semantics of schemas of the form \(<a> \{ \alpha \}\) where \(\alpha\) is a completed_phrase. Unfortunately, TAMPR transformation schemas generally have the form \(<b> \{ \beta \}\) where \(\beta\) is just a phrase (i.e., not a completed_phrase). Recall, a phrase may contain schema variables. In the next section we explore how the traditional denotational definitions can be extended in a manner that enables us to formally define the semantics of general schemas of the form \(<b> \{ \beta \}\).

### 6.5 Delta-Functions and the Semantics of Static Patterns

Traditional denotational semantics (i.e., the semantic definitions given in Section 6.3.7) are not capable of defining the semantics of schemas containing schema variables. This is because traditional denotational semantic definitions are constructed to define the semantics of SDT's whose leaves consist solely of terminal symbols. TAMPR transformation schemas, in contrast, can contain nonterminal symbols as leaf elements (i.e., schema variables) and therefore cannot be given a formal semantics by traditional denotational methods. One of the contributions of this research has been to extend, through *delta* functions, the denotational semantic paradigm to allow the semantics of schema variables to be defined. This extension amounts to using the existing denotational semantic definitions to define *delta* functions that are capable of defining the semantics of schema variables. In this section, we informally discuss the role played by *delta* functions in our proof methodology as well as giving an informal definition of how the denotational semantic definitions of a language can themselves be used to define semantics of *delta* functions.

The semantics of a schema variable (i.e., nonterminal symbol), \(s\), can be viewed as the set of the semantic expressions derivable by \(s\) with respect to the grammar one is considering (in our case the grammar is Poly). For example, suppose we have a grammar containing the productions of the form \(n \rightarrow \text{number}\), where \(\text{number}\) is a terminal symbol belonging to the set of integers \(I\). For example, the actual grammar might be something like \(n \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid \text{maxint}\). Also, suppose that the nonterminal symbol \(n\) does not appear on the left hand side of any other production in the grammar. Given the schema variable \(n\) the question is, “what semantics should be assigned to the nonterminal symbol \(n\)?” Without any information on what kind of integer \(n\) might be we would have to conclude that “\(n\) could be any one of the elements in between \(0\) and \(\text{maxint}\).” In this example, any kind of reasoning about the schema variable \(n\) must hold for all numbers between \(0\) and \(\text{maxint}\). For example, suppose we would like to prove that the expression \(\frac{2}{3} n\) has a value that is defined. Proving that \(\frac{2}{3} n\) is defined amounts to proving \(\forall \alpha \in I: \frac{2}{3} \alpha\) is defined.

Let \(<t_1>\{\\alpha_0 s_1 \alpha_1 s_2 \alpha_2 \ldots s_n \alpha_n\}\) denote an SDT in which \(s_1, s_2, \ldots, s_n\) is a list of all of the schema variables occurring in \(t_1\) when its leaves are read from left to right. For the sake of simplicity, in the following discussion we
will consider schemas having the general form \(<t_i> \{\alpha_0 s_1 \alpha_1\}\). In order to prove the correctness of a transformation having the form:

\[
T \overset{\text{def}}{=} <t_i> \{
\begin{array}{l}
\text{.sd.} \\
<t_i> \{\alpha_0 s_1 \alpha_1\} \\
\Rightarrow \\
<t_i> \{\alpha'_0 s_1 \alpha'_1\}
\end{array}
\}
\]

we need to show

\[
\text{meaning}(<t_i> \{\alpha_0 s_1 \alpha_1\}) \subseteq \text{meaning}(<t_i> \{\alpha'_0 s_1 \alpha'_1\})
\]

for all possible instantiations of the schema variable \(s_1\). An instantiation of \(s_1\) is simply an application of the productions in the grammar to the nonterminal symbol \(s_1\) in order to derive a string of terminal symbols. At first glance, one might think proving

\[
\text{meaning}(<t_i> \{\alpha_0 s_1 \alpha_1\}) \subseteq \text{meaning}(<t_i> \{\alpha'_0 s_1 \alpha'_1\})
\]

will require an infinite number of proofs if \(s_1\) has an infinite number of instantiations (e.g., a separate proof for each possible instantiation of \(s_1\)). Fortunately, this need not be the case. It turns out that in order to prove that the meaning of \(<t_i> \{\alpha_0 s_1 \alpha_1\}\) is less-defined than the meaning of \(<t_i> \{\alpha'_0 s_1 \alpha'_1\}\) it suffices to show that for all instantiations of \(s_1\), the less-defined relationship is implied by a theorem belonging to the mathematical foundation. Recall in Chapter 5 the informal proof of the transformation in Section 5.2.2 is based on a theorem stating that multiplication can be distributed over addition.

In practice, it is often not entirely obvious whether an arbitrary formula of the form

\[
\text{meaning}(<t_i> \{\alpha_0 s_1 \alpha_1\}) \subseteq \text{meaning}(<t_i> \{\alpha'_0 s_1 \alpha'_1\})
\]

is implied by a theorem belonging to the mathematical foundation, even in cases when the theorem is a simple theorem. This difficulty is due to the fact that a great many factors can come into play when determining the semantics of schemas.

For example, in most imperative programming languages the environment and store are generally used to determine the semantics of expressions. Now if side effects are allowed in the language, then different orders of evaluation of an expression can result in different values. With this in mind, consider the transformation:

\[
T_1 \overset{\text{def}}{=} <expr> \{
\begin{array}{l}
\text{.sd.} \\
<expr> \{<primary>"1"+<primary>"2"\} \\
\Rightarrow \\
<expr> \{<primary>"2"+<primary>"1"\}
\end{array}
\}
\]

This transformation reverses the order in which "primary" expressions (e.g., \(<primary>"1"\) and \(<primary>"2"\)) appear in an expression. The theorem from the mathematical foundation that we hope will assist us when we are trying to prove that this transformation is correctness preserving is that addition is commutative. Whether or not \(T_1\) can be proven correct depends on the traditional denotational semantics assigned to the nonterminal symbol \(<primary>\) in the language. Typically, a primary expression (i.e., a \(<primary>\)) is evaluated with respect to a store and an environment (i.e., in Poly, an expression is evaluated with respect to a model of computation which contains a store and an environment). The purpose served by the store and environment is that when taken together they are able to provide the values of any identifiers that are dereferenced within the primary expression. If the denotational definitions of the language permit the evaluation of a primary expression to alter the store or the environment (i.e., the evaluation of primary expressions can cause side effects), then cases can exist where reversing the evaluation order
of primary expressions will produce expressions having different values. In contrast, if the evaluation of a primary expression causes no side effects, then the order of evaluation of two primary expressions may be reversed without changing the value of the overall expression itself. In other words, only in programming languages where the evaluation of primary expressions cannot cause side effects can the theorem that *addition is commutative* (if it exists in the mathematical foundation that is used to define the semantics of the language) be used to prove the correctness of transformations like $T_1$.

This brings up an important point. Because the proof of transformation $T_1$ depends on whether or not the evaluation of primary expressions cause side effects, and because the particular primary expressions whose evaluation we are concerned about are expressed as schema variables in $T_1$, we need to define the semantics of primary expression schema variables in such a way that they capture such relevant information. In the following section we define the semantics of several delta functions that we have used in the course of proving various transformations. The definitions that we give are straightforward and quite general in nature. We leave as a topic of future research the construction of an algorithm that accepts as input a denotational description of a language and constructs refined delta functions for the schema variables of that language.

### 6.5.1 Semantic Definitions of Several Delta Functions

We define *delta* functions in the same format as the more traditional denotational semantic definitions. The only difference between these *delta* function definitions and the other definitions is that *delta* function definitions should be applied to a nonterminal symbol, $s$, only when it is known that $s$ is a schema variable (i.e., only when $s$ occurs as a leaf node in a schema).

A *delta* function simply states that there exists a semantic object that can be assigned to any instantiation of a schema variable with respect to a specific valuation function. A *delta* function does not claim to know what that semantic object is; it only indicates that such an object must exist. Within this framework, *delta* functions attempt to provide an accurate enough description of the semantic object to enable a certain amount of reasoning to be done about the object. For example, suppose we know (by inspecting the denotational definitions of a language) that the evaluations of primary expressions do not cause side effects. Also, suppose we know that in this language, expressions are evaluated with respect to a store and an environment. In this case, we can conclude that a schema variable denoting a primary expression can be defined as a mapping (i.e., a *delta* function) from a store and an environment to a value $v$ where $v$ can be any value that a primary expression might evaluate to (e.g., integer, character, real, etc.).

Below we give the semantics of several *delta* functions. These definitions are not only dependent on a specific schema variable, but they are also dependent on specific valuation functions. The reason *delta* functions need to be defined for a schema variable with respect to a specific valuation function is because different valuation functions, when given the same schema variable, can produce different values. This was discussed in Section 6.4. As a bookkeeping measure, we therefore include the valuation function responsible for creating the *delta* function as a parameter in the *delta* function itself. In addition, it should be noted that a “unique_number” parameter has been added to the parameter list of *delta* functions in order to allow different instances of a schema variable (e.g., different occurrences of the `<primary>` schema variable within a pattern) to be mapped to (potentially) different semantic objects. For example, consider the schema `<expr> {use <op_expr>="2" if <expr>="1" otherwise <expr>="3" end }` constructed from the grammar in Appendix B. As we have already mentioned in Chapter 5, the quoted integers are not part of the grammar. Their purpose is to provide us with a means to distinguish the two instances of `<expr>` that occur in the pattern. Clearly, there exist instantiations of `<expr>="1", <expr>="2", and <expr>="3"` that cannot be denoted by the same semantic object. For example, `<expr>="2"` is instantiated to the integer 56 while `<op_expr>="3"` is instantiated to the integer 57. It is for this reason that the unique number parameter is added to *delta* functions. By associating a unique number with every occurrence of a schema variable in a pattern, we assure that different occurrences of schema variables have the ability to denote potentially different semantic objects.

Below, we give a short list of the *delta* function semantics given to various schemas with respect to particular valuation functions.

1. $E_{\text{EXPR}}[expr] \equiv \lambda m. \lambda k. k(delta(E_{\text{EXPR}}, \text{unique\_number}, value))$.
2. $E_{\text{OP\_EXPR}}[\op\_expr] \equiv \lambda m. \lambda k. k(delta(E_{\text{OP\_EXPR}}, \text{unique\_number}, value))$.
3. $E_{\text{PRIMARY}}[\primary] \equiv \lambda m. \lambda k. k(delta(E_{\text{PRIMARY}}, \text{unique\_number}, value))$.
4. $E_{\text{COND\_EXPR}}[\cond\_expr] \equiv \lambda m. \lambda k. k(delta(E_{\text{COND\_EXPR}}, \text{unique\_number}, value))$.  

70
Section 6.5  Delta-Functions and the Semantics of Static Patterns

5. \( F_{\text{IDENT}}[\text{ident}] \) \text{def} \( \text{delta}(F_{\text{IDENT}}, \text{unique_number}, \text{identifier}) \)

6. \( C_{\text{STMT}\_\text{TAIL}}[\text{stmt\_tail}] \) \text{def} \( \lambda m. \lambda c_n. \text{c}_n(\text{delta}(C_{\text{STMT}\_\text{TAIL}}, \text{unique_number}, \text{model})) \).

With these \text{delta} functions we have extended the traditional semantics of the valuation functions \( E_{\text{EXPR}}, E_{\text{OPEXPR}}, E_{\text{COND}\_\text{EXPR}}, F_{\text{IDENT}}, \) and \( C_{\text{STMT}\_\text{TAIL}} \) enabling them to define the semantics of schema variables \( \text{expr, op\_expr, cond\_expr, ident, and stmt\_tail} \) respectively. Transformation patterns containing any of the above schema variables can now be given a semantics by the above valuation functions.

Earlier, we pointed out the need for the \text{unique_number} parameter in \text{delta} functions. At that time, we were interested in assuring that different instances of the same schema variable (e.g., \text{delta}) would be denoted by different \text{delta} functions. In particular, we did not consider the opposite case where a specific instance of a schema variable occurs more than once in either the input or output schemas (or both) of a transformation, and we want these occurrences to be denoted by equivalent \text{delta} functions. For example, consider the schema \( \text{<expr> \{ <primary>"1" + (<primary>"2") \}} \). Barring side-effects, in this schema, the quoted integers tell us that both occurrences of the schema variable \text{primary} denote the same semantic object. In such a case, the addition of the \text{unique_number} parameter to our \text{delta} function complicates things. If one intends to reason about the semantics of the pattern \( \text{<expr> \{ <primary>"1" + (<primary>"1") \}} \), one would need to explicitly inform the reasoning system that both occurrences of the schema \text{primary} denote the same semantic object. This can be easily accomplished by simply stating that the \text{delta} function denoting the first occurrence of \text{primary} is equal to the \text{delta} function denoting the second occurrence of \text{primary}.

The duplication problem discussed in the previous paragraph is frequently encountered when trying to reason about the input and output schemas of a transformation. For example, suppose we have the transformation:

\[
T_1 \text{def} \{ \text{expr} > \{
\begin{align*}
&.sd. \\
&\text{expr} \{ \text{primary}>"1" + \text{primary}>"2" \} \\
&\Rightarrow \\
&\text{expr} \{ \text{primary}>"2" + \text{primary}>"1" \} \\
&.sc.
\end{align*}
\}
\]

If we wanted to prove that this transformation is correctness preserving, we would need to prove the following theorem:

\[
\text{meaning}(\text{expr} \{ \text{primary}>"1" + \text{primary}>"2" \}) \\
\sqsubseteq \\
\text{meaning}(\text{expr} \{ \text{primary}>"2" + \text{primary}>"1" \}).
\]

Using the extended denotational semantics we get:

- \( \text{meaning}(\text{expr} \{ \text{primary}>"1" + \text{primary}>"2" \}) = \text{delta}(1, \text{value}) + \text{delta}(2, \text{value}), \)
- \( \text{meaning}(\text{expr} \{ \text{primary}>"2" + \text{primary}>"1" \}) = \text{delta}(3, \text{value}) + \text{delta}(4, \text{value}). \)

Recall that \text{delta} functions are objects that belong to our mathematical foundation. In order to correctly reason about the relationship between \text{delta}(1, \text{value}) + \text{delta}(2, \text{value}), and \text{delta}(3, \text{value}) + \text{delta}(4, \text{value}), we would need to inform our reasoning system that:

- \( \text{delta}(1, \text{value}) = \text{delta}(4, \text{value}), \)
- \( \text{delta}(2, \text{value}) = \text{delta}(3, \text{value}). \)

This information, combined with the fact that in our mathematical foundation addition is commutative, allows our formal reasoning system to conclude that

\[
\text{delta}(1, \text{value}) + \text{delta}(2, \text{value}) = \text{delta}(3, \text{value}) + \text{delta}(4, \text{value}).
\]

This in turn allows us to conclude that the transformation \( T_1 \) is correctness preserving.
In closing, we point out that the semantic definitions of the \textit{delta} functions in this section we obtained through informal methods. Ideally, such definitions would be obtained through formal means. However, since this problem appears to be quite complex when considered in general terms, we leave the solution of this problem to future research. Leaving the formal construction of delta functions to future research does not present a problem as far as this research is concerned because the goal of this research is to discover a general methodology that can be used to prove the correctness of TAMPR transformations (i.e., a proof of concept, so to speak). In other words, it is not the goal of this research to construct an industrial strength system capable of proving the correctness of TAMPR transformations.

6.6 The Semantics of Common Dynamic Patterns

In this section we discuss how the denotational definitions of Poly can be used to obtain the meaning of schemas containing dynamic patterns. After the meaning of such a schema has been obtained, reasoning can proceed in the standard fashion.

Transformations containing dynamic patterns can be divided into two classes. The first class, which we call \textit{common dynamic patterns}, contains all those transformations in which every occurrence of a \textit{?} symbol in the input schema of the transformation is matched by the same occurrence of the \textit{?} symbol in the output schema. For example, a transformation having the form:

\begin{align*}
\text{a}\{\alpha\{\gamma_1\}\beta\{\gamma_2\}\} & \Rightarrow \text{a}\{\alpha'\{\gamma'_1\}\beta'\{\gamma'_2\}\}
\end{align*}

would belong to the class of common dynamic patterns. The second class contains all those transformations in which there exists one or more dynamic pattern structures in the input pattern that are disgarded in the output pattern (i.e., the SDT matched by the dynamic pattern structure in the input pattern does not occur in the output pattern). Reasoning about the semantics of transformations containing dynamic patterns belonging to this second class is a topic of future research. In closing we would like to point out that if a dynamic pattern structure is copied to the output pattern via the schema variable denoting the root (or an ancestor of the root) of the dynamic pattern structure, then this output pattern could be equivalently rewritten so that the transformation adheres to the common dynamic pattern form.

This dissertation considers only how one can determine the semantics of common dynamic patterns. Initially, we investigate how one can determine the semantics of a specific common dynamic pattern (the transformation $T_2$ below). After this has been done we will then generalize our results to the class of common dynamic patterns.

As was mentioned in Section 5.4, the \textit{?} is a wild card symbol that can be used to denote very general classes of SDT patterns. For example, in Poly, it is possible to have the actual parameters of function applications be lambda expressions. Consider the transformation $T_2$ below that rewrites function applications containing actual parameters that are lambda expressions.

\begin{verbatim}
T_2 \equiv <entity> {
  .sd.
  <ident>"1" <pending args>"1" {
    ? <args> {
      ( lambda <ident>"2" @ <expr>"2" end <pending args>"2")
    } ?
  } \Rightarrow
  lambda <ident>"2" @
  <ident>"1" <pending args>"1" { ? <args> {(<expr>"2")} ? } end <pending args>"2"
  .sc.
}
\end{verbatim}

Essentially what this transformation does is move a function application containing an actual parameter that is a lambda expression whose lambda bound variable is <ident>"2" inside the scope of the lambda bound variable <ident>"2".

In order to determine the semantics of dynamic patterns like those found in $T_2$ we take advantage of the fact that the constructs in Poly are monotonic with respect to the $\sqsubseteq$ relation. A consequence of this is that if a construct
In this section we discuss some not entirely obvious aspects of proving the correctness of program transformations. Let

\[
T \overset{\text{def}}{=} \langle \text{dominating symbol} \rangle \{ \\
\text{.sd.} \\
\ t_1 \\
\Rightarrow \\
\ t_2 \\
\text{.sc.} \\
\}.
\]

As we have mentioned on numerous occasions, proving the correctness of \( T \) amounts to proving that

\[
\text{meaning}(t_1) \subseteq \text{meaning}(t_2).
\]

What does it take to prove such a relationship? If it can be shown that

\[
\text{meaning}(t_1) = \text{meaning}(t_2),
\]

then the execution of any instance of the program segment denoted by \( t_1 \) (i.e., the simplification of the semantic expression corresponding to \( \text{meaning}(t_1) \)) will have the same effect as that produced by the execution of the corresponding instance of the program segment denoted by \( t_2 \). Recall that the terms simplification and semantic ex-
expression were defined in Section 6.3.5. Note that if we can prove equivalence, then we need not concern ourselves with whether the execution of $t_1$ is undefined, because regardless of the effect produced by the execution (e.g., $\bot$ or normal termination) of $t_1$ and $t_2$ we know that both will have the same behavior. Bear in mind that our objective is to prove the correctness of the transformation $T$ and not to reason about any other aspect of the programs that $T$ might be transforming.

On the other hand, if equivalence between the input pattern and the output pattern of a transformation cannot be shown, then if

$$\text{meaning}(t_1) \supseteq \text{meaning}(t_2)$$

is to hold, it must be the case that $t_1$ is strictly less-defined-than $t_2$. Another way to say this is to say that $t_2$ is a strict refinement of $t_1$. In general, we are interested in the case where the execution of $t_1$ is defined. The reason for not considering cases where the execution of $t_1$ is undefined is because if the execution of the program we intend to transform is undefined to begin with, it really doesn’t matter what we transform this program into. Therefore when trying to prove that $t_2$ is a strict refinement of $t_1$ we will assume without loss of generality that the execution of $t_1$ is defined.

In general, because TAMPR transformations contain schema variables, we will not be able to directly use the simplification axioms (see 6.3.5) that would enable us to simplify the semantic expression, corresponding to the meaning of an input/output pattern, into a function object. This presents somewhat of a problem when trying to prove the correctness of transformations that are not strict equivalences. The problem is that in contrast to equivalences, which could be shown at the semantic expression level, in strict refinements one needs to worry about semantic expressions whose values might be undefined. Recall that a semantic expression is undefined if it cannot be simplified to function object.

For an arbitrary initial model $M_0$, let $f_1$ denote the function object that results from the simplification of meaning($t_1$). Note that we know that such a simplification is possible because we are assuming that the execution of $t_1$ is defined. In order to prove that $t_2$ is a strict refinement of $t_1$ we need to show that the simplification of meaning($t_2$) will yield a function object $f_2$ such that $f_1 \supseteq f_2$. Showing that the simplification of meaning($t_2$) will yield a function object $f_2$ amounts to showing that meaning($t_2$) is defined. That is, the simplification of meaning($t_2$) is not equal to $\bot_{\text{error}}$ or $\bot$. If this can be accomplished, then all that remains is to show that the semantic expression corresponding to meaning($t_2$) is a strict refinement of the semantic expression corresponding to meaning($t_1$).

The difficulties encountered in proving the correctness of strict refinements can perhaps best be demonstrated by an example. In the interests of simplicity, we have constructed an example that is somewhat artificial in the sense that it is hard to imagine a circumstance in which such a transformation would be used. In addition, we have somewhat oversimplified the semantic expressions (e.g., the store) that be produced if the actual denotational semantic definitions for Poly were used. We do not see any of this as a problem because the purpose of this example is to highlight, in as simple a fashion as possible, the issues that can arise when trying to prove the correctness of a transformation whose output pattern is a strict refinement of its input pattern.

$$T_1 \overset{\text{def}}{=} <\text{stmt\_tail}> \begin{array}{cl} .sd. & \begin{array}{l} <\text{stmt\_tail}> \{ \\ x = \text{expr}^\text{“1”}; \\ y = \text{expr}^\text{“2”}; \\ <\text{stmt\_tail}> \} \\ \Rightarrow \\ <\text{stmt\_tail}> \{ \\ x = \text{expr}^\text{“1”}; \\ y = \text{expr}^\text{“2”}; \\ \text{integer}\ z; \\ z = \text{expr}^\text{“1”} + \text{expr}^\text{“2”}; \\ <\text{stmt\_tail}> \} \\ .sc. \end{array} \end{array}$$

74
In this example, we will assume that \( z \) is an identifier that is unique with respect to (i.e., has not been declared in) the particular program to which transformation \( T_j \) is applied. Note that TAMPR provides a construct that allows this to be done. However, it should be noted that in a real TAMPR transformation the syntax is somewhat different than what we have given above. Let \( \varepsilon_0 \) be the environment that is associated with the point in the program after the assignment statement \( y = \langle \text{expr} \rangle \text{"2"} \). Because \( z \) is an identifier that does not occur in \( \varepsilon_0 \), it follows that updating the environment \( \varepsilon_0 \) with \( z \) will produce an environment, \( \varepsilon_1 \), that is a strict refinement of \( \varepsilon_0 \). That is, \( \varepsilon_0 \subseteq \varepsilon_1 \) and \( \neg(\varepsilon_0 \equiv \varepsilon_1) \).

However, with respect to the store this kind of reasoning is not so simple. Let \( \alpha_x \), \( \alpha_y \), and \( \alpha_z \) denote the l-value_location locations of \( x \), \( y \), and \( z \) respectively. Let \( s_0 \) denote the store prior to the assignment statement \( x = \langle \text{expr} \rangle \text{"1"} \). After the assignment to \( y \), the store function will be

\[
[\alpha_y \mapsto v_2]([\alpha_x \mapsto v_1]s_0)
\]

where \( v_1 \overset{\text{def}}{=} \delta(E_{\text{EXPR}.#1,value}) \), and \( v_2 \overset{\text{def}}{=} \delta(E_{\text{EXPR}.#2,value}) \). Recall, in \( \delta \)-functions the second arguments (e.g., #1 and #2) serve as unique identifiers that enables one to establish relationships between particular schema variables.

After the assignment to \( z \), the store function will be

\[
[\alpha_x \mapsto v_1 + v_2]([\alpha_y \mapsto v_2]([\alpha_x \mapsto v_1]s_0))
\]

At first glance it appears that this store is a strict refinement of the store produced by the input pattern. However, it is important to note that the store produced after the assignment to \( z \) is not a function object because \( v_1 + v_2 \) is an (un)simplified semantic expression. Clearly, at the semantic expression level the store produced by the output pattern is a strict refinement of the store produced by the input pattern. However, if \( v_1 + v_2 \) cannot be simplified to a function object, or if \( v_1 + v_2 \) is simplified to \( \bot_{\text{error}} \) then the execution of \( \langle \text{expr} \rangle \text{"1"} + \langle \text{expr} \rangle \text{"2"} \) (i.e., \( v_1 + v_2 \)) will result in an undefined computation. This in turn implies that the execution of the output pattern will result in an undefined computation which implies that the transformation is not correct. On the other hand, if we can somehow show that

the semantic expression \( v_1 + v_2 \) can indeed be simplified into a function object, this will allow us to conclude that the execution of the output pattern will be defined, which in turn will allow us to conclude that the store produced by the output pattern is indeed a strict refinement of the store produced by the input pattern.

From the assumption that the input pattern is defined, we can conclude that both \( v_1 \) and \( v_2 \) are function objects. However, the input pattern does not provide us with enough information to allow us to conclude that the semantic expression \( v_1 + v_2 \) can be simplified to a function object. For example, if somehow we knew that both \( v_1 \) and \( v_2 \) were constant objects of type integer, this would provide us with enough information to allow us to conclude that \( v_1 + v_2 \) could be simplified to a function object. In contrast, if we knew that \( v_1 \) and \( v_2 \) were of incompatible types (e.g., integer and character), this would provide us with enough information to allow us to conclude that the simplification of \( v_1 + v_2 \) would yield the value \( \bot_{\text{error}} \). In summary then, in order to prove the correctness of \( T_j \) our proof procedure must ultimately be able to deduce the types of \( v_1 \) and \( v_2 \). In practice, such information could be made available in the form of a precondition \( Q \) for the transformation under consideration. This typing information together with known axioms defining the behavior of the addition operation provide us with enough information to allow us to determine whether the semantic expression \( v_1 + v_2 \) can indeed be simplified to a function object (i.e., whether the evaluation of \( v_1 + v_2 \) is defined).

### 6.8 Summary

In this chapter we provided a means to reason about individual TAMPR transformations written for Poly. This was accomplished by giving a (partial) grammar for Poly (see Appendix B), which was then formally defined through denotational semantic definitions. Defining the semantics of Poly in this manner required 1) the construction of a mathematical foundation, 2) the construction of a mathematical model of computation, and 3) defining Poly grammar constructs in terms of the effects they produce in our mathematical model of computation. The mathematical model of computation that we use is a bit more complex than models that have been used to define the semantics of languages like Pascal. This complexity is due to the fact that 1) Poly is a wide-spectrum language comprising both functional and imperative constructs, and 2) our intention is to provide a semantic framework in which the relationship between functional and imperative constructs can be reasoned about.

After formally defining the semantics of Poly, we described in general terms how such a semantics can be used to
define the semantics of schemas. In order to accomplish this we first of all needed to realize that complete phrases derived from a nonterminal symbol other than the start symbol of the grammar can have multiple meanings depending on the contextual information that is available. In addition, phrases that are not complete (i.e., phrases containing schema variables) cannot be defined using the existing denotational semantic definitions that are written for Poly. In order to formally define the semantics of general phrases, the notion of delta functions was introduced. The delta function provides a mechanism in which the general properties possessed by all instantiations of a particular schema variable can be captured. We listed several of the more common delta functions that we have used during the course of this research.

The introduction of delta functions into the denotational semantic framework allow static schemas to be formally defined. However, common dynamic patterns were still not defined. By realizing that the constructs in Poly are monotonic with respect to replacement and by taking advantage of the fact that TAMPR transformations containing common dynamic patterns have matched symbols in the input and output schemas of the transformation, we were able to define the semantics of a common dynamic pattern in terms of its components. This allows correctness proofs to be attainable for TAMPR transformations containing common dynamic patterns.

In the following chapter we discuss a form of reasoning for sequences of TAMPR transformations. We also consider proving properties of transformations and transformation sequences other than correctness.
Chapter 7
Proving the Correctness of Transformation Sequences

7.1 Overview

In the previous chapter we presented a methodology that allows one to formally reason about individual TAMPR transformations for the language Poly. This methodology provided a framework enabling us to prove theorems having the form \( Q(x) \Rightarrow R(x, T(x)) \), where \( T \) can be an arbitrary TAMPR transformation having the form described in Chapter 5. It should be noted that the methodology described in Chapter 6 assumes that the precondition \( Q \) and the postcondition \( R \) are given and therefore only concerns itself with formalizing the actual proof process. The proof process essentially amounts to 1) formalizing the semantics of TAMPR transformations with respect to the language Poly, 2) using this semantics to map the input and output schemas of a transformation into semantic expressions, and 3) to then reason about these semantic expressions. What is not addressed in Chapter 6 is how one can show that an input program \( x \) indeed possesses the property \( Q \) (i.e., how one can go about showing that \( Q(x) \) holds). These two points become quite significant when one considers the fact that individual transformations, in practice, will be embedded in transformation sequences.

For example, suppose \( T_i \) belongs to the sequence \( T_{1,n} \). If we disallow the \((\_\_\_\_\_)\) transformation sequence constructor described in Chapter 5, then we can conclude that \( T_{1,n} \) must be of the form \( T_{1,i-1} ; T_i ; T_{i+1,n} \). Let \( x_1 \) denote an arbitrary Poly program to which no transformations have as yet been applied. Furthermore, let \( x_i \equiv T_{1,i-1}(x_1) \). If one wishes to show that \( T_i \) establishes the property \( R \), then it is not sufficient to simply prove \( Q_i(x_i) \Rightarrow R(x, T_i(x_i)) \). One also needs to prove that “true \( (x_1) \Rightarrow Q_i(T_{1,i-1}(x_1)) \)” Note that we will assume that the first transformation in any transformation sequence has the precondition true (i.e., \( Q_1 \equiv true \)). Recall, the predicate true holds for all programs in Poly and is defined as: “true \( (x) \equiv (x \in Poly) \). Proving “true \( (x_1) \Rightarrow Q_i(T_{1,i-1}(x_1)) \)” asserts that any Poly program \( x_1 \) that is transformed by the transformation sequence \( T_{1,i-1} \) will possess the property \( Q_i \).

We define a static transformation sequence as a transformation sequence containing transformations having static input and output schemas. Recall that static schemas were discussed in Chapter 5. In this chapter we will describe how, for a restricted class of static transformation sequences, one can formally determine the most general property \( Q_i^{def} \) that is established by the sequence. In other words, we will demonstrate how one can calculate the most general property \( Q_i^{def} \) such that “true \( (x_1) \Rightarrow Q_i^{def}(T_{1,i-1}(x_1)) \) holds. After the property \( Q_i^{def} \) has been determined, it can then be used as the precondition for a theorem of the form: \( Q_i^{def}(x_1) \Rightarrow Correct(x_1, T_i(x_1)) \). This property \( Q_i^{def} \) together with the proof of the theorem \( Q_i^{def}(x_i) \Rightarrow Correct(x_i, T_i(x_i)) \) will allow us to conclude that the transformation \( T_i \), is correctness preserving when it is embedded in the transformation sequence \( T_{1,n} \).

7.2 An Abstract Example

In this section we formally define the various theorems that need to be proven in order to prove the correctness of an arbitrary transformation sequence \( T_{1,n} \).

As we have already mentioned in Chapter 6, by using the TAMPR transformation system together with the wide spectrum language Poly, it is possible to write high-level functional specifications that can then be automatically transformed (by TAMPR) into FORTRAN 66 implementations. A functional specification is transformed into a
Chapter 7 Proving the Correctness of Transformation Sequences

FORTRAN 66 implementation by applying a transformation sequence to it. Let $T_{1,n}$ denote such a transformation sequence. In order to prove the overall correctness of $T_{1,n}$, one needs to prove the following theorems:

**Theorem 7** $\forall i : 1 \leq i \leq n : Q_i(x) \Rightarrow Correct(x, T_i(x))$

**Theorem 8** $\forall i : 1 \leq i \leq n - 1 : true(x) \Rightarrow Q_{i+1}(T_{i+1}(x))$

where $Q_1 \overset{def}{=} true$

If $T_{1,n}$ is correctness preserving, then $x \sqsubseteq T_{1,n}(x) \forall x \in Poly$. Once both of the above theorems have been proven, then by transitivity of correctness (i.e., the $\sqsubseteq$ relation) it follows that for any input program $x$ having property $true(x)$ (i.e., all Poly programs) the output program $T_{1,n}(x)$ will be correct with respect to $x$ (i.e., $x \sqsubseteq T_{1,n}(x) \forall x \in Poly$). The above example shows that proving a transformation sequence correct will in general consist of more than just proving that the individual transformations are correct. In addition to correctness proofs, what is required are general proofs showing that the input programs of an arbitrary transformation $T_i$ are the type of programs that $T_i$ can correctly transform (i.e., the input programs have property $Q_i$). An important question that should be asked is “In the above theorems, who determines the properties $Q_i$ (i.e., where do these properties come from)?” One possible approach would be to let the person responsible for proving the correctness of a particular transformation sequence determine the appropriate properties $Q_i$ that are sufficient to successfully prove the required theorems. An objection we have to this approach is that determining the appropriate $Q_i$’s becomes somewhat of an art involving trial and error.

Another difficulty is that once a property $Q_i$ has been informally determined, in what formal language will $Q_i$ be expressed? Bear in mind that $Q_i$ is a property that we wish to formally reason about. Because it is desirable to formally reason about preconditions, one generally states them in terms of first order logic expressions $[31], [58], [23], [30]$. Unfortunately, due to the fact that TAMPR transformations are, in theory, limited only by the syntactic requirements of the language (i.e., any syntactically legal rewrite is permitted), expressing preconditions in the traditional manner is not desirable. One of the contributions of this research is the discovery of a way to express properties like $Q_i$ in a manner suitable for TAMPR and in a manner that permits formal reasoning about $Q_i$.

Lastly we need to concern ourselves with the completeness of $Q_i$. For example, given a property $Q_i$ that is established by a transformation $T_{1,i-1}$, does $T_{1,i-1}$ establish any properties besides $Q_i$? In other words, is $Q_i$ a complete description of the properties established by $T_{1,i-1}$? One of the contributions of this research has been the discovery of a formal way to determine, for a restricted set of transformation sequences, a property $Q_i$ that is a complete description of the properties established by $T_{1,i-1}$. For example, let $q_i$ denote an arbitrary property that holds for all programs $T_{1,i-1}(x_1)$ where $x_1 \in Poly$. The property $Q_i$ is such that

$$Q_i(T_{1,i-1}(x_1)) \Rightarrow q_i(T_{1,i-1}(x_1)).$$

Our methodology consists of two parts. The first part concerns itself with the discovery of a formal language capable of expressing arbitrary properties like $Q_i$, and the second part concerns itself with the discovery of an algorithm that can calculate the property $Q_i$ that holds for all programs of the form $T_{1,i-1}(x_1)$.

As a final note, it should be mentioned that quite often transformations can be proven correct irrespective of any particular context. If this is the case for a particular transformation $T_i$ then $true(x) \Rightarrow Correct(x, T_i(x))$ will be a theorem. If such a theorem can be proven for every transformation in the transformation sequence, then the task of proving the overall correctness of the transformation sequence is greatly simplified. This simplification is due to the fact that proving theorems of the form $Q_i(x) \Rightarrow Correct(x, T_i(x))$ are now unnecessary since

$$(true(x) \Rightarrow Correct(x, T_i(x))) \Rightarrow (Q_i(x) \Rightarrow Correct(x, T_i(x))).$$

(Certainly if all Poly programs can be correctly transformed by $T_i$, then the subset of Poly programs possessing property $Q_i$ will also be correctly transformed.) Hence the discovery of $Q_i$ is unnecessary.

7.3 A Formal Language that can be used to Describe $Q_i$

There are several things one needs to consider when trying to construct a formal language capable of defining the
Section 7.3  A Formal Language that can be used to Describe $Q'_i$

properties that a transformation sequence can establish. First of all, the effects transformation sequences have on the programs they transform can be viewed in semantic or syntactic terms. A semantic description is perhaps the approach that might seem the most natural if a human were responsible for the discovery of the property $Q'_i$ established by the transformation sequence $T_{1,i-1}$. In such a case, a person would study a particular transformation sequence, $T_{1,i-1}$. When the person understands what $T_{1,i-1}$ is accomplishing, he or she would then be ready to write a formal description, $Q'_i$, of the property established by the transformation sequence. Bear in mind that in this approach, $Q'_i$ is the property that the person believes is established by the transformation sequence. Note, this does not present a problem as far as our methodology is concerned, because if a mistake has been made and it is actually the case that $Q'_i$ is not established by $T_{1,i-1}$ then $true(x) \Rightarrow Q'_i(T_{1,i-1}(x))$ will not be provable by formal methods.

Another issue that needs to be considered is that ultimately one must be able to formally reason about properties like $Q'_i$. For example, proving that a transformation, $T_i$, is correct in a specific context (i.e., when the input programs that $T_i$ will be applied to have the property $Q'_i$) requires formal reasoning about $Q'_i$. Because by definition $M_i$ is the only set in which we can carry out any kind of formal reasoning, properties like $Q'_i$ must ultimately be in $M_i$ (e.g., semantic expressions, function objects, or constant objects).

Finally, while some transformations alter their input programs in ways that can easily be expressed in semantic terms, other transformations may alter their input programs in ways that cannot readily be expressed in semantic terms. For example, in a Pascal-like language suppose the transformation $T_i$ takes the first declaration statement in a block (e.g., `block-begin compound statement block-end`) and moves this statement in front of the block. Clearly the effects of such transformations would be difficult to directly express in semantic terms.

However, it turns out that because TAMPR transformations manipulate programs on the syntactic level, the effect of any single transformation can easily be expressed in syntactic terms. For this reason, we have chosen to express the properties a transformation sequence establishes in syntactic terms. As we have mentioned in Chapter 5, TAMPR transformations are of the form:

$$T_i \overset{def}{=} <\text{dominating symbol}> \{ \begin{align*}
&.sd. \\
&t_{i,\text{in}} \\
&\Rightarrow \\
&t_{i,\text{out}} 
\}$$

where $t_{i,\text{in}}$ and $t_{i,\text{out}}$ are SDT schemas. Now, an exhaustive application of $T_i$ to an input program $x$ results in an output program $T_i(x)$ that does not contain any instances of the schema $t_{i,\text{in}}$. We denote the absence of the schema $t_{i,\text{in}}$ (i.e., the syntactic property) by $\neg t_{i,\text{in}}$. Certainly, one of the syntactic properties possessed by the output program $T_i(x)$ is $\neg t_{i,\text{in}}$. In the interests of clarity, we will from here on out use the notation $Q_{-i,\text{in}}$ to denote the syntactic property $\neg t_{i,\text{in}}$. Similarly we will use the notation $Q_{-i,\text{in}}$ to denote the syntactic property $t_{i,\text{in}}$.

What other syntactic properties does the output program $T_i(x)$ possess? This problem can be broken down into two parts. The first part involves trying to determine what properties $T_i(x)$ possesses. Let $Q'_i$ denote the properties possessed by the class of programs $T_{1,i-1}(x)$. The second part of the problem involves using $Q'_i$ together with $Q_{-i,\text{in}}$ to construct $Q'_{i+1}$. For the sake of simplicity, we will initially consider the case where $T_{1,i-1}(x)$ only possesses the property $Q_{-(i-1),\text{in}}$. One question that can be asked is: “Does $T_i(T_{1,i-1}(x))$ also possess the property $Q_{-(i-1),\text{in}}$?” Another question is “Can $T_i(T_{1,i-1}(x))$ possess any syntactic properties besides $Q_{-(i-1),\text{in}}$ and $Q_{-i,\text{in}}$?”

The answer to whether $T_i(T_{1,i-1}(x))$ possesses the syntactic property $Q_{-(i-1),\text{in}}$ depends on the transformation $T_i$ itself. If $T_i$ transforms its input program in such a way that schemas of the form $t_{(i-1),\text{in}}$ can result, then the output program $T_i(T_{1,i-1}(x))$ will not possess the property $Q_{-(i-1),\text{in}}$. Note that just because $T_i(T_{1,i-1}(x))$ does not possess the property $Q_{-(i-1),\text{in}}$ this does not imply that $T_i(T_{1,i-1}(x))$ must possess the property $Q_{(i-1),\text{in}}$. The reason for this is that $T_i(T_{1,i-1}(x))$ has the property $Q_{-(i-1),\text{in}}$ means that for all input programs $x$, $T_i(T_{1,i-1}(x))$ possesses the property $Q_{-(i-1),\text{in}}$. If there exists a single input program $x'$ for which $T_i(x')$ does not possess the property $Q_{-(i-1),\text{in}}$ then all we can say is that $\neg t_{(i-1),\text{in}}$ does not hold.

In this chapter we will show that (for a restricted set of static transformation sequences) $Q'_{i+1}$ together with $Q_{-i,\text{in}}$ and $T_{1,i-1}$ provide enough information for the construction of $Q'_i$. While the results of this research hold only for a restricted set of static transformation sequences, we believe that our approach can be extended to allow the construction of $Q'_{i}$ for transformations $T_{i}$ occurring in arbitrary transformation sequences.

In summary, the language in which we have chosen to express the properties that are established by a transfor-
Chapter 7  Proving the Correctness of Transformation Sequences

...ation is essentially a Boolean form of the language that is used to write TAMPR transformations themselves. This turns out to be very beneficial because in Chapter 6 the semantics of TAMPR transformation patterns were formalized. Exhaustive application of a transformation will (upon termination) establish the syntactic property that its input pattern no longer exists. This alone is not sufficient to tell us whether 1) syntactic properties established by previous transformations are preserved, or 2) the combination of previously established properties together with the currently established property provides a complete description of the syntactic properties established by the application of the transformation. The remainder of this chapter is devoted to answering the question of which previously established syntactic properties are preserved by the application of a transformation and how this information can be used to construct a property $Q_i^k$ that is complete.

### 7.4 Determining $Q_i^k$

#### 7.4.1 Overview

In this section and in Sections 7.5 and 7.6 we are interested in constructing a framework in which one can use information involving a particular relationship between a transformation $T_i$ and the transformation sequence $T_{i_1, \ldots, i_n}$ that precedes it together with the property $Q_{i_1}^k$ in order to construct $Q_i^k$. We use the name independent to denote the property (i.e., relationship) between $T_i$ and $T_{i_1, \ldots, i_n}$ in which we are interested. The independent property is formally defined in Section 7.4.3. For this overview all that is important is that independent is a predicate (i.e., a property) which asserts that the application of the transformation $T_i$ to a program of the form $T_{i_1, \ldots, i_n}(x_1)$ will preserve the syntactic property $Q_{i_1}^k$ that $T_{i_1, \ldots, i_n}(x_1)$ possesses.

Abstractly, there are three difficulties that need to be overcome in order for us to be able to construct $Q_i^k$ from $Q_{i_1}^k$ and the evaluation of the predicate independent$(T_i, T_{i_1, \ldots, i_n})$. First, we must be able to demonstrate that independent$(T_i, T_{i_1, \ldots, i_n})$ can be computed. Second we must be able to show that the result obtained from the evaluation of independent$(T_i, T_{i_1, \ldots, i_n})$ together with $Q_{i_1}^k$ can be used to construct $Q_i^k$, and third we must be able to prove that the property $Q_i^k$ that has been constructed is a sound and complete description of the properties possessed by all programs of the form $T_{i_1, \ldots, i_n}(x_1)$ where $x_1 \in \text{Poly}$.

Our research has lead us to believe that discovering an algorithm that can determine independent$(T_i, T_{i_1, \ldots, i_n})$ for arbitrary $T_i$ and $T_{i_1, \ldots, i_n}$ is a difficult and lengthy task. We therefore leave the discovery of such a general algorithm to future research. In this dissertation we focus our attention on restricting independent into a more manageable property. We place two restrictions on the general independent property. This results in a property that we refer to as independent$_{\rho}$. In Section 7.5 we present an algorithm for computing independent$_{\rho}(T_i, T_{i_1, \ldots, i_n})$ for arbitrary $T_i$ and $T_{i_1, \ldots, i_n}$. Then in Section 7.6 we show that 1) independent$_{\rho}(T_i, T_{i_1, \ldots, i_n})$ can be formally computed (i.e., it is an algorithm), 2) independent$_{\rho}(T_i, T_{i_1, \ldots, i_n})$ together with $Q_{i_1}^k$ can be used to construct $Q_i^k$, and 3) the property $Q_i^k$ that is constructed in this fashion is both a sound and complete description of the properties possessed by all programs of the form $T_{i_1, \ldots, i_n}(x_1)$ where $x_1 \in \text{Poly}$.

We would like to point out that because independent$_{\rho}$ is a restriction of independent, the latter is implied by the former. That is, independent$_{\rho}(T_i, T_{i_1, \ldots, i_n}) \Rightarrow$ independent$(T_i, T_{i_1, \ldots, i_n})$ holds for all $T_i$ and $T_{i_1, \ldots, i_n}$. In Section 7.4.3 we state the restrictions that we have imposed on the general independent property together with a discussion of their relevance. In summary, the goal of these restrictions is to characterize a class of transformation sequences $T_{i_1, \ldots, i_n}$ for which we can construct $Q_i^k$ for all $i$ between 1 and $n$.

#### 7.4.2 The Problem

Consider the transformation sequence $T_{i_1, \ldots, i_n} \overset{\text{def}}{=} T_1; T_2; T_3$. Now suppose that for transformations $T_1$, $T_2$, and $T_3$ we have proven the following theorems:

- $\text{true}(x) \Rightarrow Q_2(T_1(x))$
- $\text{true}(x) \Rightarrow \text{Correct}(x, T_1(x))$
- $Q_2(x) \Rightarrow Q_3(T_2(x))$
- $Q_2(x) \Rightarrow \text{Correct}(x, T_2(x))$
- $Q_2(x) \land Q_3(x) \Rightarrow \text{Correct}(x, T_3(x))$

Given that we have proven the above theorems, if we could in addition show that $Q_2(T_2(T_1(x)))$ holds, then...
we can conclude that the transformation $T_3$ is correctness preserving. Note that from the above theorems, it follows that $Q_3(T_2(T_1(x)))$ holds. What does not necessarily follow is $Q_2(T_2(T_1(x)))$. Clearly, we need to know if $Q_2(T_3(T_1(x)))$ holds in order to use the above theorem to conclude that $T_3$ is correctness-preserving.

In this section we present a methodology that will allow us to formally answer general questions like: “Does $Q_2(T_3(T_1(x)))$ hold?” That is, in this section we are concerned with how one can determine whether a property (e.g., $Q_2$) that holds for an arbitrary input program of a transformation (e.g., $T_2$) will hold for the output program produced by the transformation. In Section 7.6 we will then formally prove the completeness of this methodology.

Informally, if a transformation $T_i$ does not interfere with the properties established by the transformation sequence $T_{i,-1}$ that precedes it, we say that $independent(T_i, T_{i,-1})$ holds. In the above example, if $T_2$ is $independent$ of $T_1$, then the property $Q_2$ that is established by $T_1$ will be preserved by the application of $T_2$. If every transformation in a transformation sequence, $T_{1,n}$, is $independent$ of the transformation sequence that precedes it we say that $independence(T_{1,n})$ holds.

It should be noted that it is quite easy to construct a transformation that is not independent of the transformation sequence that precedes it. Consider the following two transformations:

$$T_1' \overset{def}{=} \text{<stmt tail> } \{ \text{.sd.} \text{<stmt tail> } \{ \text{x = <expr>"1";} \text{y = <expr>"2";} \text{<stmt tail>"1" } \} \Rightarrow \text{<stmt tail> } \{ \text{y = <expr>"2";} \text{x = <expr>"1";} \text{<stmt tail>"1" } \} \text{.sc.} \}$$

$$T_2' \overset{def}{=} \text{<stmt tail> } \{ \text{.sd.} \text{<stmt tail> } \{ \text{y = <expr>"1";} \text{x = <expr>"2";} \text{<stmt tail>"1" } \} \Rightarrow \text{<stmt tail> } \{ \text{x = <expr>"2";} \text{y = <expr>"1";} \text{<stmt tail>"1" } \} \text{.sc.} \}$$

For some Poly programs the above transformation sequence, $T_{1,2}' \overset{def}{=} T_1'; T_2'$ will produce an output program that is identical to the input program (i.e., for these programs $T_{1,2}'$ is the identity transformation). Clearly for such programs the transformation $T_2'$ will “undo” any syntactic properties (i.e., any properties $Q$) established by $T_1'$. We hope this simple example convinces the reader that determining whether $independent(T_i, T_{i,-1})$ and $independence(T_{1,n})$ hold is a real problem and not just a theoretical exercise.

The discovery of the properties $independent(T_i, T_{i,-1})$ and $independence(T_{1,n})$ as well as the inference rules
that can be used to determine their existence is one of the contributions of this research. In this paper when we say “independence properties” we are collectively referring to the predicates independence and independent.

On a side note, we believe that the research discussed in this chapter applies to reasoning about demodulators and paramodulators as they are used in the field of automated reasoning [68].

### 7.4.3 Independence Properties

From here on out, we will only consider a transformation in connection with the transformation sequence which precedes it. For example, if we mention the transformation $T_i$ we are implicitly assuming that $T_i$ is preceded by the transformation sequence $T_{1,i-1}$. For a language $L(G)$, the set of programs to which $T_i$ can be applied (i.e., the input set of $T_i$) is denoted by the symbol $X_i$ and is formally defined as follows:

$$X_i \overset{\text{def}}{=} \{ x \mid x = T_{1,i-1}(x') \text{ where } x' \in L(G) \}.$$

Furthermore, we will use the symbol $x_i$ to denote an arbitrary element of $X_i$. The above definition of $X_i$ is a formal characterization of the set of programs to which the transformation $T_i$ could potentially be applied. This being the case, one could argue that because the definition of $X_i$ is a complete description of the properties a program must have in order to belong to the input set of transformation $T_i$, this definition of $X_i$ provides us with $Q_i'$. While it is true that the definition of $X_i$ does provide us with a complete description of the properties possessed by the input programs to $T_i$, it is also the case that the above characterization provides us with almost no useful information as to the specific syntactic or semantic properties of the programs belonging to the input set of $T_i$. Bear in mind that our objective is to construct a property $Q_i'$ that will allow us to prove theorems like:

$$Q_i'(x) \Rightarrow R(x, T_i(x))$$

Recognizing that we ultimately must be able to incorporate $Q_i'$ into our formal reasoning we have defined a different characterization of the set $X_i$ based on independence properties. Towards this end, we now formally define the predicates independent and independence.

- **Independent:**
  
  $$\forall T_i \text{ independent } (T_i, T_{1,i-1}) \overset{\text{def}}{=} X_i \supseteq X_{i+1}$$

  where $1 \leq i$. Note that when $i = 1$ we have independent $(T_1, T_{1,0})$. This is perfectly acceptable if we let $T_{1,0}$ denote the empty transformation sequence. This in turn implies that independent $(T_1, T_{1,0})$ always holds.

- **Independence:**
  
  $$\forall T_{1,n} \text{ independence } (T_{1,n}) \overset{\text{def}}{=} X_1 \supseteq X_2 \supseteq \ldots \supseteq X_{n+1}$$

First of all, we would like to point out that:

$$\text{independence}(T_{1,n}) \equiv ((1 \leq i < j \leq n + 1) \Rightarrow Q_{-i,in}(x_j)).$$

The above relationship between independence $(T_{1,n})$ and the syntactic properties $Q_{-i,in}$ turns out to be very central to our research. Later in this chapter we will prove that for transformation sequences $T_{1,n}$ for which independence $(T_{1,n})$ hold, the expression $(1 \leq i < j \leq n + 1) \Rightarrow Q_{-i,in}(x_j)$ is an axiomatization of all the properties possessed by $x_j$. In other words, for transformation sequences possessing the independence property, the expression $(1 \leq i < j \leq n + 1) \Rightarrow Q_{-i,in}(x_j)$ is a basis for constructing $Q_j'$ for $1 \leq j \leq n + 1$.

#### 7.4.3.1 The First Restriction

In the original definition the independent predicate defines a relationship between a transformation $T_i$ and the transformation sequence $T_{1,i-1}$ that precedes it. In particular the independent predicate does not assume or require that independence $(T_{1,n})$ hold. In this section we define a restricted form of the independent predicate. We define a predicate independent$_{r1}$ which when given a transformation $T_i$ and the transformation sequence $T_{1,i-1}$ that precedes it, will only evaluate to true if independent$_{r1}$ $(T_i, T_{1,i-1})$ and if every transformation in $T_{1,i-1}$ also satisfies the independent$_{r1}$ property. Thus independent$_{r1}$ $(T_n, T_{1,n-1}) \equiv \text{independence}(T_{1,n})$.

- **Independent$_{r1}$:**
  
  - Base Case: independent$_{r1}$ $(T_1, T_{1,0}) \overset{\text{def}}{=} \text{true}$
  
  - General Case:
independent_{r1} (T_i, T_{i, i-1}) \overset{\text{def}}{=} (i > 1) \land \text{independence} (T_{i, i-1}) \land \forall x_i, j : (1 \leq j \leq i) \Rightarrow (T_i(x_i) = T_j (T_i(x_i)))

- independence (T_{i, n}) \overset{\text{def}}{=} \text{independent}_{r1} (T_n, T_{i, n-1})

In the above definitions we use the subscript \(r1\) to distinguish the property independent from independent_{r1}. The reason for not defining independent_{r1} in the “set notation” that was originally used to define independent is because our goal is to construct an effective procedure that can be used to test an arbitrary transformation to see whether independent_{r1} holds. That being the objective, it should be clear that a definition like independent (T_{i, n}) \overset{\text{def}}{=} X_1 \supseteq X_2 \supseteq \ldots \supseteq X_n is a long ways away from an effective algorithm.

### 7.4.4.3\ The Second Restriction

Let \(T^1_i\) denote a transformation having the same input and output schemas as \(T_i\), but differing from \(T_i\) in the semantics of its application. When the transformation \(T^1_i\) is applied to an input program \(x_i\), it searches for the first occurrence of \(t_{i,in}\), rewrites this schema into \(t_{i,out}\) and then halts. Note that we did not specify how one should go about finding the first occurrence of \(t_{i,in}\). The reason for this omission is because for our purposes, it does not matter how this is accomplished. Any search method (e.g., left to right and top to bottom, or right to left and bottom to top) will do. All that we are concerned with is that the difference between \(T^1_i\) and \(T_i\) is the fact that \(T^1_i\) performs a single rewrite while \(T_i\) will continue to be applied until an output program is produced containing no instances of the input schema.

In this research we limit our study to the discovery of an algorithm that can be used to determine if a transformation sequence satisfies the following restricted independence properties:

- Independent_{r2}:
  - Base Case: independent_{r2} (T_1, T_{1, 0}) \overset{\text{def}}{=} true
  - General Case:
    independent_{r2} (T_i, T_{i, i-1}) \overset{\text{def}}{=} (i > 1) \land \text{independence}_{r2} (T_{i, i-1}) \land \forall x_i, j : (1 \leq j < i) \Rightarrow (T^1_i(x_i) = T^1_j (T^1_i(x_i)))

- Independence_{r2}:
  - independence_{r2} (T_{i, n}) \overset{\text{def}}{=} \text{independence}_{r2} (T_n, T_{i, n-1})

We believe that the algorithm we will present in this chapter can be extended to determine general independence properties. However, as we mentioned earlier, we leave this extension to future research.

From a practical point of view, the relationship between independent_{r2} and independent_{r1} is that independent_{r2} \Rightarrow independent_{r1}. In other words, there exist transformation sequences \(T^1_{i, n}\), for which independent_{r2} (T_{i, n}) is not satisfied but independent_{r1} (T_{i, n}) holds. Thus by using independent_{r2} we will sometimes incorrectly conclude that the property independent_{r1} is violated when in reality it is not. We believe that these cases, while theoretically possible, will virtually never occur in practice. Nevertheless, this implies that by using the property independent_{r2} as the basis for determining whether independence (T_{i, n}) holds, our algorithm will sometimes claim that independence (T_{i, n}) does not hold when in fact it does. It should be noted that incorrectly concluding that independence (T_{i, n}) does not hold will not enable us to erroneously prove that a transformation is correct when in fact it is not. In fact quite the contrary is true.

### 7.4.4 Summary

We believe that the independence properties of an arbitrary transformation sequence, \(T_{i, n}\), can be used for the construction of the properties \(Q^1_i\) for \(1 \leq i \leq n\). For example, given a transformation sequence \(T_{i, n}\), we may begin by defining \(Q^1_i\) to be the property true, since by convention we are assuming that \(T_1\) can be applied to any Poly program. Since, by definition, independent (T_{i, 1, 0}) always holds, the property \(Q^1_2\) is simply the conjunction of \(Q^1_i\) and \(Q_{i, in}\) which can be simplified to \(Q_{i, in}\). In other words, \(Q^1_2 = Q_{2, in}\). Unfortunately, the construction of \(Q^1_3\) is a bit more problematic. In the case where independent (T_{2, 1, 1}) holds, we have been able to prove (see Section 7.6) that \(Q^1_3 = Q^1_2 \land Q_{2, in}\) is a complete description of the properties common to all programs of the form \(T_{1, 2} (x_1)\) where \(x_1 \in Poly\). How \(Q^1_3\) can be constructed in the case where independent (T_{2, 1, 1}) does not hold is a problem that we leave to future research.

We believe that, for general transformation sequences, the construction of \(Q^1_i\) is quite difficult. This difficulty is predominantly due to the fact that it is not clear how to determine \(Q^1_i\) when independent (T_{i, 1, i-1}) does not hold, and it is (possibly) even harder to determine \(Q^1_i\) when independent (T_{i, i, i-1}) holds, but independent (T_{j, j-1}) does

83
Chapter 7 Proving the Correctness of Transformation Sequences

not hold for some \( j < i \). For this reason, we have restricted the independent property so that it is more manageable. The restricted form of independent is a property which we call independent\(_{r2}\). The independent\(_{r2}\) is obtained at by placing two restrictions on the independent property. The first restriction eliminates the case we mentioned above where independent\((T_i, T_{i-1})\) holds, but independent\((T_j, T_{j-1})\) does not hold for some \( j < i \). This resulted in a property we called independent\(_{r1}\).

The properties independence and independent\(_{r1}\) are equivalent in the sense that:

\[
\text{independence}(T_n) \equiv \text{independent}_{r1}(T_n, T_{1,n-1}).
\]

However independent\(_{r1}\) is defined in a more algorithmic form than the original definition of independence. Even though the definition of independent\(_{r1}\) is algorithmic in nature, constructing an algorithm that, for an arbitrary transformation \( T_i \), computes whether independent\(_{r1}(T_{1,n}) \) holds appears to be quite difficult. For this reason we have further restricted the independent property in order to enable us to construct an algorithm that can test for this restricted form of the independent property.

The final restriction that is placed on the property independent\(_{r1}\) results in a property that we call independent\(_{r2}\). As can be seen from their definitions, independent\(_{r1}\) and independent\(_{r2}\) are not equivalent, but independent\(_{r2}\) \( \Rightarrow \) independent\(_{r1}\) holds. This means that in all cases if our algorithm for evaluating independent\(_{r2}\) determines that independent\(_{r2}(T_{1,n}) \) holds, then we will be able to correctly construct the properties \( Q_i' \) that hold for the various input sets in the transformation sequence.

### 7.5 An Algorithm for Evaluating the independent\(_{r2}\) Predicate

We begin this section with a claim (which we will later prove) that for any transformation sequence \( T_{1,n} \) satisfying the predicate independent\(_{r2}(T_{1,n}) \) the properties \( Q_{-i,\text{in}} \) for \( 1 \leq i < j \) are an axiomatization of \( Q_j' \) for \( 1 \leq j \leq n + 1 \). If for the moment we accept this claim, then we are in a position to write a formal specification of a process that determines \( Q_j' \) for an arbitrary transformation sequence \( T_{1,i-1} \). The property \( Q_j' \) is a syntactic property that is equal to \( Q_j' \) when independent\(_{r2}(T_{1,i-1}) \) holds, and is equal to the predicate true otherwise.

```
read(T_{1,i-1});
if independent\(_{r2}(T_{1,i-1}) \) then
  Q_j' = Q_{-1,in} \& Q_{-2,in} \& \ldots \& Q_{-i-1,in}
else
  Q_j' = true
```

**Figure 3**

Regardless of the syntactic property assigned to \( Q_i'' \) by the above process, \( Q_i'' \) can be used in theorems involving the transformation \( T_i \). For example the theorem that needs to be proven in order to show that \( T_i \) is correctness preserving is:

\[
Q_i''(x) \Rightarrow \text{Correct}(x, T_i(x))
\]

In this theorem, if \( Q_i'' \) has the value true, then we are assuming nothing about the syntactic structure of the input program to the transformation \( T_i \), and the above theorem becomes:

\[
true(x) \Rightarrow \text{Correct}(x, T_i(x)).
\]

On the other hand, if \( Q_i'' = Q_{-1,in} \& Q_{-2,in} \& \ldots \& Q_{-i-1,in} \) then one only needs to prove that \( T_i \) is correctness preserving when it is applied to an input program \( x \) where \( Q_i''(x) \) holds.

We now give a somewhat informal proof that the property the above process assigns to \( Q_i'' \) is sound.

**Case 1.** Letting \( Q_i'' = true \) is sound because:

\[
(true(x) \Rightarrow R(x, T_i(x))) \Rightarrow (Q_i''(x) \Rightarrow R(x, T_i(x)))
\]

**Case 2.** Letting \( Q_i'' = Q_{-1,in} \& Q_{-2,in} \& \ldots \& Q_{-i-1,in} \) is sound. This requires proving that \( Q_{-1,in} \& Q_{-2,in} \& \ldots \& Q_{-i-1,in} \) is sound which is something that we will do later in this chapter.
Section 7.5 An Algorithm for Evaluating the $\text{independence}_{r,2}$ Predicate

The only problem with the process we have defined in Figure 3 is that it is not entirely obvious that the $\text{independence}_{r,2}$ operation, as we have currently defined it, is computable. Clearly, if $\text{independence}_{r,2}$ or even $\neg \text{independence}_{r,2}$ were computable then $\text{independence}_{r,2}$ would be computable. We therefore present a definition of $\neg \text{independence}_{r,2}$ whose computability we can prove. However, before we do this, we will define several terms that will make our discussion much easier.

- $t,t_1,t_2,...$ — These symbols are used to denote arbitrary SDT’s. Note, these SDT’s may have nonterminal symbols in their leaf positions.
- $x_i$ — As we mentioned earlier, $x_i$ is an arbitrary program belonging to $X_i$. We mention $x_i$ here because we want to remind the reader that $x_i$ is also a SDT.
- $y,y'$ — These symbols are used to denote arbitrary SDT’s whose leaves must consist solely of terminals.
- $\text{error}$ — This denotes a special kind of SDT. In particular, the tree $\text{error}$ is not an SDT belonging to the language Poly.
- $t_1 = t_2$ — This expression is true if the SDT’s $t_1$ and $t_2$ are syntactically identical and false otherwise.
- $t[t_1]$ — This expression denotes the tree $t_1$, if $t_1$ is a proper subtree of $t$ and denotes $\text{error}$ otherwise.
- $t(t_1)$ — This expression evaluates to true iff $t[t_1] \neq \text{error}$. In other words $t[t_1]$ is true only when $t_1$ is a proper subtree of $t$.
- $t(t_1)$ — This expression denotes the tree $t_1$ if $t_1$ is a subtree of $t$ (i.e., either $t = t_1$ or $t[t_1]$) and denotes $\text{error}$ otherwise.
- $t(t_1)$ — This expression evaluates to true iff $t(t_1) \neq \text{error}$. In other words $t(t_1)$ is true only when $t_1$ is a subtree of $t$.
- $t_1/t_2$ — The tree $t_1$ is to be replaced with $t_2$.
- $t(t_1/t_2)$ — If $t[t_1] \neq \text{error}$, this expression denotes the tree that is created by taking the tree $t$ and replacing $t_1$ with $t_2$. If $t[t_1] = \text{error}$, then the expression $t(t_1/t_2)$ also denotes $\text{error}$.
- $t_1 \text{unifies } t_2$ — This boolean expression evaluates to true if either $t_1$ and $t_2$ are syntactically identical, or if $t_1$ can be made equal to $t_2$ by instantiating the schema variables of $t_1$ according to the rules of the grammar under consideration and false otherwise. Essentially what happens is that the schema variables occurring in $t_1$ are unified with the appropriate subtrees in $t_2$. Note, since $\text{error}$ is outside of Poly, no SDT in Poly can ever unify with $\text{error}$.
- $\text{depth}(t)$ — The length of the longest path from the root of $t$ to a leaf of $t$.
- $\text{restricted}(t,T_{i,\sim})$ — The predicate $\text{restricted}$ is true if $\forall x_i : \neg x_i(t)$, and false otherwise. What this is saying is that $\text{restricted}(t,T_{i,\sim})$ is true when $t$ does not occur in any program belonging to the set $X_i$.

### 7.5.1 An Algorithm for $\neg \text{independence}_{r,2}$

With the help of the various terms we have just defined we are now in a position to give a formal definition of $\neg \text{independence}_{r,2}$ that is computable. However, before we do this, we would like to informally describe what our algorithm is doing.

The problem we are concerned with is detecting whether $\neg \text{independence}_{r,2}(T_i,T_{i,\sim})$ holds when $\text{independence}_{r,2}(T_{i,\sim})$ holds. (Note that because $\text{independence}_{r,2}(T_{i,0})$ holds for all transformation sequences, this implies that for any transformation sequence $T_{i,n}$ for which the property $\text{independence}_{r,2}(T_{i,n})$ does not hold, there must exist an $i$ such that $\text{independence}_{r,2}(T_{i,\sim})$ and $\neg \text{independence}_{r,2}(T_{i-1}\sim)$. If $\neg \text{independence}_{r,2}(T_{i-1}\sim)$ holds, this means that for some program in $X_i$, the act of rewriting $t_{i,in}$ to $t_{i,out}$ creates an instance of an input schema of one of the transformations in $T_{i,\sim}$. In other words, there exists a program $x_i$ containing a smallest subtree $t$ that is an instance of $t_{i,in}$ (i.e., $t_{i,in} \text{unifies } t$), such that when $t$ is transformed by $T_{i}^1$ an instance of $t_{i,in}$ is produced for $1 \leq j < i$.

Let $t_j$ denote the instance of $t_{j,in}$ that is produced (i.e., $t_{j,in} \text{unifies } t_j$). Given the program $x_i(t/T_{i}^1(t))$ there are two cases that need to be considered: $1)$ $T_{i}^1(t)$ is a proper subtree of $t_1$, and $2)$ $t_1$ is a subtree of $T_{i}^1(t)$. In either case, it should be clear that $t_1$ could not have been a subtree of $x_i$ (i.e., $x_i(t_1) = \text{error}$) prior to $T_i^1(t)$ because of the assumption that $\text{independence}_{r,2}(T_{i,\sim})$ holds. What this means is that there must be some overlap between the $t_{i,in}$ portion of $t$ and the $t_{j,in}$ portion of $t_1$. The fact that this overlap must occur allows us to place the following depth restriction on the tree $t$ that needs to be considered:

$$\text{depth}(t) \leq \max(\text{depth}(t_{i,in}), \text{depth}(t_{j,in}))$$
Chapter 7 Proving the Correctness of Transformation Sequences

Our algorithm for \( \neg \text{independent}_{t,2} \) essentially consists of searching for a program \( x_i \) containing \( t \) such that the program \( T_i^1(x_i) \) contains \( t_1 \) where \( t \) and \( t_1 \) have the properties we have just discussed. Note that we have intentionally written \( T_i^1(x_i) \) instead of \( T_i^1(t) \). Recall that if \( \neg \text{restricted} (t', T_{i-1}) \) holds, there exists a program \( x_i \) that contains the tree \( t' \). Thus if we could compute \( \text{restricted} (t, T_{i-1}) \), its evaluation would determine whether or not there exists a program \( x_i \) containing \( t \). Now all that remains is to search through all possible SDT's satisfying the depth bound given above for a tree \( t \) that satisfies \( \neg \text{restricted} (t, T_{i-1}) \), and has the desired properties. In this discussion, the only operation whose computability is questionable is the \( \text{restricted} \) predicate. For the time being we will assume that the \( \text{restricted} \) predicate is computable. However, later in this chapter, a significant amount of effort will be devoted to proving the computability of this predicate.

We now give the formal description of an algorithm for computing \( \neg \text{independent}_{t,2} \).

\[
\neg \text{independent}_{t,2}(T_i, T_{i-1}) \overset{\text{def}}{=} \exists t, i, j : (1 \leq j < i) \land \text{depth}(t) \leq \max(\text{depth}(t_{i,\text{in}}), \text{depth}(t_{j,\text{in}})) \land (t_{i,\text{in}} \overset{\text{unifies}}{\rightarrow} t) \land \\
(\neg \text{restricted}(t_{j,\text{in}} \{ t_{1}/t \}, T_{i-1}) \land (t_{j,\text{in}} \overset{\text{unifies}}{\rightarrow} t_{j,\text{in}} \{ t_{1}/T_1^1(t) \}) ) \\
\lor \\
(\neg \text{restricted}(t, T_{i-1}) \land t_{j,\text{in}} \overset{\text{unifies}}{\rightarrow} T_1^1(t)(t_{1}))
\]

Figure 4

Having given an algorithm for computing \( \neg \text{independent}_{t,2} \) we need to prove that this algorithm is equivalent to the negation of our original definition of \( \text{independent}_{t,2} \).

**Proof**

- \( (\exists x_i, j : (1 \leq j < i) \land (T_i^1(x_i) \neq T_j^1(T_i^1(x_i)))) \)
  
  \( \equiv \)

- \( (\exists x_i, j, y, \forall y' : x_i(|y|) \land (T_i^1(y) \neq T_j^1(T_i^1(y))) \land (y[|y'|] \Rightarrow (T_i^1(y') = T_j^1(T_i^1(y')))) \)

  - Case 1. \( \exists y'' : T_j^1(y)[|y''|] \land (T_i^1(y)|y''/T_j^1(y'')) = T_j^1(T_i^1(y)) \)
    
    * Let \( k \overset{\text{def}}{=} j \)
    * Since \( t_{j,\text{in}}[t_1/t] \overset{\text{unifies}}{\rightarrow} y, t_{j,\text{in}}, y, \) and \( t_{i,\text{in}} \) provide enough information to obtain \( t \) and \( t_1 \).

  - Case 2. \( \exists y'' : T_i^1(y)[|y''|] \land (T_i^1(y)|y''/T_j^1(y'')) = T_j^1(T_i^1(y)) \)
    
    * Let \( k \overset{\text{def}}{=} j \)
    * Since \( t \overset{\text{unifies}}{\rightarrow} y'', t_{j,\text{in}}, y'', \) and \( t_{i,\text{in}} \) provide enough information to obtain \( t \) and \( t_1 \).

**Q.E.D.**

Having proven that the above (algorithmic) definition of \( \neg \text{independent}_{t,2} \) is equivalent to the original definition of \( \neg \text{independent}_{t,2} \), we will now informally prove that the process described in Figure 4 is indeed an algorithm.

Clearly, since \( t \) and \( j \) are bounded, the number of possible values they can acquire is finite. Also, since the input patterns of transformations are finite, and the transformation sequence \( T_{i-1} \) is finite, there are only a finite number of \( t_1 \)'s that need to be considered. In addition it should be easy to see that, with the exception of the \( \text{restricted} \) operation, all of the other operations in Figure 4 are computable. Thus, proving that the above definition of
\(-\text{independent}_{r,2}\) is an algorithm boils down to proving that the \(\text{restricted}\) operation is computable.

### 7.6 The Computability of the \(\text{restricted}\) Operation

#### 7.6.1 Overview

In this section, we will prove that the \(\text{restricted}\) operation is computable by giving an algorithm for computing it. For the sake of readability we describe this algorithm informally whenever possible and will only resort to formal notation and proofs when absolutely necessary.

The \(\text{restricted}\) operation can be computed by four inference rules. These inferences rules operate on a grammar \(G\) and a transformation sequence \(T_{i,\sim-1}\), where \(\text{independence}_{r,2}(T_{i,\sim-1})\) holds, and can be used to determine whether \(\text{restricted} (t, T_{i,\sim-1})\) holds for an arbitrary \(t\). Determining whether \(\text{restricted} (t, T_{i,\sim-1})\) holds can be accomplished in the traditional fashion of resolution style theorem provers. That is, in order to show \(\text{restricted} (t, T_{i,\sim-1})\) holds one assumes \(\neg\text{restricted} (t, T_{i,\sim-1})\) holds and, using the four inference rules mentioned above, tries to find a contradiction. If a contradiction is found, then one can conclude that \(\text{restricted} (t, T_{i,\sim-1})\) holds. This proof-by-contradiction approach has the property that if \(\text{restricted} (t, T_{i,\sim-1})\) does not hold, then no contradiction will be found. The difficulty with this is that not finding a contradiction for \(\text{restricted} (t, T_{i,\sim-1})\) does not necessarily imply that \(\neg\text{restricted} (t, T_{i,\sim-1})\) does hold. All that can be deduced from the fact that one has not found a contradiction is that either 1) \(\neg\text{restricted} (t, T_{i,\sim-1})\) does indeed hold, or 2) a contradiction has not yet been found and that running the theorem prover a little longer could yield a contradiction. Be that as it may, the inference rules we present have the property that a bound can be placed on any proof. What this means is that after a certain number of deductions have been made, if a contradiction for \(\neg\text{restricted} (t, T_{i,\sim-1})\) has not been found, this will indeed imply that \(\neg\text{restricted} (t, T_{i,\sim-1})\) holds. It is because of this bound that we can use the resolution based theorem proving process as an algorithm for computing \(\text{restricted} (t, T_{i,\sim-1})\).

#### 7.6.2 Four Inference Rules

Having informally outlined how we intend to compute the \(\text{restricted}\) operation, we are in a position to give the inference rules that will be used to perform the computation. We begin by defining several terms that will make our discussion much easier.

- \(a, a_1, a_2, ...\) — variables denoting nonterminal symbols.
- \(b, b_1, b_2, ...\) — variables denoting nonterminal symbols.
- \(w, w_1, w_2, ...\) — variables denoting terminal or nonterminal symbols.
- \(\alpha, \beta, \gamma\) — variables denoting arbitrary strings, of length greater than or equal to zero, consisting of nonterminal and terminal symbols. In Chapter 5 these strings were called phrases and completed phrases.
- \(b \Rightarrow \alpha\) — An arbitrary production in the grammar \(G\).
- \(a \Rightarrow_G \alpha\) — The nonterminal denoted by the variable \(a\) can derive the string \(\alpha\) using productions from the grammar \(G\).
- \(\neg(a \Rightarrow_G \alpha)\) — Using the productions from the grammar \(G\) it is not possible for \(a\) to derive \(\alpha\).
- \(a\{\alpha\}\) — This is the notation we gave in Chapter 5 for SDT’s. Note, since these expressions denote SDT’s, they can be used in place of any symbol (e.g., \(t, x, y\) etc.) or string of symbols (e.g., \(\alpha, \beta\) etc.) denoting an SDT. In particular, it is very important to note that if \(a_1 \Rightarrow \alpha \beta\) and \(a_2 \Rightarrow \gamma\) then \(a_1\{\alpha \beta \gamma\}\) can be equivalently written as \(a_1\{\alpha a_2\gamma\beta\}\). Also, since \(a_1 \Rightarrow a_1 a_1\{a_1\}\) is a proper notation for an SDT consisting of the single schema \(a_1\).
- \(a_{j,\text{in}}\{\alpha\}\) — This expression denotes the SDT for the input schema of the transformation \(T_j\). That is, \(a_{j,\text{in}}\{\alpha\} \overset{\text{def}}{=} t_{j,\text{in}}\).
- \(\neg a\{\alpha\}\) — This is the notation we use to denote that the four resolution rules given below are able to deduce that the schema \(a\{\alpha\}\) is restricted. Note, the expression \(\neg a\{\alpha\}\) does not make any mention of the transformation sequence \(T_{i,\sim-1}\). This does not present any problem, because the assumption here is that we are in the process of trying to determine whether \(\neg\text{independent}_{r,2}(T_i, T_{i,\sim-1})\) holds. Thus \(T_{i,\sim-1}\) is implied. As we will see \(\neg a\{\alpha\} \Rightarrow \text{restricted} (a\{\alpha\}, T_{i,\sim-1})\), but \(\neg(\text{restricted} (a\{\alpha\}, T_{i,\sim-1}) \Rightarrow \neg a\{\alpha\})\).
Chapter 7 Proving the Correctness of Transformation Sequences

- $+a\{\alpha\}$ — This is the notation we use to denote that the four resolution rules (given below) cannot deduce $-a\{\alpha\}$. Here it is the case that, $-a\{\alpha\} = +a\{\alpha\}$. As with the expression $-a\{\alpha\}$ the transformation sequence $T_{1,i-1}$ is implied.

- completed_phrase ($\alpha$) — This predicate evaluates to true iff $\alpha$ is a string containing no nonterminal symbols.

- useless ($a\{\gamma\}$) def $\neg\exists \alpha, \beta, \gamma': +a\{\gamma\} \land (S \Rightarrow \alpha \beta) \land (\gamma \Rightarrow \gamma')\land$completed_phrase ($\alpha\gamma'\beta$) $\land +S\{\alpha\gamma'\beta\}$, where $S$ is the start symbol of the grammar.

- $\mathcal{R}_{i,i-1}$ — the basic set of restricted patterns created by $T_{1,i-1}$ when independence$_{G}(T_{1,i-1})$ holds.

$$\mathcal{R}_{i,i-1} = \{ -a_{j\in \alpha}\{\alpha\}| 1 \leq j < i \}. $$

**Inference Rule 1: D-resolution.**

$-a\{ab\beta\} \Rightarrow (\forall \gamma: (b \rightarrow _G \gamma \Rightarrow -a\{ab\gamma\\beta\}))$

**Inference Rule 2: Reverse-D-resolution.**

$$(\forall \gamma: (b \rightarrow _G \gamma \Rightarrow -a\{ab\gamma\\beta\})) \Rightarrow -a\{ab\beta\}$$

**Inference Rule 3: A-resolution.**

$-b\{\gamma\} \Rightarrow (\forall a: (a \rightarrow _G ab\beta \Rightarrow -a\{ab\gamma\\beta\}))$

**Inference Rule 4: Reverse-A-resolution.**

$$(\exists a: a \rightarrow _G \alpha\beta) \land (\forall a: (a \rightarrow _G ab\beta \Rightarrow -a\{ab\gamma\\beta\})) \Rightarrow -b\{\gamma\}$$

**Theorem 9** D-resolution is sound.

**Proof:** Clearly, if $a\{ab\beta\}$ is restricted, then the schema $a\{ab\gamma\\beta\}$ which is an instance of $a\{ab\beta\}$ must also be restricted.

**Theorem 10** Reverse-D-resolution is sound.

**Proof (by contradiction):** Assume that $(\forall \gamma: b \rightarrow _G \gamma \Rightarrow -a\{ab\gamma\\beta\})$ but that $a\{ab\beta\}$ is not restricted. If $a\{ab\beta\}$ is not restricted, then by definition there must exist a program $x_i$ such that $x_i([a\{ab\beta\}])$. Since $b$ is a nonterminal there must exist a derivation of the form $b \Rightarrow \gamma'$ where $\gamma'$ is a complete-phrase. This derivation must have $b \Rightarrow \gamma$ as a first step where $b \rightarrow _G \gamma$. This contradicts our assumption that $(\forall \gamma: b \rightarrow _G \gamma \Rightarrow -a\{ab\gamma\\beta\})$. Q.E.D.

**Theorem 11** A-resolution is sound.

**Proof:** Clearly, if the schema $b\{\gamma\}$ is restricted, then any schema containing $b\{\gamma\}$ as a subschema must also be restricted.

**Theorem 12** Reverse-A-resolution is sound.

**Proof (by contradiction):** Assume that $(\exists a: a \rightarrow _G \alpha\beta) \land (\forall a: a \rightarrow _G ab\beta \Rightarrow -a\{ab\gamma\\beta\})$ but that $b\{\gamma\}$ is not restricted. If $b\{\gamma\}$ is not restricted, then by definition there must exist a program $x_i$ such that $x_i([b\{\gamma\}])$. (Note, because $(\exists a: a \rightarrow _G ab\beta)$ we know that $b$ cannot be the start symbol of $G$.) Since $b$ is not the start symbol, this means that there must exist some production $a \rightarrow _G ab\beta$ deriving $b\{\gamma\}$ in $x_i$. Thus there must exist some schema $a\{ab\gamma\\beta\}$ that is not restricted. This contradicts our assumption that $(\forall a: a \rightarrow _G ab\beta \Rightarrow -a\{ab\gamma\\beta\})$. Q.E.D.

Having proven that the above inference rules are sound, what remains to be done is to prove their completeness. Unfortunately, the inference rules as stated are not quite complete. What this means is that there can exist schemas $a\{\alpha\}$ for which restricted ($a\{\alpha\}, T_{1,i-1}$) holds, but the inference rules cannot deduce $-a\{\alpha\}$. Fortunately, it turns
out that one can prove that restricted \((a \{ \alpha \}, T_{i,i-1})\) holds iff the expression \(-a \{ \alpha \} \lor useless (a \{ \alpha \})\) holds. Without proof, we claim that given the ability to compute \(+a \{ \alpha \}\) the computation of useless \((a \{ \alpha \})\) becomes straightforward and is essentially an adaptation of the algorithm that is used, in formal language theory, to remove “useless” symbols from a grammar. Thus for the sake of brevity, we will leave the discovery of an algorithm capable of computing useless \((a \{ \alpha \})\) to future work. However, in this thesis, we will prove that a bound can be placed on the inference rules that permit the deduction of \(+a \{ \alpha \}\) in addition to \(-a \{ \alpha \}\).

**Theorem 13** (Completeness): \(-a \{ \alpha \} \lor useless(a \{ \alpha \}) \equiv restricted (a \{ \alpha \}, T_{i,i-1})\).

**Proof:**

**Part 1:** \(-a \{ \alpha \} \lor useless(a \{ \alpha \}) \Rightarrow restricted (a \{ \alpha \}, T_{i,i-1})\)

- Proof by contradiction. Assume \(-a \{ \alpha \} \lor useless(a \{ \alpha \})\) and \(!restricted (a \{ \alpha \}, T_{i,i-1})\)
  
  (a) Case 1: \(-a \{ \alpha \}\) holds.
  
  \(-a \{ \alpha \} \Rightarrow \exists x : x(a \{ \alpha \})\). This contradicts \(!restricted (a \{ \alpha \}, T_{i,i-1})\).

(b) Case 2: \(+a \{ \alpha \} \land useless(a \{ \alpha \})\) holds.

\(!restricted (a \{ \alpha \}, T_{i,i-1}) \Rightarrow \exists x : x(a \{ \alpha \})\). This contradicts useless \((a \{ \alpha \})\).

**Part 2:** restricted \((a \{ \alpha \}, T_{i,i-1}) \Rightarrow -a \{ \alpha \} \lor useless(a \{ \alpha \})\)

- Proof by contradiction. Assume restricted \((a \{ \alpha \}, T_{i,i-1})\) and \(!(-a \{ \alpha \} \lor useless(a \{ \alpha \}))\).

\(!(-a \{ \alpha \} \lor useless(a \{ \alpha \})) = +a \{ \alpha \} \land -useless(a \{ \alpha \})\). This contradicts restricted \((a \{ \alpha \}, T_{i,i-1})\).

Q.E.D.

### 7.6.3 Bounding the search for \(-a \{ \alpha \}\)

In this section, we give a bound that can be used to limit the search for the proof of \(-a \{ \alpha \}\). This means that if \(-a \{ \alpha \}\) cannot be deduced within the bound, then \(-a \{ \alpha \}\) will not be deducible and we can conclude \(+a \{ \alpha \}\). The ability to determine whether \(-a \{ \alpha \}\) or \(+a \{ \alpha \}\) holds in conjunction with useless \((a \{ \alpha \})\) provides us with a means to compute restricted \((a \{ \alpha \}, T_{i,i-1})\) thereby enabling the computation of \(-independent_{\alpha_2}(T_i, T_{i-1})\).

We begin by defining several terms that will make our discussion much easier.

- \(-a \{ \alpha \}\) — In this section, we will refer to expressions of the form \(-a \{ \alpha \}\) as clauses.
- \(-a \{ \alpha \}_m\) — Clauses may be subscripted by nonnegative integers. When a clause is subscripted, the value of the subscript denotes the depth of the clause. For example, if we write \(-a \{ \alpha \}_m\), this means that depth \((a \{ \alpha \}) = m\). The subscript on a clause is only used as a convenient means to provide the reader with the depth of the clause, and serves no other purpose.
- \(\alpha_m\) — Strings can be subscripted in a fashion similar to clauses. If \(a \Rightarrow_\alpha \), then in general \(\alpha\) will denote a forest of SDT’s. The subscript on \(\alpha\) denotes the depth of the largest SDT in \(\alpha\).
- \(C_{1,i}\) — This symbol denotes the set of clauses in \(R_{1,i}\). That is, \(C_{1,i} = R_{1,i}\).
- \(\text{max}_\text{depth}(C_{1,i})\) — Given a set of clauses \(C_{1,i}\), \text{max}_\text{depth} returns the depth of the largest clause in \(C_{1,i}\).
- \(\text{deduction}\) — A deduction is the application of one of the four inference rules (i.e., A-resolution, D-resolution, Reverse-A-resolution, or Reverse-D-resolution) to one or more clauses in order to obtain a new clause. Since we are working with four inference rules, there are four possible types of deductions:
  
  - \(\vdash\) — This symbol denotes the deduction of a clause using the A-resolution inference rule. Note, the A-resolution rule is applied to a single input clause and deduces a single output clause. We use \(\vdash\) to show
that only a single input clause participates in the deduction. For example, suppose \( a \rightarrow ab\beta \), and \( b \Rightarrow \gamma \). In this case A-resolution rule can be applied to the input clause \(-b\{\gamma\}\) deducing the output clause \(-a\{ab\{\gamma\}\}\beta\). Formally, we would write this as \( -b\{\gamma\} \overset{A}{\Rightarrow} -a\{ab\{\gamma\}\}\beta \).

Suppose now that we knew that \( \text{depth}(b\{\gamma\}) = n \). We could now write \( -a\{ab\{\gamma\}\}\beta \) and include all of the information that we know about the depth of the various parts of the clause. This would give us \( -a\{(\alpha)ab\{\gamma\}\}_n(\beta)_b \) where \( \alpha \) is the depth of \( a \) and \( \beta \) is the depth of \( b \).

- \( \vdash \) — This symbol denotes the deduction of a clause using the D-resolution inference rule. Note, the D-resolution rule is applied to a single input clause and deduces a single output clause. We use \( \vdash \) to show that only a single input clause participates in the deduction.

For example, suppose \( a_1 \overset{G}{\rightarrow} a_2 \beta \), and \( b \Rightarrow \gamma_1 \). In this case D-resolution rule can be applied to the input clause \( -b\{\gamma_1\} \) deducing the output clause \( -a_1\{a_2\beta\}\gamma_2 \). Formally, we would write this as \( -b\{\gamma_1\} \vdash -a_1\{a_2\beta\}\gamma_2 \). Since \( a_1 \overset{G}{\rightarrow} a_2 \beta \) we know that \( \text{depth}(a_1\{a_2\beta\}) = 1 \).

If we wanted to include this depth information, we could have written \( -b\{\gamma_1\}a_1\{a_2\beta\}\gamma_2 \) instead of \( -b\{\gamma_1\}a_1\{a_2\beta\}\gamma_2 \).

- \( \models \) — This symbol denotes the deduction of a clause using the Reverse-A-resolution inference rule. Note, the Reverse-A-resolution rule is, in general, applied to a set of input clauses and deduces a single output clause. We use \( \models \) to show that a set of input clauses participate in the deduction. For example, let \( a_i \overset{G}{\rightarrow} a_i b \gamma_i \) denote the entire set of productions in \( G \) in which the nonterminal symbol \( b \) occurs on the right hand side. That is, \( a_1 \overset{G}{\rightarrow} a_1 b \gamma_1 \) denotes the first production, \( a_2 \overset{G}{\rightarrow} a_2 b \gamma_2 \) denotes the second production and so on. Let \( b \Rightarrow \gamma \) and let \( -a_i\{a_i b\{\gamma\}\}\beta_i \) denote the set of clauses corresponding to \( a_i \overset{G}{\rightarrow} a_i b \gamma_i \). That is \( -a_1\{a_1 b\{\gamma\}\}\beta_1 \) corresponds to \( a_1 \overset{G}{\rightarrow} a_1 b \gamma_1 \), \( -a_2\{a_2 b\{\gamma\}\}\beta_2 \) corresponds to \( a_2 \overset{G}{\rightarrow} a_2 b \gamma_2 \), and so on. The Reverse-A-resolution rule can be applied to the set of clauses \( -a_i\{a_i b\{\gamma\}\}\beta_i \) deducing the output clause \( -b\{\gamma\} \). Formally this would be written: \( -a_i\{a_i b\{\gamma\}\}\beta_i \models -b\{\gamma\} \). If we wanted to include depth information we could have written \( -a_i\{a_i b\{\gamma\}\}_{i} \beta_i \) instead of \( -a_i\{a_i b\{\gamma\}\} \).

- \( \models \) — This symbol denotes the deduction of a clause using the Reverse-D-resolution inference rule. Note, the Reverse-D-resolution rule is, in general, applied to a set of input clauses and deduces a single output clause. We use \( \models \) to show that a set of input clauses participate in the deduction. For example, let \( b \overset{G}{\rightarrow} \gamma_i \) denote the entire set of productions in \( G \) in which the nonterminal symbol \( b \) occurs on the right hand side. That is, \( b \overset{G}{\rightarrow} \gamma_i \) denotes the first production, \( b \overset{G}{\rightarrow} \gamma_2 \) denotes the second production and so on. Suppose that for each production in \( b \overset{G}{\rightarrow} \gamma_i \), there exists a clause of the form \( -a\{ab\{\gamma_i\}\}\beta \) in \( G \). If this is the case, then all the clauses of the form \( -a\{ab\{\gamma_i\}\}\beta \) can be given as input to the Reverse-D-resolution rule allowing the deduction of \( -a\{ab\beta\} \). Formally this would be written: \( -a\{ab\{\gamma_i\}\}\beta \models -a\{ab\beta\} \). If we wanted to include depth information we could have written \( -a\{ab\{\gamma_i\}_i\}\beta \) instead of \( -a\{ab\{\gamma_i\}\}\beta \).

- deduction sequence — A deduction sequence consists of zero or more deductions. The deductions in the sequence need not be based on the same inference rule; however when they are, we will explicitly mention this. For example, a deduction sequence consisting entirely of D-resolution steps will be denoted \( \overset{D}{\vdash} \). A deduction sequence consisting entirely of RA-resolution steps will be denoted \( \overset{RA}{\models} \), and so on. Through concatenation, deduction sequences can be composed with single deductions or other deduction sequences. For example, the deduction sequence consisting of a single A-resolution step followed by a sequence of D-resolution deductions is denoted \( \overset{A}{\vdash} \overset{D}{\vdash} \).
Section 7.6  The Computability of the restricted Operation

Theorem 14  Given an initial set of clauses $C$, such that max_depth($C$) < $m$, where $m > 1$, if there exists a deduction sequence containing the deduction step:

$$-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m \overset{\mu^*}{\vDash} -a\{\alpha\}_{m-1}$$

then there must also exist a deduction sequence, having clauses of depth no greater than $m - 1$, which also deduces $-a\{\alpha\}_{m-1}$.

Proof:

- Since our original set of clauses consisted only of clauses having depth less than $m$, it follows that every clause of the form:

$$-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m$$

must have been deduced from a clause of depth $m - 1$ through an $\overset{\mu^*}{\vDash}$ or a $\overset{\mu^*}{\vdash}$ deduction sequence.

- Let $-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m$ denote the subset of clauses of the form $-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m$ that were obtained from a clause of depth $m - 1$ through an $\overset{\mu^*}{\vDash}$ deduction sequence, and let $-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m$ denote the subset of clauses that were obtained using a $\overset{\mu^*}{\vdash}$ deduction sequence.

- Let $a\{\alpha\}'$ denote the largest subtree of $a\{\alpha\}$, including the root of $a\{\alpha\}$, and satisfying depth ($a\{\alpha\}'$) = $m - 2$.

1. For clauses of the form $-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m$, let $b_i \rightarrow_G w_1w_2...w_r$, where $w_r$ denote the grammar production that is used in the $\overset{\mu^*}{\vdash}$ deduction step of the sequence $\overset{\mu^*}{\vDash}$ responsible for deducing $-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m$. There are two cases that need to be considered.

   a) A clause of the form $-a\{\alpha\}_{m-1}$ is the basis of the $\overset{\mu^*}{\vdash}$ deduction step. This cannot be the case because depth ($w_r$) = 0.

   b) The clause $-a\{\alpha\}_{m-1}$ is the basis of the $\overset{\mu^*}{\vdash}$ deduction step. In this case $-a\{\alpha\}_{m-1}$ existed prior to the deduction sequence, and therefore the deduction sequence is unnecessary.

2. Since every clause of the form $-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m$ was deduced from a clause of depth $m - 1$ through a $\overset{\mu^*}{\vdash}$ deduction sequence, it follows that for every clause of the form:

$$-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m$$

there must also exist a clause of the form:

$$-b_i\{\beta_{1,i}\}a\{\alpha\}'_{m-2}(\beta_{2,i})a_{m-1}$$

What we have been able to establish is that if $-a\{\alpha\}_{m-1}$ doesn’t already hold (as it did in 1.), then clauses of the form:

$$-b_i\{\beta_{1,i}\}a\{\alpha\}'_{m-2}(\beta_{2,i})a_{m-1}$$

must have been deduced in order to enable deductions of clauses having the form:

$$-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m$$

However, since $-b_i\{\beta_{1,i}\}a\{\alpha\}_{m-1}(\beta_{2,i})a_m \overset{\mu^*}{\vDash} -a\{\alpha\}_{m-1}$ is possible, it follows that

$$-b_i\{\beta_{1,i}\}a\{\alpha\}'_{m-2}(\beta_{2,i})a_{m-1} \overset{\mu^*}{\vDash} -a\{\alpha\}_{m-1}$$

must also be possible.

Q.E.D.
Theorem 15  Given an initial set of clauses $C$, such that $\text{max\_depth}(C) < m$, where $m > 1$, if there exists a deduction sequence containing the deduction step:

$$-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m \overset{\alpha^*}{\rightarrow} -b\{\beta_1 a_1 \{ \gamma_1 a \gamma_2 \}_{m-2} \beta_2 \}_{m-1}$$

then there must also exist a deduction sequence, having clauses of depth no greater than $m - 1$, which also deduces $-b\{\beta_1 a_1 \{ \gamma_1 a \gamma_2 \}_{m-2} \beta_2 \}_{m-1}$.

Proof:

1. For clauses of the form $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ denote the subset of clauses of the form $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ that were obtained from a clause of depth $m - 1$ through an $\overset{\alpha^*}{\rightarrow}$ deduction sequence, and let $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ denote the subset of clauses that were obtained using a $\overset{\beta^*}{\rightarrow}$ deduction sequence.

2. Since our original set of clauses consisted only of clauses having depth less than $m$, it follows that every clause of the form:

$$-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$$

must have been deduced from a clause of depth $m - 1$ through an $\overset{\alpha^*}{\rightarrow}$ or a $\overset{\beta^*}{\rightarrow}$ deduction sequence.

3. Let $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ denote the subset of clauses of the form $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ that were obtained from a clause of depth $m - 1$ through an $\overset{\alpha^*}{\rightarrow}$ deduction sequence, and let $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ denote the subset of clauses that were obtained using a $\overset{\beta^*}{\rightarrow}$ deduction sequence.

4. Given an initial set of clauses $C$, such that $\text{max\_depth}(C) < m$, where $m > 1$, if there exists a deduction sequence containing the deduction step:

$$-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m \overset{\alpha^*}{\rightarrow} -b\{\beta_1 a_1 \{ \gamma_1 a \gamma_2 \}_{m-2} \beta_2 \}_{m-1}$$

then there must also exist a deduction sequence, having clauses of depth no greater than $m - 1$, which also deduces $-b\{\beta_1 a_1 \{ \gamma_1 a \gamma_2 \}_{m-2} \beta_2 \}_{m-1}$.

Proof:

1. For clauses of the form $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ denote the subset of clauses of the form $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ that were obtained from a clause of depth $m - 1$ through an $\overset{\alpha^*}{\rightarrow}$ deduction sequence, and let $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ denote the subset of clauses that were obtained using a $\overset{\beta^*}{\rightarrow}$ deduction sequence.

2. Since our original set of clauses consisted only of clauses having depth less than $m$, it follows that every clause of the form:

$$-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$$

must have been deduced from a clause of depth $m - 1$ through an $\overset{\alpha^*}{\rightarrow}$ or a $\overset{\beta^*}{\rightarrow}$ deduction sequence.

3. Let $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ denote the subset of clauses of the form $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ that were obtained from a clause of depth $m - 1$ through an $\overset{\alpha^*}{\rightarrow}$ deduction sequence, and let $-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m$ denote the subset of clauses that were obtained using a $\overset{\beta^*}{\rightarrow}$ deduction sequence.

4. Given an initial set of clauses $C$, such that $\text{max\_depth}(C) < m$, where $m > 1$, if there exists a deduction sequence containing the deduction step:

$$-b\{\beta_1 a_1 \{ \gamma_1 a \{ \alpha_i \} \gamma_2 \}_{m-1} \beta_2 \}_m \overset{\alpha^*}{\rightarrow} -b\{\beta_1 a_1 \{ \gamma_1 a \gamma_2 \}_{m-2} \beta_2 \}_{m-1}$$

then there must also exist a deduction sequence, having clauses of depth no greater than $m - 1$, which also deduces $-b\{\beta_1 a_1 \{ \gamma_1 a \gamma_2 \}_{m-2} \beta_2 \}_{m-1}$.
Q.E.D.

**Theorem 16** If $R \vdash \neg a\{\alpha\}$ then there exists a proof of this containing patterns having depth no greater than $\max(\max(\max_{depth}(R), \text{depth}(a\{\alpha\})), 1)$.

**Proof:** (by contradiction)

There are two cases that need to be considered here. The first case is where

\[ \max(\max_{depth}(R), \text{depth}(a\{\alpha\})) = 0, \]

and the second case is where

\[ \max(\max_{depth}(R), \text{depth}((a, \alpha))) > 0. \]

Assume that $R \vdash \neg a\{\alpha\}$, and that all proofs contain at least one pattern that has a depth greater than $\max(\max_{depth}(R), \text{depth}(a\{\alpha\}))$. Let $n$ denote $\max(\max_{depth}(R), \text{depth}(a\{\alpha\}))$. Since by assumption $R \vdash \neg a\{\alpha\}$, there must exist one or more deduction sequences proving $\neg a\{\alpha\}$. For these deduction sequences there must exist a value $m$ having the property that at least one proof of $\neg a\{\alpha\}$ exists such that every pattern in the deduction sequence has a depth less than or equal to $m$, and that every deduction sequence proving $\neg a\{\alpha\}$ contains at least one pattern greater than or equal to $m$. (For the rest of this proof when we refer to a deduction sequence proving $\neg a\{\alpha\}$ what we mean is a deduction sequence proving $\neg a\{\alpha\}$ in which every pattern in the sequence has depth less than or equal to $m$.) In addition, because we have assumed that all proofs contain at least one pattern that has a depth greater than $\max(\max_{depth}(R), \text{depth}(a\{\alpha\}))$, this implies that $n < m$.

Recall, A-resolution and D-resolution operate on a single clause in order to deduce a clause, and Reverse-A-resolution and Reverse-D-resolution operate on a set of clauses in order to deduce a clause. Since all of the inference rules deduce clauses having a depth that is at most one larger or one smaller than the pattern(s) they operate on, we can without loss of generality limit our attention to the case where $n = \max(\max_{depth}(R), \text{depth}(a\{\alpha\})) = \text{depth}(a\{\alpha\})$ and $m = n + 1$. In addition, we can further limit our attention to the case where $\neg a\{\alpha\}_n$ is deduced directly from $-b\{\beta\}$ (i.e., either $-b\{\beta\} \vdash^{\text{A}} \neg a\{\alpha\}_n$, or $-b\{\beta\} \vdash^{\text{R}} \neg a\{\alpha\}_n$), where $\text{depth}(b\{\beta\}) = m$. Theorem 14 shows that $-b\{\beta\} \vdash^{\text{A}} \neg a\{\alpha\}_n$ is unnecessary, and Theorem 15 shows that $-b\{\beta\} \vdash^{\text{R}} \neg a\{\alpha\}_n$ is unnecessary.

Q.E.D.

### 7.6.4 Summary

In this section we gave four inference rules that can be used to deduce restricted patterns (i.e., clauses) from an initial set of clauses. We then proceeded to prove that the inference rules were sound and, to a certain extent, complete. The completeness theorem states that a pattern $a\{\alpha\}$ is restricted with respect to a sequence of transformations, $T_{i, i-1}$, iff the expression $\neg a\{\alpha\} \lor \text{useless}(a\{\alpha\})$ evaluated to true. At this point we claimed, without proof, that the computation of $\text{useless}(a\{\alpha\})$ is straightforward but that the details of the computation remain to be worked out. We then focused our attention on showing that an algorithm exists for determining whether $\neg a\{\alpha\}$ holds. The problem encountered here is that traditional resolution based theorem proving techniques will only provide information as to the provability of $\neg a\{\alpha\}$ after a proof has been found. Thus, by not finding a contradiction (i.e., a proof), all that one can deduce is 1) a contradiction has not yet been found and running the theorem prover a little longer might yield a contradiction, or 2) a proof will never be found. This means that, using traditional methods, one will never be able to use the fact that a proof of $\neg a\{\alpha\}$ has not been found to conclude that $+a\{\alpha\}$ holds. By bounding the search for the proof of $\neg a\{\alpha\}$ we are able to prove $\neg a\{\alpha\}$ or $+a\{\alpha\}$ for all clauses in a finite amount of time. This ability, together with the $\text{useless}(a\{\alpha\})$ operation, provides us with an effective means to compute restricted $(a\{\alpha\}, T_{i, i-1})$.

### 7.7 Generalization

Up to this point, we have focused our attention on the problem of determining what properties hold for the set of
programs \( X_i \) to which the transformation \( T_i \) can be applied. The question now is: “How can one use the knowledge of the properties that hold for the programs in \( X_i \) in order to prove theorems about the behavior of \( T_i \)?” For example, how can one use the property \( Q'_i(x_i) \) to prove the theorem:

\[
Q'_i(x_i) \Rightarrow \text{Correct}(x_i, T_i(x_i)).
\]

The problem here is that the property \( Q'_i \) is defined in syntactic terms while the property of correctness is defined in semantic terms. Bridging the gap from the syntactic domain to the semantic domain is a problem that we leave to future research. However, in this section we will informally discuss, in abstract terms, how this might be accomplished.

Suppose we have a grammar \( G \) that contains the production \( p \rightarrow a \). Furthermore, suppose that \( a \) is a nonterminal symbol that can be generated only by the production \( p \rightarrow a \). That is, if we were to remove the production \( p \rightarrow a \) from \( G \), then the nonterminal \( a \) could not be generated from the start symbol of \( G \) (i.e., \( a \) would be a useless symbol). Now suppose we construct a program \( x \) belonging to \( L(G) \), and that \( x \) is a program whose SDT contains one or more occurrences of the nonterminal symbol \( a \). Suppose we now construct a transformation \( T \) that rewrites the SDT of \( x \) in such a way that the resulting program \( T(x) \) does not contain any instance of the nonterminal symbol \( a \). The program \( T(x) \) can be viewed as a program that was derived from the language \( G - \{ p \rightarrow z \} \). This being the case, any impact due to a delta-function caused by the production \( p \rightarrow a \) needs to be removed when constructing delta-functions describing the effects of nonterminal symbols occurring in the program \( T(x) \).

Abstractly, TAMPR transformations can be viewed as functions that enable or disable certain grammar derivations in particular schemas. Because the application of a transformation to a program enables or disables a certain grammar derivation in a particular schema, transformations can be viewed as altering the grammar that is used to derive the program. Since the semantics of delta functions are highly dependent on the semantics of the underlying grammar productions, TAMPR transformations are indirectly altering the semantics of delta functions. It is in this fashion that the semantic effects brought about by transformations generally surface.

Occasionally, transformations are written whose correctness can be proven only for delta-functions having a specific semantics. If the transformation \( T_i \) given above is such a transformation, then its correctness proof will need to make use of the syntactic information expressed by \( Q'_i \).

Generalization is the term we have given to the process of determining how syntactic properties like the ones defined in \( Q'_i \) can alter the semantics of delta functions, and as we have already mentioned we leave this area to future research.
Chapter 8
Practical Results

8.1 Overview

In this chapter, we demonstrate the practical value of our work by proving the correctness of a portion (i.e., a subsequence) of a transformation sequence, \( T_{k,n} \), whose overall goal is to transform functional specifications into Poly-Fortran implementations. The subsequence whose correctness we have proven is essentially capable of transforming specifications written in terms of lambda functions into programs written in recursive Poly-Fortran. This transformation subsequence we will be considering consists of 12 transformations. Collectively we will refer to these transformations as the \textit{lisptran} transformation sequence, and we will denote the \textit{lisptran} transformation sequence by \( T_{k,i+11} \). Note that this notation implies that the \textit{lisptran} transformations are preceded by \( T_{k,i+1} \). To help the reader get a feel for what the \textit{lisptran} transformation sequence accomplishes we include a sample input and output program of \( T_{k,i+11} \). That is, we include a program \( x_i \) and its transformed form \( T_{k,i+11}(x_i) \). The example we have chosen is a specification-implementation of a variation of the eight queens problem. Informally, the problem is stated as follows:

- List all possible ways to arrange eight queens on a chessboard so that none of the queens threatens any of the others.

The formal specification of the eight queens problem can be found in Appendix C and is quite a bit longer. The reason for this additional length is due to the fact that informal terms like 1) queens, 2) chessboard, 3) arrange, and 4) threatens need to be precisely defined.

Let \( x_1 \) denote the formal specification of the eight queens problem given in Appendix C. This specification, when transformed by the TAMPR transformation sequence, \( T_{k,i+1} \), produces an output program, \( x_i \), consisting of lambda function definitions. The transformed program, \( T_{k,i+11}(x_1) \), can be found in Appendix D. It should be noted that the output program \( T_{k,i+1}(x_1) \) is still in a highly functional form and is not significantly different from the original specification.

After the transformation sequence \( T_{k,i+1} \) is applied to the original specification, \( x_1 \), the transformation sequence \( T_{k,i+11} \) is then applied to the program \( T_{k,i+1}(x_1) \) producing the output program \( T_{k,i+11}(T_{k,i+1}(x_1)) \). The program \( T_{k,i+11}(T_{k,i+1}(x_1)) \) can be found in Appendix E. Using the methodology developed in this research, we have been able to prove the correctness of the transformation sequence \( T_{k,i+11} \) (i.e., the \textit{lisptran} transformation sequence) using the automated reasoning system OTTER. The major obstacles encountered in this endeavor were:

1. Expressing the denotational semantics of Poly in a manner acceptable to the OTTER reasoning system. This is not entirely trivial due to the fact that OTTER is a first order logic system while the denotational semantics definitions of Poly are higher order.
2. Expressing the SDT's of the input and output schemas in a manner consistent with the OTTER-denotational semantic representation.
3. Providing a means for expressing relationships between schema variables. For example if two or more schema variables, occurring within a single schema or occurring in the input and output schema of a transformation, denote the same instantiation, this relationship must be expressible.
4. Constructing an environment that permits OTTER to reason in terms of the \( \sqsubseteq \) relation.
Chapter 8  Practical Results

It should be noted that for all of the transformations in the sequence $T_{i,i+11}$ we were able to prove theorems of the form:

$$\text{true}(x_j) \Rightarrow \text{Correct}(x_j, T_j(x_j)).$$

In other words, to obtain correctness proofs we did not have to resort to any of the theory laid down in Chapter 7 concerning how to determine syntactic properties $Q'$ established by transformation sequences. This also implies that in order to prove the correctness of a transformation it sufficed to use OTTER to show:

$$\text{meaning}(x_j) \subseteq \text{meaning}(T_j(x_j)).$$

which, due to the monotonic nature of Poly, can be proven by showing:

$$\text{meaning}(t_{j,in}) \subseteq \text{meaning}(t_{j,out}).$$

For the transformations $T_{i+4}, T_{i+5}, T_{i+6}$, and $T_{i+9}$ this proof could be even further simplified to showing:

$$\text{meaning}(t_{j,in}) \equiv \text{meaning}(t_{j,out}).$$

The correctness proofs of the remaining transformations requires proving, for each transformation, that its output schema is a strict refinement of its input schema. For these transformations, the fact that the output schema is a strict refinement of the input schema is generally a consequence of the fact that some expression is evaluated in the output schema with respect to an environment, $\epsilon_{out}$, that is a strict refinement of the environment, $\epsilon_{in}$, used to evaluate the same expression in the input schema of the transformation.

In this chapter we use an example (i.e., the eight queens problem) as a means to informally convey the overall effect produced by the transformation sequence $T_{i,i+11}$. However, in order to give the reader a sense of continuity, we begin with the original specification, $x_1$, which we transform by the sequence $T_{i,i-1}$. The output program, $T_{i,i-1}(x_1)$, is then transformed by $T_{i,i+11}$ yielding the program $T_{i,i+11}(T_{i,i-1}(x_1))$. After the eight queens problem is examined we discuss how OTTER can be used to prove theorems of the form:

$$\text{true}(x_j) \Rightarrow \text{Correct}(x_j, T_j(x_j)).$$

Finally, we present some of the TAMR transformations whose correctness we have proved.

### 8.2 An Example: Eight Queens

In this section we restrict our attention to a small piece of the eight queens specification. However, it should be noted that the entire specification $x_1$ as well as the transformed programs $T_{i,i-1}(x_1)$ and $T_{i,i+11}(T_{i,i-1}(x_1))$ these can be found in Appendices C, D, and E respectively.

We realize that the specification given in Appendix C has some algorithmic components to it. We therefore would like to give, for the record, a specification of the eight queens problem containing no algorithmic components.

\[
\begin{align*}
\text{coordinate}_\text{component} & \equiv \{1, 2, 3, 4, 5, 6, 7, 8\} \\
\text{coordinate} & \equiv \{(x, y) \mid (x \in \text{coordinate}_\text{component}) \land (y \in \text{coordinate}_\text{component})\} \\
\text{placement}_\text{vector} & \equiv \{(p_1, p_2, \ldots, p_8) \mid (1 \leq i \leq 8) \rightarrow (p_i \in \text{coordinate})\} \\
\text{solution} & \equiv \{(p_1, p_2, \ldots, p_8) \mid \exists(x_i, y_i), (x_j, y_j), k_1, k_2 : \\
& \quad \quad \quad \quad (1 \leq k_1 \leq 8) \land \quad (1 \leq k_2 \leq 8) \land \quad ((x_i, y_i) = p_{k_1}) \land \quad ((x_j, y_j) = p_{k_2}) \land \quad (k_1 \neq k_2) \land \quad ((x_i = x_j) \lor (y_i = y_j) \lor (\text{abs}(x_i - x_j) = \text{abs}(y_i - y_j))) \}
\end{align*}
\]
where $abs$ is the absolute value function.

Note that the above specification is essentially in a “generate and test” form. That is, the solution set is a subset of the placement_vector set and can be obtained by exhaustive examination of the elements in the placement_vector set. It should be clear that with a little amount of reasoning the number of elements in the placement_vector set that need to be examined can be greatly reduced essentially leaving us with the specification given in Appendix C. Thus, while the specification in Appendix C does contain some algorithmic components, it is not as far removed from a pure specification as one might initially think.

Bear in mind, the objective of this research is to construct a methodology that will allow formal verification of the correctness of program transformations. In particular, in this dissertation we are not interested in algorithm design. This means that the amount of algorithmic components a specification might or might not have is irrelevant as far as this research is concerned. Having said this, we now proceed with our discussion of the eight queens specification.

The piece of the eight queens specification that we have chosen for our example is the place_queens_in_rows function and is defined in the formal specification as follows:

\[
\text{fun place\_queens\_in\_rows}(row, col, board) = \\
\quad \text{use empty\_boardlist()} \\
\quad \text{if (is\_overboard(row))} \\
\quad \quad \text{otherwise} \\
\quad \quad \text{combine\_boardlists(} \\
\quad \quad \quad \text{use place\_queen\_in\_col(next\_col(col),add\_queen\_to\_board(row, col, board))} \\
\quad \quad \quad \text{if (is\_legal\_position(row, col, board))} \\
\quad \quad \quad \quad \text{otherwise empty\_boardlist()} \\
\quad \quad \quad \text{end,} \\
\quad \quad \text{place\_queens\_in\_rows(next\_row(row), col, board)} \\
\quad \text{end} \\
\text{end,}
\]

The first thing the place_queens_in_rows function does is check to see if the current row is on the chessboard. This is expressed by the use...if (is_overboard(row)) otherwise... statement. Note that is_overboard is simply the name of a function, defined within the specification, that is given a row value as its argument and checks to see if the row value is past the edge of the chessboard. If the row is off the edge of the chessboard, then the empty list is returned (i.e., there exist no solutions where a queen is placed off the edge of the chessboard).

On the other hand, if the current row is on the chessboard, the place_queens_in_rows function checks to see if a queen can be placed on the square (row, col). This is expressed by the use...if (is_legal_position(row, col, board)) otherwise... statement. Again, is_legal_position is simply the name of a function that is defined within the specification. Note that a queen can be placed on a square if it does not threaten any queen that has already been placed on the board.

If a queen can be placed on the square (row, col) without threatening any of the already placed queens, then a queen is placed on (row, col) . For a specific board, placing a queen on (row, col) is accomplished by passing row, col and board to the function add_queen_to_board. After the queen has been placed on the board, a recursive application of place_queen_in_col is made to place the remaining queens.

On the other hand, if a queen cannot be placed on the square (row, col) then this particular search fails. In either case, the next possible queen placement is checked, namely placing a queen on (row+1, col). It is because of this last step that the specification finds all possible solutions.

Transforming place_queens_in_rows by $T_{i,i-1}$ yields:

\[
\text{fun place\_queens\_in\_rows =} \\
\quad \text{lambda row @} \\
\quad \text{lambda col @} \\
\quad \text{use empty\_boardlist()} \\
\quad \text{if (is\_overboard(row))} \\
\quad \quad \text{otherwise} \\
\quad \quad \text{combine\_boardlists(} \\
\quad \quad \quad \text{use place\_queen\_in\_col(next\_col(col),add\_queen\_to\_board(row, col, board))} \\
\quad \quad \quad \text{if (is\_legal\_position(row, col, board))} \\
\quad \quad \quad \quad \text{otherwise empty\_boardlist()} \\
\quad \quad \quad \text{end,} \\
\quad \quad \text{place\_queens\_in\_rows(next\_row(row), col, board)} \\
\text{end} \\
\text{end,}
\]
As we have already mentioned, this form is not much different from the original. The main differences are that
1) the formal parameters of \texttt{place\_queens\_in\_rows} are declared as lambda bound variables, and 2) actual arguments
are passed to functions in curried form.

The lambda form of the \texttt{place\_queens\_in\_rows} function is then transformed by $T_{i,i+1}$ yielding the following
Poly-Fortran function:

```fortran
cell function place\_queens\_in\_rows ( row , col , board ) ;
    declare ;
    cell row , col , board ;
    cell iftemp00025 , combine\_boardlists00008 , place\_queens\_in\_rows00007 ,
        place\_queen\_in\_col00005 , combine\_boardlists00020 ;
    enddeclare ;
    iftemp00025 = is\_overboard ( row ) ;
    if ( iftemp00025 ) then ;
        place\_queens\_in\_rows = empty\_boardlist ( ) ;
    end else ;
    combine\_boardlists00008 = is\_legal\_position ( row , col , board ) ;
    if ( combine\_boardlists00008 ) then ;
        place\_queen\_in\_col00005 = next\_col ( col ) ;
        place\_queens\_in\_rows00007 = add\_queen\_to\_board ( row , col , board ) ;
        combine\_boardlists00020 = place\_queen\_in\_col ( place\_queen\_in\_col00005 ,
            place\_queens\_in\_rows00007 ) ;
    end else ;
    combine\_boardlists00020 = empty\_boardlist ( ) ;
    end ;
    place\_queens\_in\_rows00007 = next\_row ( row ) ;
    combine\_boardlists00008 = place\_queens\_in\_rows ( place\_queens\_in\_rows00007 ,
        col , board ) ;
    place\_queens\_in\_rows = combine\_boardlists ( combine\_boardlists00020 ,
        combine\_boardlists00008 ) ;
end ;
return ;
end ;
```

There are four major differences between the Poly-Fortran form and the lambda form. The first difference is that
the lambda bound variables in the lambda form of the program are declared as Poly-Fortran variables of type \texttt{cell}.
As we have already mentioned in Chapter 3, \texttt{cell} is essentially a universal data type. Also recall that \texttt{cell} is a data
type that is not standard to traditional Fortran 66. However, cell is a type that is available in Poly-Fortran and its function is to bridge the gap between the data type that is used in functional programming (i.e., lists) and Fortran types which can be thought of in terms of integers, reals, characters, arrays, etc. Since the objective of the overall transformation sequence (i.e., $T_{i+12,n}$) is to produce a Poly-Fortran program having the same semantics as defined by a Fortran 66 compiler, it should be clear that the sequence $T_{i+12,n}$ must ultimately express (i.e., transform) the data type cell into standard Fortran 66 constructs. In this dissertation we will not discuss how this is done, we simply mention this point as an aside.

The second major difference between the Poly-Fortran form of the place_queens_in_rows function and the lambda form from which it originated is that all actual parameters to functions and all conditional tests are expressions consisting of single “variables”. This requires the creation of a variable and a corresponding assignment statement for any actual parameter and conditional test that existed in the lambda form, not satisfying this “single variable” property. Note that such variable creations and corresponding assignments will need to be performed in a recursive fashion in cases where an actual parameter to a function is an expression that is itself a function application containing actual parameters that are not in the desired “single variable” form.

The third difference is that while lambda functions return expressions, Poly-Fortran functions must assign the value computed by a function to the function identifier.

The fourth difference is that the use-if-otherwise construct, whose evaluation incidentally yields an expression, is replaced by the if-then-else construct whose evaluation modifies the store.

In very general terms the difference between the lambda form of a function and the Poly-Fortran form is that the lambda form does not permit assignments of any kind and views a computation in terms of the evaluation of a recursively defined expression while the Poly-Fortran form requires assignments and views a computation in terms of the execution of a sequence of statements that modify a store.

This concludes our example. In the next section we briefly discuss how we were able to use OTTER to reason about the correctness of transformations, as well as some of the difficulties we encountered along the way.

8.3 OTTER

In this section we discuss the role played by the automated reasoning system OTTER in our methodology. This discussion can be broken down into four parts. They are:

1. Representing the denotational semantics of Poly in terms OTTER can manipulate.
2. Representing schemas in a manner consistent with the representation we have chosen for the denotational semantics.
3. Providing OTTER with a means to reason about the equivalence of schemas.
4. Providing OTTER with a means to reason about the less-defined relation $\sqsubseteq$.

In the following sections, we discuss each of these parts.

8.3.1 The OTTER Representation of the Denotational Semantics of Poly

Before OTTER can carry out any reasoning about schemas it must first map (i.e., translate) a schema into its meaning. To accomplish such a mapping, the basic idea was to express the denotational semantics of Poly as OTTER demodulators that could then be used to demodulate (i.e., rewrite) a schema into its denotational meaning. The most obvious and perhaps the easiest way for such an approach to work is to express schemas in some form equivalent to an SDT (i.e., not some abbreviated form like the one we have been using in this paper). If this is done, then the denotational semantic definitions can be written as OTTER functions that behave in the expected manner (i.e., a function that accepts SDT’s as input and defines them in terms of expressions constructed from the meaning of the subtrees of the input SDT).

After deciding on this approach, the first problem encountered was due to the fact that OTTER is an automated reasoning system that is based on first order logic and therefore does not directly support lambda calculus. Obtaining an acceptable first order representation of the lambda calculus required 1) representing lambda bound variables as OTTER constant functions (this is necessary in order to prevent OTTER unification with these lambda bound variables), and 2) representing $\lambda$ with a two place OTTER function “lambda($x$, $y$)” that accepted a lambda variable (i.e., an OTTER constant function) as its first argument and whose second argument consists of the expression to which the lambda variable was bound in the original lambda expression.
In addition, since most of our denotational definitions are defined in terms of continuation functions, one often encounters lambda variables denoting continuations. These continuation variables denote functions that are applied to the result obtained by the mathematical execution of the construct within whose definition the continuation variable occurs. Thus, a continuation is a higher order concept not directly supported by the first order logic available to OTTER. However, by writing a set of demodulators that “apply” a function to an argument we were able to capture most of the higher order semantics that were utilized in our denotational semantic definitions. The one thing that had not been captured by the “apply” demodulators was the alpha-conversion problem that can arise when an expression containing a lambda bound variable is passed into the scope of a second lambda bound variable sharing the same name. In order to handle this problem we gave OTTER the capability to generate unique integers. This allowed every application of a demodulator to have access to a unique integer that it could use to essentially subscript lambda bound variables. Note that it was for this reason that we expressed lambda bound variables as OTTER constant functions and not simply as constants. By using the unique integer available within the application of a demodulator as the argument of the constant function denoting lambda bound variables, we were able to simulate alpha-conversion of lambda bound variables in OTTER when necessary.

In Appendix F we give several examples of the OTTER representation we have chosen to express the denotational definitions of Poly.

### 8.3.2 Representing Static and Common Dynamic Patterns in OTTER

Since denotational semantic definitions operate on syntax derivation trees, it is necessary for the OTTER representation of schemas to contain at least as much information as SDT’s. With this in mind, the following is a possible representation for a schema:

\[
\text{schema def tree(root,index,subtreelist).}
\]

That is, a schema is a tree consisting of a root symbol, an index and a subtreelist. The root symbol is what we called the dominating symbol in Chapter 5. The index is a number that can be used to express relationships between schema variables, and the subtreelist is just a list of trees whose root symbols can be derived from the root of the original tree by the application of a single grammar production. For example, consider the pattern:

\[
< \text{expr} > \text{"1"} \{< \text{expr} > \text{"2"} + < \text{term} > \text{"4"} \}
\]

that can be derived using the expression grammar given in chapter 5. The first problem with this schema is that it is an abbreviated form of an SDT. Recall that this abbreviated was introduced as a notational convenience. We would like to thank Danit Brown of Argonne National Laboratory for producing a program that converts a schema written in abbreviated form into an appropriately annotated SDT. Thus, by using her program the above schema (essentially) becomes:

\[
\text{tree(expr,1,subtreelist(tree(expr,2, end),}
\text{subtreelist(tree(plus,3, end),}
\text{subtreelist(tree(term,4, end),
\text{end}))}
\]

Using this representation, static patterns can be completely and precisely represented in a form that OTTER can manipulate.

Correctly representing dynamic patterns proved to be a bit more complicated. In order to represent dynamic patterns and the various other transformation constructs available in TAMPR, we refined our representation of a schema to:

\[
\text{schema def tree(root,index,refinement).}
\]

Here a “refinement” is a list containing information about the nonterminal symbol root. In the case of a static schema, the refinement parameter will simply be a list containing the subtreelist associated with the root symbol. In the cases where root is a schema variable that is annotated by either a qualification expression or a dynamic pattern, the refinement parameter will be a list describing the annotation. The details of how a refinement list can express various TAMPR constructs are somewhat lengthy and uninteresting and we therefore omit them. The important
thing to note is that the refinement parameter can capture the various constructs available in TAMPR. With this representation OTTER is capable of mapping a wide variety of schemas into their denotational meaning. The only thing that remains to be done is to provide OTTER with 1) a means to perform general equality reasoning, and 2) a means to reason about the $\sqsubseteq$ relation.

### 8.3.3 Providing OTTER with a Means to Reason about the Equivalence of Schemas

Without any assistance, OTTER has the ability to conclude that two schemas are equivalent if their meanings are syntactically identical. For example, if schema $s_1$ and $s_2$ are both mapped to $m_1$ by the denotational semantic demodulators discussed in Section 8.3.1 then OTTER will conclude that $s_1 \equiv s_2$. However, in order to enable OTTER to reason about more general forms of equality we must provide equality axioms for our mathematical foundation $M_f$. To date we have dealt with this issue in an informal manner, providing OTTER with only very simple axioms and only in cases when these axioms were necessary in order to obtain a particular proof. An area of future research would be to discover a complete set of equality axioms for the mathematical foundation $M_f$.

The equality axioms we have used in our proofs have been demodulators that rewrite semantic expressions into canonical forms. For example, in Section ?? we stated that function access and function alteration is part of our mathematical foundation. Due to the constructive nature of function alteration (e.g., a list of input/output tuples is created by the function alteration operation) we have found it necessary to define a canonical form for the function access operation. This means that we have a demodulator that makes use of the equality axiom:

$$[x \mapsto value]g(x) = value.$$ 

Let $x_1$ and $x_2$ denote two distinct variables. A second equality axiom that is quite useful is:

$$[x_2 \mapsto v_2][x_1 \mapsto v_1]g \equiv [x_1 \mapsto v_1][x_2 \mapsto v_2]g.$$ 

This axiom essentially states that the order in which function alteration operations occur is not important.

It turns out that with the addition of a few equality axioms of this nature, OTTER is able to prove the correctness (i.e., the equivalence) of transformations $T_{i+4}, T_{i+5}, T_{i+6}$, and $T_{i+9}$ in the lisptran transformation sequence. However, simple equality axioms are not strong enough to allow OTTER to prove a transformation is correctness preserving in the case where the input schema of a transformation is actually less-defined than its output schema. In order to reason about the $\sqsubseteq$ relation we need to provide OTTER with an axiomatization of $\sqsubseteq$. This will be discussed in the following section.

### 8.3.4 Providing OTTER with a Means to Reason about the Less-defined Relation $\sqsubseteq$

Empirical evidence suggests that when a schema $s_1$ is less defined than a schema $s_2$ it will generally be due to the fact that some operation (e.g., expression evaluation, assignment, etc.) is being performed in $s_2$ using a store or environment function that is more-defined than the store or environment function used to perform the same operation in $s_1$. Because of this observation, we have concentrated our efforts in constructing a methodology that will allow OTTER to perform $\sqsubseteq$ reasoning in these specific cases. In other words, we have given OTTER the ability to deduce when an environment, store, or model is less-defined than another environment, store, or model and then to use this information to reason about the relationship between two schemas. An area of future research would be to construct a methodology allowing OTTER to perform general reasoning with respect to the $\sqsubseteq$ relation.

Our methodology requires three independent OTTER runs. In the first run, OTTER is presented with the canonical form of the semantic expressions (i.e., the meaning) of the input and output schemas of the transformation currently under investigation. From these semantic expressions, OTTER proceeds to extract all of the environment, store, and model functions. After this has been accomplished OTTER halts.

In the second run, OTTER uses less-definedness axioms to determine the monotonic relationship between all of the environment, store, and model functions. The axioms that are used in this run are of the form:

$$\text{defined}([x \mapsto v|\varepsilon]) \land \text{unique}(x, \varepsilon) \land \text{env}(\varepsilon) \land \text{env}([x \mapsto v|\varepsilon]) \Rightarrow \varepsilon \sqsubseteq [x \mapsto v|\varepsilon].$$

This axiom states that if the environment $[x \mapsto v|\varepsilon]$ is defined, and if the identifier $x$ is unique with respect to environment $\varepsilon$, and if the environments $\varepsilon$ and $[x \mapsto v|\varepsilon]$ exist, then $\varepsilon$ is less-defined than $[x \mapsto v|\varepsilon]$. Similar axioms exist for store functions and models. Using these axioms, OTTER is in effect creating a monotonic lattice of environment, store, and model functions. After this lattice has been completed OTTER halts. Note that the completion of this lat-
The input schema of this transformation consists of 1) an assignment of a variable to an expression where the expression is an application of a lambda function, followed by 2) a statement tail \(<\text{stmt tail}>\) \textquotedblleft 1\textquotedblright. Transformation 7a transforms this input schema into a sequence of Poly-Fortran instructions. This is accomplished by 1) declaring...
Section 8.4 Proofs

ident “2”, the formal parameter of the lambda function, as a Poly-Fortran variable of type cell, 2) assigning <expr> “2”, the actual argument to the lambda function, to <ident> “2”, and 3) assigning <expr> “1” to <var> “1”. These two assignments as well as the declaration are encapsulated in a block construct, and this block is then followed by the statement tail <stmt tail> “1”.

There are two theoretical problems that have to be dealt with in order to prove the correctness of this transformation. The first problem is that in the output schema of the transformation the expression <expr> “2” is evaluated in an environment that has just been updated with the addition of the variable <ident> “2” while in this input schema <expr> “2” is evaluated with respect to an environment that does not contain such an update. The problem here is that if there is an occurrence of <ident> “2” in <expr> “2” then the transformation will generally not be correct. One way to assure that such occurrences are not possible is by requiring that all identifier declarations be unique. This would mean that in the lambda function in the input schema, the lambda variable <ident> “2” would be unique. If uniqueness is enforced, we see that for any environment of the form:

\[ [\text{id} \mapsto \alpha]\varepsilon_0 \]

it will be the case that:

\[ \varepsilon_0 \subseteq [\text{id} \mapsto \alpha]\varepsilon_0. \]

That is, any update to an environment will always result in an environment that is a strict refinement of the original environment.

OTTER, is able to monotonically “improve” the semantic expression corresponding to the input schema until it is equivalent to the semantic expression corresponding to the output schema. From this OTTER can conclude that the output schema is a strict refinement of the input schema. However, because the output schema is a strict refinement of the input schema and not equivalent to the input schema one needs to formally show that the output schema will be defined whenever the input schema is defined in order to conclude that the transformation is correctness preserving. Recall this was discussed in Section 6.7. However, since the store function component of both the input and output schemas is equivalent, proving that the (entire) output pattern is defined is quite simple. Recall that in Poly a declaration statement changes the environment and not the store.

### 8.4.2 Transformation 7b.

<stmt tail> {

.sd.

<perfix> “1” <var> “1” =
use <expr> “2” if (<expr> “1”) otherwise <expr> “3” end;
<stmt tail> “1” ⇒
<perfix> “1” if (<expr> “1”) then;
<var> “1” = <expr> “2”; end else;
<var> “1” = <expr> “3”; end;
<stmt tail> “1” .sc.

}

The input schema of this transformation consists of 1) an assignment of a use-if-otherwise expression to a variable, followed by 2) a statement tail. Transformation 7b transforms the assignment in the input schema into a sequential if-then-else construct. The relationship between the use-if-otherwise expression and the if-then-else is straightforward and will not be described further.

In Poly, the evaluation of the arguments of the use-if-otherwise expression is not strict. This means that if the Boolean argument of the use-if-otherwise expression evaluates to true, then the argument following the keyword “use” will be returned as the value of the entire expression regardless of whether the argument following the keyword
“otherwise” is undefined. Some languages might enforce strict evaluation of all arguments to operations. In such languages all three arguments must be evaluated before they are passed to an operation. In such languages proving the correctness of a transformation that transforms if-then-else constructs into use-if-otherwise constructs will involve reasoning about the definedness of all three arguments of the use-if-otherwise construct. However, there is no need to worry about inadvertently making an omission of this nature since such issues will surface as a result of the denotational semantic definitions.

The proof of transformation 7b is obtained by showing equivalence at the semantic expression level. All that was required was rewriting (i.e., demodulating) the input and output schemas of the transformation into their respective semantic expressions and then performing a syntactic equality comparison.

This concludes our discussion of the proofs of the lisptran transformation sequence. Transformation 7a is an example of a transformation requiring a more involved proof, while 7b is an example of a transformation whose correctness proof is quite simple.

8.5 Summary

In this section we discussed our experiences involving the application of the methodology defined in Chapters 6 and 7 to a practical problem. The practical problem we chose was to prove the correctness of a portion of a transformation sequence, $T_{1,n}$, whose goal is to transform functional specifications into FORTRAN 66 implementations. We gave the name lisptran to the transformation sequence whose correctness we proved. As it turned out, proving the correctness of the lisptran transformation sequence did not require any of the theory laid down in Chapter 7.

In order to show what lisptran accomplishes we gave an example input to the lisptran sequence, and its corresponding output. The sample problem we considered was the eight queens problem. A specification of the eight queens is presented in Appendix C. This specification was then transformed by the transformations in $T_{1,n}$ that precede the lisptran sequence. At this point we have the input program to the lisptran sequence. This input specification (see Appendix D) is a functional form whose major difference from the original specification is that the formal parameters of functions are represented as lambda bound variables and function arguments are given in curried form. The output program produced by the lisptran transformation sequence is a Poly-Fortran program that is given in Appendix E.

After informally describing what the lisptran transformation sequence accomplishes, we discussed the difficulties encountered in using OTTER to automate the correctness proofs of the lisptran transformation sequence. The difficulties encountered were:

1. Expressing the denotational semantics of Poly in a manner acceptable to the OTTER reasoning system. This is not entirely trivial due to the fact that OTTER is a first order logic system while the denotational semantics definitions of Poly are higher order.
2. Expressing the SDT’s of the input and output schemas in a manner consistent with the OTTER-denotational semantic representation.
3. Providing a means for expressing relationships between schema variables. For example if two or more schema variables, occurring within a single schema or occurring in the input and output schema of a transformation, denote the same instantiation, this relationship must be expressible.
4. Constructing an environment that permits OTTER to reason in terms of the $\subseteq$ relation.

After discussing how each of these difficulties can be overcome we then presented two transformations in the lisptran transformation sequence whose correctness we have proven, thereby giving an overview of what was required in each case (i.e., equivalence and strict refinement) to obtain a proof of correctness.
Chapter 9
Conclusions and Future Work

9.1 Conclusions

In this research we have constructed a methodology that allows formal reasoning about properties established or preserved by TAMPR transformations. The first part of this research required choosing a means to formally define the semantics of the language in which one wished to perform TAMPR transformations. In our case the language we wished to transform was the wide spectrum language Poly. Poly is a language consisting of both high-level functional constructs as well as more implementation-oriented imperative constructs. Because TAMPR is a transformation system that is based on syntactic rewrites, and because it is our desire to reason about the behavior of TAMPR transformations regardless of the particular program to which they will be applied, it was imperative that the constructs of Poly be monotonic with respect to replacement. Towards this end we investigated how one could go about showing that a language such as Poly is in fact monotonic. This involved a study of fixed-point theory. The findings from this study revealed that it is quite easy to show that a language like Poly is monotonic. All that is required is showing that the objects in the semantic foundation used to define Poly are monotonic and continuous. This was discussed in Chapter 3.

Because the semantic foundation that is used to define Poly is essentially the same as the semantic foundation that is used to describe most mainstream programming languages, and because monotonicity has already been shown for many of these mainstream programming languages we concluded that Poly is also monotonic. This conclusion was arrived at through very informal means, but we believe that the simplicity of the problem does not warrant a more rigorous treatment.

Having chosen Poly as the language we wished to study, the next step was to choose a means to formally define the semantics of Poly. Of the various formal semantic methods available, we chose denotational semantics. This decision was greatly influenced by the fact that TAMPR transformations are essentially syntax derivation trees. After formally defining the semantics of Poly using denotational semantics we found that traditional denotational semantics could not be used to define general TAMPR transformations. This is because TAMPR transformations generally are not written in terms of entire programs (this being the entity that denotational semantics is capable of defining), but rather they are rewrite rules stating that a piece of code consisting of one or more language constructs should be rewritten into some other piece of code. Careful study of the denotational semantics revealed that the semantics of a piece of code can be formally defined as the set of all the valuation functions capable of defining the code segment. This solved one of our problems. However, an additional problem we encountered was that TAMPR transformations could be written in terms of schemas. A schema is a syntax derivation tree that contains one or more nonterminal symbols (schema variables) as leaf nodes. Because of schema variables, schemas are capable of describing very large, frequently infinite, classes of code segments. Thus they are much more general than standard code segments. Through delta-functions we were able to extend the traditional denotational semantics thereby enabling formal definition of the semantics of schema variables which in turn allowed formal definition of schemas themselves. At this point we had tools enabling us to formally define the semantics of various TAMPR transformations. This allowed us to prove the correctness of individual TAMPR transformations.

Being able to formally prove the correctness of TAMPR transformations has been the primary goal of this research. However, in order to prove the correctness of transformations that are applied only to programs possessing certain syntactic properties, it was necessary to extend our proof methodology to enable verification of other “con-
textual” properties (i.e., syntactic properties) that the application of TAMPR transformations might establish. This was accomplished with the addition of the concept of **independence** to our methodology. By attempting to unify the syntax derivation trees of a transformation with the preconditions of the transformation, independence is able to allow formal and precise traces of the properties that are established by the application of the transformation. In order to reason about contexts (i.e., syntactic properties) that are established by the application of transformations in a formal framework, a methodology is needed that will allow the formal reasoning mechanism to utilize the information implied by such contexts. Towards this end, our proof methodology was extended with the concept of **generalization**. Generalization allows the semantic implications of syntactic properties to be reflected in the semantics of delta-functions. Affected delta-functions in turn will affect the semantics of the schemas in which they occur. In this manner the results obtained from independence are reflected in the formal semantics of schemas.

In this dissertation we have demonstrated the workability of this methodology by proving the correctness of the lisptran transformation sequence. The lisptran transformation sequence is capable of transforming a program written in terms of lambda expressions into a program written in recursive Poly-Fortran. The proofs of correctness of the individual transformations in the lisptran transformation sequence were accomplished with the aid of the automated reasoning system OTTER. Because the transformations whose correctness we proved did not rely on independence or generalization, we did not automate these aspects of our theory. However, the theories of independence and generalization are necessary in order to show the general applicability of our work. It is one of our future goals to automate these aspects of our methodology.

In closing, while several areas of our methodology remain to be fleshed-out, this research produced a viable methodology that can be used to prove the correctness of TAMPR transformations. We would also like to point out that most of the areas that we left for “future work” have been conceptually mapped out so that only mechanical and not theoretical difficulties remain. Furthermore, Chapter 8 demonstrated that with the portion of the methodology that has been completed to date, it is possible to automatically prove the correctness of a useful class of TAMPR transformations. Summarizing then, in this research we pursued the problem of program transformation from a conceptual stage all the way through to an automated realization of the methodology that we had developed.

### 9.2 Future Work

Due to the size of this undertaking, the following aspects of our methodology have been left to future work:

1. An algorithm needs to be discovered that is capable of defining the delta functions of arbitrary nonterminals with respect to a denotational semantics. We believe that delta functions can be defined at various levels of refinement. Therefore any solution to this problem will also have to deal with this “refinement” dilemma.
2. The current methodology needs to be extended in order to permit formal definition of the semantics of schemas containing qualifications and subtransformations.
3. The ultimate goal of this transformational approach is to obtain, from a formal specification, a program (i.e., an implementation) that is known to be correct and that can be efficiently executed on a computer. In our case the goal has been to obtain, from a formal functional specification, a provably correct FORTRAN 66 program. However, since Poly-Fortran and FORTRAN 66 differ substantially in their semantics, obtaining a provably correct Poly-Fortran program from a function specification is not sufficient to allow us to conclude that the resulting program is also correct with respect to the FORTRAN 66 semantics. A problem that needs to be worked out is how one can formally show that a Poly-Fortran program, p, is in a subset of Poly-Fortran such that the semantics of p with respect to Poly is less-defined than the semantics of p with respect to FORTRAN 66.
4. The general independence problem needs to be solved. Note, among other things, this will involve being able to reason about the occurrences of schemas containing qualification expressions, dynamic patterns, and subtransformations.
5. The details of generalization need to be worked out.

The above list covers the major portions of this research that still need to be completed. However, in addition to this list there are two areas of interest that could be researched. The first area concerns itself with proving the correctness of problem domain optimizations. Note that problem domain optimizations are optimizations that are possible due to certain properties that exist for the specific problem under investigation. Proving the correctness of transformations exploiting such properties would require integrating these properties into the semantic domain in which formal reasoning about the correctness of transformations takes place.

Another area of future research is to construct a means to permit the correctness of a transformation to depend...
on the transformations that follow it. The basic idea would be for a transformation, $T_i$, that is not entirely correct, to produce a correctness obligation. If it can be shown that this correctness obligation is fulfilled by the transformations that follow $T_i$ in the sequence under investigation, then one can conclude that $T_i$ is correct with respect to its transformation sequence.
Appendix A

Proof of Transformation 7a

Level of proof is 3, length is 3.

---------------- PROOF ----------------

6 []
Equal(
  Model(Initial_Env, Alpha_0, Initial_Store),
  Model(
    UPDATE_ENV(
      delta_ident(1_b(2)),
      Tuple(2_ghostinteger, Alpha_0),
      Initial_Env
    ),
    succ(Alpha_0),
    Initial_Store
  )
).

7 []
Equal(
  Initial_Store, UPDATE_STORE(
    Alpha_0, delta_value(
      EXPR,
      1_b(2),
      Model(
        UPDATE_ENV(
          delta_ident(1_b(2)),
          Tuple(2_ghostinteger, Alpha_0),
          Initial_Env
        ),
        succ(Alpha_0),
        Initial_Store
      )
    ),
    Initial_Store
  )
).

9 []
Equal(
  Model(Initial_Env, Alpha_0, Initial_Store),
  Model(
    UPDATE_ENV(
      delta_ident(1_b(2)),
      Tuple(2_ghostinteger, Alpha_0),
      Initial_Env
    ),
    succ(Alpha_0),
    UPDATE_STORE(
      Alpha_0, delta_value(
      EXP
EXPR,
1_b(2),
Model(
  UPDATE_ENV(
    delta_ident(1_b(2)),
    Tuple(2_ghostinteger,Alpha_0),
    Initial_Env
  ),
  succ(Alpha_0),
  Initial_Store
),
Initial_Store
).

12 []
-Tree(
  cont(
    delta_model(
      STMT_TAIL,
      1_b(1),
      Model(
        Initial_Env, 
        Alpha_0, 
        UPDATE_STORE(
          ACCESS_ENV(
            delta_var(1_b(1)),
            Model(
              UPDATE_ENV(
                delta_ident(1_b(2)),
                Tuple(2_ghostinteger,Alpha_0),
                Initial_Env
              ),
              succ(Alpha_0),
              UPDATE_STORE(
                Alpha_0,
                delta_value(
                  EXPR, 
                  1_b(2),
                  Model(
                    UPDATE_ENV(
                      delta_ident(1_b(2)),
                      Tuple(2_ghostinteger,Alpha_0),
                      Initial_Env
                    ),
                    succ(Alpha_0),
                    Initial_Store
                  ),
                  Initial_Store
                ),
                Initial_Store
              ),
              Initial_Store
            ),
            Initial_Store
          ),
          Initial_Store
        ),
        Initial_Store
      ),
      Initial_Store
    ),
    Initial_Store
  ),
  Initial_Store
)
EXPR, 
1_b(1),
Model(
  UPDATE_ENV(
    delta_ident(1_b(2)),
    Tuple(2_ghostinteger,Alpha_0),
    Initial_Env
  ),
  succ(Alpha_0),
  UPDATE_STORE(
    Alpha_0,
    delta_value(
      EXPR,
      1_b(2),
      Model(
        UPDATE_ENV(
          delta_ident(1_b(2)),
          Tuple(2_ghostinteger,Alpha_0),
          Initial_Env
        ),
        succ(Alpha_0),
        Initial_Store
      )
    ),
    Initial_Store
  )
),
Initial_Store
)
);
UPDATE_STORE(
  Alpha_0,
  delta_value(
    EXPR,
    1_b(2),
    Model(
      UPDATE_ENV(
        delta_ident(1_b(2)),
        Tuple(2_ghostinteger,Alpha_0),
        Initial_Env
      ),
      succ(Alpha_0),
      Initial_Store
    )
  ),
  Initial_Store
);
)
)
).
13 []
Tree(
  cont(
    delta_model(
      STMT_TAIL,
1_b(1),
Model(Initial_Env, Alpha_0, UPDATE_STORE(
  ACCESS_ENV(
    delta_var(1_b(1)),
    Model(Initial_Env, Alpha_0, Initial_Store)
  ),
  delta_value(
    EXPR,
    1_b(1),
    Model(
      UPDATE_ENV(
        delta_ident(1_b(2)),
        Tuple(2_ghostinteger, Alpha_0),
        Initial_Env
      ),
      succ(Alpha_0),
      UPDATE_STORE(
        Alpha_0,
        delta_value(
          EXPR,
          1_b(2),
          Model(Initial_Env, Alpha_0, Initial_Store)
        ),
        Initial_Store
      )
    ),
    Initial_Store
  )
)
).
26 [para_into,13,6]
Tree(
  cont(
    delta_model(
      STMT_TAIL,
      1_b(1),
      Model(Initial_Env, Alpha_0, UPDATE_STORE(
        ACCESS_ENV(
          delta_var(1_b(1)),
          Model(Initial_Env, Alpha_0, Initial_Store)
        ),
        delta_value(
          EXPR,
          1_b(1),
          Model(
            UPDATE_ENV(
              delta_ident(1_b(2)),
              Tuple(2_ghostinteger, Alpha_0),
              Initial_Env
            ),
            succ(Alpha_0),
            UPDATE_STORE(
              Alpha_0,
              delta_value(
                EXPR,
                1_b(2),
                Model(Initial_Env, Alpha_0, Initial_Store)
              ),
              Initial_Store
            )
          ),
          Initial_Store
        )
      )
    ))
)
delta_ident(1_b(2)),
Tuple(2_ghostinteger,Alpha_0),
Initial_Env
),
succ(Alpha_0),
UPDATE_STORE(
  Alpha_0,
delta_value(
    EXPR,
    1_b(2),
    Model(
      UPDATE_ENV(
        delta_ident(1_b(2)),
        Tuple(2_ghostinteger,Alpha_0),
        Initial_Env
      ),
succ(Alpha_0),
Initial_Store
    )
  ),
Initial_Store
)
)
).

54 [para_into,26,7]
Tree(
  cont(
    delta_model(
      STMT_TAIL,
      1_b(1),
      Model(
        Initial_Env,
        Alpha_0,
        UPDATE_STORE(
          ACCESS_ENV(
            delta_var(1_b(1)),
            Model(Initial_Env,Alpha_0,Initial_Store)
          ),
          delta_value(
            EXPR,
            1_b(1),
            Model(
              UPDATE_ENV(
                delta_ident(1_b(2)),
                Tuple(2_ghostinteger,Alpha_0),
                Initial_Env
              ),
succ(Alpha_0),
UPDATE_STORE(
575 [para_into, 54, 9]

Tree(
    cont(
        delta_model(
            STMT_TAIL,  
            1_b(1),
            Model(
                Initial_Env,  
                Alpha_0,  
                UPDATE_STORE(
                    ACCESS_ENV(
                        delta_var(1_b(1)),
                        Model(
                            UPDATE_ENV(
                                delta_ident(1_b(2)),
                                Tuple(2_ghostinteger, Alpha_0),
                                Initial_Env
                            ),
                            succ(Alpha_0),
                            Initial_Store
                        ),
                        Initial_Store
                    ),
                    Initial_Store
                )
            ),
            Initial_Store
        )
    )
)
Appendix A  Proof of Transformation 7a

Tuple(2_ghostinteger,Alpha_0),
Initial_Env

),
succ(Alpha_0),
UPDATE_STORE(
  Alpha_0,
  delta_value(
    EXPR,
    1_b(2),
    Model(
      UPDATE_ENV(
        delta_ident(1_b(2)),
        Tuple(2_ghostinteger,Alpha_0),
        Initial_Env
      ),
succ(Alpha_0),
      Initial_Store
    )
  ),
  Initial_Store
)
),

delta_value(
  EXPR,
  1_b(1),
  Model(
    UPDATE_ENV(
      delta_ident(1_b(2)),
      Tuple(2_ghostinteger,Alpha_0),
      Initial_Env
    ),
succ(Alpha_0),
    UPDATE_STORE(
      Alpha_0,
      delta_value(
        EXPR,
        1_b(2),
        Model(
          UPDATE_ENV(
            delta_ident(1_b(2)),
            Tuple(2_ghostinteger,Alpha_0),
            Initial_Env
          ),
succ(Alpha_0),
          Initial_Store
        )
      ),
      Initial_Store
    )
  ),
  Initial_Store
)
),

UPDATE_STORE(
  Alpha_0,
  delta_value(
EXPR,
1_b(2),
Model(
  UPDATE_ENV(
    delta_ident(1_b(2)),
    Tuple(2_ghostinteger,Alpha_0),
    InitialEnv
  ),
  succ(Alpha_0),
  Initial_Store
)
  Initial_Store
)
).
576 [binary,575,12]
.

---------- end of proof ----------

---------- END OF SEARCH ----------
The job finished Thu May 13 16:45:23 1993
Appendix  

B  

Syntax for a subset of Poly  

1.  

Program Units  

\[
\begin{align*}
<\text{prog}> & \rightarrow <\text{prog unit cflist}> <\text{prefix}>
\\
<\text{prog unit cflist}> & \rightarrow <\text{prog unit}>
\\
& | <\text{prog unit cflist}> <\text{prog unit}>
\\
<\text{prog unit}> & \rightarrow <\text{prog part}> <\rangle>
\\
<\text{prog part}> & \rightarrow <\text{prog body}>
\\
& | <\text{p subprog stmt}> <\text{prog body}>
\\
<\text{p subprog stmt}> & \rightarrow <\text{prefix} > <\text{subprog type}> <\text{subprog head}> <\rangle>
\\
<\text{subprog type}> & \rightarrow <\text{standard type}> <\text{sfb}>
\\
& | <\text{sfb}>
\\
<\text{subprog head}> & \rightarrow <\text{ident}>
\\
& | <\text{ident} > <\text{formals}>
\\
<\text{formals}> & \rightarrow <\rangle>
\\
& | <\langle <\text{expr list} > \rangle>
\\
<\text{prog body}> & \rightarrow <\text{exec part}>
\\
<\text{exec part}> & \rightarrow <\text{range}>
\\
\end{align*}
\]

1.  

Compound Statements  

\[
\begin{align*}
<\text{range}>& \rightarrow <\text{stmt tail}>
\\
<\text{stmt tail}>& \rightarrow <\text{p end stmt}>
\\
& | <\text{p stmt} > <\text{stmt tail}>
\\
& | <\text{p spec stmt} > <\text{stmt tail}>
\\
\end{align*}
\]

2.  

Declarations  

\[
\begin{align*}
<\text{p spec stmt}>& \rightarrow <\text{prefix} > <\text{spec stmt} > <\rangle>
\\
<\text{spec stmt}>& \rightarrow <\text{spec}>
\\
<\text{spec}>& \rightarrow <\text{type spec}>
\\
<\text{type spec}>& \rightarrow <\text{standard type} > <\text{expr list}>
\\
<\text{declare clause}>& \rightarrow <\text{--declare--} > <\rangle>
\\
\end{align*}
\]

1.  

Statements
2. **Iterative and Group Statements**

- `<p stmt>` → `<prefix><stmt><>`
- `<stmt>` → `<group stmt>`  
  | `<if stmt>`  
  | `<basic stmt>`  
  | `<convenience stmt>`
- `<convenience stmt>` → `<−expressions−> <expr list>`
- `<if stmt>` → `<if clause><range>`  
  | `<if clause><range><else>><><range>`
- `<if clause>` → `<and if head><then>><>`
- `<and if head>` → `<if head>`
- `<if head>` → `<−if−><selector>`
- `<basic stmt>` → `<noref basic stmt>`  
  | `<ref basic stmt>`
- `<noref basic stmt>` → `<assignment>`  
  | `<−continue−>`  
  | `<return>`  
  | `<stop>`
- `<ref basic stmt>` → `<go to>`
- `<assignment>` → `<var> <= <expr>`
- `<stop>` → `<−stop−>`
- `<return>` → `<−return−>`
- `<go to>` → `<−goto−><label ref>`
- `<labels>` → `<[><label ref list><]>`
Appendix B  Syntax for a subset of Poly

< group stmt >  →  < loop clause > < range >
                 | < block clause > < range >
                 | < declare clause > < range >

< loop clause >  →  < iter type > < iter clause > < >
                 | < −dowhile− > < selector > < >

< iter type >  →  < −do− >

< iter clause >  →  < var > < > < iter spec >

< iter spec >  →  < expr list >

< selector >  →  < ( < expr > ) >

< block clause >  →  < −block− > < >

3. Arguments and Identifiers

< pending args >  →  < args >
                 | < args > < pending args >

< args >  →  < ( ) >
                 | < ( < expr list > ) >

< bound vars >  →  < expr list > < @ >
                 | < @ >

< ident >  →  < id >

4. Types and Constants

< const >  →  < i const >
               | < f const >
               | < char const >

< i const >  →  < i rep >

< f const >  →  < f rep >

< char const >  →  < c rep >

< standard type >  →  < cell >
                      | < integer >
                      | < real >
                      | < logical >
                      | < double precision >
                      | < complex >
                      | < character >

< sfb >  →  subroutine
               | function
5. **Prefixes**

\[
\begin{align*}
<\text{prefix}> & \rightarrow <l\ \text{part}> \\
& \quad | <\text{prefix}><\text{name}><_: > \\
<l\ \text{part}> & \rightarrow <c\ \text{part}> \\
& \quad | <\text{l\ part}><\text{label}><_: > \\
<c\ \text{part}> & \rightarrow <s\ \text{s}> \\
& \quad | <\text{c\ pack\ seq}><s\ \text{s}> \\
<s\ \text{s}> & \rightarrow <\text{empty}> \\
& \quad | <\text{mark}> \\
<c\ \text{pack\ seq}> & \rightarrow <c\ \text{pack}> \\
& \quad | <\text{c\ pack\ seq}><c\ \text{pack}> \\
<c\ \text{pack}> & \rightarrow <\text{indent}> \\
& \quad | <c><\text{indent}> \\
<c> & \rightarrow <\text{text}> \\
& \quad | <c><\text{text}> \\
& \quad | <c><\text{ident}><\text{text}> \\
& \quad | <c><\text{label}><\text{text}> \\
<\text{name}> & \rightarrow <\text{var}> \\
<\text{label\ ref}> & \rightarrow <\text{label}> \\
<\text{label}> & \rightarrow <\text{i\ const}> \\
\end{align*}
\]

6. **Expressions**
Appendix B  Syntax for a subset of Poly

\[
<\text{expr list}> \rightarrow <\text{expr}>
| <\text{expr list}> , <\text{expr}>
\]

\[
<\text{expr}> \rightarrow <\text{op expr}>
| <\text{op expr}> <\text{op}> <\text{primary}>
\]

\[
<\text{primary}> \rightarrow <\text{entity}>
\]

\[
<\text{entity}> \rightarrow <\text{basic entity}> <\text{type info}>
\]

\[
<\text{type info}> \rightarrow <\text{empty}>
\]

\[
<\text{basic entity}> \rightarrow <\text{const}>
| <\text{var}>
\]

\[
<\text{var}> \rightarrow <\text{function app}>
| <\text{function definition}>
| <\text{function expr}>
\]

\[
<\text{function app}> \rightarrow <\text{function expr}> <\text{pending args}>
\]

\[
<\text{function definition}> \rightarrow <\text{functional declaration}> <\text{end}>\]

\[
<\text{functional declaration}> \rightarrow <\ident>
| <\text{lambda abstraction}>
| <\text{cond expr}>
| <\text{expr}> <\text{expr list}> <\text{end}>
\]

\[
<\text{lambda abstraction}> \rightarrow <\text{lambda}> <\text{body}> <\text{end}>
\]

\[
<\text{body}> \rightarrow <\text{bound vars}> <\text{expr list}>
\]

\[
<\text{cond expr}> \rightarrow <\text{use}> <\text{expr list}> <\text{if head}>
| <\text{otherwise}> <\text{expr list}> <\text{end}>
\]

\[
<\text{functional declaration}> \rightarrow <\text{function declaration}>
\]

\[
<\text{function declaration}> \rightarrow <\text{fun}> <\text{rule list}>
\]

\[
<\text{rule list}> \rightarrow <\text{rule}>
\]

\[
<\text{rule}> \rightarrow <\text{expr}>=<\text{expr}>
\]

7. Operators
\[<\text{op}> \rightarrow <\text{add \, op}> \\
   | <\text{mult \, op}>\]

\[<\text{add \, op}> \rightarrow <\text{+}> \\
   | <\text{-}>\]

\[<\text{mult \, op}> \rightarrow <\text{*}> \\
   | <\text{/}> \\
   | <\text{div}> \\
   | <\text{mod}>\]
Appendix C  Eight Queens Specification

fun queens_driver() =
  print(place_queen_in_col(first_col(),empty_board()))
end,

fun place_queen_in_col(col,board) =
  use make_boardlist(board)
  if (is_overboard(col))
    otherwise place_queens_in_rows(first_row(),col,board)
  end
end,

fun place_queens_in_rows(row,col,board) =
  use empty_boardlist()
  if (is_overboard(row))
    otherwise
      combine_boardlists(
        use place_queen_in_col(next_col(col),add_queen_to_board(row,col,board))
        if (is_legal_position(row,col,board))
        otherwise empty_boardlist()
      end,
      place_queens_in_rows(next_row(row),col,board)
    )
  end
end,

fun is_legal_position(row,col,board) =
  or(
    is_empty_board(board),
    and(
      is_legal_with_respect_to_queen(row,col,first_queen(board)),
      is_legal_position(row,col,rest_board(board))
    )
  )
end,

fun is_legal_with_respect_to_queen(row,col,queen) =
  not(
    or(
      equal(row,row_of(queen)),
      equal(abs(difference(row,row_of(queen))),abs(difference(col,col_of(queen))))
    )
  )
end,

fun is_overboard(roworcol) = greaterp(roworcol,8) end,

fun first_col() = 1 end,

fun next_col(col) = plus(col,1) end,

fun first_row() = 1 end,
fun next_row(row) = plus(row,1) end,
fun empty_board() = nil end,
fun is_empty_board(board) = null(board) end,
fun add_queen_to_board(row,col,board) = cons(cons(row,col),board) end,
fun row_of(queen) = car(queen) end,
fun col_of(queen) = cdr(queen) end,
fun first_queen(board) = car(board) end,
fun rest_board(board) = cdr(board) end,
fun make_boardlist(board) = cons(board,nil) end,
fun empty_boardlist() = nil end,
fun is_empty_boardlist(board) = null(board) end,
fun combine_boardlists(boardlist1,boardlist2) = append(boardlist1,boardlist2) end,
fun first_board(boardlist) = car(boardlist) end,
fun rest_boardlist(boardlist) = cdr(boardlist) end ;
Appendix D

Lamda Form of Eight Queens

# expressions

fun queens_driver =
      lambda @
      print ( place_queen_in_col ( first_col ( ) ) ( empty_board ( ) ) )
end,
end,

fun place_queen_in_col =
      lambda col @
      lambda board @
      use make_boardlist ( board )
      if ( is_overboard ( col ) )
      otherwise place_queens_in_rows ( first_row ( ) ) ( col ) ( board )
end
end,
end,

fun place_queens_in_rows =
      lambda row @
      lambda col @
      lambda board @
      use empty_boardlist ( )
      if ( is_overboard ( row ) )
      otherwise combine_boardlists ( 
          use place_queen_in_col ( next_col ( col ) ) (add_queen_to_board (row)(col)(board))
          if ( is_legal_position ( row ) ( col ) ( board ) )
          otherwise empty_boardlist ( )
          end ) ( place_queens_in_rows ( next_row ( row ) ) ( col ) ( board ) )
end
end
end,
end,

fun is_legal_position =
      lambda row @
      lambda col @
      lambda board @
      use t
      if ( is_empty_board ( board ) )
      otherwise
      use nil
      if ( not ( is_legal_with_respect_to_queen ( row ) (col)(first_queen (board)) ) )
      otherwise is_legal_position ( row ) ( col ) ( rest_board ( board ) )
end
end
end,
end,
fun is_legal_with_respect_to_queen =
    lambda row @
        lambda col @
            lambda queen @
                not ( use t
                    if ( equal ( row ) ( row_of ( queen ) ) )
                    otherwise equal ( abs(difference(row)(row_of (queen))))(abs(difference(col)(col_of(queen))))
                end )
            end
        end
    end
end,

fun is_overboard =
    lambda roworcol @
        greaterp ( roworcol ) ( !8 )
    end
end,

fun first_col =
    lambda @
        !1
    end
end,

fun next_col =
    lambda col @
        plus ( col ) ( !1 )
    end
end,

fun first_row =
    lambda @
        !1
    end
end,

fun next_row =
    lambda row @
        plus ( row ) ( !1 )
    end
end,

fun empty_board =
    lambda @
        nil
    end
end,

fun is_empty_board =
    lambda board @
        null ( board )
    end
end,

fun add_queen_to_board =
Appendix D  Lambda Form of Eight Queens

\[
\text{lambda row @} \\
\text{ lambda col @} \\
\text{ lambda board @} \\
\quad \text{cons ( cons ( row ) ( col ) ) ( board )} \\
\quad \text{end} \\
\quad \text{end} \\
\quad \text{end} \\
, \\
\text{fun row_of =} \\
\quad \text{lambda queen @} \\
\quad \text{car ( queen )} \\
\quad \text{end} \\
\quad \text{end} \\
\text{fun col_of =} \\
\quad \text{lambda queen @} \\
\quad \text{cdr ( queen )} \\
\quad \text{end} \\
\quad \text{end} \\
\text{fun first_queen =} \\
\quad \text{lambda board @} \\
\quad \text{car ( board )} \\
\quad \text{end} \\
\quad \text{end} \\
\text{fun rest_board =} \\
\quad \text{lambda board @} \\
\quad \text{cdr ( board )} \\
\quad \text{end} \\
\quad \text{end} \\
\text{fun make_boardlist =} \\
\quad \text{lambda board @} \\
\quad \text{cons ( board ) ( nil )} \\
\quad \text{end} \\
\quad \text{end} \\
\text{fun empty_boardlist =} \\
\quad \text{lambda @} \\
\quad \text{nil} \\
\quad \text{end} \\
\quad \text{end} \\
\text{fun is_empty_boardlist =} \\
\quad \text{lambda board @} \\
\quad \text{null ( board )} \\
\quad \text{end} \\
\quad \text{end} \\
\text{fun combine_boardlists =} \\
\quad \text{lambda boardlist1 @} \\
\quad \text{lambda boardlist2 @} \\
\quad \quad \text{append ( boardlist1 ) ( boardlist2 )} \\
\quad \quad \text{end} \\
\quad \text{end}
end,

fun first_board =
    lambda boardlist @
    car ( boardlist )
end end,

fun rest_boardlist =
    lambda boardlist @
    cdr ( boardlist )
end end;

#

end;

#
Appendix E
Recursive Fortran form of Eight Queens

declare:
  cell queens_driver00022;
enddeclare;

queens_driver00022 = queens_driver ( ) ;
return;
end;

cell function queens_driver ( ) ;
  declare:
    cell print00003 , place_queen_in_col00001 , place_queen_in_col00002 ;
  enddeclare ;

  place_queen_in_col00001 = first_col ( ) ;
  place_queen_in_col00002 = empty_board ( ) ;
  print00003 = place_queen_in_col ( place_queen_in_col00001 , place_queen_in_col00002 ) ;
  queens_driver = print ( print00003 ) ;
  return;
end;

cell function place_queen_in_col ( col , board ) ;
  declare:
    cell col , board ;
    cell iftemp00023 , place_queens_in_rows00004 , place_queen_in_col00005 ;
  enddeclare ;

  iftemp00023 = is_overboard ( col ) ;
  if ( iftemp00023 ) then;
    place_queen_in_col = make_boardlist ( board ) ;
  end else ;
  place_queens_in_rows00004 = first_row ( ) ;
  place_queen_in_col = place_queens_in_rows ( place_queens_in_rows00004 , col , board ) ;
  end ;
  return;
end;

cell function place_queens_in_rows ( row , col , board ) ;
  declare:
    cell row , col , board ;
    cell iftemp00025 , combine_boardlists00008 , place_queens_in_rows00007 , place_queen_in_col00005 , combine_boardlists00020 ;
  enddeclare ;

  iftemp00025 = is_overboard ( row ) ;
if ( iftemp00025 ) then;
    place_queens_in_rows = empty_boardlist ( ) ;
end else;
    combine_boardlists00008 = is_legal_position ( row , col , board ) ;
if ( combine_boardlists00008 ) then ;
    place_queen_in_col00005 = next_col ( col ) ;
    place_queens_in_rows00007 = add_queen_to_board ( row , col , board ) ;
    combine_boardlists00020 = place_queen_in_col ( place_queen_in_col00005 ,
                                        place_queens_in_rows00007 ) ;
end else ;
    combine_boardlists00020 = empty_boardlist ( ) ;
end ;
place_queens_in_rows00007 = next_row ( row ) ;
combine_boardlists00008 = place_queens_in_rows ( place_queens_in_rows00007 ,
                                           col , board ) ;
place_queens_in_rows = combine_boardlists ( combine_boardlists00020 ,
                                           combine_boardlists00008 ) ;
end ;
return ;
end ;

cell function is_legal_position ( row , col , board ) ;
decclare ;
    cell row , col , board ;
    cell iftemp00027 , is_legal_position00011 ,
    is_legal_with_respect_to_queen00009 , iftemp00026 ;
enddeclare ;
iftemp00027 = is_empty_board ( board ) ;
if ( iftemp00027 ) then ;
    is_legal_position = t ;
end else ;
    is_legal_with_respect_to_queen00009 = first_queen ( board ) ;
    is_legal_position00011 = is_legal_with_respect_to_queen ( row , col ,
                                                        is_legal_with_respect_to_queen00009 ) ;
    iftemp00026 = not ( is_legal_position00011 ) ;
if ( iftemp00026 ) then ;
    is_legal_position = nil ;
end else ;
    is_legal_position00011 = rest_board ( board ) ;
    is_legal_position = is_legal_position ( row , col , is_legal_position00011 ) ;
end ;
end ;
return ;
end ;
appendix e recursive fortran form of eight queens

function is_legal_with_respect_to_queen (row, col, queen)

declare;
    cell row, col, queen;
    cell not00021, iftemp00028, difference00015, equal00018, abs00016, equal00017;
enddeclare;

difference00015 = row_of (queen);
iftemp00028 = equal (row, difference00015);
if (iftemp00028)
    not00021 = t;
else
    abs00016 = row_of (queen);
equal00018 = difference (row, abs00016);
equal00017 = abs (equal00018);
difference00015 = col_of (queen);
abs00016 = difference (col, difference00015);
equal00018 = abs (abs00016);
not00021 = equal (equal00017, equal00018);
end;
is_legal_with_respect_to_queen = not (not00021);
return;
end;

function is_overboard (roworcol)

declare;
    cell roworcol;
enddeclare;
is_overboard = greaterp (roworcol, '8');
return;
end;

function first_col()

declare;
enddeclare;
first_col = !1;
return;
end;

function next_col (col)

declare;
    cell col;
enddeclare;
next_col = plus (col, !1);
return;
end;
cell function first_row ( )
    declare ;
    enddeclare ;
    first_row = !1 ;
    return ;
end ;

cell function next_row ( row )
    declare ;
        cell row ;
    enddeclare ;
    next_row = plus ( row , !1 ) ;
    return ;
end ;

cell function empty_board ( )
    declare ;
    enddeclare ;
    empty_board = nil ;
    return ;
end ;

cell function is_empty_board ( board )
    declare ;
        cell board ;
    enddeclare ;
    is_empty_board = null ( board ) ;
    return ;
end ;

cell function add_queen_to_board ( row , col , board )
    declare ;
        cell row , col , board ;
        cell cons00019 ;
    enddeclare ;
    cons00019 = cons ( row , col ) ;
    add_queen_to_board = cons ( cons00019 , board ) ;
    return ;
end ;

cell function row_of ( queen ) ;
declare;
cell queen;
enddeclare;

row_of = car ( queen );
return;
end;

cell function col_of ( queen );

declare;
cell queen;
enddeclare;

col_of = cdr ( queen );
return;
end;

cell function first_queen ( board );

declare;
cell board;
enddeclare;

first_queen = car ( board );
return;
end;

cell function rest_board ( board );

declare;
cell board;
enddeclare;

rest_board = cdr ( board );
return;
end;

cell function make_boardlist ( board );

declare;
cell board;
enddeclare;

make_boardlist = cons ( board , nil );
return;
end;

cell function empty_boardlist ( );

declare;
enddeclare;
empty_boardlist = nil;
return;
end;

cell function is_empty_boardlist ( board );
    declare;
        cell board;
    enddeclare;
    is_empty_boardlist = null ( board );
    return;
end;

cell function combine_boardlists ( boardlist1 , boardlist2 ) ;
    declare;
        cell boardlist1 , boardlist2 ;
    enddeclare;
    combine_boardlists = append ( boardlist1 , boardlist2 ) ;
    return;
end;

cell function first_board ( boardlist ) ;
    declare;
        cell boardlist ;
    enddeclare;
    first_board = car ( boardlist ) ;
    return;
end;

cell function rest_boardlist ( boardlist ) ;
    declare;
        cell boardlist ;
    enddeclare;
    rest_boardlist = cdr ( boardlist ) ;
    return;
end;

#
Appendix F

Otter Representation

Below are several examples of denotational definitions that have been converted into OTTER demodulators. In order to show the differences the denotational definition for a particular input of a particular valuation function is given first, and is then followed by its OTTER representation. This list is by no means complete.

\[ Z_{\text{PROG_UNIT_CFLIST}}[[\text{prog_unit}]] \overset{\text{def}}{=} Z_{\text{PROG_UNIT}}[[\text{prog_unit}]] \]

Equal(
\[
Z(\text{PROG_UNIT_CFLIST},1_{\text{tr}}(2_{\text{Sprog_unit_cflist}}$x_1$\_index,
  1_{s}(1_{\text{tr}}(2_{\text{$\text{Sprog_unit_cflist}$}},x_2$\_index$,x_2$\_substr$),
  1_{end}$)),
  x_m)
\],

Z(\text{PROG_UNIT},1_{\text{tr}}(2_{\text{$\text{Sprog_unit}$}},x_2$\_index$,x_2$\_substr$),x_m)
).

\[ Z_{\text{PROG_UNIT_CFLIST}}[[\text{prog_unit_cflist prog_unit}]] \overset{\text{def}}{=} \lambda m. Z_{\text{PROG_UNIT_CFLIST}}[[\text{prog_unit_cflist}]] (Z_{\text{PROG_UNIT}}[[\text{prog_unit}]] m) \]

Equal(
\[
Z(\text{PROG_UNIT_CFLIST},1_{\text{tr}}(2_{\text{$\text{Sprog_unit_cflist}$}},x_1$\_index$,
  1_{s}(1_{\text{tr}}(2_{\text{$\text{Sprog_unit_cflist}$}},x_2$\_index$,x_2$\_substr$),
  1_{s}(1_{\text{tr}}(2_{\text{$\text{Sprog_unit}$}},x_3$\_index$,x_3$\_substr$),
  1_{end}$)),
  x_m)
\),

Z(\text{PROG_UNIT},1_{\text{tr}}(2_{\text{$\text{Sprog_unit}$}},x_3$\_index$,x_3$\_substr$),x_m)
).

\[ Z_{\text{STMT_TAIL}}[[\text{p_spec_stmt stmt_tail}]] \overset{\text{def}}{=} \lambda m. \]

update_env ( main )
(
  ( Tuple(Subroutine,None),Tuple(alpha(m),bottom))
  ( update_model_alpha(alpha(m)+1,m) )
)

Equal(
\[
Z(\text{STMT_TAIL},1_{\text{tr}}(2_{\text{$\text{Sstmt_tail}$}},x_1$\_index$,
  1_{s}(1_{\text{tr}}(2_{\text{$\text{Sstmt_tail}$}},x_2$\_index$,x_2$\_substr$),
  1_{s}(1_{\text{tr}}(2_{\text{$\text{Sstmt_tail}$}},x_3$\_index$,x_3$\_substr$),
  1_{end}$)),
  x_m)
\).
update_env(main, 
    Tuple(Tuple(Subroutine,None),Tuple(alpha(x_m),bottom)), 
    update_model_alpha(succ(alpha(x_m)),x_m) 
  )
).

$$LF_{\text{EXPR\_LIST}}[[\text{expr}]] \overset{\text{def}}{=} \lambda m. \lambda c_f. \lambda f. \lambda t. \
    c_f (\lambda \arg. (f(update\_env(F_{\text{EXPR\_LIST}}[[\text{expr}]])) \
        create\_arg\_object1(access\_type\_function(F_{\text{EXPR\_LIST}}[[\text{expr}]],t),\arg) \
        (m)) \
    ) 
$$

EQUAL(
    LF(EXPR\_LIST,1\_tr(2\_EXPR\_LIST$,x1$\_index, 
    1_s(1\_tr(2\_EXPR$,x2$\_index,x2$\_subtr), 
    1\_end) 
    ), 
    x_m, 
    x_c_f, 
    x_f, 
    x_t) 
  , 
  LF\_TEMP1(EXPR\_LIST,1\_tr(2\_EXPR\_LIST$,x1$\_index, 
    1_s(1\_tr(2\_EXPR$,x2$\_index,x2$\_subtr), 
    1\_end) 
    ), 
    x_m, 
    x_c_f, 
    x_f, 
    x_t, 
    ARG1($\text{UNIQUE\_NUM}$) 
)
)
).

Equal(
  LF\_TEMP1(EXPR\_LIST,1\_tr(2\_EXPR\_LIST$,x1$\_index, 
    1_s(1\_tr(2\_EXPR$,x2$\_index,x2$\_subtr), 
    1\_end) 
    ), 
    x_m, 
    x_c_f, 
    x_f, 
    x_t, 
    x_ARG1) 
  , 
  Apply(x_c_f, 
    lambda(x_ARG1, 
    Apply(x_f, 
        update\_env(F(EXPR,1\_tr(2\_EXPR$, 
        x_ARG1) 
    ) 
) 
).
Appendix F  Otter Representation

\[
x_2\_index,
\]

\[
x_2\_subtr),
\]

create_arg_object1(

access_type_function(F(EXPR,

1\_tr(2\_$expr$, x2\_index, x2\_subtr)),

x\_t),

x\_ARG1),

x\_m)

).
References

Appendix F  Otter Representation


