On Program Transformations

Sofoklis Efremidis
Ph.D Thesis

TR 94-1434
June 1994

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

ON PROGRAM TRANSFORMATIONS

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Sofoklis Efremidis
May 1994

© Sofoklis Efremidis 1994

ALL RIGHTS RESERVED

ON PROGRAM TRANSFORMATIONS

Sofoklis Efremidis, Ph.D.

Cornell University 1994

In understanding complex algorithms, the notions of encapsulation and modularization have played a key role. An algorithm is broken into several parts or modules, and understanding of each part is independent of others. In addition, each part contains details that are not needed by other parts and so can be hidden from them.

Programming languages provide support for encapsulation and modularization in many different forms. Early programming languages provided the procedure and function as a means for modularization. Later, files were introduced as a means of modularizing programs. More sophisticated mechanisms were then introduced, like modules, packages, structures, and classes. In all these cases, the interface to a module remained the procedure or function call. Programs that use such modules contain calls to functions and procedures for communicating with a module. Ideally, using the operations that are provided by a module should be done in exactly the same way as using operations of primitive types of the programming language. Primitive operations of the language and operations provided by modules should be easy to intermix. In addition, substituting one module for another that has the same functionality but different implementation should involve a minimal amount of effort.

Recently, a new programming language, *Polya*, has been designed, which attempts to support modularization and at the same time incorporate the operations that

are provided by the modules in the programming language itself. This is done by a sophisticated type-definition facility and a mechanism for transforming programs at the source-program level.

This thesis studies mechanisms for program transformation at the source program level, in the context of *Polya*. Program transformation is based on a set of transformation rules that prescribe how a part of a program is to be transformed, and a set of directives that prescribe which program variables are to be transformed.

We first give an algorithm for processing program transformations as described by the transform construct. The algorithm constructs a coordinate transformation of an abstract program based on a set of transforms and transform directives for transforming program variables. We then study the problem of transforming expressions that have compound types. Both the type constructor and the component expressions of the original expression may be transformed. No extra rules need be added to the bodies of transforms that transform the type constructor and the component expressions.

In the sequel we investigate the problem of transforming procedures and functions that have parameters that need to be transformed. Finally, the problem of transforming program-transformation rules is studied.

The program transformation techniques are applied to two well-known algorithms. The algorithms are source programs, which are subsequently transformed to programs of conventional programming languages, and then compiled and run.

BIOGRAPHICAL SKETCH

Sofoklis Efremidis was born on August 3, 1963 in Agia Paraskevi, a suburb of Athens, Greece. He attended the Anavryta Model School in Kifissia, Athens and graduated in June 1981. While in high school he pursued studies in classical music, and in June 1981 he was awarded an advanced degree in Classical Guitar performance from the National Conservatory of Athens.

In September 1981 he was admitted to the University of Patras, in Patras, Greece. In September 1982 he transferred to the Department of Electrical Engineering, National Technical University of Athens. He graduated in September 1986 with a Diploma in Electrical Engineering. He then worked as a Research Assistant at the National Technical University of Athens and at the University of Maryland, College Park.

He joined the Graduate School at Cornell University in September 1988. In August 1991 he was awarded a Master of Science Degree in Computer Science by Cornell University. He completed the requirements for the Ph.D. Degree at Cornell University in May 1994.

Sofoklis Efremidis is a civilian pilot having the Airplane Single and Multiengine Land and Instrument Airplane ratings.

Στους γονείς μου Γεώργιο και Ελεονώρα

και

στα αδέλφια μου Θανάση και Δημήτρη

Για όσα έκαναν για μένα

και

για την αγάπη και συμπαράστασή τους

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deep gratitude, respect, and thanks to Professor David Gries. This thesis would have been impossible to complete without his constant encouragement and support. An inspiring researcher, an outstanding teacher, a human full of warmth and understanding, David Gries was always willing to help and advise, especially during the critical phases of this work.

I would also like to thank Professor Steve Vavasis and Professor Anil Nerode for serving on my special committee.

I greatfully acknowledge the generous support by DARPA-ONR (Research Grant N00014–91–J–4123) for the work of this thesis.

I wish to express my sincere thanks to Aswin van den Berg, who helped immensely during this work and who was always there when I needed his expertise. I would also like to thank the other members of the *Polya* group for introducing me to the language and for their help, especially during the early stages of this work. Many thanks to my coauthors Khalid Mughal and John Reppy.

My sincere thanks also go to the support staff of the Department of Computer Science at Cornell University, especially to Jan Batzer, Diana Catley, Dorothy Marsch, Holly Mingins, and Becky Personius, for the outstanding help they provided while I was graduate student.

My friends both made my life in Ithaca more enjoyable and helped in my intellectual growth. I would like to thank each one of them. Special thanks to Caroline-Fotene Chisham for her true friendship and her pleasant companionship during my stay in Ithaca. Special thanks also to Victor and Roula Balopoulos, Alessandro Panconesi, Nikos Pitsianis, and Pankaj Rohatgi for their constant interest and encouragement on academic and non-academic matters. I also wish to thank my friends at East Hill Flying Club in Ithaca, for exposing me to the wonderful and exciting world of aviation.

Finally, I would like to deeply thank my family: my parents and my brothers for their love, encouragement, and support all these years.

TABLE OF CONTENTS

1 Introduction 1
1.1 Program Refinement . . . 3
1.2 Rewrite Systems . . . 5
1.3 About this Thesis . . . 7
1.4 Related work . . . 9

2 Preliminaries 10
2.1 Types . . . 10
2.2 Terms . . . 11

3 The Transform 14
3.1 Introduction . . . 14
3.2 Components of a transform . . . 16
3.2.1 Local declarations . . . 21
3.3 An example of a transform . . . 22
3.4 Transform directives . . . 23
3.5 Examples of using a transform . . . 24
3.6 Program transformation . . . 25

4 Transforming variables that have base types 26
4.1 Introduction . . . 26
4.2 Overview . . . 26
4.3 Pattern matching . . . 32
4.4 Replacement Instantiation . . . 33
4.5 The main algorithm . . . 36
4.6 Representations and transformation of variables . . . 36
4.7 Representations and transformations of constants . . . 42
4.8 Representations and transformations of expressions . . . 42
4.9 Transformations of statements . . . 45
4.10 Transformations of programs . . . 46
4.11 Correctness . . . 47
4.12 Complexity analysis . . . 49

5 Transforming expressions that have compound types **51**
5.1 Introduction 51
5.2 Overview 51
5.3 Transform and Transform Directives 52
5.3.1 Extensions to transforms 53
5.3.2 Notation 55
5.3.3 Steps in transforming expressions that have compound types 56
5.4 Preprocessing the transforms 56
5.4.1 Processing transform directives of the form $M(\overline{x})$ 57
5.4.2 Processing transform directives of the form $I(\overline{x})$ 58
5.4.3 Processing transform directives of the form M. 58
5.4.4 An example of preprocessing a transform for sequences 58
5.5 Transforming the abstract program 60
5.5.1 Replacing variables that are transformed 60
5.5.2 Transforming expressions 60
5.5.3 An example of transforming the elements of an array 62

6 Transforming functions, procedures, and transforms **64**
6.1 Transforming functions and procedures 64
6.1.1 Functions and procedures 64
6.1.2 Examples of transforming function and procedure calls 65
6.1.3 Transforming Functions and Procedures 67
6.1.4 Some theoretical results 69
6.2 Transforming variables in a transform 70
6.2.1 Directives for transforming variables of a transform 70
6.2.2 Processing transform directives for variables in transforms 73

7 A second approach to dealing with compound types **76**

8 Methodology and examples **80**
8.1 Interfering implementations 80
8.2 Example: The Huffman Encoding Algorithm 84
8.2.1 The algorithm 85
8.2.2 Abstract types used in the algorithm 85
8.2.3 Transforming the algorithm 89
8.3 Example: The Hopcroft-Tarjan Planarity Test Algorithm 100
8.3.1 Abstract types used in the algorithm 100
8.3.2 Input and output of the algorithm 101
8.3.3 Constructing a directed graph 102
8.3.4 Producing an embedding 103

8.3.5 Printing an embedding 108
8.3.6 The main program 108
8.3.7 Directives 108
8.3.8 The transforms used 112
8.3.9 Example of execution 119

Bibliography **121**

LIST OF FIGURES

1.1 Relationship of abstract and concrete state changes. 4

2.1 Definition of type *seq*(t). 11
2.2 Definition of function *subst*. 13

3.1 *BN* implements a variable of type *bool* with a variable of type *nat*. . . . 15

4.1 Definition of pattern matching. 34
4.2 Definition of *app*′. 35
4.3 Definition of replacement instantiation. 37
4.4 Main transformation algorithm. 38
4.5 Application of a transform rule to an abstract tree node. 43
4.6 Construction of representations of an expression. 44
4.7 Construction of conversion of representations. 45
4.8 Construction of transformation of an expression. 46

6.1 *Complex* implements a variable of type *complex* with a tuple of real numbers. 72
6.2 *Rtuple* implements a variables of type *rtuple* with a record. 72
6.3 *Complex_Rtuple* implements a variable of type *complex* with a record. . 73

8.1 *Graph* implements variables of type *dgraph*. 82
8.2 Defining a local meaning for operation *delete*. 83
8.3 Implementation of graphs according to transform *Graph*′. 83
8.4 Huffman encoding algorithm with directives. 86
8.5 Definition of type *set*(t). 87
8.6 Definition of type *seq*(t). 88
8.7 Definition of type *btree*(t). 89
8.8 Definition of type *ptr*(t). 90
8.9 Transform *Btree_Trec*. 90
8.10 Implementation of tree $\langle{:}\langle{:}tnil, d2, tnil{:}\rangle, d1, \langle{:}tnil, d3, tnil{:}\rangle{:}\rangle$ according to *Btree_Trec*. 91
8.11 Transform *Seq_Dll*, part 1. 92
8.12 Transform *Seq_Dll*, part 2. 93
8.13 Transform *Seq_Dll*, part 3. 94

8.14 Implementation of sequence $\langle\langle d_1, \ldots, d_n \rangle\rangle$ according to *Seq_Dll*. 95
8.15 Transform *Set_Seq*, part 1. 95
8.16 Transform *Set_Seq*, part 2. 96
8.17 Transformed program, part 1. 98
8.18 Transformed program, part 2. 99
8.19 Output of Huffman encoding algorithm. 99
8.20 Definition of type *dgraph*. 101
8.21 Definitions of *Readgraph* and *PrintGraph*. 102
8.22 Definition of *RenamGraph*, part 1. 104
8.23 Definition of *RenamGraph*, part 2. 105
8.24 Definition of *Planarity*, part 1. 106
8.25 Definition of *Planarity*, part 2. 107
8.26 Definition of *PrintEmbed*. 109
8.27 Definition of *PrintSeq*. 110
8.28 Main program. 110
8.29 Directives for transforming abstract program variables. 111
8.30 Transform *Seq_List*, part 1. 113
8.31 Transform *Seq_List*, part 2. 114
8.32 Implementation of $\langle\langle d_1, \ldots, d_n \rangle\rangle$ according to *Seq_List*. 115
8.33 Transform *RevSeq*. 115
8.34 Transform *Set_Seq*. 116
8.35 Transform *Graph_AdjList*, part 1. 117
8.36 Transform *Graph_AdjList*, part 2. 118
8.37 Transform *Arr*. 119
8.38 (a) Input graph. (b) A corresponding palm graph. 119
8.39 Output of the planarity algorithm. 120

Chapter 1

Introduction

Modern programming languages tend to be of higher level than those designed and implemented a few decades ago, in the sense that they are more problem-oriented than implementation-oriented. They focus on the problem to be solved and provide mechanisms for abstracting from the implementation details for solving it. Modularization and encapsulation are two primary mechanisms that most modern programming languages provide for abstraction. With modularization, a program can be split into independent modules that have well-defined interfaces. Encapsulation allows implementation details to be hidden from parts of a program where they are not needed.

Modules in modern programming languages provide for the definition and implementation of types. However they are rather restrictive, in that the interface between a module and a program is the procedure or function call and not an arbitrary statement or expression. Operations that are provided by modules do not look like primitive operations that the programming language provides. For example, most programming languages support integers as primitive data types, and a program may contain expression $a + b$ for computing the sum of two integer variables a and b. On the other hand, types like *set* may not be supported, so users of type *set* have to use modules that implement operations on sets. However, mathematical notation can not be introduced and integrated in the language. As an example, assume that an algorithm contains statement

$$s := s \cup \{e\}$$

where s is a set of integers (say) and e is an integer. A program that implements this algorithm needs to use a module that implements sets and operations on them. As mentioned before, the interface to a module is the procedure or function call. Hence, a program cannot contain a statement like the one above. Instead, depending on the module that is used in the implementation for sets, the statement may have to be

written as

$$insert(s, e)$$

or

$$assign(s, union(s, singleton(e))).$$

And, if one wants to switch back and forth between implementations, the program itself has to be changed accordingly.

The advantages of using mathematical notation in a program are apparent. Programs look like the mathematical algorithms they are supposed to implement. But more importantly, the designer of an algorithm can use a notation that facilitates focusing on the algorithm itself instead of its implementation details. The algorithm designer may use abstract mathematical objects like sets, bags, sequences, and lattices that may be needed in the algorithm, using the notation that best suits the domain.

If a programming language is to support new abstract types and operations on them in the same way it supports primitive types and operations, it needs to be tuned to the new syntax of the literals and the operations of the new types.

From a practical viewpoint, one would also like to be able to run an algorithm on a machine. This implies that the algorithm has to be changed from its abstract form that humans understand and reason about to a form that a machine can execute. Put in another way, the algorithm needs to be refined. Apparently, some sort of transformation of the original algorithm has to take place, and one would like to automate this transformational process as much as possible. In existing languages that restrict the interface of a module to be the procedure or function call, this transformation is accomplished by the compiler that implements the language. The transformational mechanism is hard-wired in the compiler. On the other hand, for a language that can be tuned to a specific syntax of interest, the transformational mechanism should be independent of the transformations that are to be applied. The transformations should be described in an appropriate language, and the transformational mechanism should apply these transformations to the abstract program. One can think of the transformations as rules that describe how parts of a program can be refined. As an example, they may describe the refinement of abstract operations like $b \cup c$ and $s := s \cup \{e\}$.

With such a mechanism, reusability comes for free: the transformations can be reused in other programs as well, as long as they implement the operations that appear in those programs.

A mechanism is also needed for specifying what transformations are to be applied to a program, i.e. a kind of directive. A directive is the link between an abstract program and the transformations that are to be applied to it. Using a directive, for example, one can specify that one set variable (and operations on it) are to be

implemented using a hash table, while another set variable is to be implemented using a binary tree. The important point about having such a mechanism is that the decisions on how to implement variables of the abstract program are localized. If one decides that a different implementation of a variable of the abstract program is desirable, one need only change the directive that specifies that transformation of that variable. The abstract program remains unchanged.

The programming language *Polya* [GV92] was designed with this philosophy in mind. The programming paradigm of *Polya* is as follows. The designer of an algorithm decides which types are needed by the algorithm. The new types and the operations they involve are formally defined, including the concrete syntax for each operation. The new types are then incorporated into the language and the newly defined types and operations can be used like the primitive types and operations provided by the language. Programs can be written that make use of the new types and can later be data-refined into a more concrete form.

Languages like *C++* [Str91] provide sophisticated mechanisms for encapsulation and modularization. One disadvantage is that these languages lack extendibility of concrete syntax. No new syntax can be defined, and only the built-in operators can be overloaded. In addition, a *class* is essentially a data type together with its implementation. This means that two variables of the same type that are to be implemented differently have to be of different classes. It also means that conversions of representation have to explicitly appear in the abstract program. This, together with the fact that the interface between an abstract program and an implementation is the procedure or function call, means that the abstract algorithm will be far removed from its implementation. Too much must be done to the abstract program by the programmer to make it efficiently implementable.

1.1 Program Refinement

The notion of program refinement is fundamental in computing science. When one refers to program refinement, one has to specify what a program is. A program may be a specification that is expressed in a formal language, like predicate calculus or temporal logic. Or, it may be a detailed description of machine instructions. Program refinement is the process by which a program that is expressed in a high-level form is changed to another one in a more concrete form. The refinement of a program need not be one big step; it may be broken down to smaller steps, which can be further subdivided to even smaller steps, and so on.

A specification language is usually very expressive and contains powerful operators and statements, and rich data types. Refinement of a specification entails refinement of the operators and statements as well as the data types that are involved in the

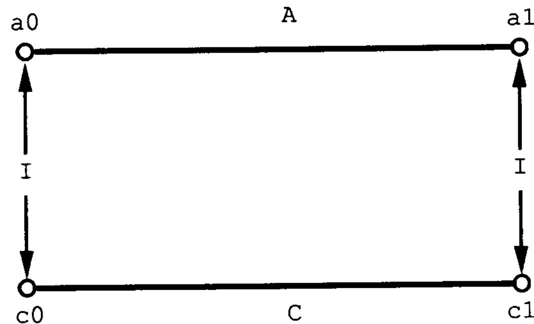


Figure 1.1: Relationship of abstract and concrete state changes.

specification. Replacing powerful operators and statements by simpler ones is known as *procedural refinement.* Replacing rich data types by simpler ones is known as *data refinement.*

[Gri81,Dij76] give a methodology for refining a specification that is expressed in predicate logic into a program that can be executed by a machine. Their work focuses mainly on procedural refinement. They give heuristics for refining specifications until they are in a suitable form so that concrete programs can be derived from them. Since the refinement takes place in a logical framework, the derived programs are provably correct with respect to their specifications.

On the other hand, [MG90,Mor89] give a calculus for refining specifications into concrete programs, focusing, mainly, on data refinement. They also give the required proof obligations for the refinement to be proved correct. [GP85] and [GV91] also give a methodology for data refinement and the required proof obligations for the refinement to be proved correct.

Figure 1.1 shows the relationship between abstract and concrete state changes. In that figure, $a0$ and $a1$ are two states of the abstract program and A describes a state change from $a0$ to $a1$, i.e. it is an abstract statement. Similarly, $c0$ and $c1$ are two states of the concrete program that is a refinement of the abstract one, and C is a state change from $c0$ to $c1$, i.e. it is a concrete statement. I is a coupling invariant, i.e. a predicate that describes the relation between abstract and concrete variables. If $c0$ is the state of the concrete program that corresponds to abstract state $a0$, then $I(a0, c0)$ holds. Execution of abstract statement A from state $a0$ results in state $a1$. If C is a refinement of A, then execution of concrete statement C from state $c0$ results in a state $c1$ such that $I(a1, c1)$ holds. In other words the diagram in Figure 1.1 commutes. This requirement is expressed as a proof obligation in [MG90, Mor89] and [GP85].

According to [MG90,Mor89], the proof obligation for concrete statement C to refine abstract statement A is

$$(\exists\, b \mid : I \wedge wp(A, X)) \Rightarrow wp(C, (\exists\, b \mid : I \wedge X)) \tag{1.1}$$

for all predicates X that contain no concrete variables. Predicate wp is the weakest precondition, as defined in [Dij76]: for a statement S and a predicate P, $wp(S, P)$ is the weakest predicate that describes the set of states $\mathcal{S}$ such that execution of S begun in a state in $\mathcal{S}$ terminates in a state described by P. Let X be a predicate that holds in state $a1$ and contains only abstract variables. Assume for simplicity that the abstract program contains one abstract variable b (say). Assume also that there exists a value for b such that $I \wedge wp(A, X)$ holds, i.e. it satisfies coupling invariant I in state $a0$. If execution of C starts in state $c0$ it will terminate in state $c1$ such that there must exist a value of b for which the coupling invariant holds, i.e. $I \wedge X$.

According to [GP85], the proof obligation for concrete statement C to refine abstract statement A is

$$I \wedge wp(A, X) \Rightarrow wp(C, \neg wp(A, \neg I)) \tag{1.2}$$

where X is a predicate that holds in state $a1$. Let $Y = wp(A, X)$ and assume that $I \wedge Y$ holds in state $c0$ where C starts execution. Then it must be the case that execution of C when started at $c0$ terminates in a state ($c1$ in the diagram of Figure 1.1) such that execution of statement A starting from state $a0$ may establish I. In other words, execution of C will not result in a state where I is not established after executing A.

[CU89] prove that the two formulations 1.1 and 1.2 are equivalent. They also give three additional equivalent formulations.

1.2 Rewrite Systems

In this thesis, we study algorithms for program transformations. We are mainly concerned with syntactic transformations of programs as opposed to semantic transformations. To this extent, the program transformations we deal with bear some similarity to transformations accomplished by rewrite-systems. In this section we briefly discuss term- and string-rewrite systems. In the next section we discuss the inapplicability of term-rewrite systems to the program transformations with which we are concerned.

Rewrite systems are mathematical formalisms that are used to compute by successive replacements. A rewrite system consists of a set of objects and a reduction relation. The reduction relation is used in replacing part of an object by another one.

Rewrite systems have the same computational power as Turing Machines [HU79]. There are two kinds of rewrite systems: term-rewrite and string-rewrite systems.

In term-rewrite systems [DJ90] the objects that are subject to replacement are terms. A term is either a variable, a constant symbol, or a function symbol applied to a list of terms. As an example, consider the set of objects generated from constant symbols a and b, function symbols f (of arity 2) and g (of arity 1), and the reduction relation

$$R = \{(f(a,b) \rightsquigarrow g(b)), (g(g(b)) \rightsquigarrow g(b)), (h(g(b), g(b)) \rightsquigarrow h(b))\}.$$

Using the rules in R, we can reduce term $h(f(a,b), g(f(a,b)))$ as follows.

$$\begin{array}{ll} & h(f(a,b), g(f(a,b))) \\ \rightarrow & h(f(a,b), g(f(a,b))) \\ \rightarrow & h(g(b), g(f(a,b))) \\ \rightarrow & h(g(b), g(g(b))) \\ \rightarrow & h(g(b), g(b)) \\ \rightarrow & h(b) \end{array}$$

A sequence like this is called a *reduction sequence.*

In a string-rewrite system [BO93] the objects that are reduced are strings of characters of a finite alphabet. As an example, consider alphabet $A = \{a, b\}$ and reduction relation

$$R = \{aabb \rightsquigarrow ab, bbaa \rightsquigarrow ba, abab \rightsquigarrow aa, baba \rightsquigarrow bb\}.$$

Using the rules in R we can reduce string $aabbaabb$ as follows.

$$\begin{array}{ll} & aabbaabb \\ \rightarrow & abaabb \\ \rightarrow & abab \\ \rightarrow & aa \end{array}$$

Depending on properties of the reduction relation, a string may or may not reduce to a unique string. For example, string $aabbaabb$ of the previous example may be reduced as follows.

$$\begin{array}{ll} & aabbaabb \\ \rightarrow & aababb \\ \rightarrow & aaab \end{array}$$

On the other hand, a string may reduce to a unique string via two different reduction sequences or it may not reduce to any string at all (i.e. give rise to an infinite reduction sequence).

Rewrite systems provide purely syntactical transformations. The reduction process assigns no semantics to the objects (terms or strings) that are reduced.

1.3 About this Thesis

In this thesis, we study the mechanics of program transformation according to the transform construct, which is described in a later chapter. The thesis focuses on the transformation system of *Polya* [GV92]. We give algorithms for transforming programs at the source level. The transformation of programs we are concerned with is completely syntactic; we do not cover issues of correctness of transforms.

First, we give an algorithm for transforming programs that contain variables and expressions that have simple types. The algorithm constructs a *coordinate transformation* of an abstract program based on a set of transforms and transform directives applied to it. A coordinate transformation of a program is an equal program in which one or more program variables and operations on them have been replaced by new ones. Then, we study the transformation of programs that contain expressions that have compound types (like *set*(*int*)). We allow both the type constructor (*set* in this case) and the component expressions (of type *int*) to be transformed, and no additional rules need be added to transforms for the type constructor. In the sequel, we study the transformation of functions and procedures that are called with arguments that need to be transformed, and then we study the transformation of program-transformation rules. Finally, we give examples that illustrate these methods.

An important methodological issue related to our work is the so-called *partial implementation of an abstract type.* An implementation of an abstract type is partial if either some operations of the type are not implemented or there are preconditions that have to be satisfied for an implementation of the operation to be correct. The main motivation for the partial implementation of an abstract type is efficiency. For example, depending on what operations on sets are used in a program, the *union-find* algorithm that implements efficiently set operations *union* and *find* may be more appropriate to use than a hash-table implementation for sets. The partial implementation of abstract types was studied in [Pri87]. In that thesis, operations of data types were implemented by explicit transform rules that describe how a part of a program can be refined to another one. The transform rules that were considered in that work were mainly rewrite rules of a rewrite system.

In the present work, we also consider *representations* of expressions and *transfor-*

mations of expressions and statements, which are formally defined in a later chapter. Representations are defined only for expressions (not for statements). As mentioned before, the coupling invariant is a predicate that connects the abstract and concrete variables.

An expression r is a *representation* of an expression e if it contains no abstract variables and the coupling invariant holds for r and e. For example, consider representing expressions of type *bool* with natural numbers, where the coupling invariant is: "a boolean expression e is *true* if and only if its representation r (a natural number) is greater than zero", i.e. $I(e, r)$ is $e \equiv r > 0$. Now consider boolean expressions $b1$ and $b2$, and assume that they are both represented with natural numbers as just explained. Let also $n1$ be the representation of $b1$, and $n2$ be the representation of $b2$. Then, the representation of $e' = b1 \wedge b2$ is $r' = n1 * n2$, and it is straightforward to prove that the coupling invariant holds for e' and r'.

A *transformation* of an expression or statement is an equal expression or statement, respectively, in which one or more variables have been replaced by new ones. The difference between the transformation and a representation of an expression can be explained as follows. Let $I(a, c)$ be the coupling invariant that connects variables a and c. Assume that abstract expression e refers to variables $a_0, \ldots, a_{n-1}$ that are to be replaced by $c_0, \ldots, c_{n-1}$, and assume that $I(a_i, c_i)$ holds for $0 \leq i < n$. Then, e may be replaced by f iff $e = f$, and f is a representation of e iff $I(e, f)$ holds. In summary,

$$\begin{array}{lll} f \text{ is a transformation of } e & \text{iff} & I(a_0, c_0) \wedge \cdots \wedge I(a_{n-1}, c_{n-1}) \Rightarrow e = f \\ f \text{ is a representation of } e & \text{iff} & I(a_0, c_0) \wedge \cdots \wedge I(a_{n-1}, c_{n-1}) \Rightarrow I(e, f). \end{array}$$

An important point to note is that a given expression may have more than one representation, and that there may be several transform rules that prescribe ways of converting one representation to another one. For example, expressions of type *bool* may also be represented with floating-point numbers, and there may be transform rules that convert a natural-number representation of a boolean expression to a floating-point-number representation, and vice versa.

The necessity for using representations of expressions becomes apparent when we consider the implementation of arbitrary expressions that contain variables to be transformed. Representations of expressions provide a way for implementing an expression from the implementations of its subexpressions.

The notion of expression-representations renders term-rewrite systems inapplicable for our work. Application of a transform rule to part of a program is similar to a term-reduction. The difference is that, in a reduction of a term with a term-rewrite system, a single redex is chosen at each reduction step and gets replaced by a new term. In our case, the choice of which representation to use for an expression may be delayed until enclosing constructs are processed. Expressing this property with

rules of a term-rewrite system would require an exhaustive set of rewrite rules that involves all possible combinations of nesting program expressions.

In addition, term-rewrite systems are rendered inapplicable for the transformation of expressions that have compound types (for example $seq(int)$). The transformation of type constructors (like *seq*) and component types (like *int*) should be independent of each other as far as transform rules are concerned. This independency requirement cannot be readily expressed with rules of a term-rewrite system.

Finally, term-rewrite systems are inapplicable for the transformation of functions and procedures whose arguments are transformed. When a call to a function or procedure involves arguments that need to be transformed, a new copy of the function or procedure need be constructed, in which the appropriate parameters are suitably transformed. This requirement cannot be expressed with rules of a term-rewrite system.

Of course the transformation of a program can be simulated by a term-rewrite system, since, as stated before, such a system has the power of a Turing Machine, but this is not our objective when carrying out program transformations.

1.4 Related work

The *CIP* project [Par90] focused on a transformational methodology for program development. The transformational methodology of *CIP* involves both procedural and data refinement. Transformations of programs in *CIP* involves semantic, as opposed to our purely syntactic transformations.

Predicate calculus and algebraic specifications are used for specifying programs and abstract types in the *CIP* framework. The program specification is subsequently transformed to a concrete program in a series of steps. The transformation process is not fully automated. Since semantic issues are involved, the system needs user input. The user of the system is responsible for choosing appropriate rules to come up with a concrete program that satisfies the specifications.

When transforming a program in *CIP* there is an interaction between the user and the system. The decisions of the user are actually transformation directives that are applied to the original program. In other words, the directives for transforming a program are distributed and are part of the transformation process. If a different transformation of the original program is desired, then different decisions need to be taken by the user.

In our case, the directives for transforming a program are centralized and are part of the program text. When a different transformation needs to be applied to a program, a single directive need be changed.

Chapter 2

Preliminaries

We give the necessary technical material for understanding the rest of the thesis.

2.1 Types

The types of expressions that we later show how to transform are essentially the data types of *Polya* [GV92]. We give a brief overview of *Polya*'s type system.

Polya's type system [Smi91] is essentially *ML*'s type system [Mil86], which is based on the type system of Damas and Milner [DM82], extended to include overloading and subtyping.

The definition of a new type in *Polya* includes definitions for literals and operations of the type. The definition of an operation involves definition of types of its components, possible constraints on these types, as well as concrete syntax for the operation. The definition of a new type in *Polya* is intended to extend the syntax of the language to include the concrete syntax of the operations of the type.

There are three kinds of operations: functions, expressions, and statements. Consider for example type $seq(t)$, the type of sequences that have elements of type t, shown in Figure 2.1. Type t can be any type; *seq* is a polymorphic type constructor. Literal *seq_nil* denotes the empty sequence, and is written as "$\langle\langle\rangle\rangle$".

Function *seq_rev* has one parameter, s, which is of type $seq(t)$ (the type of s is enclosed in parentheses). It yields a value of type $seq(t)$. If a program contains a variable $s{:}seq(t)$, then $seq_rev(s)$ is a legal function call.

Expression *seq_sel* has two components: an expression s of type $seq(t)$ and a variable i of type int. The type of the result is t. If a program contains a variable $s{:}seq(t)$ and a variable $i{:}int$, then $s\langle\langle i\rangle\rangle$ is a legal expression.

Statement *seq_iter* has three components: a variable v of type t, an expression s of type $seq(t)$, and a statement z. If a program contains a variable $s{:}seq(t)$ and

```
type seq(t)
literals
    seq_nil as "⟨⟨⟩⟩";
    ...
operations
    seq_rev exp-s (s:seq(t)) seq(t) ···;
    seq_sel exp-s exp-i (s:seq(t); i:int) t as s⟨⟨i⟩⟩;
    seq_iter var-v exp-s stmt-z (s:seq(t)) ···;
    ...
end
```

Figure 2.1: Definition of type *seq*(t).

a variable $e{:}t$, then "**foreach** e **in** s **do** $\cdots$" is a legal statement.

The concrete syntax given in the type definition is used to augment *Polya* so that operations that are defined in the type definition can appear in a program like any other primitive operation of the language. For example, given the previous type definition, the following is a syntactically correct *Polya* program fragment.

```
var s:seq(int) := ⟨⟨⟩⟩;
var i:int, k:int := 0;
foreach i in s do k := k + s⟨⟨i⟩⟩;
...
```

2.2 Terms

Let $\mathcal{C}$ be a set of constants, $\mathcal{V}$ a set of variables, and $\mathcal{F}$ a set of function symbols such that $\mathcal{C} \cap \mathcal{V} \cap \mathcal{F} = \emptyset$. With each function symbol F in $\mathcal{F}$ we associate its arity α_F, which is a positive number. The set $\mathcal{T}$ of terms is defined as follows.

- A variable $?v$ in $\mathcal{V}$ is a term.
- A constant c in $\mathcal{C}$ is a term.
- If $\bar{t}$ is a list of terms, and F in $\mathcal{F}$ is a function symbol of arity $\alpha_F = \#\bar{t}$, then $F(\bar{t})$ is a term, called a *compound term* (and the $\bar{t}_i$'s are its components).

By convention, variables begin with one or more "?" symbols.

A term that contains no variables is called a *ground term.* A term that contains at least one variable is called a *non-ground term.* For example, assume that F is

a function symbol of arity 1, C is a constant symbol, and $?s$ is a variable. Then, C and $F(F(C))$ are ground terms, and $?s$ and $F(?s)$ are non-ground terms.

For a term t in $\mathcal{T}$, denote by $Var(t)$ the set of variables that appear in t.

$$Var(t) = \begin{cases} \{?v\} & \text{if } t \text{ is } ?v \\ \emptyset & \text{if } t \in C \\ (\cup i \mid 0 \leq i < n : Var(t_i)) & \text{if } t \text{ is } F(t_0, \ldots, t_{n-1}) \end{cases}$$

Let $?v\#t$ be the number of occurrences of variable $?v$ in t. A term t is *simple* if

$$(\forall ?v \mid : ?v\#t \leq 1).$$

A *substitution* is a finite mapping from variables to terms. We write $\sigma = \{?v_0 \mapsto t_0, \ldots, ?v_{n-1} \mapsto t_{n-1}\}$ for a substitution. The *domain* of a substitution is the set of variables for which it is defined, i.e.

$$dom(\sigma) = \{?v \mid (\exists t \mid : ?v \mapsto t \in \sigma)\}.$$

For example, $dom(\sigma) = \{?v_0, \ldots, ?v_{n-1}\}$. If $?v \in dom(\sigma)$, then $\sigma(?v)$ is the term to which $?v$ is mapped by σ. If $?v \notin dom(\sigma)$, then $\sigma(?v)$ is undefined.

Define $\hat{\sigma}$ to be the extension of σ such that $\hat{\sigma}(?v) = ?v$ for variables $?v$ that are not in $dom(\sigma)$, i.e.

$$\hat{\sigma}(?v) = \begin{cases} \sigma(?v) & \text{if } ?v \in dom(\sigma) \\ ?v & \text{if } ?v \notin dom(\sigma) \end{cases}$$

A substitution $\sigma = \{?v_0 \mapsto t_0, \ldots, ?v_{n-1} \mapsto t_{n-1}\}$ is *simple* if none of the $?v_i$'s occur in the t_i's:

$$dom(\sigma) \cap (\cup i \mid 0 \leq i < n : Var(t_i)) = \emptyset.$$

Application $t \cdot \sigma$ of a substitution σ to a term t is defined as follows.

$$t \cdot \sigma = \begin{cases} \hat{\sigma}(?v) & \text{if } t \text{ is } ?v \\ t & \text{if } t \in C \\ F(t_0 \cdot \sigma, \ldots, t_{n-1} \cdot \sigma) & \text{if } t \text{ is } F(t_0, \ldots, t_{n-1}) \end{cases}$$

Theorem 2.2.1 *There is a linear algorithm that, given two simple terms s and t for which $Var(s) \cap Var(t) = \emptyset$, finds a substitution σ such that $s \cdot \sigma = t \cdot \sigma$, if such a substitution exists, otherwise it reports failure. Furthermore, if σ exists, then it is simple.*

Proof: This is a special case of the Unification Theorem [Rob65]. Call $subst(s, t, \emptyset)$ of function *subst*, shown in Figure 2.2, constructs substitution σ of the theorem if such substitution exists. Otherwise, it returns special symbol $\perp$, to indicate failure.

$$subst(?v, t, \sigma) = \sigma \cup \{?v \mapsto t\}$$
$$subst(t, ?v, \sigma) = \sigma \cup \{?v \mapsto t\}$$
$$subst(c, c, \sigma) = \sigma \qquad \textbf{Comment } \text{for } c \text{ in } \mathcal{C}$$
$$subst(F(\overline{s}), F(\overline{t}), \sigma) = subst'(\overline{s}, \overline{t}, \sigma)$$
otherwise $\perp$

$$subst'(\overline{s}, \overline{t}, \perp) = \perp$$
$$subst'([\,], [\,], \sigma) = \sigma$$
$$subst'(\overline{s}, \overline{t}, \sigma) =$$
$$\textbf{if } \#\overline{s} > 0 \land \#\overline{t} > 0 \rightarrow subst'(\overline{s}[1..], \overline{t}[1..], subst(\overline{s}_0, \overline{t}_0, \sigma))$$
$$[\,]\ \#\overline{s} = 0 \lor \#\overline{t} = 0 \rightarrow \perp$$
$$\textbf{fi}$$
otherwise $\perp$

Figure 2.2: Definition of function *subst*.

Now consider the call $subst(s', t', \sigma)$ and assume that σ is simple. Since terms s and t are simple by hypothesis, and $Var(s) \cap Var(t) = \emptyset$, it follows that for every subterm s' of s and t' of t in the call $subst(s', t', \sigma)$

$$Var(s') \cap dom(\sigma) = Var(t') \cap dom(\sigma) = \emptyset.$$

Initially $\sigma = \emptyset$, which is a simple substitution. We distinguish the following cases.

1. If s' is $?v$, since $?v \notin Var(t')$, then $\sigma \cup \{?v \mapsto t'\}$ is simple.
2. Likewise, if t' is $?v$, then $\sigma \cup \{?v \mapsto t'\}$ is simple.
3. If s' is $F(\overline{s}')$ and t' is $F(\overline{t}')$, then $subst'$ augments σ with the bindings that result from the calls $subst(\overline{s}'_i, \overline{t}'_i)$, for $0 \leq i < \#\overline{s}'$. Since $\overline{s}'$ and $\overline{t}'$ are simple, it follows by an inductive argument that the resulting substitution σ is simple.

Hence, $subst(s', t', \sigma)$ yields a simple substitution. It is also easy to see that this algorithm is linear in the size of its inputs. □

Types

As mentioned before, the types of expressions that we later show how to transform are essentially the data types of *Polya*, and they are actually terms.

In the case of types, $\mathcal{V}$ is the set of *type variables*, $\mathcal{C}$ is the set of *base* types, and $\mathcal{F}$ is the set of *type constructors.* We denote by τ the set of types. When we refer to types we use the terms *ground type*, *non-ground type*, *compound type*, and *component type*.

Chapter 3

The Transform

3.1 Introduction

A language construct, the transform (as a feature of the programming language *Polya* [GV92]), has been proposed for program transformations at the source program level [GV91]. The advantage of the transform is that it lets programmers write programs in an abstract form, which can then be transformed, using directives, to a more concrete form that can be accepted and executed by a machine. Hence, programs may be written at a much higher (more abstract) level than most conventional programming languages allow. A transform describes a coordinate transformation, i.e. the replacement of some program variables by new ones, perhaps of different types. A coordinate transformation can be a partial implementation of an abstract type [GP85], a data refinement [MG90,Mor89], or a general transformation like the replacement of a dummy variable in a loop for efficiency purposes [Knu63]. A transform directive specifies which transform to use to replace a particular program variable.

The abstract program is transformed to a more concrete form with a set of transforms. Each transform contains rules that prescribe various ways of transforming components of a program. The selection of the transformations to be applied to program variables is localized. By changing a single directive, a new transformation can be applied to the abstract program, resulting in a probably different concrete program. In this chapter we give the necessary background for transforms and transform directives.

The declaration of a transform has the following form.

$$
\begin{array}{l}
\textbf{transform } T(\overline{p}{:}\overline{t});\\
\quad \textbf{var } \overline{av}{:}\overline{at} \textbf{ into var } \overline{cv}{:}\overline{ct}\\
\quad \{\textit{coupling invariant}\}\\
\quad \textit{transform rules}
\end{array}
\qquad (3.1)
$$

with
  local declarations
end

In the sequel we call the program that is being transformed the *abstract program* and the program that is the result of the transformation the *concrete program*. The same naming convention applies to parts of the abstract and concrete programs; for example we use the terms *abstract statement*, *abstract expression*, *concrete statement*, and *concrete expression*. If the abstract program is represented by a tree, we refer to it as the *abstract tree*, which has *abstract nodes*. Similarly, we use the terms *concrete tree* and *concrete node*.

An example of a transform is given in Figure 3.1. Transform *BN* has one abstract variable, *b*:*bool*, and one concrete variable, *j*:*nat*. It has no parameters, and its coupling invariant is $b \equiv j > 0$. The parenthesized numbers on the left side of the transform rules are used only for reference purposes and do not appear in an actual transform. We will return to transform *BN* later, after we examine the components of a transform.

```
transform BN;
 (0)     var b:bool                 into  var j:nat
         {Coupling invariant: CI(b,j) = b ≡ j > 0}
 (1)   [] [[true:bool]]              =     1
 (2)   [] [[false:bool]]             =     0
 (3)   [] [[(BN¡b)]]                 =     (BN¡j)
 (4)   [] [[BN¡b¡1 ∨ BN¡b¡2]]        =     BN¡j¡1 + BN¡j¡2
 (5)   [] [[BN¡b¡1 ∧ BN¡b¡2]]        =     BN¡j¡1 * BN¡j¡2
 (6)   [] [[¬BN¡b]]                  =     if BN¡j > 0 then 0 else 1
 (7)   [] [[exp-x:bool]]             =     if x then 1 else 0
 (8)   [] BN¡b                      into  BN¡j > 0
 (9)   [] BN¡b¡1 := BN¡b¡2          into  BN¡j¡1 := BN¡j¡2
(10)   [] if BN¡b then stmt-s1
            else stmt-s2            into  if BN¡j > 0 then s1 else s2
(11)   [] var-v:bool := BN¡b        into  v := BN¡j > 0
end
```

Figure 3.1: *BN* implements a variable of type *bool* with a variable of type *nat*.

3.2 Components of a transform

Transform name and transform parameters

Each transform is uniquely identified by its name. The name of the transform in Figure 3.1 is BN. A transform may have one or more parameters, which are enclosed in parentheses following the name of the transform. If a transform has no parameters, the parentheses are omitted. The parameters are separated by commas; both the name and the type of each parameter have to be given. Transform T above has parameters $\overline{p}$, which have types $\overline{t}$, respectively.

Abstract and concrete variables

In transform skeleton T of Section 3.1, $\overline{av}$ is a list of *abstract variables* (and $\overline{at}$ is a corresponding list of their types), and $\overline{cv}$ is a list of *concrete variables* (and $\overline{ct}$ is a corresponding list of their types). The abstract variables of a transform describe the kinds of program variables on which the transform can be applied. The concrete variables of a transform describe the program variables that are generated as the result of applying the coordinate transformation that is described by the transform.

The two lists $\overline{av}$ and $\overline{cv}$ need not have the same length. Both are lists of dummies, in that their consistent renaming throughout the transform will not change the meaning of the transform.

The abstract variables of a transform are matched to the program variables on which the transform is applied (with a transform directive). The concrete variables of a transform are used for generating the appropriate program variables that result from the coordinate transformation. In a later section, we discuss how the abstract and concrete variables of a transform are used.

Coupling invariant

The *coupling invariant*, a predicate, relates the abstract and concrete variables. The coupling invariant has no bearing on the transformation process, which is purely syntactic. It is used by the author of a transform for proving the correctness of each transform rule. Later in this section, we discuss the proof obligations for the author of a transform.

Consider a transform T that transforms $v1{:}t1$ to $v2{:}t2$, and let $I(v1, v2)$ be its coupling invariant. The coupling invariant of T gives rise to the definition of a *representation* of an expression according to T. An expression $r{:}t2$ is a T-representation of an abstract expression $e{:}t1$ iff $I(e, r)$ holds. For example, for a variable $v{:}complex$

that is represented by

$$u : \textbf{record } re, im{:}real \textbf{ end}$$

the coupling invariant is $v = u.re + i \cdot u.im$.

Transform rules

Each rule in the list of *transform rules* describes a way of replacing an expression or statement that involves one or more abstract variables. Each rule may have one of the following forms.

$\{P\}$ *expression-pattern* **into** *expression-replacement*
$\{P\}$ *statement-pattern* **into** *statement-replacement*

A transform rule may be preceded by a precondition (P, in the example above) enclosed in braces. The user of a transform has the obligation to verify that the precondition of a rule is satisfied at the place in the program where the rule is applicable. The precondition has no effect on the transformation process, which is a syntactic process.

The first kind of rule prescribes the transformation of an expression that may contain abstract variables or expressions. Whenever *expression-pattern* matches an expression e of an abstract program (pattern matching is defined in Section 4.3), e can be replaced by the corresponding instance of *expression-replacement.*

The second kind of rule prescribes the transformation of a statement that may contain abstract variables or expressions. Whenever *statement-pattern* matches a statement s of an abstract program, s can be replaced by the corresponding instance of *statement-replacement.*

The rules above describe the *transformation* of an expression or statement. There is a third kind of rule, a *representation rule*, which has the following form

$\{P\}$ ⟦*expression-pattern*⟧ $=$ *expression-replacement* ¡*representation*

A transform may have rules of the third kind only if it transforms exactly one abstract variable to one concrete variable. A representation rule prescribes a way of constructing a *representation* of an expression. If *expression-pattern* matches an expression of an abstract program, then the corresponding instance of *expression-replacement* is used to construct the specified representation of the expression. The type of the corresponding instance of *expression-replacement* is the same as the type of the concrete variable of the transform whose representation is constructed. A representation of an expression depends on the representations and/or transformations of its subexpressions. The representation of a variable according to a transform T is prescribed

by the first **into** rule of T, and the representations of constants and expressions are prescribed by other transform rules of T. A pattern can refer to a specific representation of an expression, which can be used in the corresponding replacement of the transform rule. The set of representation rules typically form a recursive definition of the representation of any expression of the type.

The name of a transform identifies a representation of an abstract expression. Thus, an abstract expression may have a representation according to transform $T1$ (a $T1$-representation), and a representation according to transform $T2$ (a $T2$-representation). The *representation* after symbol "¡" in the third kind of transform rule specifies which representation of an abstract expression is constructed when the transform rule is applied. It also specifies the parameters of the transform whose representation is constructed (if the transform has parameters). If "¡*representation*" is omitted, then the transform rule prescribes the representation of an expression according to the transform in which the rule appears.

As mentioned above, an expression may have more than one representation with respect to a set of transforms. A special kind of representation rule is the one that specifies a *conversion of representation*: a function from one representation to another. If *expression-pattern* in the third kind of rule above has no component subpatterns, then the corresponding rule is a conversion of representation. The syntax of patterns and replacements of transform rules is discussed later in this section.

Proof obligations

The author of a transform has several proof obligations for proving the correctness of each transform rule, depending on the kind of a transform rule. Here, we outline the proof obligations for showing the correctness of a transform rule.

1. *Rules that prescribe the transformation of an expression*: The correctness of the rule can be shown by proving that the pattern and replacement of the rule are equal, under the assumption that the abstract and concrete variables that appear in the pattern and replacement satisfy their coupling invariants.

2. *Rules that prescribe the transformation of a statement*: The correctness of the rule can be shown by proving that the simultaneous execution of the pattern and replacement of the rule maintains the coupling invariant, under the assumption that the abstract and concrete variables that appear in the pattern and replacement of the rule satisfy their coupling invariants.

3. *Rules that prescribe the representation of an expression*: The correctness of the rule can be shown by proving that the pattern and replacement satisfy the coupling invariant, under the assumption that the abstract and concrete

variables that appear in the pattern and replacement of the rule satisfy their coupling invariants.

[MG90,Mor89] develop a calculus for data refinement of programs. In doing so, they give proof rules for the correctness of pattern-replacement pairs (in a different notation and setting). Their rules are more complicated than ours—but on the other hand they are more powerful; they can be used in place of ours. In this thesis we cover the algorithms for the transformation process, not issues of correctness of transforms.

Rule patterns and replacements

In this section, we discuss the syntax of patterns and replacements of transform rules. Examples are given using transform *BN* of Figure 3.1.

A pattern of a transform rule may have one of the following forms.

1. An abstract statement operator applied to subpatterns. For example, if *stmt_op* is an abstract statement operator of arity n and $\overline{p}$ is a list of n patterns, then *stmt_op* $\overline{p}$ is a pattern. Pattern *BN*¡*b*¡1 := *BN*¡*b*¡2 of rule (9) of *BN* is an example.

2. An abstract expression operator applied to subpatterns. For example, if *exp_op* is an abstract expression operator of arity n and $\overline{p}$ is a list of n patterns, then *exp_op* $\overline{p}$ is a pattern. Pattern *BN*¡*b*¡1 ∨ *BN*¡*b*¡2 of rule (4) of *BN* is an example.

3. A reference to a representation of an expression. For example, if T is the name of a transform and a is the name of its abstract variable, then *T*¡*a* is a pattern. Different instances of the same representation in the same pattern are distinguished by a number after a second "¡" symbol. For example, *T*¡*a*¡1 and *T*¡*a*¡2 refer to two (possibly different) T-representations of different subtrees of the abstract tree. Pattern *BN*¡*b*¡1 of rule (4) of *BN* is an example of this case.

4. **stmt**-s is a pattern, where s is an identifier. The scope of s is the transform rule in which it appears. If **stmt**-s appears more than once in a transform rule, then all occurrences of **stmt**-s refer to the same s. Pattern **stmt**-$s1$ of rule (10) of *BN* is an example.

5. **exp**-e:t is a pattern, where e is an identifier and t is a type. The scope of e is the transform rule in which it appears. If **exp**-e:t appears more than once in a transform-rule pattern, then all occurrences of **exp**-e:t refer to the same e. Pattern **exp**-e:*bool* of rule (7) of *BN* is an example.

6. **var**-v:t is a pattern, where v is an identifier and t is a type. The scope of v is the transform rule in which it appears. If **var**-v:t appears more than once in

a transform-rule pattern, then all occurrences of **var**-v:t refer to the same v. Pattern **var**-v:*bool* of rule (11) of *BN* is an example.

7. **const**-c\{*re*\}:t is a pattern, where c is an identifier and *re* is a regular expression that describes constants of type t. This pattern can be simplified to c:t if *re* is just a string c. The scope of c is the transform rule in which it appears. Pattern *false*:*bool* of rule (2) of *BN* is an example. The same pattern can be written as **const**-*false*\{*false*\}:*bool*.

8. (p) is a pattern, if p is a pattern. (*BN*¡*b*) of rule (3) of *BN* is an example.

A replacement of a transform rule has one of the following forms.

1. A concrete statement operator applied to subreplacements. For example, if *stmt_op* is an abstract statement operator of arity n and $\overline{r}$ is a list of n replacements, then *stmt_op* $\overline{r}$ is a replacement. Replacement *BN*¡*j*¡1 := *BN*¡*j*¡2 of rule (9) of *BN* is an example.

2. A concrete expression operator applied to subreplacements. For example, if *exp_op* is an abstract statement operator of arity n and $\overline{r}$ is a list of n replacements, then *exp_op* $\overline{r}$ is a replacement. Replacement *BN*¡*j*¡1 $*$ *BN*¡*j*¡2 of rule (5) of *BN* is an example.

3. A reference to a representation of an expression. For example, if T is a transform, a is the name of its abstract variable and c is the name of its concrete variable, then T¡c is a replacement. This replacement corresponds to the representation of an expression that is referred to by pattern T¡a of the same transform rule. Different instances of the T-representation in the same replacement are distinguished by a number that follows a second "¡" symbol. For example, T¡c¡1 and T¡c¡2 refer to two (possibly different) T-representations of different subtrees of the abstract tree. The *BN*¡*j*¡1 part of the replacement of transform rule (4) of *BN* is an example.

4. s is a replacement, where s is a name defined with **stmt**-s in the pattern of the corresponding transform rule. The $s1$ part of the replacement of transform rule (10) of *BN* is an example.

5. e is a replacement, where e is a name defined with **exp**-e:t in the pattern of the corresponding transform rule. The x part of the replacement of transform rule (7) of *BN* is an example.

6. v is a replacement, where v is a name defined with **var**-v:t in the pattern of the corresponding transform rule. The v part of the replacement of transform rule (11) of *BN* is an example.

7. c is a replacement, where c is a name defined with **const**-$c\{re\}$:t in the pattern of the corresponding transform rule.

8. A reference to a parameter of a transform is a replacement. For example, if p is a parameter of a transform T whose abstract variable is a, then T¡p is a replacement. This replacement corresponds to the parameter that is associated with the T-representation that is referenced by the T¡a part of the corresponding pattern of the transform rule. Different instances of the same parameter in the same replacement are distinguished by a number following a second "¡" symbol. For example, T¡p¡1 and T¡p¡2 refer to the two (possibly different) instances of parameter p of the T-representations that are referenced by T¡a¡1 and T¡a¡2, respectively, in the corresponding pattern.

9. (r) is a replacement, where r is a replacement. Replacement (BN¡j) of transform rule (3) of BN is an example.

In the case of a rule that specifies a representation of an expression, the replacement of the rule has to specify which concrete representation of the abstract expression is constructed and the values of the parameters of the corresponding transform. For example, if a rule specifies the construction of the T-representation of an expression, then this is denoted by the symbols "¡ T" that follow the replacement (assuming that transform T has no parameters). If no such symbols follow the replacement, then the rule specifies a representation according to the transform in which it appears.

Assume that T has an abstract variable a and parameters $\overline{p}$. Each reference T¡a to a T-representation of an expression has an instance of parameters $\overline{p}$ associated with it. We use the notation T¡$\overline{p}_i$ ($0 \leq i < \#\overline{p}$) to refer to parameter $\overline{p}_i$ that is associated with the T-representation of an expression.

If a transform rule specifies the construction of a T-representation for an expression, then it has to specify the values of parameters of T. For example, assume that T has one parameter p and one abstract variable a. If a rule specifies the T-representation of an expression and the associated value of parameter p is two times the value of p that is associated with T¡a, then the notation ¡$T(2 * T$¡$p)$ is used for the *representation* part of the rule. The value of the parameter of the resulting representation can be a function of the parameters of the transforms that are associated with transform references only.

3.2.1 Local declarations

A transform may contain declarations that are local to its body, which are given after keyword **with**. If a transform contains no local declarations, then **with** may be

omitted. Local declarations involve declarations of constants, functions, procedures, and types. The scope of these declarations is the rule-replacements of the transform.

3.3 An example of a transform

Consider again transform *BN*, which was shown in Figure 3.1 earlier in this chapter. *BN* is an example of a transform for the transformation of variables of type *bool*. *BN* provides a way of replacing a variable of type *bool* by a variable of type *nat* and ways of replacing boolean expressions that contain the operators $\vee$, $\wedge$, and $\neg$ by new ones that do not contain these operators.

Here we discuss some points about transform *BN*.

- As mentioned before, transform *BN* has no parameters. The precondition of each rule is *true* and is omitted. Each one of the rules (1)–(7) prescribes the construction of a *BN*-representation, so the symbol "¡*BN*" is omitted from the end of the rule.

- Transform *BN* contains rules that define the *BN*-representation of boolean constants *true* and *false* (rules (1) and (2)). According to *BN*, one representation of *true* is natural number 1 and one representation of *false* is 0.

- Transform *BN* contains rules that define the *BN*-representation of boolean expressions that are formed using operators $\vee$, $\wedge$, and $\neg$ from the *BN*-representations of their subexpressions (rules (4), (5), and (6)). For example, let $e1 \vee e2$ be a boolean expression that appears in a program (where $e1$ and $e2$ are boolean expressions). If $e1$ has a *BN*-representation $r1$ (say) and $e2$ has a *BN*-representation $r2$ (say), then according to rule (4) the *BN*-representation of $e1 \vee e2$ is $r1 + r2$.

- Rule (7) prescribes the *BN*-representation of any expression of type *bool*. It is a conversion of representation from the *default* representation of an expression (which is the expression itself) to its *BN* representation.

- Rule (8) prescribes the *transformation* of an expression that has a *BN*-representation. If expression e has a *BN*-representation r, then according to rule (8) the transformation of e is $r > 0$.

- Rule (9) prescribes the *transformation* of a statement that involves abstract variables that have *BN*-representations.

- Rule (10) prescribes the transformation of an **if** statement whose boolean expression has a *BN*-representation. (This rule is not needed in *BN* and is presented only as an example).

- Rule (11) prescribes the transformation of an assignment statement whose right-hand-side expression has a *BN*-representation. (This rule is not needed in *BN* and is presented only as an example).

The correctness of *BN* can be shown by proving each rule correct, as was discussed earlier. For example, the correctness of rule (5) of *BN* is shown as follows.

$$\begin{array}{ll} & b1 \wedge b2 \\ = & \quad \langle\!\langle CI(b1, j1), CI(b2, c2) \rangle\!\rangle \\ & j1 > 0 \wedge j2 > 0 \\ = & \quad \langle\!\langle \textit{since } j1,\ j2 \textit{ are of type nat} \rangle\!\rangle \\ & j1 * j2 > 0 \end{array}$$

3.4 Transform directives

A *transform directive* specifies a transform to be applied to a variable. There are two kinds of transform directives.

Directive

change $\overline{v}$ **using** $T(\overline{w})$

specifies that transform T, with arguments $\overline{w}$, is to be applied to variables $\overline{v}$. T is *applicable* to $\overline{v}$ only if the types of $\overline{v}$ are the same as the types of the abstract variables of T. When a transform directive is processed, $\overline{v}$ is replaced by new variables that have the same types as the concrete variables of T. Different variables of the same type may be transformed with different transforms.

For example, if abstract variable a:*bool* is to be transformed with *BN*, the transform directive would be the following.

change a **using** *BN*

Transform *BN* is applicable to a since the type of variable a is the same as the type of the abstract variable of *BN*. When this directive is given, a is replaced by a new variable a_c (say) of type *nat*. *BN* has no parameters. If *BN* had one parameter of type *int*, then a legal transform directive would had been the following.

change a **using** $BN(10)$

A second kind of directive may be given for the transformation of variables and expressions. When directive

default t **using** $T(\overline{w})$

is given, then, by default, every variable of type t that is not transformed by an explicit directive is transformed with T. In addition, every expression of type t is removed from the program.

3.5 Examples of using a transform

Consider transform BN of Figure 3.1 and suppose that a program contains the following definitions and directives.

var a:*bool*;
var b:*bool*;
...
change a **using** BN;
change b **using** BN

According to these directives, variables a and b are transformed with transform BN. The definitions of a and b are replaced by

var a_c:*nat*;
var b_c:*nat*

where a_c and b_c are fresh concrete variables that correspond to a and b, respectively. Variables a_c and b_c are the BN-representations of a and b, respectively.

Suppose that an abstract program contains subexpression $a \vee b$. According to rule (4) of transform BN, its BN-representation is $a_c + b_c$ since a_c is the BN-representation of a, and b_c is the BN-representation of b. Rule (8) of transform BN can be used to construct the following transformation of the previous expression

$$(a_c + b_c) > 0.$$

Employing rule (9) of transform BN, a transformation of statement

$$a := a \vee b$$

is the following.

$$a_c := a_c + b_c$$

Suppose that variable c is defined in the abstract program as

var c:*bool*

and that no directive is given for its transformation. Then, statement

$$c := a \vee b$$

gets transformed to

$$c := (a_c + b_c) > 0$$

as prescribed by rule (11) of BN. On the other hand, statement

$$c := c \vee b$$

gets transformed to

$$c := c \vee (b_c > 0).$$

It should be emphasized that a transform can be a *partial* implementation of a data type [Pri87]. Consequently, it may not provide implementations for all operations of the data type. In addition, different transforms may provide different partial implementations of the same data type. A given transform is suitable for the implementation of an abstract data type that is used in a program if it provides implementations for all operations of the data type that are used in the program. For example, it would be acceptable if transform BN did not contain any representation rule for operator $\vee$. In that case, BN could not be used in a program to transform a boolean expression that contains operator $\vee$.

3.6 Program transformation

A program is transformed successfully with a set of transforms $\mathcal{T}$ and a set of transform directives $\mathcal{D}$ if

1. For every list of program variables $\overline{v}$ for which a directive

   $$\textbf{change } \overline{v} \textbf{ using } T(\overline{w})$$

   is in $\mathcal{D}$ and T is applicable to $\overline{v}$, the transformation removes all occurrences of variables $\overline{v}$ from the program.

2. For a directive

   $$\textbf{default } t \textbf{ using } T(\overline{w})$$

   in $\mathcal{D}$, every variable of type t that is not transformed by an explicit directive is transformed with T, and every expression of type t is removed from the program.

If the abstract program is type correct and the transforms have been proved correct, then the transformed program is guaranteed to be type correct. Furthermore, if the abstract program is correct, and all preconditions of transform rules are satisfied at the places of the abstract program where the rules are applied, then the transformed program is correct.

Chapter 4

Transforming variables that have base types

4.1 Introduction

We present an algorithm for processing program transformations as described by the transform construct. The algorithm constructs a coordinate transformation of an abstract program based on a set of transforms and transform directives applied to it.

4.2 Overview

The algorithm for processing transforms and transform directives works in two phases. During the first phase, the source program, the transforms, and the transform directives are processed and converted to an internal representation, and the necessary lists and tables are constructed. The second phase carries out the actual program transformation. It processes the internal representation of the source program, and applies the transformations as directed by the transforms and the transform directives.

In this section, we give an overview of the transformation algorithm, we describe the representation of abstract trees, transforms, and transform directives, and we define functions that will be used later.

Symbol table entries for variables

The algorithm makes use of a symbol table where information about program identifiers is stored. The symbol table has an entry for each abstract program variable,

where its name and type is stored. It also has an entry for each concrete variable that is generated as the result of applying a transform directive to one or more abstract program variables.

Suppose transform T transforms $v1{:}t1$ to $v2{:}t2$. For a directive

$$\textbf{change } v \textbf{ using } T(\overline{w})$$

or a directive

$$\textbf{default } t1 \textbf{ using } T(\overline{w})$$

a new variable v_c (say) of type $t2$ is generated and a symbol table entry is created for it. The symbol table entry of v contains a reference to transform T, its arguments $\overline{w}$, and the corresponding concrete variable v_c.

Suppose T transforms $\overline{v1{:}t1}$ to $\overline{v2{:}t2}$. For a directive

$$\textbf{change } \overline{v} \textbf{ using } T(\overline{w})$$

a list of new variables $\overline{v_c}$ (say) of corresponding types $\overline{t2}$ is generated and a symbol table entry is created for each one of them. The symbol table entry of each v_i, $0 \leq i < \#\overline{v}$, contains a reference to transform T, its arguments $\overline{w}$, every other variable in $\overline{v}$, and the list of the corresponding concrete variables $\overline{v_c}$.

Let v be an abstract program variable. The following functions are used later and are assumed to be primitive.

1. *get_conc_vars*(v) is the list of concrete variables that are created when transform T is applied to v.

2. *get_abs_vars*(v) is the list of abstract variables that are transformed along with v when transform T is applied to v.

3. *get_trans*(v) is the transform that is applied to program variable v.

The generation of concrete variables that result from application of a transform to a list of program variables is discussed in Section 4.6.

Abstract tree nodes

As mentioned before, the first phase converts the source program into an internal representation, which is a tree (henceforth referred to as the *abstract tree*). A node of the abstract tree (an *abstract node*) has one of the following forms.

1. An abstract statement operator applied to substatements and subexpressions. For example, if *stmt_op* is an abstract statement operator of arity n and $\overline{s}$ is a list of n statements or expressions, then *stmt_op* $\overline{s}$ is an abstract statement node.

2. An abstract expression operator applied to subexpressions. For example, if *exp_op* is an abstract expression operator of arity n and $\overline{e}$ is a list of n expressions, then *exp_op* $\overline{e}$ is an abstract expression node.

3. A node that is labeled with an abstract variable *av*. For example *var* v is a node labeled with variable v.

4. A node that is labeled with a constant *ac*. For example *const* c is a node labeled with constant c.

5. A node that is labeled with a list of declarations of variables and their types. For example *decl* $\overline{v}{:}\overline{t}$ is such a node.

We assume that the source program has been type-checked and that each node of the abstract tree is annotated with its type. Statement nodes have type *void*.

With each node n of the abstract tree we associate the following values.

1. $V(n)$: is *true* iff there are variables that need to be replaced in the subtree rooted at n.

2. $C(n)$: is *true* iff there are constants that need to be replaced in the subtree rooted at n.

3. *reprs*(n): set of representations of node n.

4. *trans*(n): transformation of node n or $\perp$ if n has no transformation.

In a later section we discuss how these values are constructed.

For a node n of the abstract tree, the following functions are used later and are assumed to be primitive.

1. *type_of*(n) is the type of node n.

2. *is_var*$(n) \equiv$ "*n is a variable*".

3. *is_const*$(n) \equiv$ "*n is a constant*".

4. *get_var*(n) is the variable at node n.

5. *get_const*(n) is the constant at node n.

6. *mk_stmt_node*$($*stmt_op*$, \overline{n})$ is a statement node that has operator *stmt_op* and children $\overline{n}$.

7. *mk_exp_node*$($*exp_op*$, \overline{n})$ is an expression node that has operator *exp_op* and children $\overline{n}$.

8. *mk_var_node*(v) is a node labeled with variable v.

9. *mk_const_node*(c) is a node that is labeled with constant c.

10. *mk_decl_node*$(\overline{v}, \overline{t})$ is a declaration node labeled with variables $\overline{v}$ that have corresponding types $\overline{t}$.

In the following, the same name is used interchangeably for a node of the abstract tree and for a subtree of the abstract tree that is rooted at that node.

Transforms

The transforms are represented by a table and three lists of rules. Each transform rule r is represented by a record that contains the following information.

1. The pattern of r.

2. The replacement of r.

3. An indication if r is a representation or a transformation rule.

4. If r is a representation rule, the representation that is constructed (i.e. the name of the corresponding transform, and the expressions for constructing the associated parameters).

All representation rules that are not conversion-of-representation rules are kept in a list s_1. Similarly, all conversion-of-representation rules are kept in a list s_2 and all transformation rules form a list s_3. For a rule r, the following functions are used later, and are assumed to be primitive.

1. *pattern*(r) is the pattern of rule r.

2. *replacement*(r) is the replacement of rule r.

3. *representation*(r) is the name of the transform whose representation is constructed by rule r.

4. *parameters*(r) is the list of replacements for constructing the parameters that are associated with *representation*(r).

5. *trans_of*(r) is the name of the transform that contains rule r.

For each transform, its parameters, the list of its abstract variables and their types, and the list of its concrete variables and their types, are kept in a record. For a transform T, the following functions are used later, and are assumed to be primitive.

1. *parameters*(T) is the list of parameters of transform T.
2. *abs_vars*(T) is the list of abstract variables of transform T.
3. *conc_vars*(T) is the list of concrete variables of transform T.

In the sequel, $\mathcal{T}$ is the set of transforms that are used to transform a program.

The form of transform-rule patterns that are used in the algorithm is the following.

$$p = \mathit{stmt_op}\ \overline{p} \mid \mathit{exp_op}\ \overline{p} \mid T¡a¡k \mid \textbf{stmt-}s \mid \textbf{exp-}e{:}t \mid \textbf{var-}v{:}t \mid \textbf{const-}c\{re\}{:}t \mid (p)$$

The form of transform-rule replacements that are used in the algorithm is the following.

$$r = \mathit{stmt_op}\ \overline{r} \mid \mathit{exp_op}\ \overline{r} \mid T¡c¡k \mid s \mid e \mid v \mid c \mid T¡p¡k \mid (r)$$

Patterns and replacements are discussed in Section 3.2.

Transform directives

All transform directives are kept in a set $\mathcal{D}$. Each directive of the form

$$\textbf{change}\ \overline{v}\ \textbf{using}\ T(\overline{w})$$

is represented by a record that contains an indication that it is a **change** directive, the list of variables $\overline{v}$, the name of transform T, and the list of parameters $\overline{w}$.

Each directive of the form

$$\textbf{default}\ t\ \textbf{using}\ T\ (\overline{w})$$

is represented as a record that contains an indication that it is a **default** directive, the name of type t, the name of transform T, and the list of parameters $\overline{w}$.

For a directive D in $\mathcal{D}$, the following functions are used later, and are assumed primitive.

1. *is_change*(D) = "*D is a* **change** *directive*".
2. *is_default*(D) = "*D is a* **default** *directive*".
3. *vars_of*(D) = $\overline{v}$, if D is a **change** directive.
4. *type_of*(D) = t, if D is a **default** directive.
5. *trans_of*(D) = T.
6. *params_of*(D) = $\overline{w}$.

Program transformation

The first phase of the algorithm is essentially a front-end that processes the source program and converts it into an intermediate form [ASU86]. The details are omitted.

The second phase of the algorithm carries out the transformations described by the transforms, and the transform directives. It performs a bottom-up traversal of the abstract tree. At each leaf of the tree, the algorithm uses the transform directives to construct the representation of the node. As each node is visited, the representations and transformation for that node are constructed.

Let n be an expression, a variable, or a constant node. If $\neg(V(n) \vee C(n))$, then the transformation algorithm need not construct any transformations for it (but it tries to construct representations for the node, if possible, since they may be needed in the transformation of the program). If $V(n) \vee C(n)$, then the transformation algorithm tries to construct representations and a transformation for n, if there is no directive

$$\textbf{default } t \textbf{ using } T(\overline{w})$$

in $\mathcal{D}$, where $type_of(n) = t$.

Let n be a statement node. If $\neg(V(n) \vee C(n))$, then the transformation algorithm need not construct any transformation for n. If $V(n) \vee C(n)$, then the transformation algorithm tries to construct a transformation for it.

To construct a representation of an internal node n of the abstract tree, the algorithm tries to match the patterns of the representation rules with n. If a match is successful, the corresponding replacement of the rule is used and the concrete representation of the node is constructed as prescribed by the replacement. Constructing conversions of representation and transformations of a node is done in a similar way, by using the list of conversions-of-representation rules and the list of transformation rules, respectively.

After the construction of representations and transformations for the abstract program is complete, the algorithm checks if all directives in $\mathcal{D}$ are satisfied: if all variables v for which a **change** directive is given in $\mathcal{D}$ have been replaced by new ones and if the transformed program does not contain any expressions of type t if a directive "**default** t **using** ..." is in $\mathcal{D}$.

Given a program, a set of transforms and a set of transform directives, it may not be possible to find a transformation for the program. In such cases, the algorithm returns an indication that no transformation is possible. On the other hand, it may be possible to construct more than one transformation for the program. In such cases, heuristic methods can be employed to construct an appropriate transformation. The complexity of the structure of a pattern, the cost of the operations involved in a replacement, and the relative order of a pattern with respect to other patterns can

serve as some simple heuristics for choosing the most suitable transformation rule to apply.

4.3 Pattern matching

Let *Id* be the domain of names and $\mathcal{N}$ be the set of natural numbers. Let *AbsNodes* be the domain of abstract tree nodes. A *binding* is an ordered pair of the form (X, Y), where X is a member of $Id + (Id \times \mathcal{N})$ and Y is a member of $Id + AbsNodes$, i.e. a binding is a member of the cartesian product

$$(Id + (Id \times \mathcal{N})) \times (Id + AbsNodes).$$

An *environment* E is a set of bindings that denotes a function, i.e. for each element X of $Id + (Id \times \mathcal{N})$ there is at most one pair (X, Y) in E. The *domain* of an environment E is the set

$$dom(E) = \{X \mid : (\exists\, Y \mid : (X, Y) \in E)\}$$

and the *range* of an environment E is the set

$$rng(E) = \{Y \mid : (\exists\, X \mid : (X, Y) \in E)\}.$$

The *empty* environment is denoted by $\emptyset$. We distinguish a special environment, the *fail* environment, that is denoted by $\perp$. The fail environment is different from all other environments, including $\emptyset$. Its importance will be explained later, when the definition of pattern matching is given.

In the sequel, notation $[X \mapsto_E Y]$ is used for binding (X, Y) in E. If environment E is obvious from the context, then it is omitted from the subscript of $\mapsto$. If E is an environment and X belongs to $dom(E)$, then we write $E(X)$ for the Y in $rng(E)$ for which (X, Y) is in E.

Pattern matching is defined to be a function

$$match\text{: } pattern \times node \times environment \rightarrow environment.$$

Intuitively $match(p, n, E)$ is

- An environment that augments E by the new bindings that result from the pattern match of p and n, if p matches n with respect to E,
- $\perp$, if there is no match between p and n with respect to E.

The definition of *match* is given in Figure 4.1. ML-style pattern matching is used in the definition.

Function *match* makes use of three functions, which we describe informally here.

1. $has_repr(n, T) \equiv$ "*subtree n has a T-representation*",
2. $match_re(r, s) \equiv$ "*string s belongs to the language of regular expression r*",
3. $rank(c, \overline{c}) = (\downarrow i \mid 0 \leq i < \#\overline{c} : c = \overline{c}_i)$.

Function *has_repr* will be described in more detail later, *match_re* is assumed to be primitive. There are efficient algorithms for deciding if a string of characters belongs to the language of a regular expression [HU79].

4.4 Replacement Instantiation

As explained in Section 4.3, pattern matching is defined to be an environment that binds names and representation references of a pattern to names and nodes of the abstract tree. When a pattern p matches a tree node n, the corresponding rule that contains p can be used to construct a representation or a transformation of n. To construct such a representation or transformation of n, the corresponding replacement of the rule is used along with the environment of pattern matching.

Let r be a transform rule and n be an abstract tree node. Let also E be $match(pattern(r), n, \emptyset)$, and assume that $E \neq \perp$. Transform rule r is *applicable* with respect to environment E if

1. If s appears in $replacement(r)$, where s was defined by **stmt**-s in $pattern(r)$

$$V(E(s)) \vee C(E(s)) \Rightarrow has_trans(E(s)).$$

2. If e appears in $replacement(r)$, where e was defined by **exp**-e:t in $pattern(r)$

$$V(E(s)) \vee C(E(s)) \Rightarrow has_trans(E(s))$$

   and there is no directive "**default** t **using** ..." in $\mathcal{D}$.

3. If v appears in $replacement(r)$, where v was defined by **var**-v:t in $pattern(r)$, then there is no directive "**default** t **using** ..." in $\mathcal{D}$.

4. If c appears in $replacement(r)$, where c was defined by **const**-$c\{re\}$:t in $pattern(r)$, then there is no directive "**default** t **using** ..." in $\mathcal{D}$.

5. If the syntactic classes of the components that are used in the replacement instantiation are the same as those that are required for the operator that appears in the replacement. We distinguish four syntactic classes: *stmt*, *exp*,

$match(p, n, E)$

  $match(p, n, \bot) \rightarrow \bot$

  $match(stmt_op_1\ \overline{p}, stmt_op_2\ \overline{n}, E)$

    case $stmt_op_1 = stmt_op_2 \wedge \#\overline{p} = \#\overline{n} \rightarrow MatchList(\overline{p}, \overline{n}, E)$

    otherwise $\bot$

  $match(exp_op_1\ \overline{p}, exp_op_2\ \overline{n}, E)$

    case $exp_op_1 = exp_op_2 \wedge \#\overline{p} = \#\overline{n} \wedge type_of(p) = type_of(n) \rightarrow$

      $MatchList(\overline{p}, \overline{n}, E)$

    otherwise $\bot$

  $match(T$¡a¡$k, n, E)$

    case $(T, k) \in dom(E) \wedge E((T, k)) = n \rightarrow E$

    case $(T, k) \notin dom(E) \wedge$

      $(has_repr(n, T) \vee (is_var(n) \wedge \#abs_vars(T) > 1 \wedge$

      $rank(c, abs_vars(T)) = rank(get_var(n), get_abs_vars(get_var(n))))) \rightarrow$

      $E \cup \{[(T, k) \mapsto n]\}$

    otherwise $\bot$

  $match($**stmt**-$s, n, E)$

    case $s \in dom(E) \wedge E(s) = n \rightarrow E$

    case $s \notin dom(E) \rightarrow E \cup \{[s \mapsto n]\}$

    otherwise $\bot$

  $match($**exp**-$e{:}t, n, E)$

    case $type_of(n) = t \wedge e \notin dom(E) \rightarrow E \cup \{[e \mapsto n]\}$

    case $type_of(n) = t \wedge e \in dom(E) \wedge E(e) = n \rightarrow E$

    otherwise $\bot$

  $match($**var**-$v{:}t, n, E)$

    case $type_of(n) = t \wedge v \in dom(E) \wedge E(v) = get_var(n) \rightarrow E$

    case $type_of(n) = t \wedge is_var(n) \wedge v \notin dom(E) \rightarrow E \cup \{[v \mapsto get_var(n)]\}$

    otherwise $\bot$

  $match($**const**-$ac\{re\}{:}t, n, E)$

    case $type_of(n) = t \wedge is_const(n) \wedge match_re(re, get_const(n)) \rightarrow$

      $E \cup \{[ac \mapsto get_const(n)]\}$

    otherwise $\bot$

  otherwise $\bot$

end *match*

$MatchList(\overline{p}, \overline{n}, E)$

  if $\#\overline{p} = 1 \rightarrow match(\overline{p}_0, \overline{n}_0, E)$

  $[]\ \#\overline{p} > 1 \rightarrow MatchList(\overline{p}[1..], \overline{n}[1..], match(\overline{p}_0, \overline{n}_0, E))$

  fi

end *MatchList*

Figure 4.1: Definition of pattern matching.

var, and *const*. Function app', shown in Figure 4.2, is the definition of this requirement. Its type is

$$inst\colon \mathit{replacement} \times \mathit{environment} \rightarrow \mathit{bool}.$$

$kind_of(op, i)$ is the syntactic class of component i of operator op. We assume that it is a primitive function.

$app'(stmt_op\ \overline{r}, E) = (\wedge\, i \mid 0 \leq i < \#\overline{r} : kind_of(stmt_op, i) = kind(\overline{r}_i, E))$
$app'(exp_op\ \overline{r}, E) = (\wedge\, i \mid 0 \leq i < \#\overline{r} : kind_of(exp_op, i) = kind(\overline{r}_i, E))$
$app'(var\ v, E) = true$
$app'(const\ v, E) = true$

$kind(stmt_op\ \overline{r}, E) = stmt$
$kind(exp_op\ \overline{r}, E) = exp$
$kind(T¡c¡k, E) = \mathbf{if}\ is_var(get_nrepr(E((T, k)), T, c))\ \mathbf{then}\ var\ \mathbf{else}\ exp$
$kind(\mathbf{stmt}\text{-}s, E) = stmt$
$kind(\mathbf{exp}\text{-}e, E) = exp$
$kind(\mathbf{var}\text{-}v, E) = var$
$kind(\mathbf{const}\text{-}c, E) = const$
$kind(T¡p¡k, E) = exp$

Figure 4.2: Definition of app'.

We write $app(r, E)$ to denote that transform rule r is applicable with respect to environment E.

Replacement instantiation is defined to be a function

$$inst\colon \mathit{replacement} \times \mathit{environment} \rightarrow \mathit{node}.$$

Intuitively, $inst(r, E)$ is the instantiation of replacement r with respect to environment E.

For a transform rule r we define $inst(replacement(r), E)$ only for cases in which

$$E = match(pattern(r), n, \emptyset) \neq \perp\ \mathbf{cand}\ app(r, E)$$

where n is an abstract tree node. The definition of *inst*, given in Figure 4.3, makes use of the following functions.

1. $has_trans(n) \equiv$ "*n has a transformation*",

2. $get_repr(n, T)$ is the T-representation of node n,

3. $get_param(n, T, p)$ is the value of parameter p that is associated with the T-representation of abstract tree node n.

Functions *get_param* and *get_repr* will be described in more detail later.

4.5 The main algorithm

We present the main algorithm for processing program transformations. We assume that the abstract program, the transforms and transform directives have been preprocessed and presented to the algorithm in an internal form, as discussed in Section 4.2. We also assume that if there is a directive "**default** t **using** ..." in $\mathcal{D}$, then no transform in $\mathcal{T}$ has concrete variable of type t.

The abstract program to be transformed is presented to the algorithm as a tree n. Figure 4.4 contains the transformation algorithm. Later sections discuss parts of the algorithm, like functions *mk_repr*, *closure*, and *mk_trans*. The algorithm is invoked as

$$xform(n)$$

where n is the root of the abstract tree. It returns a tree n', which is the coordinate transformation of n according to $\mathcal{T}$ and $\mathcal{D}$, if such a tree exists, $\perp$ otherwise. Tree n' is the internal representation of the concrete program.

In several places, the algorithm in Figure 4.4 has the following form.

Definition of variable v;
if "**change** v **using** $T(\overline{w})$" $\in \mathcal{D}$ **then** ...
...
... T ...

The semantics is the following. T gets bound to the name of a transform at the conditional expression of statement **if** and this name is used later in expressions involving T.

4.6 Representations and transformation of variables

The transformation of variables is directed by transform directives. A transform directive specifies the transform to be used for the replacement of an abstract program variable. We assume that each program variable has a symbol table entry where information about the variable is stored. In addition, each concrete variable that is generated as the result of applying a transform directive to a program variable has a symbol table entry.

```
inst(r, E)
    inst(stmt_op r̄, E) = mk_stmt_node(stmt_op, InstList(r̄, E))

    inst(exp_op r̄, E) = mk_exp_node(exp_op, InstList(r̄, E))

    inst(T¡c¡k, E) = get_nrepr(E((T, k)), T, c)

    inst(s, E) =                  Comment For s defined in pattern stmt-s
        if V(E(s)) ∨ C(E(s)) → trans(E(s))
        [] ¬(V(E(s)) ∨ C(E(s))) → E(s)
        fi

    inst(e, E) =                  Comment For e defined in pattern exp-e
        if V(E(e)) ∨ C(E(e)) → idtrans(E(e))
        [] ¬(V(E(e)) ∨ C(E(e))) → E(e)
        fi

    inst(v, E) =                  Comment For v defined in pattern var-v
        mk_var_node(v)

    inst(c, E) =                  Comment For c a constant
        mk_const_node(c)

    inst(T¡p¡k, E) = get_param(E((T, k)), T, p)
end inst

InstList(r̄, E)
    if #r̄ = 0 → []
    [] #r̄ > 0 → inst(r̄₀, E) ⁀ InstList(r̄[1..], E)
    fi
end InstList

get_nrepr(n, T, c)
    if #abs_vars(T) = 1 → get_repr(n, T)
    [] #abs_vars(T) > 1 →
        mk_var_node(get_conc_vars(get_var(n))[rank(c, conc_vars(T))])
    fi
end get_nrepr
```

Figure 4.3: Definition of replacement instantiation.

Input: Program P, set of transforms $\mathcal{T}$, set of transform directives $\mathcal{D}$.
Output: Program P': the coordinate transformation of P according to $\mathcal{T}$ and $\mathcal{D}$ if such a program exists, $\perp$ otherwise.

```
xform(n:node)
    trav(n);
    if check(n) then return mk(n) else return ⊥
end xform
```

{ *check*(n) checks if the tree rooted at n has been successfully transformed, i.e.

1. For every leaf that is not in a subtree of a node that has a transformation and is labeled with a variable v, no directive in $\mathcal{D}$ is applicable to v,
2. For every node m that is not in a subtree of a node that has a transformation there is no directive "**default** t **using** ..." in $\mathcal{D}$ with $type_of(m) = t$.

}

```
check(n:node)
    check(decl v)
        return true

    check(var v)
        return "change v using ..." ∈ D ∨ "default type_of(n) using ..." ∈ D

    check(const c)
        return has_trans(n) ∨ "default type_of(n) using ..." ∈ D

    check(exp_op n̄)
        return has_trans(n) ∨ ¬("default type_of(n) using ..." ∈ D) ∨
                (∧ n | n ∈ n̄ : check(n))

    check(stmt_op n̄)
        return has_trans(n) ∨ (∧ n | n ∈ n̄ : check(n))
end check
```

Figure 4.4: Main transformation algorithm.

{ *trav*(n) traverses bottom-up the abstract tree rooted at n and constructs
the representations and transformations at each node of the tree.
}

```
trav(n:node)
  trav(decl v:t)
    if #v = 1 then
      if "change v using T(w̄)" ∈ 𝒟 ∨ "default t using T(w̄)" ∈ 𝒟 then
      begin
        Create new instance v_c having type prescribed by T
          and make a symbol table entry for it;
        get_abs_vars(v) := v; get_conc_vars(v) := v_c
      end else skip
    else
      if "change v using T(w̄)" ∈ 𝒟 then begin
        Create new instances v_c having type prescribed by T
          and make a symbol table entry for each one;
        foreach u in v do
          begin get_abs_vars(v) := v; get_conc_vars(v) := v_c end
      end
  trav(var v)
    if #abs_vars(get_trans(v)) = 1 then begin
      V(n) := "change v using T(w̄)" ∈ 𝒟 ∨
              "default type_of(n) using T(w̄)" ∈ 𝒟;
      C(n) := false;
      reprs(n) := {mk_repr(T, mk_var_node(get_conc_vars(v)[0]),
                      InstList(parameters(T), [parameters(T) ↦ w̄]))};
      closure(n); mk_trans(n)
    end else
    begin V(n) := false; C(n) := false; reprs(n) := ∅; trans(n) := ⊥ end
  trav(const c)
    V(n) := false; C(n) := "default type_of(n) using ..." ∈ 𝒟;
    reprs(n) := ∅; closure(n); mk_trans(n)
  trav(exp_op n̄)
    foreach n in n̄ do trav(n);
    V(n):=(∨ n | n ∈ n̄ : V(n)); C(n):=(∨ n | n ∈ n̄ : C(n));
    reprs(n) := ∅; mk_reprs(n); closure(n); mk_trans(n)
  trav(stmt_op n̄)
    foreach n in n̄ do trav(n); V(n):=(∨ n | n ∈ n̄ : V(n));
    C(n):=(∨ n | n ∈ n̄ : C(n)); reprs(n) := ∅; mk_trans(n)
end trav
```

Figure 4.4: (continued)

{ $mk(n)$ traverses the abstract tree rooted at n and replaces each
node with its transformation if one has been constructed.
}

mk(*n*:*node*)
  mk(*decl v*:*t*)
    if $\#v = 1$ **then**
      if "**change** v **using** $T(\overline{w})$" $\in \mathcal{D} \vee$ "**default** t **using** $T(\overline{w})$" $\in \mathcal{D}$ **then**
        return *mk_decl_node*(*get_conc_vars*(v), *conc_type*(T))
      else return n
    else
      if "**change** v **using** $T(\overline{w})$" $\in \mathcal{D}$ **then**
        return *mk_decl_node*(*get_conc_vars*(v), *conc_type*(T))
      else return n

  mk(*var* v)
    if *has_trans*(n) **then return** *get_trans*(n) **else return** n

  mk(*const* c)
    if *has_trans*(n) **then return** *get_trans*(n) **else return** n

  mk(*exp_op* $\overline{n}$)
    if *has_trans*(n) **then return** *get_trans*(n)
    else return *mk_exp_node*(*exp_op*, *mkList*($\overline{n}$))

  mk(*stmt_op* $\overline{n}$)
    if *has_trans*(n) **then return** *get_trans*(n)
    else return *mk_stmt_node*(*stmt_op*, *mkList*($\overline{n}$))
end *mk*

mkList($\overline{p}$)
  if $\#\overline{p} = 1 \rightarrow [mk(\overline{p}_0)]$
  $[\!]$ $\#\overline{p} > 1 \rightarrow mk(\overline{p}_0) \frown mkList(\overline{p}[1..])$
  fi
end *mkList*

Figure 4.4: (continued)

Suppose T transforms $v1{:}t1$ to $v2{:}t2$ and let $v{:}t1$ be a program variable. If either

$$\textbf{change } v \textbf{ using } T(\overline{w})$$

or

$$\textbf{default } t1 \textbf{ using } T(\overline{w})$$

is given, then a new variable v_c (say) of type $t2$ is generated and a symbol table entry for it is created. The declaration of v is replaced by

$$\textbf{var } v_c{:}t2$$

In addition, for every leaf n of the abstract program tree that is labeled with v

$$V(n) = \begin{cases} \mathit{true} & \text{if ``}\textbf{change } v \textbf{ using } T(\overline{w})\text{'' is in } \mathcal{D}\text{, or} \\ & \quad \text{``}\textbf{default } t \textbf{ using } T(\overline{w})\text{'' is in } \mathcal{D} \\ \mathit{false} & \text{otherwise} \end{cases}$$

and $C(n) = \mathit{false}$. In addition, the following statements are executed for constructing all representations and a transformation for node n.

$$\begin{aligned} &\mathit{reprs}(n) := \{\mathit{mk_repr}(T, \mathit{mk_var_node}(v_c), \\ &\qquad\qquad \mathit{InstList}(\mathit{parameters}(T), [\mathit{parameters}(T) \mapsto \overline{w}]))\}; \\ &\mathit{closure}(n);\ \ \mathit{mk_trans}(n) \end{aligned}$$

The T representation of v is variable v_c with associated parameters $\overline{w}$. Set *reprs*(n) contains initially the T representation of n. *closure*(n) constructs all representations that can be derived by using conversion-of-representation rules. Functions *closure*, *mk_repr*, and *mk_trans* are defined later.

Suppose T transforms $\overline{v1}{:}\overline{t1}$ to $\overline{v2}{:}\overline{t2}$. As mentioned in a previous section, for a directive

$$\textbf{change } \overline{v} \textbf{ using } T(\overline{w})$$

a list of new variables $\overline{v_c}$ (say) of corresponding types $\overline{t2}$ is generated and a symbol table entry is created for each one of them. The declaration of $\overline{v}$ is replaced by

$$\textbf{var } \overline{v_c}{:}\overline{t2}.$$

For every leaf n of the abstract program tree that is labeled v_i, for $0 \leq i < \#\overline{v}$, $V(n) = \mathit{true}$, and $C(n) = \mathit{false}$. In addition, the following statements are executed for constructing all representations and a transformation for node n.

$$\mathit{reprs}(n) := \emptyset;\ \ \mathit{trans}(n) := \bot$$

Recall that a transform like T that transforms lists of variables does not contain representation rules for expressions.

4.7 Representations and transformations of constants

Constants of a type are introduced in the type definition. The concrete form of a constant is described by a regular expression. Constants may appear in a program and may need to be transformed before further transformation of the program can proceed. The transformation of a constant $c{:}t$ can only be directed by a directive

$$\textbf{default } t \textbf{ using } T(\overline{w}).$$

A transform T that transforms $v1{:}t1$ to $v2{:}t2$ may contain rules for the transformation of constants of type $t1$. The pattern of each such rule, $\textbf{const-}id\{re\}{:}t1$, is a regular expression (re) that describes one or more constants of type $t1$. The replacement is an expression that may contain *id*.

For an abstract tree node n that is labeled with a constant $c{:}t$, $V(n) = \mathit{false}$ and

$$C(n) = \begin{cases} \mathit{true} & \text{if ``}\textbf{default } t \textbf{ using } T(\overline{w})\text{'' is in } \mathcal{D} \\ \mathit{false} & \text{otherwise.} \end{cases}$$

In addition, the following statements are executed for constructing all representations and a transformation for node n.

$$\mathit{reprs}(n) := \emptyset;\ \ \mathit{closure}(n);\ \ \mathit{mk_trans}(n).$$

4.8 Representations and transformations of expressions

If n is an expression node $\mathit{exp_op}\ \overline{n}$, then

$$\begin{aligned} V(n) &= (\vee n \mid n \in \overline{n} : V(n)) \\ C(n) &= (\vee n \mid n \in \overline{n} : C(n)) \end{aligned}$$

As mentioned in Chapter 3, each expression may have more than one representation. A representation rule provides a way of constructing a representation of an expression. These rules can be classified in two categories.

- Rules that provide a way of constructing the representation of an expression from the representations and/or transformations of its subexpressions. These rules are kept in a set s_1.

- Conversion-of-representation rules, which convert one representation of an expression into another. All conversion-of-representation rules are kept in a set s_2.

Application of a transform rule to a node is a function

$$apply\colon rule \times node \rightarrow repr + \perp$$

Intuitively, $apply(r, n)$ is

- A representation that is constructed for abstract tree node n using rule r.
- $\perp$, if r can not be applied to node n.

Type *repr* is discussed shortly. The definition of *apply* is given in Figure 4.5. Function *apply* uses *app* whose definition is given in Section 4.4.

```
apply(r, n)
    var E:environment;
    E := match(pattern(r), n, ∅);
    if E ≠ ⊥ cand app(r, E) then
        mk_repr(trans_of(r), inst(replacement(r), E), InstList(parameters(r), E))
    else ⊥
end apply
```

Figure 4.5: Application of a transform rule to an abstract tree node.

Function *apply* makes use of *mk_repr*, which we describe here. $mk_repr(T, r, rl)$ is a T-representation of an expression whose replacement instantiation is r and list of replacement instantiations rl for the transform parameters that are associated with the T-representation. Each such representation is implemented with a record that contains the following information.

1. The name of the transform whose representation is constructed.
2. The representation of the expression according to this transform.
3. A list of the values of the parameters of the transform whose representation is constructed. The values of the parameters appear in the same order as the parameters of the transform in the transform declaration.

We denote $[T, r, rl]$ such a record. Type *repr* is the type of these records. Functions *has_repr*, *get_repr*, and *get_param* that were used previously are defined as follows.

1. $has_repr(n, T) \equiv (\exists\, r, rl \mid : [T, r, rl] \in reprs(n))$.
2. $get_repr(n, T) = (\exists\, rl \mid [T, r, rl] \in reprs(n) : r)$.

3. $get_param(n, T, p) =$
$$(\exists\, r \mid [T, r, rl] \in reprs(n) : rl[rank(p, parameters(T))]).$$

With each expression node n of the abstract tree is associated a set $reprs(n)$ of all representations of n. At each expression node n of the abstract tree, the transformation algorithm constructs the set $reprs(n)$ of representations of n from the representations and/or transformations of the subtrees of n. Function *mk_reprs* of Figure 4.6 takes a node of the abstract tree as an argument and constructs the representations of the node.

```
mk_reprs(n)
    var t:repr + ⊥;
    for r in s₁ do
        t := apply(r, n);
        if t ≠ ⊥ then
            if has_repr(n, representation(r)) then
                if (∃ T, rl | t = [r, T, rl] : r) ≻ get_repr(n, representation(r)) then
                    reprs(n) := reprs(n) − {get_repr(n, representation(r))} ∪ {t}
                else skip
            else reprs(n) := reprs(n) ∪ {t}
    end
end mk_reprs
```

Figure 4.6: Construction of representations of an expression.

If two different patterns match the same node n, then one of the two has to be chosen for the representation of n to be constructed. It does not matter which one is chosen. Predicate $\succ$ decides which representation is preferable. Since the test for choosing the appropriate pattern to be used is localized, more elaborate heuristics may be employed for choosing the most suitable pattern.

The conversion-of-representation rules are like the other representation rules, with the exception that the corresponding pattern has no subpatterns. These rules describe the conversion of one representation of an expression to another. Before a conversion of representation rule can be applied to convert the $T1$ representation of an expression to a $T2$ representation, the $T1$ representation has to be constructed. This can be accomplished by maintaining a set $S2$ that contains the conversion of representation rules that have not been used yet. As mentioned in a previous section, all conversion-of-representation rules are kept in set s_2. The algorithm for applying the conversion of representation rules in the appropriate order, when abstract tree node n is visited is shown in Figure 4.7. If n is an abstract tree

node, then *closure*(n) augments *reprs*(n) with representations that are obtained with conversions-of-representation rules.

```
closure(n)
    var S2 := s2;
    var t:repr + ⊥;
    do
        for r in S2 do
            t := apply(r, n);
            if t ≠ ⊥ then begin
                S2 := S2 − {r};
                if has_repr(n, representation(r)) then
                    if (∃ T, rl | t = [r, T, rl] : r) ≻ get_repr(n, representation(r)) then
                        reprs(n) := reprs(n) − {get_repr(n, representation(r))} ∪ {t}
                    else skip
                else reprs(n) := reprs(n) ∪ {t}
            end
        end
    until S2 is unchanged
end closure
```

Figure 4.7: Construction of conversion of representations.

The transformation of an expression is constructed in the same way as the representations of the expression. The only difference is that set s_3, which contains the transformation rules, is used instead. Figure 4.8 shows the algorithm for constructing a transformation of an expression. $\max_{\succ}(a, b)$ is the maximum of a and b with respect to relation $\succ$. We assume that $a \succ \perp$ for all a. Again, heuristics can be employed for choosing the most appropriate transformation rule when more than one rule can be applied to an abstract tree node.

The transformation algorithm performs the following steps when it visits an expression node n for constructing its representations and a transformation for it.

$$reprs(n) := \emptyset;\ \ mk_reprs(n);\ \ closure(n);\ \ mk_trans(n)$$

4.9 Transformations of statements

If n is a statement node *stmt_op* $\overline{n}$, then

$$\begin{aligned} V(n) &= (\vee n \mid n \in \overline{n} : V(n)) \\ C(n) &= (\vee n \mid n \in \overline{n} : C(n)). \end{aligned}$$

```
mk_trans(n)
    var E:environment;
    trans(n) := ⊥;
    for r in s3 do
        E := match(pattern(r), n, ∅);
        if E ≠ ⊥ cand app(r, E) then
            trans(n) := max≻(inst(replacement(r), E), trans(n))
    end
end mk_trans
```

Figure 4.8: Construction of transformation of an expression.

The transformation of a statement is constructed in the same way as the transformation of an expression. When the transformation algorithm visits a statement node n, it executes the following statements.

$$reprs(n) := \emptyset;\ mk_trans(n)$$

Again, special heuristics can be employed for choosing the most appropriate transformation rule when more than one rule can be applied to an abstract statement node.

4.10 Transformations of programs

Let $\mathcal{W}$ be the set of abstract tree nodes that are the highest nodes in the abstract tree for which a transformation has been constructed. The transformation of the original program is successful if

1. For every leaf that is not in a subtree of a node in $\mathcal{W}$ and is labeled with a variable v, no directive in $\mathcal{D}$ is applicable to v,

2. For every node n that is not in a subtree of a node in $\mathcal{W}$, there is no directive

   default t **using** $T \ldots$

   in $\mathcal{D}$ with $type_of(n) = t$.

These conditions are checked by function *check* that was shown in Section 4.5.

The transformed program consists of the original tree where each node in $\mathcal{W}$ is replaced by its transformation. Function *mk*, which was presented in Section 4.5, constructs the concrete tree.

4.11 Correctness

In this section we show the correctness of the transformation algorithm. First, we define *validity* of a transformation with respect to a set of transforms and transform directives. Then we show that the algorithm constructs valid transformations of a program with respect to the transforms and transform directives that appear in the program.

A program P' is a *valid* transformation of program P with respect to a set of transforms $\mathcal{T}$ and a set of transform directives $\mathcal{D}$ if

1. For every directive "**change** $\overline{v}$ **using** $T(\overline{w})$" in $\mathcal{D}$, where T is in $\mathcal{T}$, all free instances of $\overline{v}$ have been eliminated from P, as prescribed by T.

2. For every directive "**default** t **using** $T(\overline{w})$" in $\mathcal{D}$, where T is in $\mathcal{T}$, every variable $v{:}t$ for which no **change** directive is given is replaced using $T(\overline{w})$ and every expression of type t is removed from P.

We write $P \models^{\mathcal{T}}_{\mathcal{D}} P'$ to denote that P' is a valid transformation of P with respect to $\mathcal{T}$ and $\mathcal{D}$.

We show that the transformation algorithm produces a program P' such that $P \models^{\mathcal{T}}_{\mathcal{D}} P'$ iff such a program exists.

Let n' be a representation of an abstract expression node n or a transformation of an abstract statement or expression node n. Define properties $P_V(n, n')$ and $P_C(n, n')$ as follows.

1. $P_V(n, n')$: If $V(n)$ then n' contains no instances of variables in the subtree rooted at n that need to be transformed.

2. $P_C(n, n')$: If $C(n)$ then n' contains no instances of constants in the subtree rooted at n that need to be transformed.

Let r be a representation of an abstract expression node n and t be a transformation of an abstract expression or statement node n. During program transformation, the transformation algorithm maintains the following invariant for each node n.

$$I_n = P_V(n, r) \wedge P_V(n, t) \wedge P_C(n, r) \wedge P_C(n, t)$$

Lemma 1 *Let n be an abstract tree node. If for every descendent m of n properties $P_V(m, m')$ and $P_C(m, m')$ hold (where m' is a transformation or a representation of node m), then for every rule r for which $E = match(pattern(r), n, \emptyset)$ and $app(r, E)$, $inst(replacement(r), E)$ has properties P_V and P_C.*

Proof: Let n be an abstract tree node for which $V(n) \vee C(n)$. Assume that for every descendent m of n properties $P_V(m, m')$ and $P_C(m, m')$ hold (i.e. I_m is true),

where m' is a transformation or a representation of node m. Let r be a transform rule and $E = match(pattern(r), n, \emptyset)$.

Since $V(n) = (\vee n \mid n \in \overline{n} : V(n))$ and $C(n) = (\vee n \mid n \in \overline{n} : C(n))$, it follows from the definition of *app* in Section 4.4 that whenever rule r such that $app(r, E)$ is used to construct a representation or a transformation of n, the constructed representation or transformation does not contain any instances of variables or constants in the subtree rooted at n. Functions *apply* and *mk_trans* construct a representation and a transformation of a node, respectively. When transform rule r is applied using environment $E = match(pattern(r), n, \emptyset)$ (i.e. when $inst(replacement(r), E)$ is used), condition $app(r, E)$ holds.

Hence, for a representation r or a transformation t of n

$$P_V(n, r) \wedge P_V(n, t) \wedge P_C(n, r) \wedge P_C(n, t)$$

i.e. I_n is maintained during program transformation. □

Lemma 2 *Let n be a leaf of the abstract tree. I_n is true.*

Proof: We distinguish the following cases.

1. Let n be a node labeled with variable $v{:}t$. If "**change** v **using** $T(\overline{w})$" is in $\mathcal{D}$, then $V(n) = true$ and $C(n) = false$. In this case v is replaced by a new variable v' as prescribed by T, and v' is v's T-representation. Hence $P_V(n, v') \wedge P_C(n, v')$ trivially. In addition, every other conversion of representation r that is derived from v' can not contain any instance of v, hence $P_V(n, r) \wedge P_V(n, r)$. Similarly for a transformation t. Hence I_n is true in this case.

2. Let n be a node labeled with variable $v{:}t$. If no "**change** v **using** ..." is in $\mathcal{D}$, but "**default** t **using** ..." is in $\mathcal{D}$, then $V(n) = true$ and $C(n) = false$. The proof of this case is the same as the previous one.

3. Let n be a node labeled with variable $v{:}t$. If no transform directive is applicable to v, then v is not replaced by another variable, and $V(n) = C(n) = false$. Hence I_n is trivially true.

4. Let n be a node labeled with constant $c{:}t$. If "**default** t **using** ..." is in $\mathcal{D}$, then $V(n) = false$ and $C(n) = true$. In this case the representations and transformations of n are constructed. Hence $P_V(n, n') \wedge P_C(n, n')$ trivially for every representation and transformation n' of n.

5. Let n be a node labeled with constant $c{:}t$. If no transform directive is applicable to c, then $V(n) = C(n) = false$. Hence I_n is trivially true.

Hence for a leaf n of the abstract tree I_n is true. □

It follows from the previous two lemmas that I_n is true at every node n of the abstract tree. This implies that every representation and transformation that is constructed for a node n of the abstract tree observes all transform directives in $\mathcal{D}$. In addition, the definition of a variable v on which a transform directive D in $\mathcal{D}$ is applicable is replaced by a new definition according D. At the end, the algorithm checks if the original tree has any variables that ought to be transformed and they are not. In this case no rules could be used to transform these variables and the expressions that contain them so the algorithm returns $\perp$. Hence part (1) of the definition of validity is satisfied.

From the definition of *app* it follows that every expression, variable or constant that is used in constructing a representation or a transformation of a program fragment is not of type t where "**default** t **using** ..." is in $\mathcal{D}$. At the end, function *check* checks if the resulting program contains any remaining expressions of type t where "**default** t **using** ..." is in $\mathcal{D}$. If not, then the transformed program is constructed by function *mk*. Hence part (2) of the definition of validity is satisfied and we have the following soundness theorem.

Theorem 4.11.1 *Given a program P, a set of transforms $\mathcal{T}$ and a set of transform directives $\mathcal{D}$, the transformation algorithm produces a program P' such that $P \models^{\mathcal{T}}_{\mathcal{D}} P'$ if such a program exists.*

Conversely since the algorithm applies all transform rules and constructs all possible representations for expressions, if there is a program P' such that $P \models^{\mathcal{T}}_{\mathcal{D}} P'$ the algorithm will construct it. Hence we have the following completeness theorem.

Theorem 4.11.2 *Given a program P, a set of transforms $\mathcal{T}$ and a set of transform directives $\mathcal{D}$, if there is a program P' such that $P \models^{\mathcal{T}}_{\mathcal{D}} P'$ then the transformation algorithm will construct it.*

4.12 Complexity analysis

Let n be the size of the abstract program P. Program P is represented internally by an abstract tree that has size $O(n)$. Let also t be the number of transform rules in all transforms in $\mathcal{T}$, p be the maximum pattern size, r be the maximum replacement size and k be the maximum number of transform parameters a transform may have.

At each internal node n of the abstract tree, the transformation algorithm computes $V(n)$ and $C(n)$. These values are the disjunction of the corresponding values associated with the children of node n. Computing $V(n)$ and $C(n)$ takes time $O(w)$ where w is a constant and is the maximum number of children a node can have.

Constructing a representation or a transformation takes time $O(pn + kr)$ since both the pattern and the replacement of a rule are traversed and the matching may require testing equality of subtrees of P.

Since there are at most t rules, the algorithm spends time $O(t(pn + kr))$. We assume that predicate $\succ$ takes time proportional to the size of the abstract tree. The time spent by $\succ$ at each node of the tree is thus $O(tn)$.

Hence, at each node of the tree the algorithm spends $O(tn + t(pn + kr) + w)$ time, which is $O(t(pn + kr))$. Therefore, the total time spent by the transformation algorithm is $O(nt(pn + kr))$, which is $O(n^2)$.

Chapter 5

Transforming expressions that have compound types

5.1 Introduction

We consider expressions that have compound types, i.e. types of the form $F(\overline{t})$, where F is a type constructor and $\overline{t}$ is a list of types. Any type that appears in $\overline{t}$ can have a similar form, i.e. it can be a type constructor applied to a list of types. Type constructor F as well the component expressions of the original expression may be transformed. We show how to transform expressions that have such types. The important point that should be emphasized is that no extra transform rules need be added to the bodies of transforms that transform type constructor F and the component expressions of the original expression.

5.2 Overview

Assume we want to transform a program variable x of type "sequence of booleans". Assume also that we have a transform *Seq* for implementing sequences of any type, and a transform *Bool* for implementing booleans. We want to transform x using these two transforms. We can define x and give a directive for transforming it as follows.

var x: *seq*(*bool*);
...
change x **using** *Seq*(*Bool*)

Assume that expression $x\langle\langle i\rangle\rangle \oplus x\langle\langle j\rangle\rangle$ appears in a program, where i and j are integer variables and $\oplus$ is an operation on booleans. We want to transform this expression,

given the directive for transforming x. The transformation of the expression is accomplished using the rules in transforms *Seq* and *Bool*. No extra rules for transforming expressions of the form $x\langle\langle i\rangle\rangle$ need be added to the body of transform *Bool*.

We illustrate how such expressions can be transformed. Below is part of the type definition for sequences and the operation for selecting an element of a sequence.

> **type** *seq*(t)
> **operations**
> *seq_sel* **exp**-s **exp**-i (s:*seq*(t) i:*int*) t **as** s "⟨⟨" i "⟩⟩"
> ...
> **end**

This type definition defines type constructor *seq*. The name t of the component type is used in the operations that are defined in the type definition. Identifier t is a dummy name and stands for any type. Since t can be any type, operation *seq_sel* is *polymorphic.* In the type definition, operation *seq_sel* is defined to take two expressions as operands. The first is s, which is an expression of type *seq*(t). The second is i, which is an expression of type *int*. The type of the result of *seq_sel* is t. The concrete syntax for the operation is given after keyword **as**.

We now outline how operation $x\langle\langle i\rangle\rangle \oplus x\langle\langle j\rangle\rangle$ is transformed when x is to be implemented by *Seq*(*Bool*). The declaration x:*seq*(*bool*) indicates that parameter t of type *seq*(t) should be instantiated with *bool*. Operation $\cdot\langle\langle\cdot\rangle\rangle$ is actually *seq_sel*. The type of *seq_sel* is

$$seq(bool) \times int \rightarrow bool$$

and is implemented as

$$Seq(Bool) \times _ \rightarrow Bool$$

In other words the definition of *seq_sel* along with the directive to implement x using *Seq*(*Bool*) can then be envisioned to mean that the instance $x\langle\langle i\rangle\rangle$ of *seq_sel* is

> *seq_sel* **exp**-x **exp**-i (x:*seq*(*bool*)/*Seq*(*Bool*) i:*int*) *bool*/*Bool* **as** x "⟨⟨" i "⟩⟩"

Thus, we see that, automatically, the result of $x[i]$ is of type *bool* represented by *Bool*. Similarly $x[j]$ is of type *bool* represented by *Bool*. Hence, both operands of operator $\oplus$ are represented by *Bool*. An appropriate transform rule of transform *Bool* can be applied to transform the original expression $x[i] \oplus x[j]$.

5.3 Transform and Transform Directives

A transform describes the replacement of a program variable of some type by another variable. A transform directive specifies which transform to use to replace a particular program variable. In Chapter 4, we give an algorithm for processing transforms

and transform directives when the variables to be transformed have base types. As an example, let *Int* be a transform that describes the implementation of variables of type *int*. A program may contain the following declaration of variable i and directive for implementing it using transform *Int*.

var i:*int*;
...
change i **using** *Int*

We also want to be able to transform program variables that have ground compound types and operations on such variables. For example, suppose we define type *seq*(t), where t can be any type, and transform *Seq* for implementing variables of this type. Suppose a program contains the following definition of variable s.

var s:*seq*(*int*)

The implementation of s depends on the implementation of type constructor *seq* and the implementation of expressions of type *int*. We would like to keep the implementation of the *seq* and its elements independent. For transforming s, one of the following transform directives might be given.

change s **using** *Seq*(_)	Change the implementation of variable s using transform *Seq*. Do not use any implementation for the elements of s.
change s **using** _(*Int*)	Change the implementation of the elements of variable s using transform *Int*. Do not use any implementation for s.
change s **using** *Seq*(*Int*)	Change the implementation of variable s using transform *Seq* and the implementation of the elements of s using transform *Int*.

The expression after keyword **using** in a transform directive is called a *transform expression* (*transexp* for short). Transexps are defined formally later.

5.3.1 Extensions to transforms

The above kinds of directives do force us to extend the notation we use in transforms. We need to be able to name arbitrary transforms with transform variables, so that transforms like *Int* can be communicated. To this end, we allow the types of variables in a transform to be annotated. Consider, for example, the following transform.

```
transform Seq;
    var s:seq(?t/??T) into var c:C(?t/??T);
    ...
    s := s ^ exp-e:?t/??T into ...
end
```

By convention, all transform variables begin with two question marks, while all type variables begin with one.

When a variable *v*:*seq*(*int*) is directed to be implemented using directive

change *v* **using** *Seq*(*Int*)

type variable ?*t* in transform *Seq* is bound to *int*, transform variable ??*T* is bound to transform *Int*, and a new copy of *Seq* is generated to effect the transformation.

```
transform Seq';
    var s:seq(int/Int) into var c:C(int/Int);
    ...
    s := s ^ exp-e:int/Int into ...
end
```

The transexp in a **change** directive may contain symbol _ (as shown in some examples of the previous section). Symbol _ is synonym for *I*, the identity transform, which transforms a variable to itself.

There are several possibilities for transforming the components of a variable that has a compound type, depending on the transform variables that appear in the declaration of the abstract variable of the transform used in the transformation. The following examples show the effect of transform variables in *Seq* in transforming a program variable.

transform *Seq*;
  var v:*seq*(*int*) **into** ...
  ...
end

Transform program variables of type *seq*(*int*). If a transform is specified for the transformation of the component expressions (of type *int*) of the variable, it has no effect on the transformation of the component expressions.

transform *Seq*;
  var v:*seq*($?t$) **into** ...
  ...
end

Transform program variables of type *seq*(t) (for any ground type t). If a transform is specified for the transformation of the component expressions (of type t) of the variable, it has no effect on the transformation of the component expressions.

transform *Seq*;
  var v:*seq*(*int*/$??T$) **into** ...
  ...
end

Transform program variables of type *seq*(*int*). If a transform (in a transform directive) is specified for the transformation of the components of a variable that is transformed with *Seq*, it gets bound to $??T$.

transform *Seq*;
  var v:*seq*($?t/??T$) **into** ...
  ...
end

Transform program variables of type *seq*(t) (for any ground type t). If a transform (in a transform directive) is specified for the transformation of the components of a variable that is transformed with *Seq*, it gets bound to $??T$.

5.3.2 Notation

Consider a transform M defined as follows.

transform M;
  var aM:TM **into var** cM:CM
  ...
end

Denote by ∇TM the type in τ that results from TM when all transform-variable annotations are removed. As an example, let TM be $C(C_1(?t_1/??T_1)/??T_2, ?t_3)$. Then ∇TM is $C(C_1(?t_1), ?t_3)$. We also need a notation for the transexp that describes how an abstract variable is to be implemented; we use $\triangle(TM, M)$ for this purpose. $\triangle(TM, M)$ is constructed as follows.

- Fully annotate TM, using M for the outermost constructor and I for unannotated types. In our running example, $C(C_1(?t_1/??T_1)/??T_2, ?t_3/I)/M$ is the full annotation of TM.

- In the full annotation of TM produced in the previous step, replace each type by its annotation and then remove all annotations to yield the transexp $\triangle(TM, M)$. In our running example this step yields $M(??T_2(??T_1), I)$.

We also need a notation for annotating a type in τ, given a transexp. Denote by $\triangleright t/x$ the annotated type that is produced when t and all its component types get annotated by transexp x and its corresponding components. The definition of $\triangleright t/x$ is the following.

- If t is a base type or a type variable, then $\triangleright t/x$ is t/x .
- If t is compound type $C(t_0, \ldots, t_{n-1})$ and x is $X(x_0, \ldots, x_{n-1})$, then $\triangleright t/x$ is $C(\triangleright t_0/x_0, \ldots, \triangleright t_{n-1}/x_{n-1})/X$. If X is _, replace it by I.
- If t is $C(t_0, \ldots, t_{n-1})/X$, then $\triangleright C(t_0, \ldots, t_{n-1})/X$ is $C(\triangleright t_0, \ldots, \triangleright t_{n-1})/X$.

For example, $\triangleright C(C_1(?t_1), ?t_3)/_(??T_2(??T_1), I)$ is

$$C(C_1(?t_1/??T_1)/??T_2, ?t_3/I)/I.$$

From now on, we will freely annotate a type t or any of its components and we will always mean $\triangleright t$. In particular we will use expressions like $\nabla TM/\triangle(TM, M)$ and $\triangleright TM$ interchangeably.

5.3.3 Steps in transforming expressions that have compound types

There are two main steps involved in the transformation of expressions that have compound types. First, we need to prepare for the transformation. For this, the transforms that describe the transformation of variables that have compound types need to be processed and a new set of transforms that will be used in carrying out the transformation of the abstract program need to be generated. Second, the newly generated transforms need to be applied to the abstract program. Section 5.4 discusses the first step. how to preprocess the transforms to prepare for the transformation. Section 5.5 discusses how to carry out the transformation.

5.4 Preprocessing the transforms

We show how to preprocess the transforms and prepare for the transformation. In the following we use the terminology of Chapter 2.

5.4.1 Processing transform directives of the form $M(\overline{x})$

A lot of checking must be done to make sure that a directive like

change v **using** *Seq*(*Int*)

is well defined and to determine the transforms to be used in replacing v. To outline what must transpire, let us view the situation a bit more abstractly. Consider again transform M of the previous section.

transform M;
  var $aM{:}TM$ **into var** $cM{:}CM$
  ...
end

Let $v{:}T$ be a variable, where T is a ground type, and suppose directive

change v **using** $M(\overline{x})$

is given for its transformation, where $\overline{x}$ is a list of transexps. We assume that $M(\overline{x})$ contains no transform variables. Then, if the following process succeeds, the directive is valid.

1. Unify T and ∇TM to yield a substitution σ.
2. TM has an associated transexp $\triangle(TM, M)$, which describes the implementation of aM. Unify $M(\overline{x})$ and $\triangle(TM, M)$ to yield a substitution δ (for transform variables that appear in TM).
3. Let M' be a new transform that is obtained from M by applying substitutions σ and δ to M. M' has the following form.

   transform M';
   $\left(\begin{array}{l} \textbf{var } aM'{:}TM \textbf{ into var } cM'{:}CM \\ \ldots \end{array} \right) \cdot \sigma \cdot \delta$
   end

   Process M' to remove all annotations, yielding a transform $\widehat{M}$.
4. Let $n = \#\overline{t}$ and let $v'_i{:}\overline{t}_i$ be a fresh variable, for $0 \leq i < n$. Then, recursively, process each of $v'_i{:}\overline{t}_i$, $\overline{x}_i$, for $0 \leq i < n$, to check validity and produce a fresh transform N_i.

Annotate program variable v with $\widehat{M}(N_0, \ldots, N_{n-1})$.

5.4.2 Processing transform directives of the form $I(\overline{x})$

Let $v{:}T(\overline{t})$ be a variable, where $T(\overline{t})$ is a ground type, and suppose directive

change v **using** $_(\overline{x})$

is given for its transformation, where $\overline{x}$ is a list of transexps. Each component of type $\overline{t}_i$ has to be implemented as described by x_i, $0 \leq i < \#\overline{t}$. The directive is valid if the following process succeeds.

Let $n = \#\overline{t}$ and let $v'_i{:}\overline{t}_i$ be a fresh variable, for $0 \leq i < n$. Process each of $v'_i{:}\overline{t}_i$, $\overline{x}_i$, for $0 \leq i < n$, to check validity and produce a fresh transform N_i.

Annotate program variable v with $_(N_0, \ldots, N_{n-1})$.

5.4.3 Processing transform directives of the form M.

These directives are processed as described in Chapter 4.

5.4.4 An example of preprocessing a transform for sequences

Let $seq(t)$ be the type of sequences that have elements of type t; t can be any type, since seq is a polymorphic type constructor of arity 1.

Consider also transform Seq that is used to transform variables of type $seq(t)$.

```
transform Seq;
    var s:seq(?t/??T) into var r:C(?t/??T)
    ...
end
```

C is a type constructor. For example, Seq may implement variables of type $seq(?t)$ with doubly-linked lists, so C could be ptr—the type constructor for pointers.

Assume we have the following definitions and directives in a program.

```
var s:seq(seq(bool));
...
change s using Seq(Seq(BN))
```

According to the directive for transforming s, both instances of type constructor seq are transformed with Seq and the elements of the elements of s are transformed with BN. Transform BN transforms variables of type $bool$ to variables of type nat (BN is shown in Chapter 4).

In this example, type T is $seq(seq(bool))$, transexp TM is $seq(?t/??T)$, ∇TM is $seq(?t)$, and $\Delta(TM, Seq)$ is $Seq(??T)$. The algorithm performs the following steps.

Step 1: Unify T and $\bigtriangledown TM$, yielding substitution $\sigma = \{?t \mapsto seq(bool)\}$.
Step 2: Unify $\triangle(TM, Seq)$ and $Seq(Seq(BN))$, yielding substitution

$$\delta = \{??T \mapsto Seq(BN)\}.$$

Step 3: Apply substitutions σ and δ to Seq to yield transform Seq'.

transform Seq';
 var $s{:}seq(seq(bool)/Seq(BN))$ **into var** $r{:}C(seq(bool)/Seq(BN))$
 ...
end

Transforming Seq' to remove its annotations, yielding transform $\widehat{Seq}$.

transform $\widehat{Seq}$;
 var $s{:}seq(seq(bool)/Seq(BN))$ **into var** $r{:}C(C(nat))$
 ...
end

At this point transform $\widehat{Seq}$ has been generated, and it contains no annotations.
Step 4: Recursively process the components of the original transexp. For this, consider a fresh variable $v'{:}seq(bool)$ and directive **change** v' **using** $Seq(BN)$. We now give the steps of the recursive call to show the new transform that is generated as part of the recursion.

For the recursive step, type T is $seq(bool)$, transexp TM is $seq(?t/??T)$, $\bigtriangledown TM$ is $seq(?t)$ and $\triangle(TM, Seq)$ is $Seq(??T)$.
Step 4.1: Unify T and $\bigtriangledown TM$, yielding substitution $\sigma = \{?t \mapsto bool\}$.
Step 4.2: Unify $\triangle(TM, Seq)$ and $Seq(Seq(BN))$, yielding substitution

$$\delta = \{??T \mapsto BN\}.$$

Step 4.3: Apply substitutions σ and δ to Seq, yielding a transform Seq''.

transform Seq'';
 var $s{:}seq(bool/BN)$ **into var** $r{:}C(bool/BN)$
 ...
end

Transform Seq'' to remove its annotations, yielding a transform $\widehat{\widehat{Seq}}$.

transform $\widehat{\widehat{Seq}}$;
 var $s{:}seq(bool/BN)$ **into var** $r{:}C(nat)$
 ...
end

Step 4.4: Recursively process the components of the component transexp. The next level of recursion does not yield any new transforms (substitutions σ and β are empty).

Program variable s gets annotated with $\widehat{Seq}(\widehat{\widehat{Seq}}(BN))$.

5.5 Transforming the abstract program

The transformation of the abstract program proceeds in the same way as described in Chapter 4, by traversing the tree bottom-up and applying the rules of the transforms that have been constructed during the processing of the transform directives.

5.5.1 Replacing variables that are transformed

Program variables for which transform directives are given get replaced as follows, using $ctyp(M)$ to denote the type of the concrete variable of transform M.

- A program variable v that is annotated with $\widehat{M}(\overline{N})$ gets replaced by a new variable $v'{:}ctyp(\widehat{M})$.
- A program variable v that is annotated with $_(\overline{N})$ gets replaced by a new variable $v'{:}T(ctyp(N_0), \ldots, ctyp(N_{n-1}))$.
- A program variable v that is annotated with M gets replaced by a new variable $v'{:}ctyp(M)$.

In the example of the previous section, program variable s of type $seq(seq(bool))$ is annotated with $\widehat{Seq}(\widehat{\widehat{Seq}}(BN))$. It gets replaced by a new variable $s'{:}C(C(nat))$.

5.5.2 Transforming expressions

When transforming expressions that contain variables that are transformed we distinguish two cases, depending on how the variable that is transformed is annotated.

Let v be a variable that is annotated with $M(\overline{N})$ and assume that a program contains an operation o that involves v. When the directive for the transformation of v was processed, transforms M, $\overline{N}$ were generated. Operation o gets replaced by a new one according to the transform rules of M. Representations and transformations of expressions are constructed the same way as described in Chapter 4. The only difference is that in pattern matching transexps M and $\widehat{M}$ can match, where M and $\widehat{M}$ are transforms such that $\widehat{M}$ was constructed from M during preprocessing. Since types can be annotated, a pattern **exp**-$e{:}t/T$ matches an expression $e'{:}t$ that

has a T-representation. The replacement of a rule may contain $T¡e$ which is the T-representation of expression e.

The other case involves transformation of operations that contain program variables and that are annotated with $_(\overline{N})$; we discuss this case now.

An example

Suppose type $T(t1, t2)$ defines abstract operation

$$op\colon\ T(t1, t2) \times t1 \rightarrow t2$$

with concrete syntax $\cdot[\cdot]$. Suppose program variable $u\colon T(s1, s2)$ has been defined and it has been annotated with $_(S1, S2)$, where $S1$, $S2$ are the following transforms.

transform $S1$;	**transform** $S2$;
var $a{:}s1$ **into var** $c{:}s1'$	**var** $a{:}s2$ **into var** $c{:}s2'$
...	...
end	**end**

The annotation of u indicates that the first component of u (of type $s1$) is to be implemented by $S1$ and the second component of u (of type $s2$) is to be implemented by $S2$. The definition of u is replaced by

$$\textbf{var}\ u'\colon T(s1', s2').$$

Now consider a concrete operation $u[e{:}s1]$, which in abstract form is

$$op(u, e) : T(s1, s2) \times s1 \rightarrow s2$$

This occurrence of variable u has to be eliminated; to do it, we proceed as follows. The definition of u and the directive for its transformation indicate that $t1$ of type $T(t1, t2)$ gets bound to $s1$, which is implemented by $S1$, and $t2$ of the type gets bound to $s2$, which is implemented by $S2$. Variable $u\colon T(s1, s2)$ with its first component of type $s1$ implemented with $S1$ and its second component of type $s2$ implemented with $S2$ is annotated with $_(S1, S2)$ and has been replaced by u'. The representations of expression $e{:}s1$ have been constructed from the recursion. If expression $e{:}s1$ has an $S1$-representation, then the previous operation is replaced by

$$op(u', [\![e]\!]_{S1}) : T(s1', s2') \times s1' \rightarrow s2'$$

which in concrete syntax is $u'[[\![e]\!]_{S1}]$ ($[\![e]\!]_{S1}$ is the $S1$-representation of e), and is annotated with $S2$.

Replacing operations that contain variables that are transformed

Let the type definition for $T(\overline{t})$ contain operation

$$op\colon T(\overline{t}) \times \overline{r} \rightarrow r0$$

where each element of $\overline{r}_i$ and $r0$ may include any of $\overline{t}_i$.

Let $v\colon T(\overline{s})$ be a variable that is annotated with $_(\overline{x})$. Binding σ associates type $\overline{s}_i$ and transexp $\overline{x}_i$ with $\overline{t}_i$.

$$\sigma = \{\cup i \mid 0 \leq i < \#\overline{t} \;:\; \overline{t}_i \mapsto \overline{s}_i/\overline{x}_i\}$$

The declaration of v is replaced by

$$\textbf{var } v'\colon T(ctyp(\overline{x}_1), \ldots, ctyp(\overline{x}_n))$$

where, again, $ctyp(T)$ is the type of the concrete variable of transform T.

Let $op(v, \overline{e})$ be an operation that involves variable v and expressions $\overline{e}{:}\overline{y}$. Annotating the components of op with their types and the transexps that implement them yields

$$op(v\colon T(\overline{s})/_(\overline{x}), \ldots, \overline{e}_i{:}\overline{r}_i \cdot \sigma, \ldots). \tag{5.1}$$

The representations for each expression $\overline{e}_i$ have been constructed recursively. In the last annotated operation, each component that is annotated with $/M$ (say) gets replaced by its M-representation, if such a representation exists for the corresponding component. The original operation that involves abstract variable v gets replaced by

$$op(v', [\![\overline{e}_1]\!], \ldots, [\![\overline{e}_n]\!])$$

where for each expression $\overline{e}_i$, $0 \leq i < \#\overline{e}$, $[\![\overline{e}_i]\!]$ is the appropriate representation that is required by annotation 5.1. The original operation is annotated with the transexp that corresponds to $r0 \cdot \sigma$.

5.5.3 An example of transforming the elements of an array

Suppose the following definitions and directive appear in a program.

var x:**array** [100] **of** *bool*;
var i, j:*int*;
...
change x **using** $_(BN)$

According to this directive, type constructor *array* is not transformed, but the elements of array x are transformed with BN, shown in Chapter 4.

When transform directive above is given for the transformation of variable x, x gets annotated with $_(BN)$. The definition of x is replaced by

$$\textbf{var } x'\textbf{:array of } nat.$$

Assume now that expression $x[i]$ appears in a program. The type definition for $array(t)$ contains the type for the array-element selector $\cdot[\cdot]$.

$$array_selector : array(?t) \times int \rightarrow ?t.$$

In expression $x[i]$, *array_selector* is applied to variables $x{:}array(bool)/_(BN)$, and $i{:}int$.

Hence, $?t$ in the type of the array-element selector is bound to *bool* represented by BN.

Element $x[i]$ has a BN-representation, as shown below.

$$\begin{aligned} & [\![(x\textbf{:array of } bool/_(BN))[i{:}int]{:}bool/BN]\!] \\ = \;& (x'\textbf{:array of } nat)[i{:}int]{:}nat \end{aligned}$$

Assume now that expression $x[i] \vee x[j]$ appears in a program. Like $x[i]$, subexpression $x[j]$ has type *bool* and associated transexp BN. Consider the type definition for *bool* the contains the type of operator $\vee$.

$$bool \times bool \rightarrow bool.$$

Consider also transform BN, shown in Chapter 4.

transform BN;
(0) **var** $b{:}bool$ **into** **var** $j{:}nat$
{*Coupling invariant*: $CI(b,j) = b \equiv j > 0$}
...
(4) $[\!]$ $[\![BN¡b¡1 \vee BN¡b¡2]\!] \quad = \quad BN¡j¡1 + BN¡j¡2$
...
end

Rule (4) of BN prescribes the representation of a boolean expression that is formed using operator $\vee$ from the representations of its subexpressions.

Pattern $BN¡b¡1 \vee BN¡b¡2$ of rule (4) of BN matches the original expression $x[i] \vee x[j]$, since both $x[i]$ and $x[j]$ are annotated with BN. Rule (4) of BN constructs a BN-representation; hence it can be applied to $x[i] \vee x[j]$ to construct its BN-representation. The constructed representation is of type *nat* and its associated transexp is BN.

Chapter 6

Transforming functions, procedures, and transforms

We show how to transform procedures and functions when they are called with arguments that are transformed. We also show how to transform transforms that contain directives for the transformation of local variables.

6.1 Transforming functions and procedures

The transformation of programs is discussed for programs that contain calls to functions or procedures, in which one or more arguments are expressions that must be transformed. An algorithm is presented for transforming such programs. The original call is replaced by a new one where the arguments are the transformations of the arguments of the original call. A new version of the function or procedure that is called may need to be constructed.

6.1.1 Functions and procedures

We assume that the source language supports functions and procedures. The definition of a function is as follows.

function $f(\overline{p}{:}\overline{t}){:}t$;
expression

Each function has a name (f in this example) and a return type (t). A function may have zero or more parameters ($\overline{p}$) with corresponding types ($\overline{t}$). We write

$$f\colon \overline{t} \rightarrow t$$

for a function f that has parameters of types $\overline{t}$ and returns a result of type t.

A parameter can be either **var** or non-**var** (as in Pascal). For a **var** parameter, any change of its value in the body of f is reflected in an immediate change in the corresponding argument of the function call. For a non-**var** parameter, any change of its value in the body of f is local to the function and is not reflected in the corresponding argument of the function call. A **var** parameter is indicated with the keyword **var** in front of its name. The *body* of the function is any expression of type t.

The definition of a procedure is similar to the definition of a function.

$$\textbf{procedure } p(\overline{p}{:}\overline{t});$$
$$\textit{statement}$$

Each procedure has a name (p in this example). It may have zero or more parameters ($\overline{p}$) with corresponding types ($\overline{t}$). We write

$$p{:}\ \overline{t} \rightarrow \textit{void}$$

for a procedure p that has parameters of types $\overline{t}$. We assume that a procedure returns a result of type *void*. Here, the *body* is any statement.

6.1.2 Examples of transforming function and procedure calls

We give some examples to illustrate the issues involved when one or more arguments of a procedure or a function call need to be transformed. We use transform BN, shown in Figure 3.1, as an example of a transform.

Assume that a program contains a procedure $P{:}\textit{bool} \rightarrow \textit{void}$ and a variable v.

var $v{:}\textit{bool}$;	**procedure** $P(\textbf{var } p{:}\textit{bool})$;
...	$p := p \wedge \textit{true}$
$P(v)$;	
...	
change v **using** BN	

Then, variable v is to be replaced by a fresh variable v_c (say) of type *nat*, and the call $P(v)$ should be changed to $P(v_c)$ (note that procedure P has one **var** parameter so any argument used for calling P must be a variable). However, the type of the argument of the call is now incorrect. What is required is to construct a version P' of P in which parameter p_c and the body of P' are transformed as well.

var $v_c{:}\textit{nat}$;	**procedure** $P'(\textbf{var } p_c{:}\textit{nat})$;
...	$p_c := p_c * 1$
$P'(v_c)$;	
...	

Assume now that a program contains a procedure Q:*bool* $\rightarrow$ *void* and a variable u, as follows.

var u:*bool*;
...
$Q(u)$;
...
change u **using** BN

procedure $Q(p$:*bool*$)$;
  $p := p \wedge$ *true*

Then, variable u is to be replaced by a fresh variable u_c (say) of type *nat*, and the call $Q(u)$ must be replaced by a new one. Procedure Q has one non-**var** parameter of type *bool* so any expression of type *bool* can be used as an argument for calling Q. Since transformation $u_c > 0$ of u can be constructed using the rules of transform BN, the original call can be replaced by $Q(u_c > 0)$. In this case the type of the argument of the call is correct. No new version of Q is required to be constructed.

var u_c:*nat*;
...
$Q(u_c > 0)$;
...

procedure $Q(p$:*bool*$)$;
  $p := p \wedge$ *true*

Assume now that rule (8) of BN does not exist. Hence, no transformation can be built for u. Then the call $Q(u)$ is replaced by $Q'(u_c)$ where Q' is a new instance of Q. Procedure Q' is defined as follows.

procedure $Q'(p_c$:*nat*$)$;
  $p_c := p_c * 1$

Procedure Q' has one parameter p_c that is of type *nat*, since the argument to call $Q'(u_c)$ is variable u_c of type *nat*. Parameter p_c is a BN-representation of an expression, hence the rules of transform BN can be used for transforming the body of Q to obtain Q'.

Assume that the program contains a call $Q(u \wedge u)$, where variable u is transformed as before. Let T be the following transform.

transform T;
  var b:*bool* **into var** f:*float*
  $[\![BN¡b¡1 \wedge BN¡b¡2]\!] = n2f(BN¡j¡1) *_f n2f(BN¡j¡2)$
end

Transform T transforms a variable of type *bool* to a variable of type *float*. Function *n2f* converts a value of type *nat* to a value of type *float* and $*_f$ is multiplication of values of type *float*. Now the argument of call $Q(u \wedge u)$ has a T-representation, namely $n2f(u_c) *_f n2f(u_c)$. Since transform T contains no rules for operators $\vee$

and :=, the previous representation can not be used for transforming the body of Q. Finally, assume that rule (4) of transform BN is

$$(4')\ [\,]\ [\![T¡b¡1 \vee BN¡b¡1]\!] = T¡f¡1 + BN¡j¡1.$$

To successfully transform call $Q(u \wedge u)$ we need both BN- and T-representation of argument $u \wedge u$. In this case the previous call is transformed to

$$Q''(u_c * u_c, n2f(u_c) *_f n2f(u_c))$$

where Q'', a new instance of Q, is defined as follows.

procedure $Q''(p_c{:}nat, q_c{:}float)$;
  $p_c := q_c + p_c$

In the case of a function, the body, in general, is an expression that may have several representations. Selecting one of the representations for the result of a function depends on the context of the call of the function. For example, consider function call $F(u)$ where F is defined as

function $F($**var** $p{:}bool){:}bool$;
  $\neg p$

and u is a variable of type *bool* that is transformed with BN. Function call $F(u)$ is transformed to $F'(u_c)$ where F' is defined as follows.

function $F'($**var** $p_c{:}nat){:}bool$;
  (**if** $p_c > 0$ **then** 0 **else** 1) > 0

But if rule (8) of transform BN is missing, then the call is transformed to $F''(u_c)$ where F'' is defined as follows.

function $F''($**var** $p_c{:}nat){:}nat$;
  if $p_c > 0$ **then** 0 **else** 1

In this case the result of call $F(u)$ is a BN-representation. If $F(u)$ appears in a more complicated expression, then further transformation may be possible. For example, expression $u \wedge F(u)$ can be further transformed using rule (5) of transform BN. Its BN-representation is $u_c * F''(u_c)$ and its transformation is $u_c * F''(u_c) > 0$.

6.1.3 Transforming Functions and Procedures

Assume that procedure p has been defined as follows.

procedure $p(\overline{p}{:}\overline{t})$;
  statement

For call $p(\overline{e})$, the transformation algorithm creates a new instance

procedure $p_x_1_\cdots_x_n(\overline{p}':\overline{t}')$;
statement'

of p where $x_1, \ldots, x_n$ are strings, $\overline{p}'$ are names, and $\overline{t}'$ are types that are generated as explained below. The body of the new procedure is the transformation of the body of the original one. The algorithm then replaces the original call with $p_x_1_\cdots_x_n(\overline{e}')$, where $\overline{e}'$ are arguments that are generated as explained below.

1. Suppose $p_i{:}t_i$ is a **var** parameter. Then e_i has to be a variable of type t_i. There are two possibilities.
   - No transform directive is applicable to e_i. Then e_i' is e_i, p_i' is p_i, and x_i is I, the identity transformation.
   - Expression e_i is transformed with T, which transforms $v{:}t_i$ to $u{:}t$. Then e_i' is $(e_i)_c$, where $(e_i)_c$ is the concrete variable that is generated when transform T is applied to e_i, p_i' is a **var** parameter of type t, and x_i is T.
2. Suppose p_i is a non-**var** parameter. There are three possibilities.
   - Expression e_i need be transformed. Then e_i' is e_i, p_i' is p_i, t_i' is t_i, and x_i is I,
   - Expression e_i must be transformed, and has a transformation *trans*(e_i). Then e_i' is *trans*(e_i), p_i' is p_i, t_i' is t_i, and x_i is I.
   - Expression e_i must be transformed and does not have any transformation but it has representations $(e_i)_1, \ldots, (e_i)_k$ according to transforms $T_1, \ldots, T_k$, respectively. Then the basic transformation algorithm is applied to the body b of procedure p. If the transformation of b is successful, then it finds which representations of p_i were needed for the transformation of b. Let $(e_i)_{h_1}, \ldots, (e_i)_{h_r}$ be the representations according to transforms $T_{h_1}, \ldots, T_{h_r}$, respectively, that were needed for the transformation of b. Assume that T_{h_1} transforms $v_{h_1}{:}t_{h_1}$ to $u_{h_1}{:}t_{h_1}', \ldots, T_{h_r}$ transforms $v_{h_r}{:}t_{h_r}$ to $u_{h_r}{:}t_{h_r}'$. Then e_i' is $[(e_i)_{h_1}, \ldots, (e_i)_{h_r}]$, p_i' is $[(p_i')_{h_1}, \ldots, (p_i')_{h_r}]$, t_i' is $[t_{h_1}', \ldots, t_{h_r}']$ and x_i is $T_{h_1} \cdots T_{h_r}$.

Then, the *statement* of p is processed like any other program part. If there are directives for the transformation of any variables that are defined locally in *statement* then *statement* (body of of p) is transformed. The transformed statement is the body of the newly generated procedure.

Assume that function f has parameters $\overline{p}$ with corresponding types $\overline{t}$ and that it returns a result of type t. The same steps as before are executed for transforming

a call to f. The transformation algorithm constructs the representations and transformation of the result expression. The decision of which representation to use is made when the transformation of the program that contains the call to f is constructed.

6.1.4 Some theoretical results

When a function or a procedure call appears in a program such that one or more of the arguments has several representations, a new version of the function or procedure needs to be constructed. If more than one representation is required for transforming the function or procedure call, then the algorithm of the previous section uses all representations that are necessary for transforming the call. An implementation may choose one only representation for each argument of the call for transforming the body of the function or procedure. In the case of a function call, the result may have several representations and an implementation may chose only one for transforming the rest of the program. Selecting an appropriate representation to use for each argument of a function call and an appropriate representation for the result of the call is a hard task. The following theorem characterizes the complexity of the problem.

Theorem 6.1.1 *The following problem P is NP-complete* [GJ79].
Instance: A set of transforms $\mathcal{T} = \{T_0, \ldots, T_{k-1}\}$, a function $f\colon \overline{t} \rightarrow t$, a list of expressions $\overline{e}$, each having a T-representation, for T in $\mathcal{T}$, and a transform T in $\mathcal{T}$.
Question: Does the call $f(\overline{e})$ have a T-representation?

Proof: We show that the special case of P in which $\mathcal{T} = \{T_0, T_1\}$ and $T = T_1$ is NP-complete. It follows that the more general case is also NP-complete.

P is trivially in NP. A machine chooses nondeterministically a T_0 or a T_1-representation for each argument $\overline{e}_i$, $0 \leq i < n$ and transforms f. If a T_1 representation can be constructed for the call $\overline{f}$ then the machine accepts its input, otherwise it rejects its input.

P is hard for NP. We reduce 3-*SAT* to P. Let $\mathcal{B} = (\wedge\, i \mid 0 \leq i < n : (l_{i_1} \vee l_{i_2} \vee l_{i_3}))$, be a boolean formula which contains literals l_j, for $0 \leq j < m$ and each literal l_j is either p_j or $\neg p_j$. We construct a set of transforms $\mathcal{T} = \{T_0, T_1\}$, a function f and a list of expressions $\overline{e}$ such that $\mathcal{B}$ is *true* iff $f(\overline{e})$ has a T_1-representation.

transform T_0;

var a:*bool*	**into**	**var** b:*nat*
▯ ⟦*true*⟧	=	0
▯ ⟦$\neg T_1$¡a¡1⟧	=	$f_{T_1 \rightarrow T_0}$(T_1¡b¡1)

end

transform T_1;

var a:*bool*	**into**	**var** b:*nat*
▯ ⟦*true*⟧	=	1
▯ ⟦$\neg T_0$¡a¡1⟧	=	$f_{T_0 \rightarrow T_1}$(T_0¡b¡1)
▯ ⟦T_1¡a¡1 $\wedge$ T_1¡a¡2⟧	=	$f_\wedge$(T_1¡b¡1, T_1¡b¡2)
▯ ⟦T_0¡a¡1 $\vee$ T_0¡a¡2 $\vee$ T_1¡a¡3⟧	=	$f_{\vee 1}$(T_0¡b¡1, T_0¡b¡2, T_1¡b¡3)
▯ ⟦T_0¡a¡1 $\vee$ T_1¡a¡2 $\vee$ T_0¡a¡3⟧	=	$f_{\vee 2}$(T_0¡b¡1, T_1¡b¡2, T_0¡b¡3)
▯ ⟦T_0¡a¡1 $\vee$ T_1¡a¡2 $\vee$ T_1¡a¡3⟧	=	$f_{\vee 3}$(T_0¡b¡1, T_1¡b¡2, T_1¡b¡3)
▯ ⟦T_1¡a¡1 $\vee$ T_0¡a¡2 $\vee$ T_0¡a¡3⟧	=	$f_{\vee 4}$(T_1¡b¡1, T_0¡b¡2, T_0¡b¡3)
▯ ⟦T_1¡a¡1 $\vee$ T_0¡a¡2 $\vee$ T_1¡a¡3⟧	=	$f_{\vee 5}$(T_1¡b¡1, T_0¡b¡2, T_1¡b¡3)
▯ ⟦T_1¡a¡1 $\vee$ T_1¡a¡2 $\vee$ T_0¡a¡3⟧	=	$f_{\vee 6}$(T_1¡b¡1, T_1¡b¡2, T_0¡b¡3)
▯ ⟦T_1¡a¡1 $\vee$ T_1¡a¡2 $\vee$ T_1¡a¡3⟧	=	$f_{\vee 7}$(T_1¡b¡1, T_1¡b¡2, T_1¡a¡3)

end

Function f has n parameters, p_j for $0 \leq j < n$. The body of f is $\mathcal{B}$.

Let each $\overline{e}_i$, $0 \leq i < n$, be *true*. Hence, each $\overline{e}_i$ has a T_0 and a T_1-representation and exactly one has to be chosen for transforming the call $f(\overline{e})$.

Each parameter of f can be thought of as a boolean variable, such that its T_0-representation corresponds to *false* and its T_1-representation corresponds to *true*. It is easy to show that $\mathcal{B}$ is *true* iff $f(\overline{e})$ has a T_1-representation. □

In practice, the number of parameters of a function or procedure is typically small, so deciding which representation to choose for an argument is not prohibitively expensive, in case only one representation must be chosen for each argument.

6.2 Transforming variables in a transform

The transformation of variables that appear in the body of a transform is discussed. When directives for the transformation of such variables are given, the transform is expanded and a new transform is constructed automatically. An algorithm is presented for deciding if a set of transforms that contain directives for the transformation of variables that appear in their body does not lead to infinite expansion.

6.2.1 Directives for transforming variables of a transform

The declaration of a transform has the following form.

transform $T(\overline{p}{:}\overline{t})$;
  var $\overline{av}{:}\overline{at}$ **into var** $\overline{cv}{:}\overline{ct}$
  {*coupling invariant*}
  transform rules
end

The various components of a transform are discussed in detail in Chapter 3. Here we extend a transform by allowing directives of the form

change c **using** $T'(\overline{w})$

in the body of T, where c is a concrete variable of T or a variable that appears in a rule-replacement and T' is a transform. Such a directive has the following effect.

1. If c is a concrete variable, then its definition is replaced by a new one c_c (say), as prescribed by T'. In addition, the replacements of all transform rules that contain c are replaced by new ones in which c is eliminated,

2. If c is a variable that is declared in a transform-rule replacement (for example a block-expression or a block-statement that is part of a transform-rule replacement), then its declaration is replaced by a new one c_c (say) as prescribed by T'. In addition, all instances of c in the transform-rule replacement are eliminated.

The transformation of a variable that appears in the body of a transform and for which a directive like the one above is given is accomplished based on transform rules in T'.

An example

Transform *Complex*, shown in Figure 6.1, transforms variables of type *complex* to tuples of real numbers. Transform *Complex* contains a directive

change t **using** *Rtuple*

for transforming its concrete variable t with transform *Rtuple*.

Consider now transform *Rtuple*, shown in Figure 6.2. *Rtuple* transforms a variable of type *rtuple* (a tuple of real numbers) to records with two fields of real numbers.

Assume now that directive

change v **using** *Complex*

is given in a program. This is equivalent to directive

change v **using** *Complex_Rtuple*

```
transform Complex;
    var c:complex   into   var t:rtuple
    {Coupling invariant: c = fst(t) + i · snd(t)}
  [] [[c1 + c2]]      =     (fst(t1) + fst(t2), snd(t1) + snd(t2))
  [] [[c1 * c2]]      =     (fst(t1) * fst(t2) − snd(t1) * snd(t2),
                             fst(t1) * snd(t2) + snd(t1) * fst(t2))
  [] Re(c)          into   fst(t)
  [] Im(c)          into   snd(t)
    ...
    change t using Rtuple
end
```

Figure 6.1: *Complex* implements a variable of type *complex* with a tuple of real numbers.

```
transform Rtuple;
    var t:rtuple                          into   var r:record f, s:real end
    {Coupling invariant: fst(t) = r.f ∧ snd(t) = r.s}
  [] [[(exp-e1:real, exp-e2:real)]]     =     [f = e1, s = e2]
  [] fst(t)                             into   r.f
  [] snd(t)                             into   r.s
    ...
end
```

Figure 6.2: *Rtuple* implements a variables of type *rtuple* with a record.

where transform *Complex_Rtuple* is constructed automatically by the transformation algorithm. *Complex_Rtuple* is shown in Figure 6.3.

Transform *Complex_Rtuple* is constructed automatically from *Complex* when its rule-replacements are transformed with rules of transform *Rtuple*. *Complex_Rtuple* is used to transform variables of type *complex* into records that contain two fields of type *real*.

```
transform Complex_Rtuple;
    var c:complex  into  var idr:record f, s:real end
    {Coupling invariant: c = r.f + i · r.s}
  [] [[c1 + c2]]       =     [f = r1.f + r2.f, s = r1.s + r2.s]
  [] [[c1 * c2]]       =     [f = r1.f * r2.f - r1.s * r2.s,
                              s = r1.f * r2.s + r1.s * r2.f]
  [] Re(c)            into   r.f
  [] Im(c)            into   r.s
    ...
end
```

Figure 6.3: *Complex_Rtuple* implements a variable of type *complex* with a record.

6.2.2 Processing transform directives for variables in transforms

Assume $T1$ and $T2$ are transforms like the ones below, where a directive is given in the body of each one for transforming its concrete variable.

```
transform T1;                        transform T2;
    var v1:t1 into var v2:t2             var v1:t2 into var v2:t1
    ...                                  ...
    change v2 using T2                   change v2 using T1
end                                  end
```

When a directive is given for transforming a variable c that appears in the body of a transform, a new transform is generated, in which c is replaced according to the directive. In the example of $T1$ and $T2$ above, the expansion procedure never terminates. Concrete variable $v2$ of $T1$ is replaced by a new variable $v2'$ according to $T2$, which is replaced by a new variable $v2''$ according to $T1$, and so on.

In this section, we give an algorithm that decides if a set of transforms that contain transform directives for variables that appear in their bodies results in an infinite expansion.

When directive

$$\textbf{change } c \textbf{ using } T'(\overline{w})$$

appears in the body of T, where c is a variable that appears in T (where T' is a possibly different transform), the basic transformation algorithm described in Chapter 3 is used to replace c and the replacements of the transform rules. Such directive has the following effect.

1. If c is a concrete variable, then its definition is replaced by a new one c_c (say), as prescribed by T'. In addition, the replacements of all transform rules that contain c are replaced by new ones in which c is eliminated,

2. If c is a variable that is declared in a transform-rule replacement (for example a block-expression or a block-statement that is part of a transform-rule replacement), then its declaration is replaced by a new one c_c (say) as prescribed by T'. In addition, all instances of c in the transform-rule replacement are eliminated.

The definition of concrete variable c is replaced by a new one c_c (say), as prescribed by T'. In addition, the replacements of all transform rules that contain c are replaced by new ones in which c is eliminated. The transformation of replacements is accomplished based on transform rules in T'.

Since, by following blindly the transform directives, the transformation algorithm may keep expanding forever, we need to know in advance if such a situation may result.

Let $\mathcal{T}$ be a set of transforms that contain directives for the transformation of their concrete variables and let $\mathcal{D}$ be the set of transform directives that appear in transforms in $\mathcal{T}$.

For $\mathcal{T}$ a set of transforms, we define directed graph $G = (V, E)$ as follows. For every T in $\mathcal{T}$ where

transform T;
  var $\overline{v}$:$\overline{t1}$ **into var** $\overline{u}$:$\overline{t2}$
  ...
end

let V_T be the set of variables that appear in the replacements of the transform rules of T (possibly qualified by their scope) for which a transform directive is given. Define $G_T = (V_T, E_T)$ as follows.

$$\begin{aligned} V_T &= \{v \mid v \in \overline{v}\} \cup \{T\} \cup \{u \mid u \in \overline{u}\} \\ E_T &= \{(v, T) \mid v \in \overline{v}\} \cup \{(T, u) \mid u \in \overline{u}\} \end{aligned}$$

Assume T transforms $\overline{v}$:$\overline{t1}$ to $\overline{u}$:$\overline{t2}$ and T' transforms $\overline{u}$:$\overline{t2}$ to $\overline{w}$:$\overline{t3}$. For every directive D in $\mathcal{D}$ of the form

change $\overline{x}$ **using** $T'(\overline{w})$

that appears in T, define $G_D = (V_D, E_D)$ as follows.

$$V_D = \emptyset, \quad E_D = \{(x, T') \mid x \in \overline{x}\}.$$

Let G be the graph (V, E) where

$$V = \bigcup_{T \in \mathcal{T}} V_T, \quad E = \bigcup_{T \in \mathcal{T}} E_T \cup \bigcup_{D \in \mathcal{D}} E_D.$$

If G has a cycle, then attempting to transform transforms in $\mathcal{T}$ results in an infinite expansion. Conversely, if G has no cycle, then the expansion terminates, producing a set of transforms whose concrete variables have been replaced according to the corresponding directives.

The algorithm is linear in the number of transforms. Efficient algorithms exist for testing if a directed graph contains a cycle [AHU78].

Chapter 7

A second approach to dealing with compound types

We show a second approach to transforming expressions that have compound types, which has its advantages and disadvantages with respect to the one of Chapter 5. The approach is based on the notion of transform parameters, which we explain below.

We allow transforms to have parameters that are other transforms. For example, a transform whose concrete variable is further transformed can be declared as follows.

> **transform** $M[SUB]$;
>   **var** $a{:}t0$ **into var** $c{:}t1$
>   ...
>   **change** c **using** SUB
> **end**

Parameter SUB may appear at any place in transform M where a transform name is expected.

A transform parameter gets bound to a transform name with a transform directive. For example, directive

change v **using** $M[N]$

creates a new instance M' of M (M' has no parameters) in which SUB has been replaced by N. The directive for transforming the concrete variable of M' becomes

change c **using** N.

A mechanism is provided for accessing the type of the abstract variable of a transform (for a transform with exactly one abstract variable). Construct

domain$[SUB]$

is the type of the abstract variable of transform SUB. In our running example, in M', this is the type of the abstract variable of N. Similarly,

$$\mathbf{range}[SUB]$$

is the type of the concrete variable of transform SUB (defined only if SUB has exactly one concrete variable).

The main use of having transforms as parameters to other transforms is to allow for the transformation of expressions that have compound types, where the type constructor and the component expressions are transformed independently. For example, the following transform may be used for transforming sequences.

transform *Seq*[*SUB*];
  var *s*:*seq*(**domain**[*SUB*]) **into var** *c*:*C*(**range**[*SUB*])
  ...
  $[\![\ s \rhd$ **exp**-*e*:**domain**[*SUB*] $]\!] \ = \ \cdots\ SUB_{\text{¡}}e$
  ...
end

$SUB_{\text{¡}}e$ is the *SUB*-representation of expression e. Suppose *s*:*seq*(*int*) is transformed using

$$\textbf{change } s \textbf{ using } Seq(Int) \tag{7.1}$$

where transform *Int* is defined as follows.

transform *Int*;
  var *i*:*int* **into var** *j*:*bit_vector*
  ...
end

Then, a parameterless version *Seq′* of *Seq* is created, in which *SUB* has been replaced with *Int*.

transform *Seq′*;
  var *s*:*seq*(*int*) **into var** *c*:*C*(*bit_vector*)
  ...
  $[\![\ s \rhd$ **exp**-*e*:*int* $]\!] \ = \ \cdots\ Int_{\text{¡}}e$
  ...
end

In the approach taken in Chapter 5, transform *Seq* would be defined as follows.

transform *Seq*;
  var s:*seq*$(?t/??T)$ **into var** c:$C(?t/??T)$
  ...
  $[\![s \rhd$ **exp**-e:$?t/??T]\!] = \cdots ??T_{\text{¡}}e$
  ...
end

When directive 7.1 is given, a new version of *Seq* is created as follows.

transform $\widehat{Seq}$;
  var s:*seq*(int/Int) **into var** c:$C(bit_vector)$
  ...
  $[\![s \rhd$ **exp**-e:$int/Int]\!] = \cdots Int_{\text{¡}}e$
  ...
end

This version of *Seq*′ is essentially the same as the one shown above.

The approaches of this chapter and of Chapter 5 differ on the amount of control given to the user of the transform. The approach of this chapter allows, and sometimes requires, the user of a transform to know how the arguments are used in the body of the transform. For example, if the transform that is passed as argument is used for transforming the concrete variable of another transform, the user has to know the type of the concrete variable of the transform.

In the approach taken in Chapter 5, this knowledge is confined within the body of the transform. Only the author of the transform can direct the implementation of the concrete variables of the transform. The user need only know what operations the transform implements. To illustrate the point, consider transform *RevSeq*.

transform *RevSeq*$[SUB1, SUB2]$;
  var s:*seq*(**domain**$[SUB1]$) **into** **var** r:*seq*(**range**$[SUB1]$)
{*Coupling invariant*:
  $I(s,r) = \#s = \#r \wedge (\forall\ i \mid 0 \leq i < \#s : s\langle\!\langle i\rangle\!\rangle = r\langle\!\langle \#r - i - 1\rangle\!\rangle)$}
  $[\!]$ $[\![\langle\!\langle\rangle\!\rangle]\!]$ **into** $\langle\!\langle\rangle\!\rangle$
  $[\!]$ $[\![\langle\!\langle$**exp**-e:**domain**$[SUB1]\rangle\!\rangle]\!]$ **into** $\langle\!\langle SUB2_{\text{¡}}e\rangle\!\rangle$
  $[\!]$ $\#s > 0$ **into** $\#r > 0$
  $[\!]$ $[\![s\langle\!\langle$**exp**-e:$int\rangle\!\rangle]\!] = r\langle\!\langle \#r - e - 1\rangle\!\rangle$
  $[\!]$ $[\![s_{\text{¡}}1$ **cat** $s_{\text{¡}}2]\!] = r_{\text{¡}}2$ **cat** $r_{\text{¡}}1$
  $[\!]$ $[\![s \rhd e$:**domain**$[SUB1]]\!] = SUB1_{\text{¡}}e \lhd r$
  $[\!]$ $[\![e$:**domain**$[SUB1] \lhd s]\!] = r \rhd SUB1_{\text{¡}}e$
  change r **using** $SUB2$
end

The specification for *RevSeq* is the following.

1. *RevSeq* implements a variable of type $seq(t)$ with its reversed version.
2. *RevSeq* has two parameters, $SUB1$ and $SUB2$. $SUB1$ is used to transform the elements of the sequence, and $SUB2$ is used to transform its concrete variable.
3. *RevSeq* provides implementations for the following operations.
   - The empty sequence $\langle\langle\rangle\rangle$.
   - A singleton sequence $\langle\langle e\rangle\rangle$, where expression e is of type t.
   - Testing if a sequence is non-empty ($\#s > 0$).
   - Selecting an element of a sequence ($s\langle\langle e\rangle\rangle$).
   - Catenating two sequences ($s1$ **cat** $s2$).
   - Prepending an element to a sequence ($e \lhd s$).
   - Appending an element to a sequence ($s \rhd e$).

This specification reveals more about transform *RevSeq* than it should. In particular, the user need not know what the concrete variable of *RevSeq* is, and how is to be implemented. This is to be decided by the author of the transform who knows the operations on the concrete variable that appear on rule-replacements.

Using transform-parameters can lead to certain abuses of a transform. For example, the rule for prepending an element to a sequence might have been the following.

$$[\![\, e{:}\mathbf{domain}[SUB2] \lhd s \,]\!] = r \rhd SUB2¡e$$

This rule can be used only when the elements of a sequence are transformed with $SUB2$, the same transform that is used for transforming the concrete variable of *RevSeq*. Even though this rule is a bit contrived, it is used to illustrate another point about this approach: allowing transforms as arguments to other transforms can lead to tricky implementations.

The main advantage of this approach is its simplicity of implementation. The transform parameters allow a lot of flexibility but can easily lead to obscure transforms. When parameters are used only for transforming the components of the abstract variable, the approach is the same as the one of Chapter 5.

Chapter 8

Methodology and examples

We discuss some methodological issues for writing transforms. Then, we give examples to illustrate the use of transforms in practice.

A working system based on the approach of Chapter 7 has been implemented on the Synthesizer Generator [RT88]; the examples we give in this chapter are based on that approach. Since complete type inference is not yet supported by the implementation, type annotations are required at places of a program where types can not be inferred. These type annotations have the form $e{:}t$ where e is an expression and t is its type. For the same reason, transform directives require type annotations for the types of the abstract variables they transform. These type annotations have the form $d{::}t$ where d is a directive and t is the type of the abstract variable it is applied to. For example

$$\textbf{change } s \textbf{ using } \mathit{Seq}(\mathit{Int})::\mathit{seq}(\mathit{int})$$

is a directive for transforming variable $s{:}\mathit{seq}(\mathit{int})$.

The examples in this chapter, two well-known algorithms, were written in their algorithmic form in *Polya*, automatically transformed into *C* [KR87], and compiled and executed.

8.1 Interfering implementations

A goal of the transform approach is to allow algorithms to be written in the problem domain, using the types of the domain, and then to have the algorithms automatically transformed to use efficient implementations of variables of types. Achieving efficiency in an implementation may require more interdependency in the implementation of operations than appears at the abstract level. We explain this as follows.

Independent implementations of operations do not interfere with each other; they may be combined in unlimited ways in the abstract program, and there will be no

interference between their corresponding implementations.

On the other hand, operations of an abstract type may be implemented in a dependent fashion. The main reason for this is to optimize the implementation. One implication is that the transform author cannot consider the implementation of an operation independently of the implementation of other operations. In such cases, special transform rules must be written for dealing with such dependencies.

We illustrate dependency of implementations with an example of type *dgraph*, which may be defined as follows.

type *dgraph*
  ...
  for **exp-**g **exp-**v **var-**w **stmt-**s $(g{:}dgraph;\ v{:}vertex;\ w{:}vertex)$ $(w(s))$
    as for w **adjacent to** v **in** g **do** s
  ...
  delete **exp-**g **exp-**v **exp-**w $(g{:}dgraph;\ v{:}vertex;\ w{:}vertex)$
    as delete "(" v "→" w ")" **from** g
  ...
end

The two statements we consider here are *for* and *delete*. Statement

for w **adjacent to** v **in** g **do** s

executes statement s for every vertex w that is adjacent to vertex v in graph g. Statement

delete $(v \rightarrow w)$ **from** g

deletes directed edge (v, w) from graph g.

Consider now a program that contains the following statement.

for w **adjacent to** v **in** g **do**
  if ···
  [] ··· **delete** $(v \rightarrow w)$ **from** g
  ...
  fi

Consider also transform *Graph*, shown in Figure 8.1, which implements variables of type *dgraph* with adjacency lists. Transform *Graph* implements operations *for* and *delete* so that there is no interference between them. The drawback of this implementation is lack of efficiency. The implementation of operation *for* requires construction of a new set, and the implementation of operation *delete* traverses the adjacency list from its beginning. It would be nice to be able to write a specialized rule for implementing *delete* so that deletion of an edge is done "in place" in cases like the one above, i.e. while operating on vertices that are adjacent to a vertex.

```
transform Graph;
    var g:dgraph into var c:array of ptr(vnode)
    ...
    for id-w:vertex adjacent to exp-v:vertex in g do stmt-s into
        block
            var w:vertex;
            var S:set(vertex) := set of vertices adjacent to v in g;
            do w ∈ S → s od
        end
    ...
    delete (exp-v:vertex → exp-w:vertex) from g into
        Traverse v's adjacency list and remove w from it
    ...
with type vnode = record v:vertex, next:ptr(vnode) end
end
```

Figure 8.1: *Graph* implements variables of type *dgraph*.

Transform *Graph'*, shown in Figure 8.2, implements operation *for* by traversing the adjacency list of vertex v. It maintains two pointers pw, and ppw during traversal of the adjacency list: pw points to the node of the list that contains w, and ppw points to the previous node. Figure 8.3 depicts the situation. It is clear that operation *delete* cannot be implemented as in transform *Graph*. Simply traversing the adjacency list that is associated with vertex v, and removing the node that contains w would interfere with the implementation of operation *for*, since it would invalidate pointer pw. So, when *delete* appears in the context of a *for* statement, its implementation has to "coordinate" with the implementation of *for*.

There are various ways to solve this problem. One is to change the abstract program (perhaps by introducing new abstract operations) so that the dependencies of the implementations of operations no longer exist. In the previous example, one may define a new variable s:*set*(*vertex*) and change the abstract program as follows.

```
s := set of vertices adjacent to v in g;
for w ∈ s do
    if ···
    [] ··· delete (v → w) from g ···
    ...
    fi
```

So, what was part of the implementation (transform rules) becomes part of the

transform *Graph'*;
  var g:*dgraph* **into var** c:**array of** *ptr*(*vnode*)
  ...
  for id-w:*vertex* **adjacent to exp**-v:*vertex* **in** g **do stmt**-s **into**
    block
      var *pw*, *ppw*:*ptr*(*vnode*);
      procedure *Del*(v', w':*vertex*) =
        if $v = v' \wedge w = w' \rightarrow \cdots$
        ...
      end;
      ...
      Traverse v's adjacency list and execute statement s
    end
  ...
  delete (**exp**-u:*vertex* $\rightarrow$ **exp**-v:*vertex*) **from** g **into** *Del*(u, v)
  ...
with
  procedure *Del*(v', w':*vertex*) =
    Default implementation of delete
  end;
  type *vnode* = **record** v:*vertex*, *next*:*ptr*(*vnode*) **end**
end

Figure 8.2: Defining a local meaning for operation *delete*.

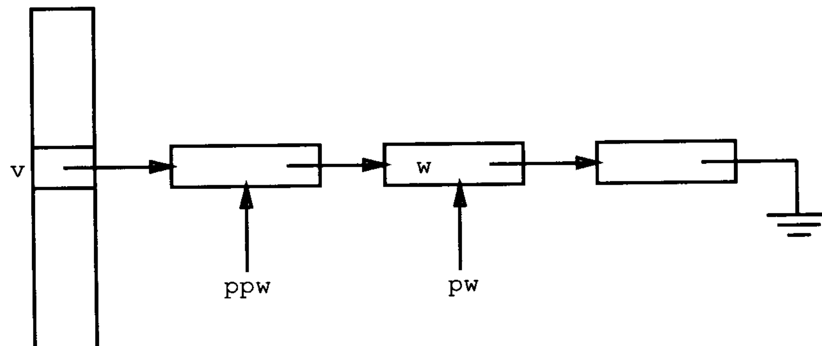


Figure 8.3: Implementation of graphs according to transform *Graph'*.

abstract program. Note that the program stays in abstract form, in the sense that all implementation details are hidden.

There are a few drawbacks with this approach. First, the original algorithm is modified to accommodate an implementation (for graphs in this case), and one of our goals is to leave the abstract program unchanged as much as possible. Second, efficiency of the implementation may be degraded.

Another way to solve this problem is to use already existing mechanisms for transforming programs. Essentially, we want an implementation of operation *delete* that is local to operation *for*. The scope rules of the language already provide mechanisms for doing this. We can implement operation *delete* by a call to procedure *Del* (say) and define *Del* within the implementation of operation *for*. Figure 8.2 shows a way to do this. For a statement **delete** (u, w) **from** g that does not appear within a *for* statement, procedure *Del*, which is defined after **with** in *Graph'*, is called. If, on the other hand, *delete* appears in a *for* statement, procedure *Del* that is defined locally in the replacement of the transform rule for *for* is invoked.

A third approach would be to introduce a new kind of transform rule that is powerful enough to describe special-case implementations of operations. For example, we may invent a new kind of transform rule that has one or more *local* transform rules as follows.

for w **adjacent to** v **in** g **do** s **into** $\cdots$
  local
    delete $(v \rightarrow w)$ **from** g **into** $\cdots$
  end

The semantics is as follows. Whenever operation **delete** $(v \rightarrow w)$ **from** g appears in the body of **for** w **adjacent to** v **in** g **do** s, the local rule will be used for its implementation. Transform rules that contain local rules may be implemented by appropriately generating local procedures and calls to them. Since statement **local** is not implemented in *Polya*, in our examples we use the alternative method of defining local procedures.

8.2 Example: The Huffman Encoding Algorithm

We show how to implement the algorithm for constructing Huffman codes [Huf51] using transforms. Given non-negative weights $\overline{w}$, the Huffman encoding algorithm constructs a binary tree with $\#\overline{w}$ leaves, such that there is a one-to-one correspondence between its leaves and weights $\overline{w}_i$, $0 \leq i < \#\overline{w}$, and $\overline{l} \cdot \overline{w}$ is minimum over all

binary trees, where $\overline{l}_i$ is the depth of the leaf labeled $\overline{l}_i$ ($a \cdot b$ is the dot product of vectors a and b).

8.2.1 The algorithm

Figure 8.4, a screen dump of *Polya*'s program editor, contains the conventional algorithm for constructing Huffman codes. The details of the Huffman encoding algorithm can be found in [Huf51,AHU85]. Here we focus on how we can implement the algorithm in *Polya* using transforms.

The code in Figure 8.4 makes use of abstract types *set* and *btree* ("binary trees of values of some type"). These abstract data types are defined with *Polya*'s type definition editor and are discussed in Section 8.2.2. Procedure *read_set* reads integer values (the weights of the algorithm) from the input (we use *C* input functions for reading numbers) and constructs a set of binary trees of integers. Each element of the set that is being constructed is a binary tree with exactly one node that contains a weight. Procedure *print_btree* prints a binary tree of integers.

Variable S:*set*(*btree*(*int*)) of the main program is initialized with the weights that are read from the input (by calling procedure *read_set*) for which Huffman codes will be computed. Initially, each element of S is a binary tree with no children. The Huffman code for the input weights is given by the single tree in S when the algorithm terminates. The algorithm also uses two variables x, y:*btree*(*int*).

The key here is that the algorithm is written at a high level, in terms of trees and sets of trees. Implementation details for these types are not part of the algorithm.

8.2.2 Abstract types used in the algorithm

We briefly describe the abstract types for sets, sequences, and binary trees that are used in the Huffman encoding algorithm of Figure 8.4. Sequences are actually used for implementing sets, and their use will be shown later when we discuss the transforms that are used for transforming the algorithm. The types discussed in this section are defined with *Polya*'s type definition editor. Here, we have placed operations on these types that are generally useful. After some time and use, these would be fleshed out to types that most people would use, over and over again.

Figure 8.5 contains a screen dump of polymorphic type *set*(t): the type of sets having elements of type t, where t can be any type. Most of the operations that are defined for type *set*(t) are the "usual" ones. We briefly describe these operations.

- Expression $\{e_0, \ldots, e_{n-1}\}$ constructs a set from its elements $e_0, \ldots, e_{n-1}$.
- Expressions *compr1*, and *compr2* are two forms of set comprehension.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
(*--Program Editor*)

procedure read_set(var sb: set(btree(int))) =
 sb := {} ;
 do ¬eof(input) -->
  var e: int ;
  read(e) ;
  sb := sb∪{<:tnil,e,tnil:>}
 od
end ;
procedure print_btree(t: btree(int), margin: int) =
 if t≠tnil -->
  block
   var i: int := 0 ;
   do i<margin --> print(" ") ; i := i+1  od ;
   print(•t: int) ;
   print("\n") ;
   print_btree(←t, margin+5) ;
   print_btree(→t, margin+5)
  end
 ▯ t=tnil --> skip
 fi
end ;
var S: set(btree(int)) ;
read_set(S) ;
do #S≠1 -->
 var x: btree(int) ;
 var y: btree(int) ;
 x := S↓↓ ;
 S := S-{S↓↓} ;
 y := S↓↓ ;
 S := S-{S↓↓} ;
 S := S∪{<:x,•x+•y,y:>}
od ;
print_btree(S↓↓, 0)

##DIRECTIVES##

change S into s using Set_Seq[I]::set(btree(int));
  Seq_Dll[Btree_Trec[I]::btree(int)]::seq(btree(int))
change x, y using Btree_Trec[I]::btree(int)
main display function :C
```

Figure 8.4: Huffman encoding algorithm with directives.

- Expressions $s1 \cup s2$, $s1 \cap s2$, and $s1$--$s2$ are the set union, set intersection, and set difference, respectively, of sets $s1$ and $s2$.
- Expression $e{\in}{\in}s$ ($e{\notin}{\notin}s$) is *true* iff e belongs (does not belong) to set s.
- Expression $s1 \subset s2$ ($s1 \supset s2$) is *true* iff $s1$ is a proper subset (superset) of $s2$.
- Expression $s1 \subseteq s2$ ($s1 \supseteq s2$) is *true* iff $s1$ is a subset (superset) of $s2$.
- Expression $s \downarrow\downarrow$ ($s \uparrow\uparrow$) is a minimal (maximal) element of set s according to an ordering relation for its elements.
- Expression $\#\#s$ is the size of set s.
- Statement **forall** i $\in\in$ s **do** b performs statement b on each element i of set s.
- Function *choose*(s) returns an arbitrary element of set s.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
type set(t)
operations
  set_make  {[exp-e]}  (e: t)  set(t) as "{"  {[e]-","}  "}";
  union  (s: set(t);  t: set(t))  set(t)  infix "∪";
  intersection  (s: set(t);  t: set(t))  set(t)  infix "∩";
  minus  (s: set(t);  t: set(t))  set(t)  infix "-";
  compr1  id-x  exp-e  (x: t;  e: bool;  (x) e)  set(t)
   as "{"  x  "|"  e  "}";
  compr2  [id-x  type-t1]  exp-r  exp-e
   (x: t1;  r: bool;  e: t;  (x) r;  (x) e)  set(t)
   as "{"  [x  "∈"  t1]-","  ":"  r  ":"  e  "}";
  minimum  (s: set(t))  t  postfix "↓↓";
  maximum  (s: set(t))  t  postfix "↑↑";
  subset  (s: set(t);  t: set(t))  bool  infix "⊆";
  superset  (s: set(t);  t: set(t))  bool  infix "⊇";
  subsetp  (s: set(t);  t: set(t))  bool  infix "⊂";
  supersetp  (s: set(t);  t: set(t))  bool  infix "⊃";
  member  (e: t;  s: set(t))  bool  infix "∈";
  not_member  (e: t;  s: set(t))  bool  infix "∉";
  forall  id-i  exp-s  stmt-b  (i: t;  s: set(t);  (i) b)
   as "forall"  i  "∈"  s  "do"  b  "end" ;
  size  (s: set(t))  int  prefix "#";
  choose  (s: set(t))  t
end
```

Figure 8.5: Definition of type *set*(t).

Figure 8.6 contains a screen dump of type *seq*(t). Type *seq*(t) is also polymorphic, it is the type of sequences having elements of some type t; t can be any type. We briefly describe the operations that are defined in type *seq*(t).

- Expression $\langle\!\langle e_0, \ldots, e_{n-1} \rangle\!\rangle$ constructs a sequence from its elements $e_0, \ldots, e_{n-1}$.
- Expression $s \rhd e$ ($e \lhd s$) appends (prepends) e to sequence s.
- Expression $s1$ **cat** $s2$ catenates two sequences $s1$ and $s2$.
- Expression $s\langle\!\langle i \rangle\!\rangle$ yields element i of sequence s.
- Expressions $s\langle\!\langle i.. \rangle\!\rangle$, $s\langle\!\langle ..j \rangle\!\rangle$, $s\langle\!\langle i..j \rangle\!\rangle$ yield a subsequence of sequence s.
- Expression $e \in s$ is *true* iff e is an element of sequence s.
- Expression $\#s$ is the size of sequence s.
- Expression $\leftrightarrow s$ is the reverse sequence of s.
- Statement **for** e **inseq** s **do** st performs statement st on each element e of sequence s.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
type seq(t)
operations
  seq_make  {[exp-e]}  (e: t)  seq(t) as "«"  {[e]-","}  "»";
  append  (s: seq(t);  e: t)  seq(t)  infix "▷";
  prepend  (e: t;  s: seq(t))  seq(t)  infix "◁";
  cat  (e: seq(t);  s: seq(t))  seq(t)  infix "cat";
  index  exp-s  exp-i  (s: seq(t);  i: int)  t as s  "«"  i  "»";
  initial_segment  exp-s  exp-i  (s: seq(t);  i: int)  seq(t)
   as s  "«"  i  ".."  "»";
  middle_segment  exp-s  exp-i  exp-j
   (s: seq(t);  i: int;  j: int)  seq(t) as s  "«"  i  ".."  j  "»";
  final_segment  exp-s  exp-i  (s: seq(t);  i: int)  seq(t)
   as s  "«"  ".."  i  "»";
  member  (e: t;  s: seq(t))  bool  infix "∈";
  not_member  (e: t;  s: seq(t))  bool  infix "∉";
  size  (s: seq(t))  int  prefix "#";
  reverse  (s: seq(t))  seq(t)  prefix "↔";
  for  id-e  exp-s  stmt-st  (e: t;  s: seq(t))
   as "for"  e  "inseq"  s  "do"  st
end
```

Figure 8.6: Definition of type *seq*(t).

Figure 8.7 contains a screen dump of polymorphic type *btree*(t): the type of binary trees having elements of some type t. The type definition defines literal *tnil*, which denotes the empty binary tree. We briefly describe the operations that are defined in *btree*(t).

- Expression $\langle : t1, e, t2 : \rangle$ constructs a binary tree that has $t1$ as its left subtree, $t2$ as its right subtree and e as the value of its root.
- Expression $\bullet t$ returns the value stored at the root of tree t.
- Expression $\leftarrow t$ $(\rightarrow t)$ returns the left (right) subtree of t.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
type btree(t)
literals
  btree_nil  as "tnil"
operations
  mktree  exp-e  exp-t1  exp-t2
   (e: t;  t1: btree(t);  t2: btree(t))  btree(t)
   as "<:"  t1  ","  e  ","  t2  ":>";
  node_val  (e: btree(t))  t  prefix "•";
  lsub  (e: btree(t))  btree(t)  prefix "←";
  rsub  (e: btree(t))  btree(t)  prefix "→"
end
```

Figure 8.7: Definition of type *btree*(t).

Figure 8.8 contains a screen dump of polymorphic type *ptr*(t): the type of pointers to values of type t. Type *ptr*(t) is used for implementing sequences and binary trees. The type definition defines literal *nil*, which denotes the empty pointer. The operations that are defined in *ptr*(t) are the following.

- Expression $@e$ returns a pointer to expression e.
- Expression $\hat{\ }p$ dereferences p and returns the value pointed to by it.
- Expression **dispose** p disposes the value that is pointed to by p. This operation is included because we transform a program to produce C code.

8.2.3 Transforming the algorithm

We want to transform the algorithm in Figure 8.4 into one that uses appropriate data structures for implementing variables S, x, and y. We do this by employing transforms *Btree_Trec*, *Set_Seq*, and *Seq_Dll*. The formal coupling invariants are not given in the transforms; instead, they are discussed informally below.

Note that there are general transforms, which could be placed in a library and used in many situations. Thus, they become "off-the-shelf", reusable parts.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
type ptr(t)
literals
  ptr_nil  as "nil"
operations
  ref  (e: t)  ptr(t)  prefix "@";
  deref  (p: ptr(t))  t  postfix "^";
  dispose  exp-p  (p: ptr(t)) as "dispose"  p
end
```

Figure 8.8: Definition of type *ptr*(t).

Figure 8.9 contains a screen dump of transform *Btree_Trec*, which implements a variable of type "binary tree" with a pointer to record *trec*. The definition of *trec* is given after keyword **with** at the end of the transform. Each node of the binary tree contains a value and two pointers. one to the left subtree and one to the right subtree. Figure 8.10 depicts the representation of a binary tree according to *Btree_Trec*.

Transform *Btree_Trec* implements the empty binary tree (*tnil*) and *btree* expression $\langle{:}t1, e, t2{:}\rangle$, where $t1$ and $t2$ are binary trees and e is a value. It also contains rules for accessing the value at the root, and the left and right subtree of a binary tree. Finally, it contains rules for comparing two binary trees according to the ordering of the values at their roots, and assigning one binary tree to another.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
(* Btree_Trec implements binary trees with pointers to records *)
transform  Btree_Trec[SUB2];
  var b: btree(domain[SUB2]) into t: ptr(trec)
coupling invariant : <expression>
|  | tnil |  =  nil
|  | <:b¡1,exp- d:domain[SUB2],b¡2:> |
 =  @[left = t¡1; data = SUB2¡|d|; right = t¡2]
|  | ←b |  =  (t^).left
|  | →b |  =  (t^).right
| exp  •b                      into  t^.data
| exp  b¡1<b¡2                 into  t¡1^.data<t¡2^.data
| exp  b¡1=b¡2                 into  t¡1=t¡2
| exp  b¡1≠b¡2                 into  t¡1≠t¡2
| stmt b¡1  := b¡2             into  t¡1  := t¡2
with   type trec =
    record left: ptr(trec); data: range[SUB2]; right: ptr(trec) end
end
```

Figure 8.9: Transform *Btree_Trec*.

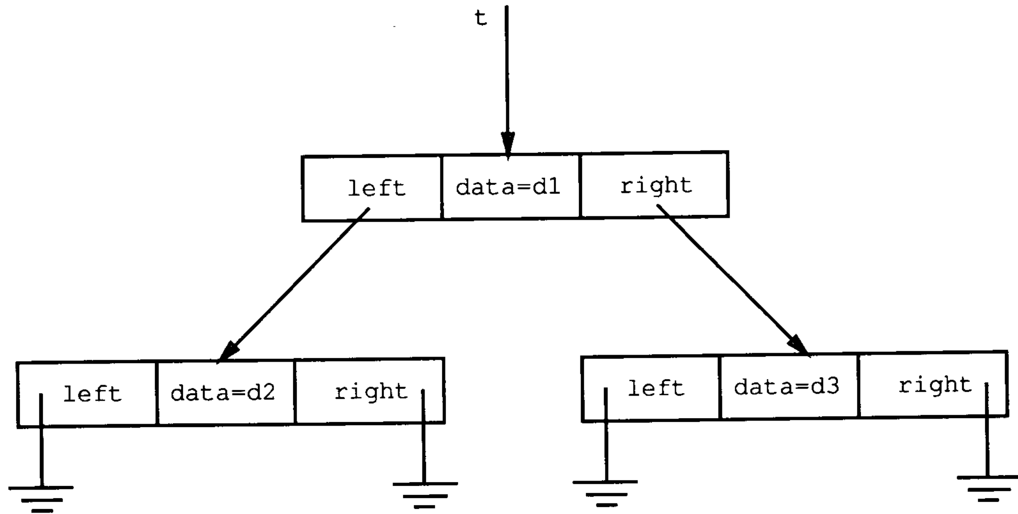


Figure 8.10: Implementation of tree ⟨:⟨:*tnil*, *d*2, *tnil*:⟩, *d*1, ⟨:*tnil*, *d*3, *tnil*:⟩:⟩ according to *Btree_Trec*.

Transform *Seq_Dll* implements a variable of type "sequence of elements of some type" with another variable of type "doubly linked list of elements of the same type". Figures 8.11–8.13 contain screen dumps of *Seq_Dll*. According to *Seq_Dll*, a variable of type "sequence of elements of some type" is implemented as a record (of type *rec*; this type is defined after **with** at the end of transform *Seq_Dll*) that contains the size of the sequence (which is also the length of the doubly-linked list) and two pointers to a doubly-linked list: one to the head and one to the tail of the list. Each element of the doubly-linked list (of type *cell*) contains an element of the sequence. Figure 8.14 depicts the implementation of a sequence according to *Seq_Dll*.

Transform *Seq_Dll* implements an empty sequence, and a sequence having a single element. It implements expressions for prepending and appending an element to a sequence, for catenating two sequences, for accessing an element of a sequence, for extracting a subsequence of a sequence, for membership testing, and for the size of a sequence. Finally, it provides implementations for assigning a new value to element i of a sequence, and for assigning one sequence to another.

Transform *Set_Seq* implements a variable of type *set* with a sequence. The sequence is maintained in *heap order*, i.e. the first element of the sequence is the smallest element of the heap. Heaps and heap operations are studied in [AHU78]. Figures 8.15 and 8.16 contain transform *Set_Seq*. *Set_Seq* actually implements bags, since it does not check for membership when adding a new element to a set. Transform *Set_Seq* implements the size of a set, the empty set, and singleton sets. It also provides imple-

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help

(* Seq_Dll implements sequences with doubly linked lists *)
transform  Seq_Dll[SUB1];
  var s: seq(domain[SUB1]) into c: rec
coupling invariant : <expression>
▮  | «» |  =  [n = 0; hd = nil; tl = nil]
▮  | «exp- e:domain[SUB1]» |  =  Single(SUB1¡|e|)
▮  | s▷exp- e:domain[SUB1] |  =  Append(c, SUB1¡|e|)
▮  | exp- e:domain[SUB1]◁s |  =  Prepend(SUB1¡|e|, c)
▮  | s¡0 cat s¡1 |  =  Cat(c¡0, c¡1)
▮ SUB1¡ | s«exp- i:int» |  =  Index(c, i)
▮  | s«exp- i:int..» |  =  CopyEnd(c, i)
▮  | s«exp- i:int..exp- j:int» |  =  Copy(c, i, j)
▮  | s«..exp- i:int» |  =  Copy(c, 0, i)
▮ exp  exp- e:domain[SUB1]∈s
    into  from
           var p: ptr(cell) := c.hd ;
           do p≠nil cand (p^).d≠SUB1¡|e| --> p := (p^).r  od
          yield p≠nil
▮ exp  exp- e:domain[SUB1]∉s
    into  from
           var p: ptr(cell) := c.hd ;
           do p≠nil cand (p^).d≠SUB1¡|e| --> p := (p^).r  od
          yield p=nil
▮ exp  #s                       into  c.n
▮ stmt s«exp- i1:int», s«exp- i2:int» :=
     s«exp- i2:int», s«exp- i1:int»
                                into  block
       var j: int := 0 ;
       var p1: ptr(cell) := c.hd ;
       var p2: ptr(cell) ;
       if i1=i2 --> skip
       ▮ i1≠i2 -->
        const max = (i1↑i2): int ;
        const min = (i1↓i2): int ;
        do j<min --> p1 := (p1^).r ; j := j+1  od ;
        p2 := p1 ;
        do j<max --> p2 := (p2^).r ; j := j+1  od ;
        (p1^): cell.d, (p2^): cell.d := (p2^).d: Range, (p1^).d: Range

       fi
      end
▮ stmt s¡0 := s¡1               into  c¡0 := CopyEnd(c¡1, 0)
```

Figure 8.11: Transform *Seq_Dll*, part 1.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                    Help
▮ stmt s«exp- i:int» := exp- e:domain[SUB1]
 into  block
      var j: int := 0 ;
      var p: ptr(cell) := c.hd ;
      do j<i --> p := (p^).r ; j := j+1  od ;
      (p^).d := SUB1¡▮e▮
     end
with  type cell =
    record l: ptr(cell); r: ptr(cell); d: range[SUB1] end ;
   type Range = range[SUB1] ;
   type rec = record n: int; hd: ptr(cell); tl: ptr(cell) end ;
   const Single =
    fn (e: range[SUB1])
    from
     var cp: ptr(cell) := @[l = nil; r = nil; d = e]
    yield [n = 1; hd = cp; tl = cp] ;
   const Index =
    fn (cc: rec, i: int)
    from
     var cp: ptr(cell) := cc.hd ;
     var j: int := 0 ;
     do j<i --> cp := (cp^).r ; j := j+1  od
    yield (cp^).d: range[SUB1] ;
   const Copy =
    fn (p: rec, m: int, n: int)
    from
     var i: int := 0 ;
     var t: ptr(cell) := p.hd ;
     var a: ptr(cell) ;
     var b: ptr(cell) ;
     var q: rec ;
     do i<m --> t := (t^).r ; i := i+1  od ;
     q := [n = (n-m+1)↑0; hd = nil; tl = nil] ;
     a := nil ;
     b := nil ;
     do i≤n -->
      b := @[l = a; r = nil; d = (t^).d] ;
      if a≠nil --> (a^).r := b  ▮ a=nil --> q.hd := b  fi ;
      a := b ;
      i := i+1 ;
      t := (t^).r
     od ;
     q.tl := b
    yield q ;
   const CopyEnd = fn (p: rec, m: int) Copy(p, m, p.n-1) ;
```

Figure 8.12: Transform *Seq_Dll*, part 2.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                    Help
  const Append =
   fn (p: rec, e: range[SUB1])
   (if p.n=0 then Single(e)
     else from
            var cc: rec := CopyEnd(p, 0) ;
            var cp: ptr(cell) := @[l = cc.tl; r = nil; d = e] ;
            (cc.tl^).r := cp ;
            cc.tl := cp ;
            cc.n := cc.n+1
           yield cc): rec ;
  const Prepend =
   fn (e: range[SUB1], p: rec)
   (if p.n=0 then Single(e)
     else from
            var cc: rec := CopyEnd(p, 0) ;
            var cp: ptr(cell) := @[l = nil; r = cc.hd; d = e] ;
            (cc.hd^).l := cp ;
            cc.hd := cp ;
            cc.n := cc.n+1
           yield cc): rec ;
  const Cat =
   fn (c1: rec, c2: rec)
   if c1.n=0 -> CopyEnd(c2, 0)
   | c2.n=0 -> CopyEnd(c1, 0)
   | otherwise ->
     from
      var cc0: rec ;
      var cc1: rec ;
      cc0 := CopyEnd(c1, 0) ;
      cc1 := CopyEnd(c2, 0) ;
      cc0.n := cc0.n+cc1.n ;
      (cc1.hd^).l := cc0.tl ;
      (cc0.tl^).r := cc1.hd ;
      cc0.tl := cc1.tl
     yield cc0
   fi
end
```

Figure 8.13: Transform *Seq_Dll*, part 3.

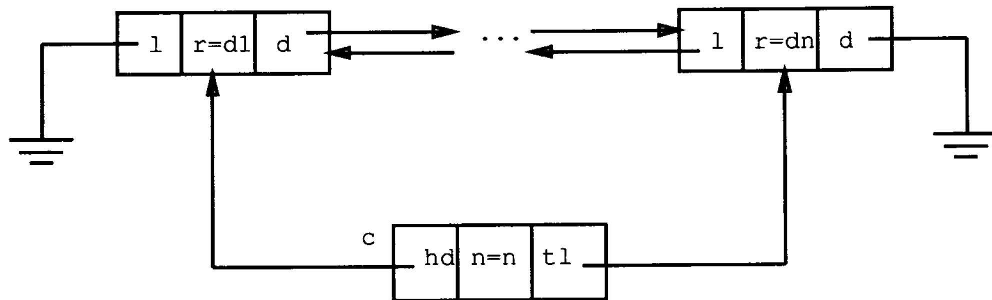


Figure 8.14: Implementation of sequence $\langle\!\langle d_1, \ldots, d_n \rangle\!\rangle$ according to *Seq_Dll*.

mentations of operations for selecting the minimum element of a set (recall that, since the sequence is maintained in heap order the minimum element of the set is the first element of the sequence), for removing the minimum element of a set (implemented by removing the first element of the corresponding sequence), and for adding an element to a set. Since the heap order must be maintained among elements of the sequence, the sequence must be reorganized after adding or removing an element from it. This is accomplished by procedures *bubble_up* and *bubble_down*, which are defined at the end of transform *Set_Seq*. Finally, transform *Set_Seq* provides an implementation for assigning one set to another.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                    Help
(* Set_Seq implements sets with sequences *)
transform  Set_Seq[SUB, CONT];
  var s: set(domain[SUB]) into a: seq(range[SUB])
coupling invariant : <expression>
| exp  #s                        into  #a
| stmt s := s-{s↓↓}
                                 into  if #a>2 -->
      block a := a«#a-1»◁a«1..#a-2» ; bubble_down(a)  end
     | #a=2 --> a := «a«1»»
     | #a<2 --> a := «»
     fi
|  | {} |  =  «»
|  | {exp- x:domain[SUB]} |  =  «SUB¡‖x‖»
| SUB¡ | s↓↓ |  =  a«0»
| stmt s¡1 := s¡2               into  a¡1 := a¡2
| stmt s := s∪{exp- x:domain[SUB]}
          into  block a := a▷SUB¡‖x‖ ; bubble_up(a)  end
```

Figure 8.15: Transform *Set_Seq*, part 1.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
with   procedure bubble_up(var aa: seq(range[SUB])) =
    var i: int := #aa-1 ;
    do i>0 cand aa«i»<aa«(i-1)/2» -->
     aa«i», aa«(i-1)/2» := aa«(i-1)/2», aa«i» ;
     i := (i-1)/2
    od
   end ;
   procedure bubble_down(var aa: seq(range[SUB])) =
    var i: int := ~1 ;
    var j: int := 0 ;
    var left: int ;
    var right: int ;
    do j≠i -->
     i := j ;
     left := 2*i+1 ;
     right := 2*i+2 ;
     j :=
      (if (left<#aa cand aa«left»<aa«i»)
        then (if (right<#aa cand aa«right»<aa«i»)
         then (if aa«left»<aa«right» then left else right)
         : int else left): int else (if (right<#aa cand aa«right»<aa«i»
        ) then right else i): int): int ;
     aa«i», aa«j» := aa«j», aa«i»
    od
   end

 ##DIRECTIVES##

  change a into b using CONT::seq(range[SUB])
 end
```

Figure 8.16: Transform *Set_Seq*, part 2.

The concrete variable a of transform *Set_Seq* is of type *seq*, which is an abstract type. The operations on a that are used on rule-replacements of *Set_Seq* were defined in type *seq*(t), shown in Figure 8.6. The directive at the end of *Set_Seq* requests that this variable be implemented as a doubly-linked list. This illustrates that variables in an implementation can themselves be transformed.

The transform directives

The directives for implementing the abstract variables of the algorithm are shown in Figure 8.4. Variable S:*set*(*btree*(*int*)) is implemented as follows. First, it is implemented with a sequence, and then the sequence is implemented with a doubly-linked list. The elements of the set (each of type *btree*(*int*)), and variables x and y are implemented with pointers to records, using transform *Btree_Trec*.

The transformed program

Figures 8.17 and 8.18 contain screen dumps of parts of the transformed program. The transformed program is quite long; we do not show all of it here. Variable S of the original program has been replaced by variable s of the appropriate type. The body of the **do**-loop has been transformed as well. The size of S has been replaced by field n of record s. The assignment to variable x has been replaced by the code for accessing the first element of the doubly-linked list. The removal of x from set S (recall that x is the least element of S) has been implemented by code that removes the first element of the doubly linked list and a call to *bubble_down* to maintain the heap order. Similar transformations take place for the operations on variable y. Finally, the union of S with the singleton set has been replaced by code that appends an element to the doubly-linked list and calls *bubble_up* to maintain the heap order among elements of the sequence.

Naturally, the programmer need not look at the transformed program. But it is interesting to see how the short algorithm of Figure 8.4 is turned into this expanded code through the transformation process.

The transformed program is then automatically transformed into a C program that can be compiled and run. The size of the object code that is produced by the C compiler is approximately 7 Kbytes. As an example, when the compiled C program runs on input `1 2 3 4`, it produces the output shown in Figure 8.19.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
(*--Program can be transformed*)

;
type _Btree_Trec__I__btree_int_trec =
 record
  _Btree_Trec__I__btree_int_left: ptr(_Btree_Trec__I__btree_int_trec);
  _Btree_Trec__I__btree_int_data: int;
  _Btree_Trec__I__btree_int_right: ptr(_Btree_Trec__I__btree_int_trec)
 end ;
type _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_cell =
 record
  _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_l:
   ptr(_Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_cell);
  _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_r:
   ptr(_Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_cell);
  _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_d:
   ptr(_Btree_Trec__I__btree_int_trec)
 end ;
type _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_Range =
 ptr(_Btree_Trec__I__btree_int_trec) ;
type _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_rec =
 record
  _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_n: int;
  _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_hd:
   ptr(_Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_cell);
  _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_tl:
   ptr(_Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_cell)
 end ;
const _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_Single =
 fn (e: ptr(_Btree_Trec__I__btree_int_trec))
 from
  var cp:
   ptr(_Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_cell) :=
    @[_Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_l = nil;
      _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_r = nil;
      _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_d = e]
 yield
  [_Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_n = 1;
   _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_hd = cp;
```

Figure 8.17: Transformed program, part 1.

```
Polya File Edit View Tools Options Structure Text                    Help
procedure print_btree_Btree_Trec__I__btree_int_I(
 _Btree_Trec__I__btree_int_t_t: ptr(_Btree_Trec__I__btree_int_trec),
 margin: int) =
 if _Btree_Trec__I__btree_int_t_t≠nil -->
  block
   var i: int := 0 ;
   do i<margin --> print(" ") ; i := i+1  od ;
   print(_Btree_Trec__I__btree_int_t_t^._Btree_Trec__I__btree_int_data
          : int) ;
   print("\n") ;
   print_btree_Btree_Trec__I__btree_int_I((_Btree_Trec__I__btree_int_t_
                                          ._Btree_Trec__I__btree_int_le
                                          , margin+5) ;
   print_btree_Btree_Trec__I__btree_int_I((_Btree_Trec__I__btree_int_t_
                                          ._Btree_Trec__I__btree_int_ri
                                          , margin+5)
  end
 | _Btree_Trec__I__btree_int_t_t=nil --> skip
 fi
end ;
var s: _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_rec ;
read_set_Set_Seq__I_Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int__s
   ;
do s._Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_n≠1 -->
 var _Btree_Trec__I__btree_int_t_x:
  ptr(_Btree_Trec__I__btree_int_trec) ;
 var _Btree_Trec__I__btree_int_t_y:
  ptr(_Btree_Trec__I__btree_int_trec) ;
 _Btree_Trec__I__btree_int_t_x :=
  _Seq_Dll__Btree_Trec__I__btree_int__seq_btree_int_Index(s, 0) ;
```

Figure 8.18: Transformed program, part 2.

```
10
     4
     6
          3
          3
               1
               2
```

Figure 8.19: Output of Huffman encoding algorithm.

8.3 Example: The Hopcroft-Tarjan Planarity Test Algorithm

The Hopcroft-Tarjan Planarity Algorithm [HT74,GX88] tests whether an undirected graph is planar and, if it is, constructs a planar embedding for it. The algorithm is linear in the number of vertices of the graph. The algorithm is quite complex and subtle, and a clear, precise presentation of it is difficult to achieve. Its presentation in *Polya* illustrates very well the usefulness of the paradigm of presenting an algorithm at a high level of abstraction and then directing the implementation of its variables using transforms.

The version of the algorithm that we show how to implement is due to Gries and Xue [GX88]. Although we give some explanations on how the algorithm works, one should refer to [GX88] for details and a proof of correctness. We also use terms from [GX88] without repeating the definitions here.

8.3.1 Abstract types used in the algorithm

The algorithm uses directed graphs, sets, and sequences. The type definitions for sets and sequences were given in Section 8.2. The type definition for directed graphs that have vertices numbered with natural numbers and no weights on the edges is given in Figure 8.20.

Throughout this section we use the convention that a graph G is undirected iff $(u, v) \in G \equiv (v, u) \in G$, for all vertices u, v.

The type definition of graphs, shown in Figure 8.20, contains the following operations.

1. Expression **newgraph** e yields a new graph whose vertices are numbered $0, \ldots, (e-1)$, and has no edges.

2. Statement **newedge** $(v0, v1)$ **in** g adds edge $(v0, v1)$ to graph g.

3. Statement **deledge** $(v0, v1)$ **from** g removes edge $(v0, v1)$ from graph g.

4. Expression **vertnum** g is the number of vertices of graph g.

5. Expression **edgenum** g is the number of edges of graph g.

6. Statement **deladjedges of** $v0$ **from** g removes all edges in g that are adjacent to vertex $v0$.

7. Statement **for** (u, v) **edge of** g **do** s executes statement s for every edge (u, v) of graph g.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                    Help
type dgraph
operations
  newdgraph exp-e (e: int) dgraph as "newdgraph" e;
  newedge exp-g exp-v0 exp-v1 (g: dgraph; v0: int; v1: int)
   as "newedge" "(" v0 "," v1 ")" "in" g ;
  deledge exp-g exp-v0 exp-v1 (g: dgraph; v0: int; v1: int)
   as "deledge" "(" v0 "," v1 ")" "from" g ;
  vertnum exp-g (g: dgraph) int as "vertnum" g;
  edgenum exp-g (g: dgraph) int as "edgenum" g;
  deladjedges exp-g exp-v0 (g: dgraph; v0: int)
   as "deladjedges" "of" v0 "from" g ;
  for_edge id-u id-v exp-g stmt-s (u: int; v: int; g: dgraph)
   as "for" "(" u "," v ")" "edge" "of" g "do" s ;
  for_adj exp-u id-v exp-g stmt-s (u: int; v: int; g: dgraph)
   as "for" v "adjacent" "to" u "in" g "do" s ;
  for_adj_cond exp-u id-v exp-g exp-e stmt-s
   (u: int; v: int; g: dgraph; e: bool)
   as "for" v "adjacent" "to" u "in" g "cond" e "do" s
end
```

Figure 8.20: Definition of type *dgraph*.

8. Statement **for** v **adjacent to** u **in** g **do** s executes statement s for every vertex v that is adjacent to vertex u in graph g.

9. Statement **for** v **adjacent to** u **in** g **cond** e **do** s executes statement s for every vertex v that is adjacent to vertex u in graph g, as long as condition e holds.

The algorithm makes use of an abstract notion of a "distinguished" or "first" vertex that is associated with a vertex of the graph. This notion is specific to the algorithm and is not a general operation on graphs. Hence, it is not defined in the type definition for *dgraph*. In Section 8.3.8 we discuss how we can implement such an operation.

8.3.2 Input and output of the algorithm

The planarity test algorithm is given in Figures 8.21–8.28. Figure 8.21 contains the definitions of *Readgraph* and *PrintGraph* for reading and printing a graph.

The input to the algorithm is an undirected graph, which, by the convention we have adopted, means that for each edge (u, v), (v, u) is also in the graph. *Readgraph* reads a graph having the following format.

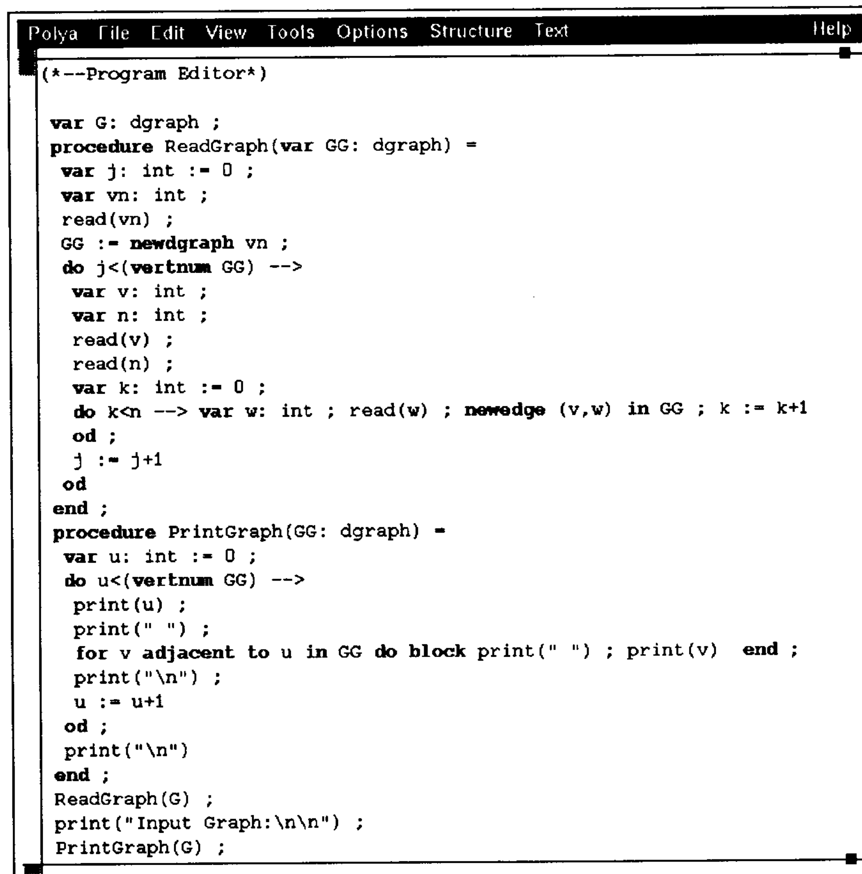
- Number of vertices of the graph, followed by an equal number of lines.

- Each line contains a vertex number, the number of adjacent vertices to it, and a list of the adjacent vertices.

We use *C* functions for reading numbers.

PrintGraph outputs a list of lines. Each line contains a vertex number followed by the vertices that are adjacent to it.

The representation of the graph is hidden at this point. Both procedures are expressed in abstract terms, using the operations of type *dgraph*.




```
(*--Program Editor*)

var G: dgraph ;
procedure ReadGraph(var GG: dgraph) =
 var j: int := 0 ;
 var vn: int ;
 read(vn) ;
 GG := newdgraph vn ;
 do j<(vertnum GG) -->
  var v: int ;
  var n: int ;
  read(v) ;
  read(n) ;
  var k: int := 0 ;
  do k<n --> var w: int ; read(w) ; newedge (v,w) in GG ; k := k+1
  od ;
  j := j+1
 od
end ;
procedure PrintGraph(GG: dgraph) =
 var u: int := 0 ;
 do u<(vertnum GG) -->
  print(u) ;
  print(" ") ;
  for v adjacent to u in GG do block print(" ") ; print(v)  end ;
  print("\n") ;
  u := u+1
 od ;
 print("\n")
end ;
ReadGraph(G) ;
print("Input Graph:\n\n") ;
PrintGraph(G) ;
```

Figure 8.21: Definitions of *Readgraph* and *PrintGraph*.

8.3.3 Constructing a directed graph

The planarity test algorithm consists of two parts. First, the undirected graph (which is actually a directed one in which for every edge (u, v), (v, u) is also in the graph) is

converted to a directed one while the vertices of the original graph are renamed. This is accomplished by procedure *RenamGraph*, which calls *DFS* for a depth-first order traversal of the graph. For each pair $(u, v), (v, u)$ of directed edges, *DFS* removes exactly one of them; thus, constructing a directed graph. During the traversal, a renumbering of the vertices of the original graph is computed. This is accomplished by maintaining three arrays, L, LL, and N, as follows.

- During the depth-first traversal of the graph, for a vertex v, $N[v]$ is the number of vertices processed before v.
- For a vertex v, $L[v]$ is the minimum value of $N[w]$ for vertices w reachable from v along span-frond paths.
- For a vertex v, $LL[v]$ is the second minimum value of $N[w]$ for vertices w reachable from v along span-frond paths (or v if there is no such second minimum value).

After *DFS* traverses the graph, array N contains the map for renumbering the vertices: $N[v]$ will be the new number of vertex v in the renamed graph.

After the depth-first traversal and the construction of the corresponding directed graph, *RenamGraph* renames the vertices of the graph, thus constructing a *palm* graph. Procedure *RenamGraph* is shown in Figures 8.22 and 8.23.

8.3.4 Producing an embedding

Procedure *Planarity* works on the graph that is produced by *RenamGraph*. It is given a cycle in the graph (whose lowest-numbered vertex is *cstart*), the beginning of a spine (defined by vertices u and v), and a partition of the vertices processed thus far (sequence b). Parameter b is a sequence of blocks, and each block is a set of segments. Each element B of b contains two fields.

- $B.I$: the segments of B that are inside of cycle c, and
- $B.O$: the segments of B that are outside of cycle c.

For each segment B, $B.I.S$ ($B.O.S$) is a sequence of segment-numbers for segments that are inside (outside) of cycle c. For each segment B, $B.I.A$ ($B.O.A$) is a sequence of attachments to cycle c.

If the segment defined by u and v can be embedded in the plane with respect to the segments in b, then x is set to *true*, and b is augmented with the new segment resulting in an augmented embedding, otherwise x is set to *false*. The details of the algorithm and its correctness are given in [GX88]. Procedure *Planarity* is shown in Figures 8.24 and 8.25.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                    Help
var N: array [vertnum G] of int ;
procedure RenamGraph(var GG: dgraph) =
 var L: array [vertnum GG] of int ;
 var LL: array [vertnum GG] of int ;
 block
  var i: int := 0 ;
  do i<(vertnum GG) --> N[i] := ~1 ; i := i+1  od
 end ;
 var n: int := 0 ;
 procedure DFS(u: int, v: int) =
  N[v] := n ;
  n := n+1 ;
  L[v] := v ;
  LL[v] := v ;
  for w adjacent to v in GG do if w≠u∧N[w]≥0∧N[v]>N[w] -->
   if N[w]<L[v] --> LL[v] := L[v] ; L[v] := N[w]
   | N[w]=L[v] --> skip
   | N[w]>L[v] --> LL[v] := LL[v]↓N[w]
   fi
  | w=u∨N[v]<N[w] --> deledge (v,w) from GG
  | N[w]=~1 -->
   DFS(v, w) ;
   if L[w]<L[v]∧L[w]<LL[w] --> LL[v] := L[v]↓LL[w] ; L[v] := L[w]
   | L[w]<L[v]∧L[w]=LL[w] --> LL[v] := L[v] ; L[v] := L[w]
   | L[w]=L[v]∧L[w]<LL[w] --> LL[v] := LL[v]↓LL[w]
   | L[w]=L[v]∧L[w]=LL[w] --> skip
   | L[w]>L[v] --> LL[v] := LL[v]↓L[w]
   fi
  fi
 end ;
 const theta =
  fn (v: int, w: int)
  if N[v]>N[w] -> 2*N[w]: int
  | N[v]≤N[w]∧LL[w]=w -> 2*L[w]
  | N[v]≤N[w]∧LL[w]<w -> 2*L[w]+1
  fi ;
 type gedge = record v0: int; v1: int end ;
 var Bucket: array [2*(vertnum GG)] of set(gedge) ;
 DFS(0, 0) ;
```

Figure 8.22: Definition of *RenamGraph*, part 1.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                    Help
block
 var j: int := 0 ;
 do j<2*(vertnum GG) --> Bucket[j] := {} ; j := j+1  od
end ;
for (v,w) edge of GG do block
 var th: int := theta(v, w) ;
 Bucket[th] := Bucket[th]∪{[v0 = v; v1 = w]: gedge}
end ;
block
 var v: int := 0 ;
 do v<(vertnum GG) --> deladjedges of v from GG ; v := v+1  od
end ;
block
 var j: int := 2*(vertnum GG)-1 ;
 do j≥0 -->
  var e: gedge ;
  do Bucket[j]≠{} -->
   block e := choose(Bucket[j]) ; Bucket[j] := Bucket[j]-{e}  end ;
   newedge (N[e.v0],N[e.v1]) in GG
  od ;
  j := j-1
 od
end
end ;
RenamGraph(G) ;
print("Renaming of edges:\n\n") ;
block
 var i: int := 0 ;
 do i<(vertnum G) -->
  print(i) ;
  print(" -> ") ;
  print(N[i]) ;
  print("\n") ;
  i := i+1
 od ;
 print("\n")
end ;
print("Renamed Directed Graph:\n\n") ;
PrintGraph(G) ;
```

Figure 8.23: Definition of *RenamGraph*, part 2.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
type BlockT = record A: seq(int); S: seq(int) end ;
type PartT = record I: BlockT; O: BlockT end ;
var Side: array [(edgenum G)-(vertnum G)+1] of bool ;
var N_Side: int := 0 ;
procedure Planarity(cstart: int,
                    u: int,
                    v: int,
                    var x: bool,
                    var b: seq(PartT)) =
 var spi: seq(int) ;
 var w: int ;
 const Lace = fn (t: seq(int)) (t≠«» cand t«~1»>w): bool ;
 procedure Merge =
  b«~2»: PartT.I.A := b«~2»: PartT.I.A cat b«~1».I.A ;
  b«~2»: PartT.O.A := b«~2»: PartT.O.A cat b«~1».O.A ;
  b«~2»: PartT.I.S := b«~2»: PartT.I.S cat b«~1».I.S ;
  b«~2»: PartT.O.S := b«~2»: PartT.O.S cat b«~1».O.S ;
  b := b«0..#b-2»
 end ;
 procedure FixSide(var t: seq(int), X: bool) =
  do t≠«» --> Side[t«~1»] := X ; t := t«0..#t-2»  od
 end ;
 procedure Deletefrom(var t: seq(int), y: int) =
  do t≠«» cand t«~1»≥y --> t := t«0..#t-2»  od
 end ;
 procedure Purge(var q: seq(int), y: int) =
  var h: bool := q≠«» ;
  do h -->
   Deletefrom(q«~1».I.A, y) ;
   Deletefrom(q«~1».O.A, y) ;
   if q«~1».I.A=«»∧q«~1».O.A=«» -->
    FixSide(q«~1».I.S, true) ;
    FixSide(q«~1».O.S, false) ;
    q := q«0..#q-2» ;
    h := q≠«»
   ▮ q«~1».I.A≠«»∨q«~1».O.A≠«» --> h := false
   fi
  od
 end ;
 spi := «» ;
 w := v ;
 do w>u --> spi := spi⊳w ; w := firstv(w, G)  od ;
```

Figure 8.24: Definition of *Planarity*, part 1.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
block
 var v: PartT ;
 if w>cstart --> v.I.A := «w»  | w=cstart --> v.I.A := «»  fi ;
 v.O.A := «» ;
 v.I.S := «N_Side» ;
 v.O.S := «» ;
 N_Side := N_Side+1 ;
 b := b▷v
end ;
do x∧#b>1 cand Lace(b«~2».I.A) cand Lace(b«~2».O.A) --> x := false
| x∧#b>1 cand ¬Lace(b«~2».I.A) cand Lace(b«~2».O.A) --> Merge
| x∧#b>1 cand Lace(b«~2».I.A) cand ¬Lace(b«~2».O.A) -->
 b«~2».I: BlockT, b«~2».O: BlockT :=
  b«~2».O: BlockT, b«~2».I: BlockT ;
 Merge
od ;
var q: seq(PartT) ;
q := «» ;
do x∧spi≠«» -->
 var y: int := spi«~1» ;
 spi := spi«0..#spi-2» ;
 Purge(q, y) ;
 for r adjacent to y in G cond x do if r=firstv(y, G) --> skip
 | ¬(r=firstv(y, G)) --> Planarity(w, y, r, x, q)
 fi
od ;
Purge(q, u) ;
block
 var t: seq(int) ;
 t := «» ;
 do x∧q≠«» -->
  if q«~1».I.A≠«»∧q«~1».O.A≠«» --> x := false
  | q«~1».I.A=«»∧q«~1».O.A≠«» -->
   q«~1».I: BlockT, q«~1».O: BlockT :=
    q«~1».O: BlockT, q«~1».I: BlockT
  | q«~1».O.A=«» --> skip
  fi ;
  FixSide(q«~1».I.S, true) ;
  FixSide(q«~1».O.S, false) ;
  t := q«~1».I.A cat t ;
  q := q«0..#q-2»
 od ;
 b«~1».I.A := b«~1».I.A cat t
 end
end ;
```

Figure 8.25: Definition of *Planarity*, part 2.

8.3.5 Printing an embedding

Procedure *PrintEmbed*, shown in Figure 8.26, prints a planar embedding of the graph, if the graph is planar. *PrintEmbed* is similar to *Planarity*: it recursively traverses the graph produced by *RenamGraph* but it does not process any partitions. It has three parameters: a cycle c and two vertices u and v that define a spine emanating from c. First, it constructs spines *spi* and *spine* that are defined by vertices u and v. Spine *spi* does not contain any vertices of c. On the other hand, *spine* contains both attachments to cycle c. In other words

$$spine = u \frown spi \frown w$$

where w is a vertex in c.

In the sequel, *PrintEmbed* indicates if *spi* is inside or outside cycle c. It then constructs cycle $c2$: the cycle that is defined by $seam(c, s) \frown spi$ (where s is the segment defined by the spine that is defined by vertices u and v). Following the construction of $c2$, *PrintEmbed* calls itself recursively on all vertices (but the first) that are adjacent to every vertex of spine *spine*. Figure 8.27 contains the definition of procedure *PrintSeq*, which prints a sequence of integers.

8.3.6 The main program

The main program is shown in Figure 8.28. It initializes variable b to the empty partition and then calls *Planarity*. If the graph is planar, (variable x is set to *true*) it calls *PrintEmbed* to print a planar embedding of it.

8.3.7 Directives

Figures 8.21–8.28 have shown the algorithm at a very high level, for ease of understanding. Typically, to implement this algorithm, one would have to reprogram it in a language like *C*, *Pascal*, or *Scheme*, a time consuming and error-prone method. We can do better by using off-the-shelf transforms for implementing the variables and the operations on them. By doing so, the algorithm *is* the program, it need not be changed even if a different implementation of the variables is desired.

The directives for transforming program variables are given in Figure 8.29. We employ transforms *RevSeq* for implementing a sequence by its reverse, and *Seq_List* for implementing a sequence with a linked list. All sequences that are used by the algorithm are implemented as reversed linked lists, and sets are implemented as sequences with transform *Set_Seq*. We implement graph G using transform *Graph*. These transforms are discussed in the next section.

Polya File Edit View Tools Options Structure Text Help

```
procedure PrintEmbed(c: seq(int), u: int, v: int) =
 var spine: seq(int) ;
 var spi: seq(int) ;
 var c1: seq(int) ;
 var c2: seq(int) ;
 var w: int ;
 spine := «» ;
 spi := «u» ;
 w := v ;
 do w>u --> spine := spine▷w ; spi := spi▷w ; w := firstv(w, G)  od
   ;
 spi := spi▷w ;
 print("Segment: ") ;
 print(N_Side) ;
 print("\n  Spine: ") ;
 PrintSeq(spi) ;
 print("\n") ;
 if Side[N_Side] --> print("  Inside of cycle: ")
 ▮ ¬Side[N_Side] --> print("  Outside of cycle: ")
 fi ;
 PrintSeq(c) ;
 c1 := CopySeq(c) ;
 do c1«-1»≠u --> c1 := c1«0..#c1-2»  od ;
 c1 := c1«0..#c1-2» ;
 c2 := «» ;
 do c1«-1»≠w --> c2 := c1«-1»◁c2 ; c1 := c1«0..#c1-2»  od ;
 if c2≠«» --> c2 := w◁c2  ▮ c2=«» --> skip  fi ;
 c2 := c2 cat spi ;
 print("\n") ;
 N_Side := N_Side+1 ;
 do spine≠«» -->
  var y: int := spine«-1» ;
  spine := spine«0..#spine-2» ;
  for r adjacent to y in G do if r=firstv(y, G) --> skip
  ▮ ¬(r=firstv(y, G)) --> PrintEmbed(c2, y, r)
  fi
 od ;
 do spi≠«» --> spi := spi«0..#spi-2»  od ;
 do c1≠«» --> c1 := c1«0..#c1-2»  od ;
 do c2≠«» --> c2 := c2«0..#c2-2»  od
end ;
```

Figure 8.26: Definition of *PrintEmbed*.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help

procedure PrintSeq(s: seq(int)) =
 var first: bool := true ;
 print("<") ;
 for e inseq s do block
  if ¬first --> print(" ")  | first --> first := false  fi ;
  print(e: int)
 end ;
 print(">")
end ;
```

Figure 8.27: Definition of *PrintSeq*.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help

var x: bool := true ;
var p: seq(PartT) ;
p := «» ;
Planarity(0, 0, firstv(0, G), x, p) ;
if x -->
 var cycle: seq(int) ;
 cycle := «0» ;
 cycle := cycle⊳«0» ;
 print("Graph is planar.\nEmbedding:\n\n") ;
 N_Side := 0 ;
 PrintEmbed(cycle, 0, firstv(0, G))
| ¬x --> print("Graph is non-planar.\n\n")
fi
```

Figure 8.28: Main program.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                    Help
##DIRECTIVES##

change RenamGraph;Bucket into B using Arr[Set_Seq[I]::set(gedge);
  RevSeq[I]::seq(gedge); Seq_List[I]::seq(gedge)]::array of set(gedge)
change Planarity;spi into SPI using RevSeq[I]::seq(int);
  Seq_List[I]::seq(int)
change Planarity;t into T using RevSeq[I]::seq(int);
  Seq_List[I]::seq(int)
change Planarity;q into Q using RevSeq[I]::seq(PartT);
  Seq_List[I]::seq(PartT)
change PrintEmbed;spine into SPINE using RevSeq[I]::seq(int);
  Seq_List[I]::seq(int)
change PrintEmbed;spi into SPI using RevSeq[I]::seq(int);
  Seq_List[I]::seq(int)
change PrintEmbed;c1 into C1 using RevSeq[I]::seq(int);
  Seq_List[I]::seq(int)
change PrintEmbed;c2 into C2 using RevSeq[I]::seq(int);
  Seq_List[I]::seq(int)
change p into P using RevSeq[I]::seq(PartT); Seq_List[I]::seq(PartT)
change cycle into CYCLE using RevSeq[I]::seq(int);
  Seq_List[I]::seq(int)
change A into AA using RevSeq[I]::seq(int); Seq_List[I]::seq(int)
change S into SS using RevSeq[I]::seq(int); Seq_List[I]::seq(int)
change G into g using Graph_AdjList
main display function :C
```

Figure 8.29: Directives for transforming abstract program variables.

8.3.8 The transforms used

Figures 8.30 and 8.31 show transform *Seq_List* that implements sequences as lists, and Figure 8.32 depicts how *Seq_List* implements a sequence.

Transform *Seq_List* implements operations for testing if a sequence is empty, and if a sequence has more than one elements. It has representation rules for constructing an empty sequence, and a sequence that has only one element. It also has rules for extracting an element of a sequence, for appending and prepending an element to a sequence, and for catenating two sequences. The constant-time implementation of catenation is worth noting, since $s0$ **cat** $s1$ is destructive on its second operand ($s1$). Any changes to $s1$ after $s0$ **cat** $s1$ is evaluated are reflected to $s0$. If an element is appended to $s1$, it will automatically be appended to $s0$. If, on the other hand, the first element of $s0$ is removed, $s1$ will be left with a dangling pointer to it. Finally, transform *Seq_List* provides two ways of copying one sequence to another. Function *CopySeq* creates a fresh copy of its argument. On the other hand, operator := is implemented by assignment of pointers.

Transform *RevSeq* of Figure 8.33 implements a sequence with a reversed sequence. Its coupling invariant $I(s, rs)$ is

$$\#s = \#rs \wedge (\forall\, i \mid 0 \leq i < \#s \,:\, s\langle\langle i\rangle\rangle = rs\langle\langle \#rs - i - 1\rangle\rangle).$$

Transform *Set_Seq* of Figure 8.34 implements sets with sequences. It is different from the one that is shown in Section 8.2. Its concrete variable, a sequence, is itself transformed by a directive at the end of *Set_Seq*. Transform *Set_Seq* implements the size of a set, and provides operations for testing if a set is empty, and choosing an element of a set. It also provides implementations for constructing the union of two sets, for constructing an empty set, for constructing a singleton set, for adding an element to a set, and for assigning one set to another.

Figures 8.35–8.36 show transform *Graph_AdjList*, which implements a graph with an adjacency list. A graph of n vertices is implemented as an array that has n entries. Entry v, $0 \leq v < n$, of the array is a list of vertices that are adjacent to v. Transform *Graph_AdjList* provides implementation for most operations of type *dgraph*, shown in Figure 8.20.

As mentioned before, the algorithm uses an abstract notion of a "first" vertex. A vertex of a graph can have at most one "first" vertex associated with it. The invariant used in transform *Graph* is that the last vertex added to the adjacency list of a vertex v of graph g is the "first" vertex associated with v. Function *firstv*(v, g) is implemented so that it returns the first value on the adjacency list of vertex v in graph g.

Finally, Figure 8.37 shows transform *Arr* for transforming the elements of an array. The implementation requires such a transform to be used. *Arr* implements

```
Polya  File  Edit  View  Tools  Options  Structure  Text                    Help
(* Seq_List implements sequences as linked lists and has destructive modifiers *)
transform  Seq_List[SUB];
  var s: seq(domain[SUB]) into c: rec
coupling invariant : <expression>
| exp  s=«»                  into  c.head=nil
| exp  s≠«»                  into  c.head≠nil
| exp  #s>0                  into  c.head≠nil
| exp  #s>1                  into  c.head≠c.tail
| SUB¡ | s«0» |  =  (c.head^).vv
| SUB¡ | s«1» |  =  ((c.head)^.next)^.vv
| SUB¡ | s«-1» |  =  (c.tail^).vv
|  | «» |  =  [head = nil; tail = nil]
|  | «exp- e:domain[SUB]» |
 =
       (from
         var _pt: ptr(cell) ;
         _pt := @[vv = SUB¡|e|; next = nil]
        yield [head = _pt; tail = _pt]: rec): rec
|  | s⊳exp- e:domain[SUB] |
 =
       (from
         var pt: ptr(cell) ;
         pt := @[vv = SUB¡|e|; next = nil] ;
         if c.head=nil --> c.head := pt
         | c.head≠nil --> (c.tail^).next := pt
         fi ;
         c.tail := pt
        yield c: rec): rec
|  | exp- e:domain[SUB]⊲s |
 =
       (from
         var pt: ptr(cell) ;
         pt := @[vv = SUB¡|e|; next = c.head] ;
         if c.head=nil --> c.tail := pt  | c.head≠nil --> skip  fi ;
         c.head := pt
        yield c: rec): rec
|  | s¡0 cat s¡1 |
 =
     if c¡0.head=nil -> c¡1: rec
     | c¡1.head=nil -> c¡0
     | c¡0.head≠nil∧c¡1.head≠nil ->
       from (c¡0.tail)^.next := c¡1.head ; c¡0.tail := c¡1.tail yield c¡0: rec
     fi
```

Figure 8.30: Transform *Seq_List*, part 1.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
▮  ▮ s«1..#s-1» ▮
 -
       (from
         var pt: ptr(cell) ;
         pt := c.head ;
         c.head := (pt^).next ;
         if c.head=nil --> c.tail := nil  ▮ c.head≠nil --> skip  fi ;
         dispose pt
        yield c: rec): rec
▮  ▮ CopySeq(s) ▮  =  CopyList(c)
▮ stmt s¡1  := s¡2            into  c¡1  := c¡2
▮ stmt for id- i:domain[SUB] inseq ↔s do stmt- st
      into  block
      var i: range[SUB] ;
      procedure ForReverse(_pt: ptr(cell)) =
       if _pt=nil --> skip
       ▮ ¬(_pt=nil) --> ForReverse(_pt^.next) ; i := _pt^.vv ; stmt- st
       fi
      end ;
      ForReverse(c.head)
     end
with   type rec = record head: ptr(cell); tail: ptr(cell) end ;
   type cell = record vv: range[SUB]; next: ptr(cell) end ;
   const CopyList =
    fn (l: rec)
    from
     var _pt: ptr(cell) := l.head ;
     var _p0: ptr(cell) := nil ;
     var r: rec ;
     r.head := nil ;
     do _pt≠nil -->
      var _q0: ptr(cell) := @[vv = (_pt^).vv; next = nil] ;
      if _p0=nil --> r.head := _q0  ▮ _p0≠nil --> (_p0^).next := _q0  fi ;
      _p0 := _q0 ;
      _pt := (_pt^).next
     od ;
     r.tail := _p0
    yield r: rec
end
```

Figure 8.31: Transform *Seq_List*, part 2.

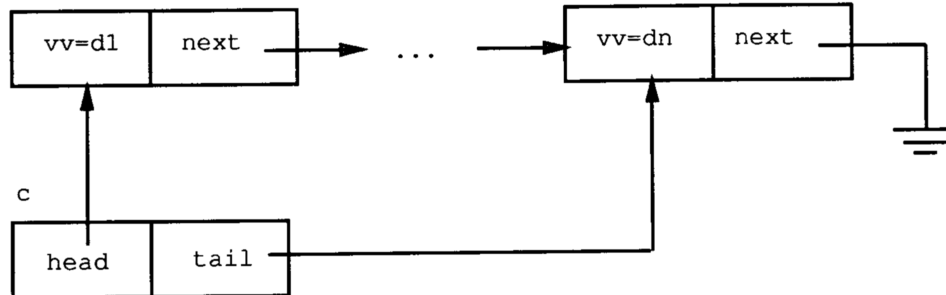


Figure 8.32: Implementation of $\langle\langle d_1, \ldots, d_n \rangle\rangle$ according to *Seq_List*.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
(* RevSeq reverses sequences *)
transform  RevSeq[SUB, CONT];
  var s: seq(domain[SUB]) into rs: seq(range[SUB])
coupling invariant : <expression>
| exp   s=«»                        into  rs=«»
| exp   s≠«»                        into  rs≠«»
| exp   #s>0                        into  #rs>0
| exp   #s>1                        into  #rs>1
| SUB¡ | s«0» |  =  rs«~1»
| SUB¡ | s«~1» |  =  rs«0»
| SUB¡ | s«~2» |  =  rs«1»
| SUB¡ | s«exp- i:int» |  =  rs«#rs-i-1»
|   | «» |  =  «»
|   | «exp- e:domain[SUB]» |  =  «SUB¡|e|»
|   | s¡1 cat s¡2 |  =  rs¡2 cat rs¡1
|   | s▷exp- e:domain[SUB] |  =  SUB¡|e|◁rs
|   | exp- e:domain[SUB]◁s |  =  rs▷SUB¡|e|
|   | s«0..#s-2» |  =  rs«1..#rs-1»
|   | s«1..#s-1» |  =  rs«0..#rs-2»
|   | CopySeq(s) |  =  CopySeq(rs)
| stmt s¡1 := s¡2                  into  rs¡1 := rs¡2
| stmt for id- i:domain[SUB] inseq s do stmt- st
        into  for i inseq ↔rs do stmt- st
| stmt for id- i:domain[SUB] inseq ↔s do stmt- st
      into  for i inseq rs do stmt- st

##DIRECTIVES##

 change rs into RS using CONT::seq(range[SUB])
end
```

Figure 8.33: Transform *RevSeq*.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
(* Set_Seq implements sets using sequences *)
transform  Set_Seq[SUB, CONT];
  var s: set(domain[SUB]) into a: seq(range[SUB])
coupling invariant : <expression>
| exp  #s                         into  #a
| stmt block
     exp- v:domain[SUB] := choose(s) ;
     s := s-{exp- v:domain[SUB]}
    end
                                  into  block
       v := a«~1» ;
       a := a«0..#a-2»
      end
|  | s¡1∪s¡2 |  =  a¡1 cat a¡2
|  | {} |  =  «»
|  | {exp- x:domain[SUB]} |  =  «SUB¡|x|»
| exp  s={}                       into  a=«»
| exp  s≠{}                       into  a≠«»
| stmt s := s∪{exp- x:domain[SUB]}
           into  block a := a⊳SUB¡|x|  end
| stmt s¡1 := s¡2                 into  a¡1 := a¡2

##DIRECTIVES##

 change a into b using CONT::seq(range[SUB])
end
```

Figure 8.34: Transform *Set_Seq*.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                                          Help
(* Graph_AdjList implements graphs as adjacency lists *)
transform  Graph_AdjList;
  var g: dgraph into c: gr
coupling invariant : <expression>
| stmt g := newdgraph exp- e:int
 into  block
        var _a: array [e] of ptr(vnode) ;
        var _i: int := 0 ;
        do _i<e --> _a[_i] := nil ; _i := _i+1  od ;
        c.vnum := e ;
        c.enum := 0 ;
        c.adj := _a
       end
| exp  firstv(exp- v:int, g)      into  ((c.adj[v])^).val
| stmt newedge (exp- v:int,exp- w:int) in g
 into  block
        var _pt: ptr(vnode) ;
        _pt := @[val = w; next = c.adj[v]] ;
        c.adj[v] := _pt ;
        c.enum := c.enum+1
       end
| stmt deledge (exp- v0:int,exp- v1:int) from g          into  Del(v0, v1, c)
| exp  vertnum g            into  c.vnum
| exp  edgenum g            into  c.enum
| stmt deladjedges of exp- v0:int from g
    into  block
        var y: ptr(vnode) ;
        var z: ptr(vnode) := c.adj[v0] ;
        do z≠nil --> y := z ; z := (z^).next ; dispose y ; c.enum := c.enum-1  od ;
        c.adj[v0] := nil
       end
| stmt for (id- u:int,id- v:int) edge of g do stmt- s
   into  block
        var u: int := 0 ;
        var v: int ;
        do u<(c.vnum) -->
         var z: ptr(vnode) := c.adj[u] ;
         do z≠nil --> v := (z^).val ; stmt- s ; z := (z^).next  od ;
         u := u+1
        od
       end
```

Figure 8.35: Transform *Graph_AdjList*, part 1.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
▮ stmt for id- v:int adjacent to exp- u:int in g cond exp- e:bool do stmt- s
   into  block
      var v: int ;
      var _z: ptr(vnode) := c.adj[u] ;
      do e∧_z≠nil --> v := (_z^).val ; stmt- s ; _z := (_z^).next  od
     end
▮ stmt for id- v:int adjacent to exp- u:int in g do stmt- s
        into  block
      var v: int ;
      var _y: ptr(vnode) := nil ;
      var _z: ptr(vnode) := c.adj[u] ;
      var _flag: bool ;
      do _z≠nil -->
       v := (_z^).val ;
       _flag := false ;
       procedure Del(u1: int, v1: int, cc: gr) =
        if u=u1∧v=v1 -->
         _flag := true ;
         if _y=nil --> c.adj[u] := (_z^).next
         ▮ _y≠nil --> (_y^).next := (_z^).next
         fi ;
         var r: ptr(vnode) := _z ;
         _z := (_z^).next ;
         dispose r ;
         cc.enum := cc.enum-1
        ▮ u≠u1∨v≠v1 --> skip
        fi
       end ;
       stmt- s ;
       if ¬_flag --> _y := _z ; _z := (_z^).next  ▮ _flag --> skip  fi
      od
     end
with  type vnode = record val: int; next: ptr(vnode) end ;
   type gr = record vnum: int; enum: int; adj: array of ptr(vnode) end ;
   procedure Del(v0: int, v1: int, cc: gr) =
    var y: ptr(vnode) := nil ;
    var z: ptr(vnode) := cc.adj[v0] ;
    do z≠nil cand (z^).val≠v1 --> y := z ; z := (z^).next  od ;
    var r: ptr(vnode) := z ;
    if y=nil --> cc.adj[v0] := (z^).next  ▮ y≠nil --> (y^).next := (z^).next  fi ;
    z := (z^).next ;
    dispose r ;
    cc.enum := cc.enum-1
   end
end
```

Figure 8.36: Transform *Graph_AdjList*, part 2.

selection of an array element and assignment of one array to another one.

```
Polya  File  Edit  View  Tools  Options  Structure  Text                Help
(* Arr transforms array elements *)
transform  Arr[SUB];
  var a: array [exp- size:int] of domain[SUB] into b: array [size] of range[SUB]
coupling invariant : <expression>
| SUB¡ | a[exp- e:int] |  =  b[e]
| stmt a¡1 := a¡2              into  b¡1 := b¡2
end
```

Figure 8.37: Transform *Arr*.

8.3.9 Example of execution

The abstract program shown in the previous sections can be transformed according to the directives for the implementation of its abstract variables. We do not show the transformed program here; it is quite long. The transformed program can be automatically transformed into a *C* program that can be compiled and executed.

The object code that is produced by the *C* compiler is approximately 24 Kbytes. Figure 8.38 (a) shows an input graph to the algorithm. Figure 8.38 (b) shows the palm graph that is produced after renaming its vertices. When run on the input graph of Figure 8.38, the output that is produced is shown in Figure 8.39.

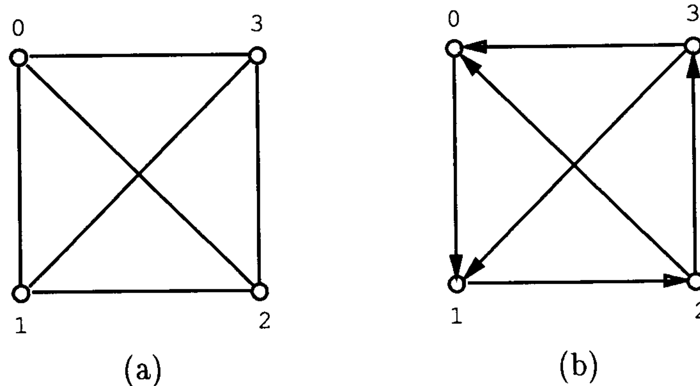


Figure 8.38: (a) Input graph. (b) A corresponding palm graph.

```
Input Graph:

0  3 2 1
1  3 2 0
2  3 1 0
3  2 1 0

Renaming of edges:

0 -> 0
1 -> 3
2 -> 2
3 -> 1

Renamed Directed Graph:

0  1
1  2
2  3 0
3  0 1

Graph is planar.
Embedding:

Segment: 0
  Spine: <0 1 2 3 0>
  Outside of cycle: <0 0>
Segment: 1
  Spine: <3 1>
  Outside of cycle: <0 1 2 3 0>
Segment: 2
  Spine: <2 0>
  Inside of cycle: <0 1 2 3 0>
```

Figure 8.39: Output of the planarity algorithm.

Bibliography

[AHU78] Alfred Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1978.

[AHU85] Alfred Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1985.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Tools and Techniques.* Addison-Wesley, Reading, MA, 1986.

[BO93] Ronald V. Book and Friedrich Otto. *String-Rewriting Systems.* Springer-Verlag, New York, N.Y., 1993.

[CU89] Wei Chen and Jan T. Udding. Towards a calculus of data refinement. In *Lecture Notes in Computer Science, No. 375: Mathematics of program construction.* Springer-Verlag, 1989.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.

[DJ90] N. Dershowitz and J. P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 6, pages 243–320. Elsevier Science Publishers, B.V., 1990.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[GJ79] Michael Garey and Steven Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, San Fransisco, CA, 1979.

[GP85] David Gries and Jan F. Prins. A new notion of encapsulation. In *Proceedings SIGPLAN 1985 Symposium on Language issues in Programming Environments*, volume 20 of *SIGPLAN Notices*, pages 131–139, July 1985.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, New York, N.Y., 1981.

[GV91] David Gries and Dennis Volpano. The transform—a new language construct. *Structured Programming*, 11:1–10, 1991.

[GV92] David Gries and Dennis Volpano. The Definition of Polya. Technical report, Department of Computer Science, Cornell University, 1992.

[GX88] David Gries and Jinyun Xue. The Hopcroft-Tarjan Planarity Algorithm, Presentations and Improvements. Technical report, Department of Computer Science, Cornell University, 1988.

[HT74] John E. Hopcroft and Robert E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, October 1974.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[Huf51] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings IRE Conference*, volume 20, pages 1098–1101, 1951.

[Knu63] Donald E. Knuth. *The Art of Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1963.

[KR87] Brian Kerninghan and Dennis Ritchie. *The C programming language, second edition.* Prentice Hall, Englewood Cliffs, N.J., 1987.

[MG90] Carol Morgan and Philip H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[Mil86] Robin Milner. *The Definition of ML.* Oxford Press, Oxford, England, 1986.

[Mor89] J. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[Par90] Helmut A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, New York, N.Y., 1990.

[Pri87] Jan F. Prins. *Partial Implementations in Program Derivations.* Ph.D. dissertation, Department of Computer Science, Cornell University, 1987.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[RT88] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-based Editors.* Springer-Verlag, New York, N.Y., 1988.

[Smi91] Geoffrey S. Smith. *Polymorhpic type inference for languages with overloading and subtyping.* Ph.D. dissertation, Department of Computer Science, Cornell University, 1991.

[Str91] Bjarne Stroustrup. *The C++ Programming Language, second edition.* Addison-Wesley, Reading, MA, 1991.