SCHEMA-BASED LOGIC PROGRAM TRANSFORMATION

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

by

Halime Büyükyıldız

August 1997

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ass't Prof. Pierre Flener (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ass't Prof. Nihan Kesim Çiçekli

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Ass't Prof. İlyas Çiçekli

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of Institute of Engineering and Science

ABSTRACT

SCHEMA-BASED LOGIC PROGRAM TRANSFORMATION

Halime Büyükyıldız
M.S. in Computer Engineering and Information Science
Supervisor: Ass't Prof. Pierre Flener
August 1997

In traditional programming methodology, developing a correct and efficient program is divided into two phases: in the first phase, called the synthesis phase, a correct, but maybe inefficient program is constructed, and in the second phase, called the transformation phase, the constructed program is transformed into a more efficient equivalent program. If the synthesis phase is guided by a schema that embodies the algorithm design knowledge abstracting the construction of a particular family of programs, then the transformation phase can also be done in a schema-guided fashion using transformation schemas, which encode the transformation techniques from input program schemas to output program schemas by defining the conditions that have to be verified to have a more efficient equivalent program.

Seven program schemas are proposed, which capture sub-families of divide-and-conquer programs and the programs that are constructed using some generalization methods. The proposed transformation schemas either automate transformation strategies, such as accumulator introduction and tupling generalization, which is a special case of structural generalization, or simulate and extend a basic theorem in functional programming (the first duality law of the fold operators) for logic programs. A prototype transformation system is presented that can transform programs, using the proposed transformation schemas.

Keywords: logic programming, program development, program transformation, program schema, transformation schema, generalization, duality laws.

ÖZET

TASLAĞA DAYALI MANTIK PROGRAMI DÖNÜŞTÜRME

Halime Büyükyıldız
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Pierre Flener
Ağustos 1997

Geleneksel programlama metodolojisinde, doğru ve etkili program geliştirme iki aşamaya ayrılır: birinci aşamada, sentez aşaması denir, doğru, fakat yeterince etkili olmayabilen bir program yapılır, ve ikinci aşamada, dönüştürme aşaması denir, yapılan program daha etkili eşdeğer bir programa dönüştürülür. Eğer sentez aşaması belirli bir program ailesinin yapımını özetleyebilen algoritma plan bilgisini içeren program taslağı rehberliğindeyse, dönüştürme aşaması da giren program taslağından çıkan program taslağına tanımlanmış dönüşüm tekniklerini daha etkili eşdeğer bir program elde etmeyi sağlayacak gerekli koşulları tanımlayarak kodlayan dönüşüm taslakları kullanarak yapılabilir.

Böl-ve-fethet ve genelleme metodlarını kullanarak sentezlenebilecek program ailelerini temsil eden yedi program taslağı sunuluyor. Sunulan dönüşüm taslakları ya içine birikeç sokmak ve yapısal genellemenin özel bir hali olan çoğullama genellemesi gibi dönüşüm tekniklerinin otomasyonunu sağlar, ya da fonksiyonel programlamanın temel teoremlerinden birini (fold operatörlerinin ilk ikilik kuralını) mantıksal programlamaya geliştirerek uygular. Sunulan dönüşüm taslaklarını kullanarak program dönüştürebilen prototip bir sistem geliştirilmiştir.

Anahtar Sözcükler: mantıksal programlama, program geliştirme, program dönüştürme, program taslağı, dönüşüm taslağı, genelleme, ikilik kuralları.

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Pierre Flener, due to his supervision, suggestions, and understanding throughout the development of this thesis.

I would like to thank Fergus Henderson, Mark Stickel, and Dan Sahlin, for their enourmous help in understanding and using Mercury, PTTP, and Mixtus. I would also like to thank the participants of the LOPSTR'97 workshop (especially Yves Deville, Andreas Hamfelt, and Norbert Fuchs) for their valuable comments and suggestions.

I am also indebted to Ass't Prof. Nihan Kesim Çiçekli and Ass't Prof. İlyas Çiçekli for showing keen interest to the subject matter and accepting to read and review this thesis.

I would like to thank Gülşen Demiröz, Gökmen Gök, Bilge Aydın, Bilge Say, my drama class friends, my other friends all over the world, and my family for their moral support and friendship.

I would also like to thank Bilkent University, which enabled this research environment and supported the presentation of this work at LOPSTR'97.

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

S_r	: Specification of the relation r
$\mathcal{I}_r$	: Input condition of the relation r
$\mathcal{O}_r$	: Output condition of the relation r
S	: Program schema (or schema pattern)
C	: Steadfastness constraints of a program schema
t	: Number of tails of the induction parameter
p	: Position of the head in the composition of the result parameter
e	: A special constant existing in program schema patterns for initializing the composition
LR	: Left-to-Right Composition
RL	: Right-to-Left Composition
A	: Applicability conditions of a transformation schema
O	: Post-optimizability conditions of a transformation schema
DC	: Divide-and-Conquer
TG	: Tupling Generalization
DG	: Descending Generalization
TDG	: Simultaneous Tupling-and-Descending Generalization

Chapter 1

Introduction

In traditional programming methodology, developing a correct and efficient program is divided into two phases: in the first phase, called the *synthesis phase*, a correct, but maybe inefficient program is constructed, and in the second phase, called the *transformation phase*, the constructed program is transformed into a more efficient equivalent program. However, it is better to divide logic program development into 5 steps like Deville did in [16], as in the figure below:

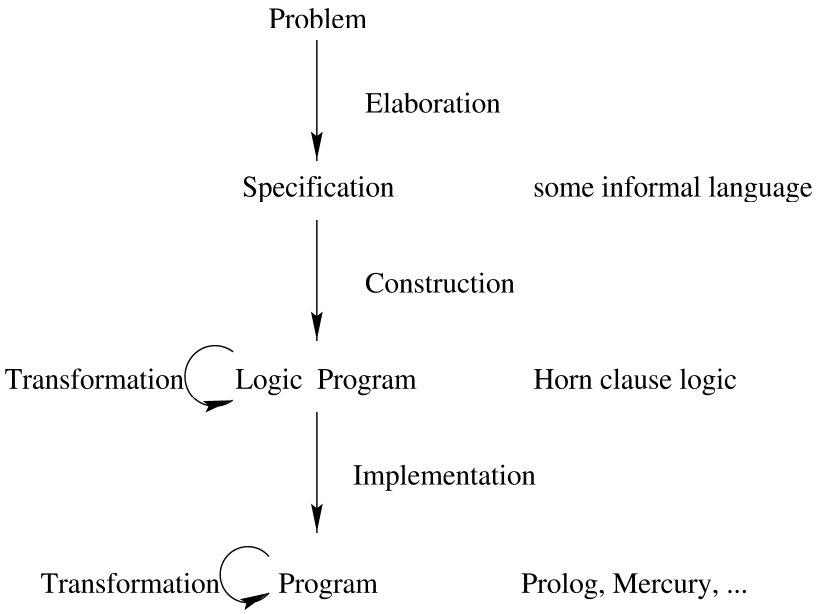

Figure 1.1. Program Development Methodology

The first step in Deville's program development methodology is the elaboration of a specification of the problem given, and this is the step that can't be (semi-)automated, and the step where most of the mistakes in program development occur. The second step is the construction of a logic program (logic description in [16]) from the specification of the problem. There is a considerable amount of work in literature that try to (semi-)automate this process, and they have shown improvements in this subject (refer to [18, 23, 22]). The third step is to derive a program from the logic program. This step deals with the computational and compiler-specific issues that make programming in a given language different from programming in logic. There are also some works in literature that automate this step (e.g. the Mercury compiler or abstract interpretation systems like Le Charlier's GAIA [35]). The two transformation steps in program development have the objective of increasing the efficiency of programs. Logic program transformation deals with logic, without any procedural aspects, and therefore it will be easier to carry out while preserving the correctness. However, transforming programs written in a logic programming language deals with the operational semantics of that language, and must have a suitable introduction of control. Deville proposed this methodology of program development, since it systematizes the logic programming adage "think logically first, then consider the procedural behaviour".

In this thesis, I only deal with the declarative semantics of programs in program transformation. The research results for the logic program transformation step of the above methodology are presented, where some well known methods like generalization and the duality laws in functional programming are used in a schema-guided way.

The objective of this research is to pre-compile the logic program transformation techniques that are proposed in the literature, after constructing most general definitions of the notions in the schema-based logic program transformation. I first examine the work done in the logic program transformation area so as to properly define the underlying theory of this research. The definitions are constructed by extending the proposed ideas and methods in the schema-based logic program transformation literature. These definitions and a summary of the related work in logic program transformation are presented

in Chapter 2. Generalization of the divide-and-conquer programs is worked out in this research. So, the program schemas, which abstract sub-families of divide-and-conquer programs are proposed next in Chapter 3.

I propose some generalization schemas that pre-compile the generalization methods proposed by Deville [16], namely tupling and descending generalization, in Chapter 4. One more category is added, namely simultaneous tupling-and-descending generalization, which can be thought of as a combination of the other two. The generalization schemas are more general than the generalization schemas that are proposed by Flener and Deville [20], in the sense that they deal with the transformation of more generic program families, by benefiting from the strength of the extended theory.

I propose some more transformation schemas in Chapter 5 that simulate and extend a basic theorem in functional programming (the first duality law of the fold operators) to logic programs. These schemas result from the ideas captured during the pre-compilation of generalization techniques. The similarity of this work with the work done in functional programming helps us to automate these transformations easily.

Although the transformation schemas proposed in this thesis only deal with the declarative semantics of programs, they are also evaluated by making performance tests on the input and output programs of these transformation schemas in a logic programming language setting. The performance tests of the input and output programs of these transformation schemas for some selected problems show that the post-optimizability conditions have a key role in ensuring an efficiency gain. The results of these performance tests and a detailed discussion are thereof given in Chapter 6.

Using the results of the theoretical part of this research and the evaluation of the transformation schemas, a prototype transformation system is developed, which is the main practical objective of this research. This system is explained in detail in Chapter 7. It is shown that our transformation schemas can really be used in a real practical transformation setting.

There exist a lot of future work directions of this research, since the constructed theory is new and seems to be powerful enough to pre-compile some more transformation techniques like loop merging. Some extensions in the theory will also help to extend the prototype system so as to become a complete transformation system that can be integrated into a schema-based logic program development environment. The contributions of this research and the future work directions are summarized in Chapter 8.

Chapter 2

Basic Concepts

In this chapter, the most general definitions of the notions that are used throughout this thesis are presented (Section 2.1), then the related work done in logic program transformation is summarized (Section 2.2).

2.1 Terminology

I first define the notions; program and specification in Section 2.1.1. Next, the correctness and equivalence criteria of programs are presented in Section 2.1.2. The general definitions of the notions in program transformation are given in Section 2.1.3. Program schemas and the related notions are defined in Section 2.1.4. I present the definition of a transformation schema in Section 2.1.5. Finally, problem generalization methods, which are used in this thesis, are discussed in Section 2.1.6.

2.1.1 Programs and Specifications

Definition 1 An *atom* is a first-order formula of the form $r(t_1, \ldots, t_n)$, where r is a relation symbol of arity n, and $t_1, \ldots, t_n$ are terms constructed out of variables, constants, and function symbols.

Example 1 $p([HL|TL], R, [HL|TS])$ and $q([\,], 0)$ are atoms.

Definition 2 A *typed definite clause* is a formula of the form:

$$\forall X_1 : \mathcal{X}_1, \ldots, X_n : \mathcal{X}_n \; r(X_1, \ldots, X_n) \leftarrow \mathcal{B}[X_1, \ldots, X_n]$$

where $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are the sorts (or: types) of $X_1, \ldots, X_n$, respectively, atom $r(X_1, \ldots, X_n)$ is called the *head* of the clause, and $\mathcal{B}[X_1, \ldots, X_n]$ is called the *body* of the clause, which is a (possibly empty) conjunction of formulas, which are either atoms or disjunctions.

Example 2 The formula below is a typed definite clause:

$$\begin{aligned} \forall L : list(int), \forall S : int.\ sum(L, S) \leftarrow \quad & L = [HL|TL], sum(TL, TS), \\ & S \; is \; HL + TS \end{aligned}$$

Definition 3 A *typed definite logic procedure* is a finite set of typed definite clauses whose heads have the same relation symbol with the same arity.

Example 3 Below is a typed definite logic procedure:

$$\begin{aligned} \forall L : list(int), \forall S : int.\ sum(L, S) \leftarrow \quad & L = [\,], S = 0 \\ \forall L : list(int), \forall S : int.\ sum(L, S) \leftarrow \quad & L = [HL|TL], sum(TL, TS), \\ & S \; is \; HL + TS \end{aligned}$$

Definition 4 A *typed definite logic program* is the union of a set of typed definite procedures.

Example 4 Below is a typed definite logic program:

$$\begin{aligned} \forall A : int, \forall B : int. \quad int\_eqq(A, B) \leftarrow \quad & A = B \\ \forall A : int, \forall B : int, \forall C : int. \quad add(A, B, C) \leftarrow \quad & A \; is \; B + C \\ \forall L : list(int), \forall E : int. \quad mem(L, E) \leftarrow \quad & L = [HL|TL], int\_eqq(HL, E) \\ \forall L : list(int), \forall E : int. \quad mem(L, E) \leftarrow \quad & L = [HL|TL], mem(TL, E) \end{aligned}$$

Throughout the thesis, the word program (respectively, procedure and clause) is used to mean typed definite logic program (respectively, procedure and clause), and I drop the quantifications wherever they are either irrelevant or known in context.

Definition 5 A non-primitive relation that appears in the clause bodies of a program, but does not appear in any heads of the clauses of that program is called an *undefined* (or *open*) relation, otherwise it is called a *defined* relation.

Definition 6 An *open program* is a program where some of the relations are undefined. If all the relations in the program are defined, then the program is called a *closed program.*

Example 5 The program below is an open program:

$$\begin{aligned} sort(L,S) \leftarrow\ & L = [\,], S = [\,] \\ sort(L,S) \leftarrow\ & L = [HL|TL], sort(TL,TS), insert(HL,TS,S) \end{aligned}$$

since the relation $insert/3$ is undefined in the program. If we construct a new program by taking the set union of the program above and the program below:

$$\begin{aligned} insert(E,L,R) \leftarrow\ & L = [\,], R = [E] \\ insert(E,L,R) \leftarrow\ & L = [HL|TL], HL \geq E, R = [E|L] \\ insert(E,L,R) \leftarrow\ & L = [HL|TL], HL < E, \\ & insert(E,TL,TR), R = [HL|TR] \end{aligned}$$

then the new program is a closed program, assuming $= /2$, $\geq /2$, and $< /2$ are primitives.

Definition 7 A clause is said to be *recursive* iff its head relation also occurs in an atom of its body. A program is said to be *recursive* iff one or more of its clauses is recursive.

Definition 8 [41] A program is *tail recursive* iff it has one and only one recursive subgoal and its last clause has the form

$$r(t) \leftarrow L, r(u)$$

where L is deterministic. When the last clause of a program has this form but the program has more than one recursive subgoal, the procedure is said to be *semi-tail recursive.*

Definition 9 A *formal specification* of a program for a relation r of arity 2 is a first-order formula written in the format:

$$\forall X : \mathcal{X}.\ \forall Y : \mathcal{Y}.\ \mathcal{I}_r(X) \Rightarrow [r(X,Y) \Leftrightarrow \mathcal{O}_r(X,Y)]$$

where $\mathcal{X}$ and $\mathcal{Y}$ are the sorts (or types) of X and Y, respectively, $\mathcal{I}_r(X)$ denotes the *input condition* that must be fulfilled before the execution of the program, and $\mathcal{O}_r(X,Y)$ denotes the *output condition* that will be fulfilled after the execution.

Example 6 Below is the formal specification of any program for the problem of sorting an integer list:

$$\forall L : list(int).\ \forall S : list(int).\ true \Rightarrow [sort(L,S) \Leftrightarrow$$

$$permutation(L,S) \wedge ordered(S)]$$

where L and S are integer-lists, the input condition of $sort(L,S)$ is $true$, and the output condition of $sort(L,S)$ is the conjunction $permutation(L,S) \wedge ordered(S)$.

I give the definition of the formal specification of a relation r of arity 2 for pedagogical reasons, the definition can be generalized to relations of arity n. Also, for some of the problems worked out in this thesis, sometimes I give *informal specifications*, which are rewritings of the formal specifications in a "natural" language.

2.1.2 Correctness and Equivalence Criteria

In this section, I give correctness and equivalence criteria by using the notion of framework [21]. Throughout the section, when I write "a relation r", it means

"a relation r of arity 2", but these definitions can be generalized for relations of arity n. In the definitions below, I do not consider mutually recursive programs. However, these definitions can be reconstructed for mutually recursive programs as well.

Definition 10 (Frameworks [21])
A *framework* $\mathcal{F}$ is a full first-order logical theory (with identity) with an intended model. An *open framework* consists of:

* a (many-sorted) signature of
  - both *defined* and *open* sort names;
  - function declarations, for declaring both *defined* and *open* constant and function names;
  - relation declarations, for declaring both *defined* and *open* relation names;
* a set of first-order axioms each for the (declared) *defined* and *open* function and relation names, the former possibly containing induction schemas;
* a set of theorems.

Thus, an open framework $\mathcal{F}$ is also denoted as $\mathcal{F}(\Pi)$, where Π are the open names, or parameters, of $\mathcal{F}$. The definition of a *closed framework* is the same as the definition of an open framework, except that a closed framework has no open names. Therefore, a closed framework is just an extreme case of an open one, namely where Π is empty.

The definitions of correctness of a logic program and equivalence of two programs are given only for programs in closed frameworks.

Example 7 (Closed Frameworks) A typical closed framework is (first-order) Peano arithmetic $\mathcal{NAT}$ [21]: [1]

[1] The most external universal quantifiers will be omitted.

Framework $\mathcal{NAT}$;

SORTS: Nat;

FUNCTIONS:
$$0 \;:\; \rightarrow Nat;$$
$$s \;:\; Nat \rightarrow Nat;$$
$$+, * \;:\; (Nat, Nat) \rightarrow Nat;$$

AXIOMS:
$$\neg 0 = s(x) \wedge s(a) = s(b) \rightarrow a = b;$$
$$x + 0 = x;$$
$$x + s(y) = s(x + y);$$
$$x * 0 = 0;$$
$$x * s(y) = x + x * y;$$
$$H(0) \wedge (\forall i.H(i) \rightarrow H(s(i))) \rightarrow \forall x.H(x).$$

This framework defines the abstract data type $\mathcal{NAT}$ as follows: the sort Nat of natural numbers is constructed *freely* from the constructors 0 (*zero*) and s (*successor*); the *freeness axiom* for these constructors is the first axiom; the functions $+$ (*sum*) and $*$ (*product*) on Nat are axiomatized by the next four axioms (in a primitive recursive manner). Note in particular that the last axiom in $\mathcal{NAT}$ can be used for reasoning about properties of $+$ and $*$ that can't be derived from the other axioms, e.g. associativity and commutativity. This illustrates the fact that in a framework we may have more than just an abstract data type definition.

Definition 11 (Correctness of a Closed Program)
Let P be a closed program for relation r in a closed framework $\mathcal{F}$. We say that P is (*totally*) *correct* wrt its specification S_r iff, for any ground term t of $\mathcal{X}$ such that $\mathcal{I}_r(t)$ holds, the following condition holds: $P \vdash r(t, u)$ iff $\mathcal{F} \models \mathcal{O}_r(t, u)$, for every ground term u of $\mathcal{Y}$.

If we replace 'iff' by 'implies' in the condition above, then P is said to be *partially correct* wrt S_r, and if we replace 'iff' by 'if', then P is said to be *complete* wrt S_r.

This kind of correctness is not entirely satisfactory, for two reasons. First, it defines the correctness of P in terms of the procedures for the relations

in its clause bodies, rather than in terms of their specifications. Second, P must be a closed program, even though it might be desirable to discuss the correctness of P without having to fully implement it. So, the abstraction achieved through the introduction (and specification) of the relations in its clause bodies is wasted. This leads us to the notion of steadfastness (also known as parametric correctness) [21] (also see [16]).

Definition 12 (Steadfastness of an Open Program)
In a closed framework $\mathcal{F}$, let:

- P be an open program for relation r
- $q_1, \ldots, q_m$ be all the undefined relation names appearing in P
- $S_1, \ldots, S_m$ be the specifications of $q_1, \ldots, q_m$.

We say that P is *steadfast* wrt its specification S_r in $\{S_1, \ldots, S_m\}$ iff the (closed) program $P \cup P_S$ is correct wrt S_r, where P_S is any closed program such that

- P_S is correct wrt each specification S_j $(1 \leq j \leq m)$
- P_S contains no occurrences of the relations defined in P.

Let's illustrate with an example the reason why we can't rephrase the last sentence above as:

> We say that P is *steadfast* wrt its specification S_r in $\{S_1, \ldots, S_m\}$ iff, for any closed programs $P_1, \ldots, P_m$ that are correct wrt $S_1, \ldots, S_m$, respectively, and that contain the open programs for $q_1, \ldots, q_m$, we have that the (closed) program $P \cup P_1 \cup \ldots \cup P_m$ is correct wrt S_r.

Example 8 I use propositional logic, since it helps to understand the example easily. Let the open program P be:

$$r \leftarrow p, q$$

To show the steadfastness of P, suppose we choose the closed program P_p as

$$p \leftarrow t, s$$

$$t \leftarrow s$$

$$s \leftarrow u$$

where u is a primitive, and P_p is correct wrt S_p. Also suppose we choose the closed program P_q as

$$q \leftarrow t$$

$$t \leftarrow v$$

where v is a primitive, and P_q is correct wrt S_q. To say that P is steadfast wrt S_r in $\{S_p, S_q\}$, the (closed) program $P \cup P_p \cup P_q$ would have to be correct wrt S_r. But note that the set union $P \cup P_p \cup P_q$ has two different programs for proposition t, which makes the regular set union inapplicable in this context. □

The steadfastness definition yields the following interesting property, which is actually a high-level recursive algorithm to check the steadfastness of an open program.

Property 1 In a closed framework $\mathcal{F}$, let:

- P be an open program for relation r of the specification S_r
- $p_1, \ldots, p_t$ be all the defined relation names appearing in P (including r thus)
- $q_1, \ldots, q_m$ be all the undefined relation names appearing in P
- $S_1, \ldots, S_m$ be the specifications of $q_1, \ldots, q_m$.

For $t \geq 2$, the program P is steadfast wrt S_r in $\{S_1, \ldots, S_m\}$ iff every P_i $(1 \leq i \leq t)$ is steadfast wrt the specification of p_i in the set of the specifications of all

undefined relations in P_i, where P_i is a program for p_i, such that $P = \bigcup_{i=1}^{t} P_i$. When $t = 1$, the definition of steadfastness is directly used, since the only defined relation is the relation r. Thus, $t = 1$ is the stopping case of this recursive algorithm.

Example 9 I use propositional logic, since it helps to understand the example easily. In a closed framework $\mathcal{F}$, let the open program P be:

$$r \leftarrow p, w$$

$$p \leftarrow q$$

To show the steadfastness of P, suppose we choose the closed program P_S as

$$q \leftarrow t$$

$$w \leftarrow v$$

where t and v are primitives in $\mathcal{F}$, and P_S is correct wrt S_w and S_q. By Definition 12, P is steadfast wrt S_r in $\{S_w, S_q\}$ iff the closed program $P \cup P_S$ is correct wrt S_r in $\mathcal{F}$. By Definition 11, $P \cup P_S$ is correct wrt S_r in $\mathcal{F}$ iff the following condition holds:

$$\{r \leftarrow p, w, p \leftarrow q, q \leftarrow t, w \leftarrow v\} \vdash r \text{ iff } \mathcal{F} \models \mathcal{O}_r$$

By resolution:

$$\{p \leftarrow q, q \leftarrow t, w \leftarrow v\} \vdash p, w \text{ iff } \mathcal{F} \models \mathcal{O}_p \wedge \mathcal{O}_w$$

The formula above can be written as:

$$(\{p \leftarrow q, q \leftarrow t\} \vdash p \text{ iff } \mathcal{F} \models \mathcal{O}_p) \wedge (\{w \leftarrow v\} \vdash w \text{ iff } \mathcal{F} \models \mathcal{O}_w)$$

The second part of the conjunction is *true*, since P_S is correct wrt S_w and $\{w \leftarrow v\}$ is the program of w in P_S.

By Definitions 11 and 12, the first part of the conjunction means that the program P_p below

$$p \leftarrow q$$

is steadfast wrt S_p in $\{S_q\}$ iff the closed program $P_p \cup P_q$ is correct wrt S_p, where P_q is

$$q \leftarrow t$$

and it is correct wrt S_q.

If we use the property of steadfastness, for $t = 2$, the program P is steadfast wrt S_r in $\{S_w, S_q\}$, iff P_p is steadfast wrt S_p in $\{S_q\}$. After we prove the steadfastness of P_p, t reduces to 1 and we directly use Definition 12 for proving the steadfastness of P_r wrt S_r in $\{S_p, S_w\}$ where $P = P_p \cup P_r$. The algorithm summarizes what we did bottom up in this example for proving steadfastness of P wrt S_r in $\{S_w, S_q\}$.

Thus, Property 1 proposes an efficient algorithm to prove the steadfastness of an open program. □

For program equivalence, we do not require the two programs to have the same models, because this would not make much sense in some program transformation settings, where the transformed program features relations that were not in the initially given program. That is why our program equivalence criterion establishes equivalence wrt the specification of a common relation (usually the root of their call-hierarchies).

Definition 13 (Equivalence of Two Open Programs)
In a closed framework $\mathcal{F}$, let P and Q be two open programs for a relation r. We say that P is *equivalent to* Q wrt the specification S_r iff the following two conditions hold:

(*a*) P is steadfast wrt S_r in $\{S_1, \ldots, S_m\}$, where $S_1, \ldots, S_m$ are the specifications of $p_1, \ldots, p_m$, which are all the undefined relation names appearing in P

(*b*) Q is steadfast wrt S_r in $\{S'_1, \ldots, S'_t\}$, where $S'_1, \ldots, S'_t$ are the specifications of $q_1, \ldots, q_t$, which are all the undefined relation names appearing in Q.

Since the 'is equivalent to' relation is symmetric, we also say that P and Q are *equivalent* wrt S_r.

Sometimes, in program transformation settings, there exist some conditions that have to be verified related to some parts of the initial and/or transformed program in order to have a transformed program that is equivalent to the initially given program wrt the specification of the top-level relation. Hence the following definition.

Definition 14 (Conditional Equivalence of Two Open Programs) In a closed framework $\mathcal{F}$, let P and Q be two open programs for a relation r. We say that P is *equivalent to* Q wrt the specification S_r *under* conditions C iff P is *equivalent to* Q wrt S_r provided that C hold.

2.1.3 Transformation

In this section, I give the definitions of the following concepts: program transformation, transformation techniques, transformation strategies, and transformation rules.

Definition 15 A *program transformation* is the replacement of a subset of the clauses of a program with another clause set such that the resulting program is equivalent to the initial program wrt the specification of the top-level relation.

Definition 16 A *transformation rule* is a rule that takes an input program and produces another program, which is equivalent to the input program wrt the specification of the top-level relation.

Example 10 An example transformation rule is replacing the clause of a program that has the conjunction $H = [], append(H, T, R)$ in its body, with a clause that is the same as the previous one, except that it has the literal $R = T$ in place of that conjunction. □

A program transformation process starting from a given initial program P_0 can also be viewed as a sequence of programs $P_0, \ldots, P_n$, called *transformation sequence*, such that program P_{k+1}, with $0 \leq k < n$, is obtained from P_k by the application of a transformation rule, which may depend on $P_0, \ldots, P_k$. However, the problem is that an efficiency improvement is not ensured by an undisciplined application of transformation rules one after another. So a better approach is using a transformation strategy.

Definition 17 A *transformation strategy* is some form of a meta-rule that takes an input program and produces another program, which is equivalent to the first one wrt a given semantics, by applying a suitable sequence of transformation rules.

Example 11 The *loop merging* strategy transforms the "naive" program

$$p(L, R) \leftarrow \quad \ldots, sum(L, S), length(L, N), \ldots$$

into the optimized program

$$p(L, R) \leftarrow \quad \ldots, sumLength(L, S, N), \ldots$$

and generates a new program for $sumLength$ from those for sum and $length$. □

Definition 18 A *transformation technique* improves program efficiency by using a combination of transformation strategies.

Efficiency improvement is the main objective of transformation techniques.

In the remaining part of this section, I present four basic transformation rules, namely *unfolding*, *folding*, *definition introduction*, and *goal replacement* for definite programs. The definitions below are similar to the definitions in [41], but they are adapted to our terminology. The reader may refer to [41] for more transformation rules, the variations of the transformation rules below for different semantics, and their relevant properties.

Definition 19 (Unfolding) Let P_k be the program $\{E_1, \ldots, E_r, C, E_{r+1}, \ldots, E_s\}$ where E_i $(1 \leq i \leq s)$ is a clause, and let C be the clause $H \leftarrow F, A, G$, where A is an atom and F and G are conjunctions of atoms. Suppose that:

(1) $\{D_1, \ldots, D_n\}$, with $n > 0$, is the subset of all clauses in a program P_j, with $0 \leq j \leq k$, such that A is unifiable with $head(D_1), \ldots, head(D_n)$, with most general unifiers $\theta_1, \ldots, \theta_n$, respectively, and

(2) C_i is the clause $(H \leftarrow F, body(D_i), G)\theta_i$, for $i = 1, \ldots, n$.

If we *unfold* C *wrt* A *using* $D_1, \ldots, D_n$ *in* P_j, we derive the clauses $C_1, \ldots, C_n$ and we get the new program $P_{k+1} = \{E_1 \ldots, E_r, C_1, \ldots, C_n, E_{r+1}, \ldots, E_s\}$. A simpler terminology, like "to unfold C wrt A using P_j", can also be used.

Example 12 Let $C = p(X) \leftarrow q(t(X)), s(X)$ be a clause in P_k and let the definition of q in P_j, with $0 \leq j \leq k$, consist of the following clauses:

$$\begin{aligned} q(a) &\leftarrow \\ q(t(b)) &\leftarrow \\ q(t(a)) &\leftarrow \quad r(a) \end{aligned}$$

Then, by unfolding C wrt $q(t(X))$ using P_j, the following clauses are derived:

$$\begin{aligned} p(b) &\leftarrow \quad s(b) \\ p(a) &\leftarrow \quad r(a), s(a) \end{aligned}$$

Thus P_{k+1} is obtained by replacing the subset $\{C\}$ in P_k by the set of derived clauses above. □

Definition 20 (Folding) Let P_k be the program $\{E_1, \ldots, E_r, C_1, \ldots, C_n, E_{r+1}, \ldots, E_s\}$ and let $\{D_1, \ldots, D_n\}$ be a subset of clauses in a program P_j, with $0 \leq j \leq k$. Suppose that there exists an atom A such that, for $i = 0, \ldots, n$:

(1) $head(D_j)$ is unifiable with A via a most general unifier θ_i,

(2) C_i is the clause $(H \leftarrow F, body(D_j), G)\theta_i$, where F and G are conjunctions of atoms, and

(3) for any clause D of P_j not in the subset $\{D_1, \ldots, D_n\}$, $head(D)$ is not unifiable with A.

If we *fold* $C_1, \ldots, C_n$ *using* $\{D_1, \ldots, D_n\}$ *in* P_j, we derive the clause $H \leftarrow F, A, G$, call it C, and the new program is $P_{k+1} = \{E_1, \ldots, E_r, C, E_{r+1}, \ldots, E_s\}$.

The folding rule is the inverse of the unfolding rule, in the sense that given a transformation sequence $P_0, \ldots, P_k, P_{k+1}$, where P_{k+1} has been obtained from P_k by unfolding, there exists a transformation sequence $P_0, \ldots, P_k, P_{k+1}, P_k$, where (the last occurrence of) P_k has been obtained from P_{k+1} by folding.

Example 13 The clauses

$$C_1: \quad p(t(X)) \leftarrow q(X), r(X)$$
$$C_2: \quad p(u(X)) \leftarrow s(X), r(X)$$

can be folded using

$$D_1: \quad a(X, t(X)) \leftarrow q(X)$$
$$D_2: \quad a(X, u(X)) \leftarrow s(X)$$

thereby deriving

$$C: \quad p(Y) \leftarrow a(X, Y), r(X)$$

Notice that by unfolding clause C using $\{D_1, D_2\}$, we get again $\{C_1, C_2\}$. □

Definition 21 (Definition Introduction) Let P_k be the program $\{E_1, \ldots, E_n\}$, a new program P_{k+1} can be obtained by the set union of P_k and P_r where P_r is a program for relation r such that r does not occur in $P_0, \ldots, P_k$.

Example 14 Let P_k be the program:

$$p \leftarrow \quad q$$
$$p \leftarrow \quad fail$$
$$q \leftarrow$$

By definition introduction, P_{k+1} will be the program:

$$\begin{aligned} p &\leftarrow q \\ p &\leftarrow fail \\ q &\leftarrow \\ newp &\leftarrow q \end{aligned}$$

iff $newp$ does not occur in $P_0, \ldots, P_k$. □

Definition 22 (Goal Replacement) A *replacement law* is a pair $S \equiv T$, where S and T are conjunctions of atoms. Let $\{X_1, \ldots, X_n\}$ be the set containing the variables both in S and in T (i.e., $vars(T) \cap vars(S)$), and let us consider the following two clauses:

$$\begin{aligned} C_S: &\ p(X_1, \ldots, X_n) \leftarrow S \\ C_T: &\ p(X_1, \ldots, X_n) \leftarrow T \end{aligned}$$

where p is any new relation name. We say that $S \equiv T$ is *valid* wrt the program P_k iff the program $P_k \cup C_S$ is equivalent to the program $P_k \cup C_T$ wrt the specification of the top-level relation. Let

$$C: \ H \leftarrow F, S, G$$

be a clause in P_k such that:

1. $S \equiv T$ is a valid replacement law wrt P_k, and
2. $vars(H, F, G) \cap vars(S) = vars(H, F, G) \cap vars(T) = \{X_1, \ldots, X_n\}$.

By *replacement of* S *in* C *using* $S \equiv T$ we derive the clause

$$R: \ H \leftarrow F, T, G$$

and we get P_{k+1} by replacing C by R in P_k.

Example 15 (Goal Replacement [41]) Let P_k be the program below:

$$\begin{aligned} C_1: &\ sublist(N, X, Y) \leftarrow length(X, N), append(V, X, W), append(W, Z, Y) \\ C_2: &\ append(L, R, Z) \leftarrow L = [\,], Z = R \\ C_3: &\ append(L, R, Z) \leftarrow L = [HL|TL], append(TL, R, TZ), Z = [HL|TZ] \end{aligned}$$

The replacement law

$$append(V, X, W), append(W, Z, Y) \equiv append(X, L, M), append(K, M, Y)$$

(which expresses a weak form of associativity of *append*) is valid wrt P_k. Indeed, if we consider the clauses:

$$C_S: \quad p(X, Y) \leftarrow append(V, X, W), append(W, Z, Y)$$
$$C_T: \quad p(X, Y) \leftarrow append(X, L, M), append(K, M, Y)$$

we have that $P_k \cup C_S$ is equivalent to the program $P_k \cup C_T$ wrt the specification of the top-level relation. Thus by goal replacement of

$$append(V, X, W), append(W, Z, Y)$$

in C_1, we derive the clause:

$$C_1': \quad sublist(N, X, Y) \leftarrow length(X, N), append(X, L, M), append(K, M, Y)$$

□

In [9], I use the transformation rules unfolding and folding for proving the equivalence of the input and output programs of the transformations explained in the remaining chapters of this thesis. The definition introduction and goal replacement rules are used to define the transformation strategies that were proposed in the literature, as we will see in Section 2.2.1.

2.1.4 Program Schemas and Schema Patterns

I gave the definition of a program in Section 2.1.1, now I will give the definitions of a program schema and a program schema pattern.

Definition 23 In a closed framework $\mathcal{F}$, a *program schema* for a relation r is a pair $\langle T, C \rangle$, where T is an open program for r, called the *template*, and C is a set of specifications of the open relations of T in terms of each other and the input/output conditions of the closed relations of T. The specifications in C, called the *steadfastness constraints*, are such that, in $\mathcal{F}$, T is steadfast wrt its specification S_r in C.

Example 16 Let GT be the generate_and_test program schema for relation r of arity 2, then GT contains the template program:

$$\forall X : \mathcal{X}.\forall Y : \mathcal{Y}. \quad r(X,Y) \leftarrow \quad generator(X,Y), tester(Y)$$

Note that most programs can be classified as GT programs according to the template above, if no semantic constraints on the open relations are given. Informally, the semantics (i.e. meaning) of the template above is that, for a given input X of type $\mathcal{X}$, the relation *generator* generates a possible output Y of type $\mathcal{Y}$ until Y satisfies the condition specified by the relation *tester*. So the steadfastness constraints of GT are:

$$\mathcal{I}_r(X) \Rightarrow [generator(X,Y) \Leftrightarrow \mathcal{O}_g(X,Y)]$$

$$\mathcal{O}_g(X,Y) \Rightarrow [tester(Y) \Leftrightarrow \mathcal{O}_r(X,Y)]$$

where $\mathcal{I}_r(X)$ is the input condition of the relation r, and $\mathcal{O}_r(X,Y)$ (respectively, $\mathcal{O}_g(X,Y)$) is the output condition of the relation r (respectively, *generator*). □

Definition 24 In a closed framework $\mathcal{F}$, a program P for a relation r is an *instance* of program schema $S = \langle T, C \rangle$ for a relation r if it has the form $T \cup E$, where E is a closed program defining all the open relations in T, such that E is totally correct wrt each specification in C (i.e., such that P is totally correct wrt its specification S_r).

Example 17 For instance, the closed program

$$\begin{aligned} r(X,Y) &\leftarrow \quad generator(X,Y), tester(Y) \\ generator(X,Y) &\leftarrow \quad perm(X,Y) \\ tester(Y) &\leftarrow \quad ordered(Y) \end{aligned}$$

is an instance of the generate-and-test GT schema in the list framework, assuming that *perm* and *ordered* are primitives. □

Sometimes, a series of schemas are quite similar, in the sense that they only differ in the number of arguments of some relations, or in the number of

calls to some relations, etc. For instance, one may want to write a GT schema for relations having n result arguments. For this purpose, rather than having a proliferation of similar schemas, I introduce the notions of *schema pattern* (compare with [10]) and *particularization.*

Definition 25 A *schema pattern* is a schema where term, conjunct, and disjunct ellipses are allowed in the template and in the steadfastness constraints.

I do not formally define the ellipsis notation here, assuming that their semantics is quite straightforward. For instance, $TX_1, \ldots, TX_t$ is a term ellipsis, and $\bigwedge_{i=1}^{t} r(TX_i, TY_i)$ is a conjunct ellipsis.

Example 18 The following is the template of a GT schema pattern, called GTP:

$$\forall X : \mathcal{X}. \forall Y_1, \ldots, Y_n : \mathcal{Y}. \quad r(X, Y_1, \ldots, Y_n) \leftarrow \begin{array}{l} generator_1(X, Y_1), tester_1(Y_1), \\ \ldots, \\ generator_n(X, Y_n), tester_n(Y_n) \end{array}$$

□

Definition 26 A *particularization* of a schema pattern is a schema obtained by eliminating the ellipses, i.e., by binding the (mathematical) variables denoting their lower and upper bounds to natural numbers.

Example 19 The schema GT is the particularization of GTP for $n = 1$ (assuming that indexes are dropped when ellipses reduce to singletons). □

2.1.5 Transformation Schemas

In Section 2.1.3, I gave the definitions of a program transformation and a transformation technique. Now, it is time to give the definition of a transformation schema that is the counterpart of the transformation techniques in the strategy-based approach.

Definition 27 A *transformation schema* encoding a transformation technique is a 5-tuple $\langle S_1, S_2, A, O_{12}, O_{21} \rangle$, where S_1 and S_2 are program schemas (or schema patterns), A is a set of *applicability conditions*, which ensure the equivalence of the templates of S_1 and S_2 wrt the specification of the top-level relation, and O_{12} (respectively, O_{21}) is a set of *optimizability conditions*, which ensure the optimizability of the output program schema (or schema pattern) S_2 (respectively, S_1).

The reader may find the example below too easy and providing not much efficiency gain as a transformation and little generic as a transformation schema, but I give this example so that the reader will have an intuitive understanding of the notion. Many realistic examples of transformation schemas will be found in the remaining chapters.

Example 20 Let TS be the example transformation schema that is a 5-tuple $\langle S_1, S_2, A, O_{12}, O_{21} \rangle$, where S_1 has the template:

$$r(X, Y) \leftarrow id(E), Z = [E], comp_1(Z, X, Y)$$

and the steadfastness constraints of S_1 are the specifications of the relations r, id, and $comp_1$. Then, S_2 has the template:

$$r(X, Y) \leftarrow id(E), Z = [E], comp_2(X, Z, Y)$$

with a subset of the steadfastness constraints of S_1 that are the specifications of relations r, id, and $comp_2$.

The set A of the applicability conditions of TS contains the formula:

$$\mathcal{O}_{c1}(Z, X, Y) \Leftrightarrow \mathcal{O}_{c2}(X, Z, Y)$$

where $\mathcal{O}_{c1}$ and $\mathcal{O}_{c2}$ are the output conditions of $comp_1$ and $comp_2$.

O_{12}, which is the set of the optimizability conditions of S_2 in TS, is the set containing the formula:

$$Z = [E] \Rightarrow [\mathcal{O}_{c1}(Z, X, Y) \Leftrightarrow Y = [E|X]]$$

and O_{21}, which is the set of the optimizability conditions of S_1 in TS, is the set containing the formula:

$$Z = [E] \Rightarrow [\mathcal{O}_{c2}(X, Z, Y) \Leftrightarrow Y = [E|X]]$$

assuming that the two schemas are defined in the list framework. □

Definition 28 A transformation schema $\langle S_1, S_2, A, O_{12}, O_{21} \rangle$ is *correct* iff the templates of program schemas (or schema patterns) S_1 and S_2 are equivalent wrt the specification of the top-level relation under the applicability conditions A.

In program transformation, for proving the correctness of a transformation schema $\langle S_1, S_2, A, O_{12}, O_{21} \rangle$, I have to prove the conditional equivalence of T_1 and T_2, which are the templates of $S_1 = \langle T_1, C_1 \rangle$ and $S_2 = \langle T_2, C_2 \rangle$. I assume that the template T_i of the input program schema $S_i = \langle T_i, C_i \rangle$ (where $i = 1, 2$) is steadfast wrt the specification of the top-level relation, say S_r, in C_i, then the correctness of the transformation schema is proven by establishing the steadfastness of the template T_j of the output program schema (or schema pattern) $S_j = \langle T_j, C_j \rangle$ (where $j = 1, 2$ and $j \neq i$) wrt S_r in C_j using the applicability conditions A.

At the program-level, the transformation of a given closed program P for a relation r into a new closed program Q for r then reduces to:

(1) *selection* of an applicable transformation schema $\langle S_1, S_2, A, O_{12}, O_{21} \rangle$, where $S_1 = \langle T_1, C_1 \rangle$ and $S_2 = \langle T_2, C_2 \rangle$ such that P is an instance of S_1 (i.e., $P = T_1 \cup E$), or an instance of S_2 (i.e., $P = T_2 \cup E$);

(2) *verification* of the applicability of the transformation schema by verification of whether E satisfies the conditions A, in the considered closed framework $\mathcal{F}$, i.e., whether $E \vdash_{\mathcal{F}} A$;

(3) *verification* of the efficiency gain by the transformation schema by verification of whether E satisfies the conditions O_{12}, or O_{21}, in the considered closed framework $\mathcal{F}$, i.e., whether $E \vdash_{\mathcal{F}} O_{12}$, or $E \vdash_{\mathcal{F}} O_{21}$;

(4) *computation* of Q as an instance of S_2, or S_1, i.e., $Q = T_2 \cup E$, or $Q = T_1 \cup E$;

(5) *optimization* of Q.

If schema-guided synthesis of P was performed (e.g., if P is a-priori known to be a particularization of S_1), then Q can be obtained automatically, namely Q will be the corresponding particularization of S_2.

2.1.6 Problem Generalization

Not only in mathematics, but also in many fields of computer science, such as machine learning, theorem proving, and so on, generalization techniques are used to ease the process of solving a problem. Here generalization is used to transform a possibly inefficient program into a more efficient one, because the generalization process may provoke a complexity reduction by loop merging and because the output program may be (semi-)tail-recursive (which can be further transformed into an iterative program by an optimizing interpreter). The problem generalization techniques that are used in this thesis are explained in detail in [16], and using these techniques for synthesizing and/or transforming a program in a schema-guided fashion was first proposed in [16, 17], and then extended in [20].

Given a program, the generalization process works as follows: first the specification of the initial program is generalized, then a recursive program for the generalized specification is synthesized, and finally a non-recursive program for the initial problem can be written, since the initial problem is a particular case of the generalized one. The two generalization approaches used here are:

1. *Structural generalization*: The intended relation is generalized by generalizing the structure (or: type) of a parameter. If a problem dealing with a term is generalized to a problem dealing with a *list* of terms, then this generalization is called *tupling generalization.*

2. *Computational generalization*: The intended relation is generalized so as to express the general state of a computation in terms of what has

been done and what remains to be done. *Ascending* and *descending* generalizations are two particular cases of computational generalization, where in ascending generalization, information about what has already been done is also needed, but in descending generalization the information about what remains to be done is enough.

Definition 29 If output program schema (or schema pattern) of the transformation schema is obtained by any method of generalization described above, then the transformation schema is called a *generalization schema.*

In the remainder of this section, I illustrate the generalization process described above on two examples; in the first one, I use tupling generalization, and in the second one, I use descending generalization.

Example 21 (Tupling Generalization) Let $sort/2$ be our initial problem, and its specification is:

$sort(L, S)$ iff integer-list S is the sorted version of integer-list L in ascending order.

Let's assume that $sort/2$ program below is constructed as the initial program, which is not very efficient in time and space, although it is better than most of the $sort/2$ programs that can be constructed.

$$\begin{aligned} sort([\,],[\,]) \leftarrow & \\ sort([HL|TL],S) \leftarrow\ & partition(TL,HL,TL1,TL2), \\ & sort(TL1,TS1), sort(TL2,TS2), \\ & append(TS1,[HL|TS2],S) \end{aligned}$$

with a correct program for $partition/4$, which has the specification below:

$partition(L, H, T1, T2)$ iff integer-list $T1$ has all the elements of integer-list L that are less than integer H, and integer-list $T2$ has all the remaining elements of L that are greater or equal to H.

and a correct program for $append/3$, having the specification:

$append(L1, L2, L3)$ iff list $L3$ is the concatenation of the lists $L1$ and $L2$.

Using tupling generalization, by generalizing the parameter L in the specification, the $sort/2$ problem can be generalized to the $sort\_tupling/2$ problem, which has the specification below:

$sort\_tupling(Ls, S)$ iff integer-list S is the concatenation of the sorted versions of the integer-lists in list Ls.

The next step in the generalization process is to synthesize a program for the generalized specification. Keeping the $sort/2$ program above in mind, the program for $sort\_tupling/2$ is:

$$
\begin{aligned}
sort\_tupling([\,],[\,]) \leftarrow & \\
sort\_tupling([[\,]|TLs],S) \leftarrow & \quad sort\_tupling(TLs,S) \\
sort\_tupling([[HL|TL]|TLs],[HL|TS]) \leftarrow & \quad partition(TL,HL,TL1,TL2), \\
& \quad TL1 = [\,], \\
& \quad sort\_tupling([TL2|TLs],TS) \\
sort\_tupling([[HL|TL]|TLs],S) \leftarrow & \quad partition(TL,HL,TL1,TL2), \\
& \quad TL1 \neq [\,], \\
& \quad sort\_tupling([TL1,[HL|TL2]|TLs],S)
\end{aligned}
$$

also with a correct program for $partition/4$.

Finally, the non-recursive program for the initial problem is:

$$sort(L,S) \leftarrow \quad sort\_tupling([L],S)$$

The resulting tupling generalized program is much more efficient than the initial program, both in time and space, since the call to *append* is eliminated, and the generalized program can be made semi-tail recursive, when L is the input parameter and S is the result parameter. □

Example 22 (Descending Generalization) Our initial problem is $reverse/2$, which has the specification below:

$reverse(L, R)$ iff list R is the reverse of list L.

For the $reverse/2$ problem, a "naive" program can be constructed as below:

$$\begin{aligned} reverse([\,],[\,]) &\leftarrow \\ reverse([HL|TL],R) &\leftarrow \quad reverse(TL,TR), \\ &\qquad\quad HR = [HL], append(TR,HR,R) \end{aligned}$$

with a correct program for $append/3$, which has the specification as the one given in Example 21.

The "naive" reverse program given above is not adequate, in the sense that it is not space efficient, since it generates too much intermediate data structures, and it will be time inefficient, if we don't have a linear-time program for *append*. Using descending generalization principles, our initial specification of $reverse/2$ can be generalized to the specification $S_{reverse\_desc}$, namely:

$reverse\_desc(L, R, A)$ iff list R is the concatenation of list A to the end of the reverse of list L.

The reader, who may wonder how I achieve this generalization of the initial specification, can refer to [16] for details. I will explain other methods for descendingly generalizing a specification in Sections 2.2.2 and 4.2.1.

The next step in the generalization process is to develop a program for $S_{reverse\_desc}$, which can be:

$$\begin{aligned} reverse\_desc([\,],R,R) &\leftarrow \\ reverse\_desc([HL|TL],R,A) &\leftarrow \quad reverse\_desc(TL,R,[HL|A]) \end{aligned}$$

Finally, the non-recursive program for the initial problem $reverse/2$ is:

$$reverse(L,R) \leftarrow \quad reverse\_desc(L,R,[\,])$$

The resulting descendingly generalized program is much more efficient than the initial program, both in time and space, since the call to *append* is eliminated, and the generalized program can be made tail recursive, when L is the input parameter and R is the result parameter. □

2.2 Related Work

The program transformation approach to the development of programs was first advocated by Burstall and Darlington [7] for functional programs that were written as sets of recursive equations. Burstall and Darlington divided the task of developing a correct and efficient program into two subtasks [7]:

1. develop an initial, maybe inefficient program whose correctness can be easily verified,
2. transform that initial program into a more efficient program.

Their transformation approach is based on the "rules+strategies" approach (i.e. they proposed transformation techniques that use a combination of some basic transformation strategies based on the transformation rules unfolding and folding). The extensive use of program transformation is strongly related to the development of functional and logic languages, since some simple tools, which will be explained in detail in Sections 2.2.1 and 2.2.2, can be easily used for program manipulations in these languages.

In this section, I present a summary of what has already been done in the *logic* program transformation area. I divided the transformation approaches into *strategy-based approaches* and *schema-based approaches.* However, most of the researchers in both fields work on program transformation in a given procedural semantics, which is the one of Prolog in most of the cases. I will later take a different approach, namely program transformation in declarative semantics. In Section 2.2.1, I present the strategy-based approaches to logic program transformation by using the categorization of Pettorossi and Proietti [41]. So, for a more detailed survey of strategy-based approaches to logic program transformation, the reader is invited to read [41], and similarly for transformation approaches in functional programming [42]. In Section 2.2.2, I present the schema-based logic program transformation techniques found in the literature.

2.2.1 Strategy-based Transformation Approaches

Before explaining the techniques that were proposed under the strategy-based approaches, I will first give the definitions of an *unfolding tree*, which represents the process of unfolding a given clause using a given program, and an *unfolding selection rule*, which definitions are taken from [41]. Then, I will give the definitions of some of the transformation strategies that were given in [41, 42], since they were widely used in the techniques that I will explain.

Definition 30 (Unfolding tree [41]) Let P be a program and let C be a clause. An *unfolding tree* for $P \cup \{C\}$ is a (finite or infinite) non-empty labeled tree such that:

(i) the root is labeled by the clause C;

(ii) if M is a node labeled by a clause D, then:

either M has no sons,

or M has $n(\geq 1)$ sons labeled by the clauses $D_1, \ldots, D_n$ obtained by unfolding D wrt an atom of its body using P,

or M has one son labeled by a clause obtained by goal replacement from D.

Definition 31 (Unfolding selection rule [41]) An *unfolding selection rule* is a function that, given an unfolding tree and one of its leaves, tells us whether or not to unfold the clause in that leaf, and, in the affirmative case, tells us the atom wrt which that clause should be unfolded.

Definition 32 (Generalization Strategy [42]) Given a clause C of the form

$$H \leftarrow A_1, \ldots, A_m, B_1, \ldots, B_n$$

we define a new predicate *genp* by a clause G of the form

$$genp(X_1, \ldots, X_k) \leftarrow GenA_1, \ldots, GenA_m$$

where $(GenA_1 \ldots, GenA_m)\theta = A_1, \ldots, A_m$, for a given substitution θ, and $\{X_1, \ldots, X_k\}$ is a superset of the variables that are necessary to fold using a clause whose body is $GenA_1, \ldots, GenA_m$. We then fold C using G and we get

$$H \leftarrow genp(X_1, \ldots, X_k)\theta, B_1, \ldots, B_n.$$

We finally look for the recursive definition of the predicate *genp*. A suitable form of the clause G introduced by the *generalization strategy* can often be obtained by matching clause C against one of its descendants, say D, in the unfolding tree, which is considered during program transformation. In particular, we will consider the case where:

1. D is the clause $K \leftarrow E_1, \ldots, E_m, F_1, \ldots, F_r$ and D has been obtained from C by applying no transformation rules, except rearrangement of goals and deletion of duplicate goals in a clause, which preserve the correctness in declarative semantics, to $B_1, \ldots, B_n$;
2. for $i = 1, \ldots, m$, the atom E_i has the same predicate as A_i;
3. for $i = 1, \ldots, m$, the atom E_i is not an instance of A_i;
4. the goal $GenA_1 \ldots, GenA_m$ is the most specific generalization of $A_1, \ldots, A_m$ and $E_1, \ldots, E_m$;
5. $\{X_1, \ldots, X_k\}$ is the minimum subset of $vars(GenA_1 \ldots, GenA_m)$ (where $vars(t)$ denotes the set of variables occurring in term t), which is necessary to fold both C and D using a clause whose body is $GenA_1, \ldots, GenA_m$.

The *loop absorption strategy*, which is formally introduced by Proietti and Pettorossi [43], can be viewed as a particular case of the generalization strategy, which can be applied if the conditions 1, 2, 4, and 5 hold in the definition of the generalization strategy, and for $i = 1, \ldots, m$, E_i is an instance of A_i.

The strategies above were also called *auxiliary strategies* [41], since they can be used by a more general strategy, called the *predicate tupling strategy.*

Definition 33 (Predicate Tupling Strategy [42]) This strategy, also called *tupling*, for short, consists of selecting some atoms, say $A_1, \ldots, A_n$, with $n \geq 1$,

occurring in the body of a clause C. We introduce a new predicate *newp* defined by a clause T of the form:

$$newp(X_1, \ldots, X_k) \leftarrow A_1, \ldots, A_n$$

where $X_1, \ldots, X_k$ are the *linking variables* in C (i.e., the variables occurring in $A_1, \ldots, A_n$, and also in the head and in the remaining atoms in the body of C). We then look for the recursive definition of the predicate *newp* by performing some unfolding, and two more transformation rules (i.e, goal replacement and clause deletions, which were defined in [41]) followed by some folding steps using clause T. We finally fold the atoms $A_1, \ldots, A_n$ in the body of C using clause T.

Now, I explain some of the work done in the program transformation field using a strategy-based approach. The techniques can be categorized under the following titles: compiling control, composing programs, changing data representation, recursion removal, annotations and memoing, and partial evaluation.

COMPILING CONTROL

Programs that are written with the left-to-right computation rule of Prolog in mind are often not very efficient, because of the amount of nondeterminism during the execution of these programs in Prolog.

Compiling control was defined as a different approach to program transformation [41], in the sense that a given program is transformed into a program that behaves under the naive evaluator (i.e. the execution mechanism) of Prolog as the given program would behave under an enhanced evaluator that uses a better control strategy.

The *filter promotion strategy* was proposed with a similar idea in functional programming by Bird [4], which is a general method to transform an input program into a more efficient program by exploiting the recursive structure in the dominant term of an algorithmic expression. In [41], Pettorossi and Proietti categorized the transformation technique that was proposed by Seki and Furukawa [49], as a technique similar to compiling control and the filter

promotion strategy, for transforming generate-test programs into more efficient programs. However, I will categorize their method under synthesis of programs.

In [41], basic techniques of compiling control are characterized as follows:

> Given a program P_1, a set Q of queries, and a computation rule C, compiling control derives a new program P_2 by first constructing a suitable unfolding tree, say T, and then applying the loop absorption strategy.

COMPOSING PROGRAMS

Compositional programming is a popular style of programming, which consists of decomposing a given goal in smaller and easier subgoals, then writing programs to solve these subgoals, and finally composing these programs in an appropriate way [41]. However, the disadvantage of this style is that the composition of the programs that are written to solve the subgoals results in inefficient programs, since this composition does not take into account the interactions that may occur while evaluating these subgoals.

For functional and imperative programs, various transformation methods have been proposed in the literature, which can be classified under this category, e.g., *finite differencing* [40], *deforestation* [59], and *super-compilation* [56, 51].

Loop merging, in Section 2.1.3, (also called *loop fusion* by Debray [14]) is one of the transformation techniques that was proposed for improving programs that were written in compositional style. This technique transforms the program for a relation that is defined as the composition of two independent recursive relations into a program where a new relation is introduced, which does all the computations done by these two recursive relations. *Unnecessary variable elimination* is another technique, proposed by Proietti and Pettorossi [44], for deriving programs without unnecessary variables, and uses the predicate tupling strategy. A variable X of a clause C is *unnecessary* if at least one of the following two conditions holds [44]:

- X occurs more than once in the body of C (in this case, X is a *shared* variable);
- X does not occur in the head of C (in this case, X is an *existential* variable).

The loop merging and the unnecessary variable elimination methods avoid multiple traversals of data structures as well as the construction of intermediate data structures.

CHANGING DATA REPRESENTATION

Choosing the appropriate data representation is an important issue to develop an efficient program, but this is not an easy process in most of the cases, and, further, complex data representations complicate the correctness proofs of programs. Program transformation was proposed as a solution to the problem above. In logic programming, transformation of programs that use lists into equivalent programs that use *difference-lists* is the best-known example of program transformation by changing data representation.

A *difference-list*, denoted by $L\backslash R$, where L and R are lists, can be used to represent a third list X, such that the concatenation of X and R is L. A single list can be represented by many difference-lists. The main advantage of difference-lists is that the concatenation of two difference-lists can be performed in constant time, unlike in the simple list representation, where the concatenation of two lists takes linear time wrt the length of the first list.

Programs that use lists are often easier to write and understand than programs that use difference-lists. Let us illustrate this on an example for the *reverse* relation.

Example 23 The program for *reverse* that uses simple lists was given in Example 22. The desired transformation can be achieved by applying the definition introduction rule, and introducing a new relation *reverse_d* with the following initial definition:

$$reverse\_d(X, L\backslash R) \leftarrow reverse(X, Y), append(Y, R, L)$$

Performing some unfolding and goal replacement steps, a new program for $reverse\_d$ can be obtained, and finally, the transformed program, which uses a difference-list, can be written as:

$$\begin{aligned} reverse(L,R) &\leftarrow reverse\_d(L, R\backslash[\,]) \\ reverse\_d([\,], L\backslash L) &\leftarrow \\ reverse\_d([HL|TL], L\backslash R) &\leftarrow reverse\_d(TL, L\backslash[HL|R]) \end{aligned}$$

□

In [31], Hansson and Tärnlund proposed a semi-automatic technique to derive a program using difference-lists from a program that uses simple lists, by introducing a function that maps a simple list to a difference-list. Their data structure mapping takes away the *append* procedure, which is the concatenation relation defined for simple lists. In [62], Zhang and Grant proposed an automatic transformation technique towards difference-list manipulation, which applies under control the transformation rules folding and unfolding, and some other transformation techniques. Their technique also made use of semantic information on the relations that are used in the program, e.g., associativity. In [39], Marriott and Søndergaard proposed an automatic three staged transformation technique that transforms list-processing programs into programs that use difference-lists by first doing data flow analysis of the input Prolog program to determine whether the transformation is applicable to the input program. In this first part of the method, data structure transformation is performed by converting the *append* calls into variations of *append*. Then, the most efficient version of *append* in that case is chosen for the procedural semantics preserving concatenation of difference-lists. Finally, the non-logical calls added during the previous stages are removed.

The new relation, which has to be introduced in all the methods ($reverse\_d$ in our example), can also be viewed as the invention of an *accumulator variable* in the *accumulation strategy*, which was first introduced in [4] for transforming functional programs. Simply put, the accumulation strategy achieves the generalization of the initial problem by the inclusion of an extra parameter, which is called *accumulator*. Indeed, in Example 23, the new relation $reverse\_d(X, L\backslash R)$ can be written as $reverse\_acc(X, L, R)$, where R is the

accumulator parameter. The reader may also notice that descending generalization also comes to the same conclusion with its different underlying idea. Also note that the accumulator strategy and descending generalization provide more than a conversion to difference-list representation, since any difference structure can be represented by these methods. I will further discuss this in Sections 2.2.2 and 4.2.

RECURSION REMOVAL

Although recursion is the main control structure for declarative programs, the extensive use of recursive relations may lead to programs that are inefficient in time and space. In logic programming, recursion removal means transforming a recursive program into a tail recursive program.

In [13], Debray proposed a transformation technique to transform an *almost-tail recursive* program into a tail recursive one. He defined an *almost-tail recursive* clause as a recursive clause where the atoms following the last recursive call in the body involve only primitive computations. So, a program is said to be *almost-tail recursive* iff all its recursive clauses are either tail recursive or almost-tail recursive, and there has to be at least one almost-tail recursive clause in that program. His technique introduces an auxiliary relation, like the definition introduction in transformation towards difference-lists, in the first stage. Then, the most efficient recursive program for the new relation is obtained by using the unfolding/folding transformation rules. Finally, his method converts the new program to a tail recursive version, if it was not already, by using the syntactic structure of the recursive calls, and the semantic properties of the primitive operations, which are called lastly in the recursive clauses, e.g., associativity, commutativity, and so on.

ANNOTATIONS and MEMOING

In the literature, the transformation techniques that make use of the extra-logical features of logic languages, like cuts, asserts, and so on, are also studied widely. These techniques are called *program annotations*, which was first used to define similar techniques in functional programming. Prolog program transformation techniques that are based on the usage of the extra-logical predicates

of Prolog, the computation, and the search rule of Prolog are explained in detail by Deville [16].

A typical technique, which was given in [16, 41], transforms a given Prolog program into an efficient annotated program by adding the cut operator, which is denoted by "!". Let us illustrate this on an example.

Example 24 Let the input Prolog program be as follows:

$$r(X) \leftarrow A, C_1$$
$$r(X) \leftarrow not(A), C_2$$

where C_1 and C_2 are conjunctions of atoms, A is an atom, and $not(A)$ denotes the negation of the atom A. The program above can be transformed (if A has no side-effects) into

$$r(X) \leftarrow A, !, C_1$$
$$r(X) \leftarrow C_2$$

The output program is more efficient than the initial program, since it behaves like an if-then-else statement. □

Memoization is another technique that can be classified under program annotations, where the results of the previous transformations are stored in a table for further use.

PARTIAL EVALUATION

Partial evaluation [33] (also called *partial deduction* in the case of logic programming) is a program transformation technique that takes as input a program and a query, and produces an output program optimized for all instances of that query. For a detailed explanation and further references, the reader can refer to [41]. I will illustrate partial evaluation using the example which was given in [41].

Example 25 Let the program P be:

$$
\begin{aligned}
p([\,],Y) &\leftarrow \\
p([H|T],Y) &\leftarrow \quad q(T,Y) \\
q(T,Y) &\leftarrow \quad Y = b \\
q(T,Y) &\leftarrow \quad p(T,Y)
\end{aligned}
$$

and let the query Q be $\leftarrow p(X,a)$. If we use the *unfolding strategy* U [41], which performs unfolding steps starting from the query $\leftarrow p(X,a)$ until each leaf of the SLDNF-tree is either a success or a failure or has predicate p, then, finally, the SLDNF-tree in Figure 2.1 below will be obtained.

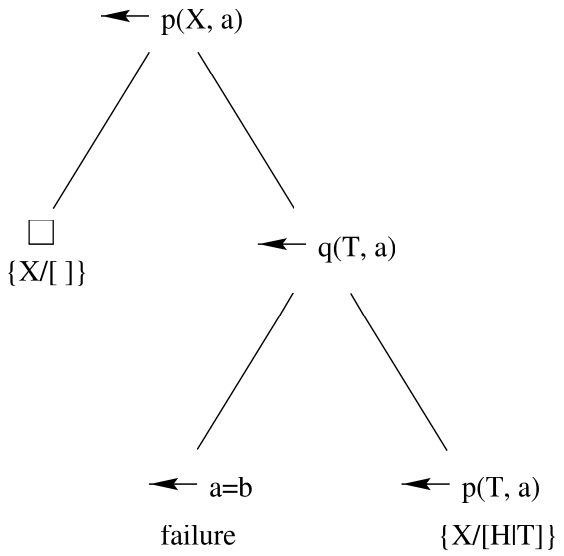

Figure 2.1. An SLDNF-tree of $P \cup \{\leftarrow p(X,a)\}$ using U

After collecting the goals and and the substitutions corresponding to the leaves of that tree, the output program of the partial evaluation of the program P and the query Q is as follows:

$$
\begin{aligned}
p([\,],a) &\leftarrow \\
p([H|T],a) &\leftarrow \quad p(T,a)
\end{aligned}
$$

The final program does not contain the clauses for q, since p does not depend on q in the output program. □

There exist (semi-)automatic partial evaluators that use the idea of partial evaluation to transform programs into more efficient programs for the case where some information about the input parameters of the program is a-priori

known, e.g., Mixtus [48] (for a summary of Mixtus and its integration details into another transformation system, refer to Chapter 7).

2.2.2 Schema-based Transformation Approaches

Logic program schemas have proven useful in various fields of logic programming: teaching logic programming to novices [25], synthesizing logic programs [52, 17, 19, 22], and also transforming logic programs [20, 24, 57, 58, 27]. The basic ideas for using schemas for synthesizing and transforming programs were introduced first for functional programs, e.g., the transformation schemas for improving recursive functions [32].

The strategy-based approaches to logic program transformation, which were explained in Section 2.2.1, are actually sequences of transformation rules that are not predefined. A strategy thus needs a global plan for the application of transformation rules, since at each point a check must be made whether the application of a possible transformation rule will result in the most efficient program at the end. The schema-based approaches to program transformation, on the other hand, consist of a database of predefined transformations, which are called *transformation schemas*. There exist different definitions for the notion of transformation schema [57, 27]. However, our definition of a transformation schema in Section 2.1.5 is the most general one, in the sense that it is possible to represent all the transformation schemas in the literature up to now by our definition.

Most of the transformation schemas that I am going to explain are represented as higher-order logic programs. So, the selection of the applicable transformation, which is the first step of the transformation at the program-level, becomes the most time-consuming step, because of the higher-order matching that has to be performed.

Since Gegg-Harrison did not give a unified definition for transformation schemas in [27], and represented the transformation schemas either as a triple (an input program, an output program, and the conditions, which have to be satisfied for achieving that transformation), or as a quadruple (two input

programs, one output program, and applicability conditions if they exist), I will not repeat his definitions here. However, it is better to examine the definition of the transformation schema that was first proposed by Fuchs and Fromherz [24], and was then extended by Vasconcelos and Fuchs [57, 58] by also augmenting the program schema representation, since their representation is more formal and easy to examine, and also because Gegg-Harrison's work can be represented using their transformation schema definition. Below is their definition of transformation schemas (which they called *schema-based transformation*) [57]:

> A transformation schema T is a quadruple of the form
>
> $$\langle < G_1, \ldots, G_n >, < S_1, \ldots, S_n >, < H_1, \ldots, H_m >, < T_1, \ldots, T_m > \rangle$$
>
> where $< G_1, \ldots, G_n >$ and $< H_1, \ldots, H_m >$ are conjunctions of subgoals, and $< S_1, \ldots, S_n >$ and $< T_1, \ldots, T_m >$ are input and output program schemas respectively.

The applicability conditions of their transformation schemas are either implicitly checked, or attached to the program schema representations, since they did not have a fifth component for them in their transformation schemas. If $n = 1$ and $m = 1$ in the definition above, which means that an individual procedure is transformed into another one, this can be represented in our transformation schema definition by taking the input program schema as $\{C_1\} \cup S_1$ and the output program schema as $\{C_2\} \cup T_1$, where C_1 is the clause $r(X_1, \ldots, X_k) \leftarrow G_1(X_1, \ldots, X_k)$, and C_2 is the clause $r_t(X_1, \ldots, X_l) \leftarrow H_1(X_1, \ldots, X_l)$, where r_t is the new relation, which is introduced by the transformation, and k and l are respectively indexes indicating the number of arguments of the relations r and r_t. For the cases where $n > 1$ and $m \geq 1$, since we allow nested programs (where the relations are defined as an instance of a program schema in the extension of these programs), the transformation schema above also can be represented in our notation, where the input program has as the template the single clause $r(X_1, \ldots, X_k) \leftarrow G_1(X_1, \ldots, X_k), \ldots, G_n(X_1, \ldots, X_k)$ and the extension $\{S_1, \ldots, S_n\}$, and the output program has as the template the single clause $r_t(X_1, \ldots, X_l) \leftarrow H_1(X_1, \ldots, X_l), \ldots, H_m(X_1, \ldots, X_l)$ and the extension $\{T_1, \ldots, T_m\}$. This will cause us to extend the definition of schema-based

transformations to capture recursive schema-based transformations.

In [57], Vasconcelos and Fuchs categorized the work done in the schema-based logic program transformation field into three categories, depending on the integration of the transformation steps in program construction [58]:

(1) efficiency issues are considered *during* the program construction using the programming techniques that are standard logic programming constructs, and guarantee a good computational behavior of the constructed programs. For instance, Prolog programming techniques are extensively studied in the literature (e.g., [53]);

(2) efficiency issues are considered *after* the program is constructed, by transforming the code of the program (i.e., the second transformation step in Deville's logic program development methodology);

(3) efficiency issues are considered *during* the synthesis of a program whenever possible, such that a program is synthesized using a program technique, *and* the information, which is gained during the synthesis of the logic program, will be used in transforming the logic program *before* translating it into a program, which is written in a given language. Actually, this category was born as a result of Deville's methodology in schema-based logic program development (e.g., [19, 1, 20]).

I do not give examples of the work done under the first category, since these techniques fully meld the transformation step in the construction of programs. The transformation schemas that are proposed in this thesis fit into the third category, since this work is actually an extension of the ideas proposed in [20, 1]. Most of the transformation schemas that will be explained in this section are examples of the second category. So, if I do not indicate under which category the work can be classified, then this means that the work is under the second category. Otherwise, I will explicitly indicate to which category the work belongs.

I will categorize schema-based approaches using the categorization made in Section 2.2.1 for strategy-based approaches. However, nearly all the papers

in the schema-based logic program transformation literature can be classified under two categories out of the six categories in Section 2.2.1, namely recursion removal and composing programs. There exist some exceptions, e.g., the transformation schemas proposed by Seki and Furukawa [49] for reducing the amount of nondeterminism of generate-test programs, were classified under the category compiling control in Section 2.2.1. As I indicated before, I will categorize their work as a method for synthesizing a program using program schemas.

RECURSION REMOVAL

In [5], Brough and Hogger proposed two transformation schemas, where the second one further improves the output program of the first one, if the applicability conditions are satisfied. The first transformation schema transforms an input program, which has to be a member of a subclass of recursive programs for relations of arity 2, into an output program, which is also recursive and has a time complexity similar to the input program, by checking the applicability conditions, which are the associativity and the closeness property of the computation relation, which is the last call in the body of the recursive clause of the input program. For instance, the almost-tail recursive programs, defined by Debray [13], are a subset of the input programs that can be transformed by this transformation schema. The second transformation schema takes as an input program the output program of the transformation schema mentioned above, and transforms it into a tail recursive program if the applicability conditions, namely the right-identity and functionality properties of the computation relation, are satisfied by the input program.

In [6], the same authors proposed two more transformation schemas, where the second one is a more generic version of the first one, in the sense that the input program family that can be transformed by the first transformation schema is a sub-family of the input program family that can be transformed by the second transformation schema. These schemas were constructed by investigating the analogy between grammars and logic programs, where they assumed the logic programs were fully declarative (i.e., their transformation schemas can be classified under the third category according to Vasconcelos and Fuchs'

categorization). The first transformation schema, namely *forward-simulation transformation*, is the analogue of one of the important rules for grammars, namely the *Greibach-Foster transformation*, which takes as an input a left-recursive grammar and produces as an output a right-recursive grammar. The normalized template of the input logic program schema analogue of the left-recursive grammar in the forward-simulation transformation can be represented in our notation as:

$$\begin{aligned} r(X) &\leftarrow \quad d(X) \\ r(X) &\leftarrow \quad r(Y), c(X,Y) \end{aligned}$$

Then, the template of the output program schema analogue of the right-recursive grammar is:

$$\begin{aligned} r(T) &\leftarrow \quad d(X), s(X,T) \\ s(T,T) &\leftarrow \\ s(Y,T) &\leftarrow \quad c(Z,Y), s(X,T) \end{aligned}$$

This transformation provides left-recursive elimination (i.e., provides tail recursion by introducing an accumulator parameter). The second transformation schema was constructed by also using the analogy above for a class of programs that are larger than the input program family of the forward-simulation transformation.

In [27], Gegg-Harrison proposed two transformation schemas for transforming single recursive programs into tail recursive programs. The first transformation schema is the counterpart of the accumulation strategy in the strategy-based approaches. The applicability condition of the transformation schema is defined as the associativity of the computation relation in the input schema, which computes the final version of the result parameter. The second transformation schema was proposed to transform a single recursive program, which he called a *forward-processing* program (i.e., a program that processes its input list from the head and one of the outputs, which is a number, from its actual value down to 0), into another single recursive program, where the number argument is processed from 0 up to its actual value.

In [24], Fuchs and Fromherz proposed a transformation schema that simulates the accumulation strategy for transforming recursive list-processing programs into tail recursive list-processing programs. In [58], Vasconcelos and Fuchs proposed an extension of the transformation schema that was introduced in [24]. The applicability conditions of the transformation schemas above consist of the needed declarative properties of the relations, and also the properties related to the operational semantics of Prolog.

In [20], Flener and Deville proposed two transformation schemas that automate the tupling generalization and the descending generalization, which are explained in Section 2.1.6. So, they called these transformation schemas *generalization schemas*. The tupling generalization schema can transform an input program, which is an instance of a program schema that abstracts a subclass of recursive programs, into an output tail recursive program iff some of the open relations of the input template satisfy some properties, which are the applicability conditions, e.g., associativity of the relation that computes the result parameter. The descending generalization schema transforms a single recursive program into a tail recursive program iff the applicability conditions of the generalization schema are satisfied. They also indicated the analogy between the descending generalization schema and the accumulation strategy in strategy-based approaches. So, these generalization schemas mechanize the generalization of a restricted sub-family of recursive programs, where this generalization process was thought to be necessarily under human control before Flener and Deville's work. The reason is mainly that the generalization process introduces a new relation, which defines the generalized problem, and this definition introduction step (i.e., the eureka discovery step) needs human interaction. Flener and Deville showed that this step can be eliminated by using the transformation schemas proposed for a restricted sub-family of programs.

Using the ideas in [20], Batu pre-compiled some more generalization techniques for different families of programs. These generalization schemas can be found in [1]. The generalization schemas that will be presented in this thesis are actually extensions of Flener and Deville's, and Batu's generalization schemas by extending the program schema and the transformation schema representations, and the eureka discovery step is fully eliminated by the generalization

schemas that we have in this thesis.

Note that the transformation schemas that are counterparts of the accumulation strategy can also be classified as 'changing data representation', since these transformations represent the transformations towards difference-structures implicitly. The descending generalization of the relation *reverse*, which is given in Example 22, can be achieved by the transformation schemas that simulate the accumulation strategy, since *reverse* is a list-processing single recursive program with *append* as the composition relation, which satisfies the applicability conditions of these transformation schemas.

COMPOSING PROGRAMS

In [27], Gegg-Harrison proposed a set of transformation schemas that can transform a program that is written in a compositional style into a more efficient program by merging the logic programs written for the subgoals, which are instances of the list-processing recursive program schemas, and they have common arguments.

In [58], Vasconcelos and Fuchs presented two transformation schemas in the appendix that were also pre-compiled in their transformation system, where the second one is more generic than the first one, and both are counterparts of the loop merging in the strategy-based approaches. The first schema can merge two programs manipulating the same single recursive data-structure, whereas the second one can merge two data-structure manipulating programs, even if these programs have different possibilities of recursions.

The loop merging example, namely Example 11, can be achieved by using the transformation schemas in [58]. I will not illustrate the schemas above by an example, since their schema representations have to be explained in detail.

Later, in [47], Richardson and Fuchs proposed a methodology for development of provably correct program transformation schemas, by abstracting the program transformation operations to transformation operations on program schemas. They have defined abstract unfold operation on program schemas to mirror the concrete unfold operations on programs. They also indicate a way

to define the fold operation on program schemas. Unfortunately, much has to be done on this work to be useful, e.g., correctness proofs of these operations.

Chapter 3

Divide-and-Conquer Logic Program Schemas

The divide-and-conquer methodology is one of the most effective program construction methodologies, since it is applicable to a large variety of problems, and the programs that are constructed by this methodology are easy to understand. The *divide-and-conquer methodology* solves a problem in three steps: [11]

i. *divide* a problem into sub-problems, unless it can be trivially solved;

ii. *conquer* the sub-problems by solving them recursively;

iii. *combine* the solutions to the sub-problems into a solution to the initial problem.

If a (sub)problem can be solved trivially (without dividing any more and recursion), it is called a *minimal* case, otherwise it is called a *non-minimal* case.

The program schema patterns given in this chapter abstract sub-families of divide-and-conquer (DC) programs. They are restricted to binary predicates with X as the induction parameter and Y as the result parameter, to reflect the program schema patterns that can be represented by the prototype transformation system explained in Chapter 7. Another restriction in the schema patterns is that when X is non-minimal, then X is decomposed into *one* head

HX and t tails $TX_1, \ldots, TX_t$, so that Y is composed from one head HY (which is the result of processing HX) and t tails $TY_1, \ldots, TY_t$ (which are the results of recursively calling the predicate with $TX_1, \ldots, TX_t$, respectively) by p-fix composition (i.e. Y is composed by putting HY between TY_{p-1} and TY_p).

These program schema patterns are called $DCLR$ and $DCRL$ (the reason why I call them $DCLR$ and $DCRL$ will be explained after I give the schema patterns). Template 1 (respectively, Template 2) is the template of the $DCLR$ schema pattern (respectively, the $DCRL$ schema pattern).

Logic Program Template 1

$r(X, Y) \leftarrow$

$minimal(X),$

$solve(X, Y)$

$r(X, Y) \leftarrow$

$nonMinimal(X),$

$decompose(X, HX, TX_1, \ldots, TX_t),$

$r(TX_1, TY_1), \ldots, r(TX_t, TY_t),$

$I_0 = e,$

$compose(I_0, TY_1, I_1), \ldots, compose(I_{p-2}, TY_{p-1}, I_{p-1}),$

$process(HX, HY), compose(I_{p-1}, HY, I_p),$

$compose(I_p, TY_p, I_{p+1}), \ldots, compose(I_t, TY_t, I_{t+1}),$

$Y = I_{t+1}$

Logic Program Template 2

$r(X, Y) \leftarrow$

$$
\begin{aligned}
& minimal(X), \\
& solve(X, Y) \\
r(X, Y) \leftarrow & \\
& nonMinimal(X), \\
& decompose(X, HX, TX_1, \ldots, TX_t), \\
& r(TX_1, TY_1), \ldots, r(TX_t, TY_t), \\
& I_{t+1} = e, \\
& compose(TY_t, I_{t+1}, I_t), \ldots, compose(TY_p, I_{p+1}, I_p), \\
& process(HX, HY), compose(HY, I_p, I_{p-1}), \\
& compose(TY_{p-1}, I_{p-1}, I_{p-2}), \ldots, compose(TY_1, I_1, I_0), \\
& Y = I_0
\end{aligned}
$$

The steadfastness constraints of these schema patterns (i.e., the specifications of the open relations in these templates) are the same, since these templates have the same open relations, and these constraints are shown in [21]. For example, the specifications of *solve* and *decompose* are:

$$\forall X : \mathcal{X}.\ \forall Y : \mathcal{Y}.\ \ \mathcal{I}_r(X) \wedge \mathcal{O}_m(X) \Rightarrow [solve(X, Y) \Leftrightarrow \mathcal{O}_r(X, Y)$$

$$
\begin{aligned}
& \forall X, TX_1, \ldots, TX_t : \mathcal{X}.\ \forall HX : \mathcal{HX}.\ \ \mathcal{O}_{nm}(X) \Rightarrow \\
& \quad [decompose(X, HX, TX_1, \ldots, TX_t) \Leftrightarrow \\
& \quad Dec(X, HX, TX_1, \ldots, TX_t) \wedge \bigwedge_{i=1}^{t} \mathcal{I}_r(TX_i) \wedge \bigwedge_{i=1}^{t} TX_i \prec X]
\end{aligned}
$$

where $\mathcal{I}_r$ is the input condition of the top level relation r, $\mathcal{O}_r$ (respectively, $\mathcal{O}_m$ and $\mathcal{O}_{nm}$) is the output condition of r (respectively, $minimal$ and $nonMinimal$), and $\prec$ is a well-founded order over the sort of the induction parameter X.

Now, I explain the underlying idea why we have two different schema patterns for DC, and why we call them $DCLR$ and $DCRL$. If we denote the functional version of the *compose* predicate with $\oplus$, then the composition of Y in template $DCLR$ by *left-to-right* (LR) *composition ordering* can be written as

$$Y = ((((((e \oplus TY_1) \oplus \ldots) \oplus TY_{p-1}) \oplus HY) \oplus TY_p) \oplus \ldots) \oplus TY_t \qquad (3.1)$$

The composition of Y in $DCRL$ by *right-to-left* (RL) *composition ordering* can be written as

$$Y = TY_1 \oplus (\ldots \oplus (TY_{p-1} \oplus (HY \oplus (TY_p \oplus (\ldots \oplus (TY_t \oplus e)))))) \qquad (3.2)$$

Each example program in this chapter is an instance of a particularization of the schema pattern that it belongs to, namely for $t = 2$ and p varying between 1 and 3, for prefix, infix, and postfix composition, respectively.

Three problems (to give a better understanding of p-fix composition) are given for traversing binary trees. In all the problems, the constant *void* is used to represent the empty binary tree, and the compound term $bt(L, E, R)$ is used to represent a binary tree of root E, left subtree L, and right subtree R. Because of properties of *compose*, we can construct two programs, which are instances of the DC schema patterns above, for each problem. For these problems, equations 3.1 and 3.2 can be further simplified resulting in an equal composition of the result parameter as:

$$Y = TY_1 \oplus \ldots \oplus TY_{p-1} \oplus HY \oplus TY_p \oplus \ldots \oplus TY_t$$

Example 26 For the prefix traversal of a binary tree, we have the specification below:

$prefix\_flat(B, F)$ iff list F is the prefix representation of binary tree B,

where *prefix representation* means the list representation of the prefix traversal of the tree.

Program 1 below is a program for the $prefix\_flat/2$ problem, and it is an instance of the $DCLR$ schema pattern.

$$
\begin{aligned}
&prefix\_flat(B, F) \leftarrow \\
&\quad B = void, \\
&\quad F = [\,] \\
&prefix\_flat(B, F) \leftarrow \\
&\quad B = bt(\_, \_, \_), \\
&\quad B = bt(L, E, R), \\
&\quad prefix\_flat(L, FL), prefix\_flat(R, FR), \\
&\quad I_0 = [\,], \\
&\quad HF = [E], append(I_0, HF, I_1), \\
&\quad append(I_1, FL, I_2), append(I_2, FR, I_3), \\
&\quad F = I_3
\end{aligned}
$$

Logic Program 1

Since Program 1 is an instance of the $DCLR$ schema pattern for $t = 2$ and $p = 1$ (i.e. prefix composition), the calls

$$compose(I_0, TY_1, I_1), \ldots, compose(I_{p-2}, TY_{p-1}, I_{p-1})$$

in the non-minimal case reduce to the empty conjunction (i.e. *true*), during particularization.

Program 2 below is another program for the $prefix\_flat/2$ problem, and it is an instance of the $DCRL$ schema pattern.

$$prefix\_flat(B, F) \leftarrow$$

$$B = void,$$

$$F = [\,]$$

$$prefix\_flat(B, F) \leftarrow$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$prefix\_flat(L, FL), prefix\_flat(R, FR),$$

$$I_3 = [\,],$$

$$append(FR, I_3, I_2), append(FL, I_2, I_1),$$

$$HF = [E], append(HF, I_1, I_0),$$

$$F = I_0$$

Logic Program 2

Similarly, for Program 2, which is an instance of the $DCRL$ schema pattern, the calls

$$compose(TY_{p-1}, I_{p-1}, I_{p-2}), \ldots, compose(TY_1, I_1, I_0)$$

in the non-minimal case reduce to the empty conjunction (i.e. *true*), during particularization.

Example 27 For the infix traversal of a binary tree, we have the specification below:

$infix\_flat(B, F)$ iff list F is the infix representation of binary tree B,

where *infix representation* means the list representation of the infix traversal of the tree.

Program 3 below is a program for the $infix\_flat/2$ problem, and it is an instance of the $DCLR$ schema pattern.

$$
\begin{aligned}
&infix\_flat(B, F) \leftarrow \\
&\quad B = void, \\
&\quad F = [\,] \\
&infix\_flat(B, F) \leftarrow \\
&\quad B = bt(\_, \_, \_), \\
&\quad B = bt(L, E, R), \\
&\quad infix\_flat(L, FL), infix\_flat(R, FR), \\
&\quad I_0 = [\,], \\
&\quad append(I_0, FL, I_1), \\
&\quad HF = [E], append(I_1, HF, I_2), \\
&\quad append(I_2, FR, I_3), \\
&\quad F = I_3
\end{aligned}
$$

Logic Program 3

Program 4 below is another program for the $infix\_flat/2$ problem, and it is an instance of the $DCRL$ schema pattern.

$$
\begin{aligned}
&infix\_flat(B, F) \leftarrow
\end{aligned}
$$

$$B = void,$$

$$F = [\,]$$

$$infix\_flat(B, F) \leftarrow$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$infix\_flat(L, FL), infix\_flat(R, FR),$$

$$I_3 = [\,],$$

$$append(FR, I_3, I_2),$$

$$HF = [E], append(HF, I_2, I_1),$$

$$append(FL, I_1, I_0),$$

$$F = I_0$$

Logic Program 4

Example 28 For the postfix traversal of a binary tree, we have the specification below:

$postfix\_flat(B, F)$ iff list F is the postfix representation of binary tree B,

where *postfix representation* means the list representation of the postfix traversal of the tree.

Program 5 below is a program for the $postfix\_flat/2$ problem, and it is an instance of the $DCLR$ schema pattern.

$$postfix\_flat(B, F) \leftarrow$$

$$B = void,$$

$$F = [\,]$$

$$postfix\_flat(B, F) \leftarrow$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$postfix\_flat(L, FL), postfix\_flat(R, FR),$$

$$I_0 = [\,],$$

$$append(I_0, FL, I_1), append(I_1, FR, I_2),$$

$$HF = [E], append(I_2, HF, I_3),$$

$$F = I_3$$

Logic Program 5

Program 6 below is another program for the $postfix\_flat/2$ problem, and it is an instance of the $DCRL$ schema pattern.

$$postfix\_flat(B, F) \leftarrow$$

$$B = void,$$

$$F = [\,]$$

$$postfix\_flat(B, F) \leftarrow$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$postfix\_flat(L, FL), postfix\_flat(R, FR),$$

$$I_3 = [\,],$$

$$HF = [E], append(HF, I_3, I_2),$$

$$append(FR, I_2, I_1), append(FL, I_1, I_0),$$

$$F = I_0$$

Logic Program 6

By the same reasoning that we use in explaining the instances of programs for $prefix\_flat/2$, for $postfix\_flat/2$, where $t = 2$ and $p = 3$ (i.e. postfix composition), both the calls

$$compose(I_p, TY_p, I_{p+1}), \ldots, compose(I_t, TY_t, I_{t+1})$$

in $DCLR$ and the calls

$$compose(TY_t, I_{t+1}, I_t), \ldots, compose(TY_p, I_{p+1}, I_p)$$

in $DCRL$ reduce to the empty conjunctions (i.e. *true*), during particularization.

Chapter 4

Problem Generalization Schemas

In Section 2.1.6, I summarized the logic program generalization techniques that were proposed by Deville in [16], and illustrated these techniques on two example problems. He proposed these techniques for logic program development. As I mentioned in Section 2.2.2, these techniques were further used for computer-aided synthesis and transformation of logic programs [17, 20]. The main objective of this research is to extend the usage of these generalization techniques in schema-based logic program transformation, since in the referenced papers above, the automation of these generalization methods was proposed for restricted sub-families of *DC* programs.

In this chapter, I present the generalized generalization schemas that are constructed by extending the ideas proposed in [20, 1]. The generalized tupling generalization schemas are given in Section 4.1 and the generalized descending generalization schemas are given in Section 4.2, together with the complexity analysis of these schemas. In Section 4.3, simultaneous-tupling-and-descending generalization schemas are given with their complexity analysis. For the correctness proofs of these generalization schemas, the reader is invited to consult [9].

4.1 Tupling Generalization

The definition of tupling generalization (TG) was given in Section 2.1.6, and the tupling generalization process was illustrated in Example 21 for the problem of sorting integer lists. In Section 2.2.2, the automation process of tupling generalization of a restricted sub-family of DC programs, which was proposed by Flener and Deville [20], was summarized.

Thus, firstly in Section 4.1.1, I give two tupling generalization schemas. The time and space complexity analysis of the programs of these generalization schemas is discussed in Section 4.1.2.

4.1.1 Tupling Generalization Schemas

I do not separate the tupling generalized program schema pattern into two schema patterns, as the $TGLR$ schema pattern and the $TGRL$ schema pattern, like I did for the DC program schema patterns, since one of the objectives of tupling generalization is to reduce the number of recursive calls of the intended relation by generalizing the problem to a new single-recursive relation (i.e. the composition of the result parameter reduces to head-tail composition), which is achieved by generalizing the structure (or: type) of the induction parameter of the input relation. So, it is not too much helpful and meaningful to give two different tupling generalized program schema patterns, although it is possible. Therefore, in this section, I give only two TG transformation schemas (one for each DC program schema pattern), rather than four.

Take a relation r, which has the specification S_r as:

$$\forall X : \mathcal{X}.\ \forall Y : \mathcal{Y}.\ \mathcal{I}_r(X) \Rightarrow [r(X,Y) \Leftrightarrow \mathcal{O}_r(X,Y)]$$

where $\mathcal{X}$ and $\mathcal{Y}$ denote the types of X and Y, $\mathcal{I}_r(X)$ denotes the *input condition* that must be fulfilled before the execution of the procedure, and $\mathcal{O}_r(X,Y)$ denotes the *output condition* that will be fulfilled after the execution.

If a program is given for r as an instance of $DCLR$ (or $DCRL$), then the specification of the new tupling generalized problem of r, namely $S_{r\_tupling}$ is:

$$\forall Xs : list(\mathcal{X}).\ \forall Y : \mathcal{Y}.\ (\forall X : \mathcal{X}.\ X \in Xs \Rightarrow \mathcal{I}_r(X)) \Rightarrow$$

$$[r\_tupling(Xs, Y) \Leftrightarrow (Xs = [\,] \wedge Y = e)$$

$$\vee\ (Xs = [X_1, X_2, \ldots, X_n] \wedge \bigwedge_{i=1}^{n} \mathcal{O}_r(X_i, Y_i)\ \wedge I_1 = Y_1$$

$$\wedge \bigwedge_{i=2}^{n} \mathcal{O}_c(I_{i-1}, Y_i, I_i)\ \wedge Y = I_n)]$$

where $\mathcal{O}_c$ is the output condition of *compose* and $n \geq 1$.

The tupling generalization schemas are:

TG_1 : $\langle$ $DCLR$, TG, A_{t1}, O_{t112}, O_{t121} $\rangle$ where

A_{t1} : - *compose* is associative
- *compose* has e as the left and right identity element, where e appears in $DCLR$ and TG
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \Rightarrow [\neg minimal(X) \Leftrightarrow nonMinimal(X)]$

O_{t112} : partial evaluation of the conjunction $process(HX, HY), compose(HY, TY, Y)$ results in the introduction of a non-recursive relation

O_{t121} : partial evaluation of the conjunction $process(HX, HY), compose(I_{p-1}, HY, I_p)$ results in the introduction of a non-recursive relation

TG_2 : $\langle$ $DCRL$, TG, A_{t2}, O_{t212}, O_{t221} $\rangle$ where

A_{t2} : - *compose* is associative
- *compose* has e as the left and right identity element, where e appears in $DCRL$ and TG
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \Rightarrow [\neg minimal(X) \Leftrightarrow nonMinimal(X)]$

O_{t212} : partial evaluation of the conjunction

$process(HX, HY), compose(HY, TY, Y)$
results in the introduction of a non-recursive relation

O_{t221} : partial evaluation of the conjunction
$process(HX, HY), compose(HY, I_p, I_{p-1})$
results in the introduction of a non-recursive relation

where the template of the common schema pattern TG is Logic Program Template 3 below:

Logic Program Template 3

$r(X, Y) \leftarrow$

$r\_tupling([X], Y)$

$r\_tupling(Xs, Y) \leftarrow$

$Xs = [\,],$

$Y = e$

$r\_tupling(Xs, Y) \leftarrow$

$Xs = [X|TXs],$

$minimal(X),$

$r\_tupling(TXs, TY),$

$solve(X, HY),$

$compose(HY, TY, Y)$

$r\_tupling(Xs, Y) \leftarrow$

$Xs = [X|TXs],$

$nonMinimal(X),$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$minimal(TX_1), \ldots, minimal(TX_t),$$

$$r\_tupling(TXs, TY),$$

$$process(HX, HY),$$

$$compose(HY, TY, Y)$$

$$r\_tupling(Xs, Y) \leftarrow$$

$$Xs = [X|TXs],$$

$$nonMinimal(X),$$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$minimal(TX_1), \ldots, minimal(TX_{p-1}),$$

$$(nonMinimal(TX_p); \ldots; nonMinimal(TX_t)),$$

$$r\_tupling([TX_p, \ldots, TX_t|TXs], TY),$$

$$process(HX, HY),$$

$$compose(HY, TY, Y)$$

$$r\_tupling(Xs, Y) \leftarrow$$

$$Xs = [X|TXs],$$

$$nonMinimal(X),$$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$(nonMinimal(TX_1); \ldots; nonMinimal(TX_{p-1})),$$

$$minimal(TX_p), \ldots, minimal(TX_t),$$

$$minimal(U_1), \ldots, minimal(U_{p-1}),$$

$$decompose(N, HX, U_1, \ldots, U_{p-1}, TX_p, \ldots, TX_t),$$

$$r\_tupling([TX_1, \ldots, TX_{p-1}, N|TXs], Y)$$

$$r\_tupling(Xs, Y) \leftarrow$$

$$Xs = [X|TXs],$$

$$nonMinimal(X),$$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$(nonMinimal(TX_1); \ldots; nonMinimal(TX_{p-1})),$$

$$(nonMinimal(TX_p); \ldots; nonMinimal(TX_t)),$$

$$minimal(U_1), \ldots, minimal(U_t),$$

$$decompose(N, HX, U_1, \ldots, U_t),$$

$$r\_tupling([TX_1, \ldots, TX_{p-1}, N, TX_p, \ldots, TX_t|TXs], Y)$$

Note that, in the TG template, I have only used $= /2$, which is a built-in of all the logic program compilers, and all the open predicates of $DCLR$ (or $DCRL$), and no other new predicates. In other words, Lavoisier's Principle ("rien ne se crée, rien ne se perd") also applies to transformation schemas.

The applicability conditions of TG_1 (respectively, TG_2) ensure the equivalence of the $DCLR$ (respectively, $DCRL$) and TG programs for a given problem. The optimizability conditions ensure that the output program of these generalization schemas is more efficient than the input program. The optimizability conditions, together with some of the applicability conditions, check whether the *compose* calls in the template TG can be eliminated. In the second clause of *r_tupling*, the conjunction $solve(X, HY), compose(HY, TY, Y)$ can be simplified to $Y = A$, if relation r maps the minimal form of X into e, and e is

also the right identity element of *compose*. This is already checked by the second and third applicability conditions of TG_1 and TG_2. In the third and fourth clauses of *r_tupling*, the conjunction $process(HX, HY), compose(HY, TY, Y)$ can be partially evaluated, resulting in the disappearance of that call to *compose*, and thus in a merging of the *compose* loop into the r loop in the template $DCLR$ (or $DCRL$). The optimizability condition O_{t112} (or O_{t212}) checks whether this *compose* call can be eliminated in the corresponding clauses.

In this section, I illustrate tupling generalization using the TG generalization schemas on the $prefix\_flat$ and $infix\_flat$ problems.

Example 29 The specification of the $prefix\_flat$ problem, and its $DCLR$ and $DCRL$ programs are in Example 26 in Chapter 3. These DC programs can be tupling generalized both resulting in Program 7 below, since the open relations in the schema pattern $DCLR$ (respectively, $DCRL$) satisfy the applicability conditions A_{t1} (respectively, A_{t2}), and the optimizability conditions O_{t112} (respectively, O_{t212}) of TG_1 (respectively, TG_2). So, the $prefix\_flat$ problem can be tupling generalized, resulting in the specification of a program for tupling generalized problem of $prefix\_flat$ as:

$prefix\_flat\_t(Bs, F)$ iff F is the concatenation of the prefix representations of the elements in binary tree list Bs.

The word "concatenation" in the specification above reflects the composition done by the *compose* operators in $S_{r\_tupling}$. Then, Program 7 below is the tupling generalized program for $prefix\_flat$ as an instance of TG.

$$prefix\_flat(B, F) \leftarrow$$

$$prefix\_flat\_t([B], F)$$

$$prefix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [\,],$$

$$F = [\,]$$

$$prefix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = void,$$

$$prefix\_flat\_t(TBs, TF),$$

$$HF = [\,],$$

$$append(HF, TF, F)$$

$$prefix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = void, R = void,$$

$$prefix\_flat\_t(TBs, TF),$$

$$HF = [E], append(HF, TF, F)$$

$$prefix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$(L = bt(\_, \_, \_); R = bt(\_, \_, \_)),$$

$$prefix\_flat\_t([L, R|TBs], TF),$$

$$HF = [E], append(HF, TF, F)$$

Logic Program 7

The reader may notice that in Program 7 we have only five clauses, although we have seven clauses in the TG template. The fifth and sixth clauses of $prefix\_flat\_t$ reduce to $false$, during particularization, since the disjunction $(nonMinimal(TX_1); \ldots; nonMinimal(TX_{p-1}))$ in the fifth and sixth clauses of $r\_tupling$ in TG becomes an empty disjunction (i.e. $false$), because of $p = 1$ in $prefix\_flat$. □

Example 30 The specification of the $infix\_flat$ problem, and its $DCLR$ and $DCRL$ programs are in Example 27 in Chapter 3. The $infix\_flat$ problem can also be tupling generalized using the TG transformation schemas above resulting in Program 8 below, since the $infix\_flat$ and $prefix\_flat$ DC programs have the same open relations, which satisfy the applicability and optimizability conditions of the TG transformation schemas. So, the specification of the tupling generalized problem of $infix\_flat$ is:

$infix\_flat\_t(Bs, F)$ iff F is the concatenation of the infix representations of the elements in binary tree list Bs.

Program 8 below is the tupling generalized program for $infix\_flat$ as an instance of TG.

$$infix\_flat(B, F) \leftarrow$$

$$infix\_flat\_t([B], F)$$

$$infix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [\,],$$

$$F = [\,]$$

$$infix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = void,$$

$$infix\_flat\_t(TBs, TF),$$

$$HF = [\,],$$

$$append(HF, TF, F)$$

$$infix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = void, R = void,$$

$$infix\_flat\_t(TBs, TF),$$

$$HF = [E], append(HF, TF, F)$$

$$infix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = void,$$

$$R = bt(\_, \_, \_),$$

$$infix\_flat\_t([R|TBs], TF),$$

$$HF = [E], append(HF, TF, F)$$

$$infix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = bt(\_, \_, \_),$$

$$R = void,$$

$$U_L = void,$$

$$N = bt(U_L, E, R),$$

$$infix\_flat\_t([L, N|TBs], F)$$

$$infix\_flat\_t(Bs, F) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$(L = bt(\_, \_, \_); R = bt(\_, \_, \_)),$$

$$U_L = void, U_R = void,$$

$$N = bt(U_L, E, U_R),$$

$$infix\_flat\_t([L, N, R|TBs], F)$$

Logic Program 8

□

Although the tupling generalization schemas are constructed to tupling generalize DC programs (i.e. to transform DC programs into TG programs), these schemas can also be used in the reverse direction, such that they can be used to transform TG programs into DC programs, if the optimizability conditions for the corresponding DC program schema pattern are satisfied, since the applicability conditions hold in both directions. These generalization schemas can be used in the reverse direction, since it is sometimes possible that we have a TG program, which is not efficient (e.g., the *compose* call in the non-minimal case of *r_tupling* cannot be eliminated), and we want to transform it to a more efficient program, which will be a $DCLR$ program (most probably), since it is possible to eliminate the *compose* call in the non-minimal case in $DCLR$, because of the verification of the optimizability conditions O_{t112} of TG_1. Further discussion of this can be found in Section 4.1.2.

4.1.2 Complexity Analysis

In this section, I present the complexity analysis of the input and output programs of the tupling generalization schemas, and I will use this complexity analysis to discuss the efficiency gain obtained by the tupling generalization schemas.

I use the $infix\_flat$ problem, whose informal specification is given in Chapter 3. If the $prefix\_flat$ and $postfix\_flat$ DC programs are also tupling generalized and the complexity analysis is done for these problems, similar results will be obtained. Therefore, I consider only the programs for $infix\_flat$ here. Logic Program 9 below is the optimized version of Program 3, which is an instance of the program schema pattern $DCLR$ for the $infix\_flat$ problem.

$$infix\_flat(B, F) \leftarrow$$

$$B = void, F = [\,]$$

$$infix\_flat(B, F) \leftarrow$$

$$B = bt(L, E, R),$$

$$infix\_flat(L, FL), infix\_flat(R, FR),$$

$$append(FL, [E], I), append(I, FR, F)$$

Logic Program 9

Logic Program 10 below is the optimized version of Program 4, which is an instance of the program schema pattern $DCRL$ for the $infix\_flat$ problem.

$$infix\_flat(B, F) \leftarrow$$

$$B = void, F = [\,]$$

$$infix\_flat(B, F) \leftarrow$$

$$B = bt(L, E, R),$$

$$infix\_flat(L, FL), infix\_flat(R, FR),$$

$$I = [E|FR], append(FL, I, F)$$

Logic Program 10

If n is the number of elements in tree B, then Programs 9 and 10 have $O(n^2)$ time complexity in the worst case, because composition is done through *append*, whose complexity is linear in the number of elements in its first parameter. If we analyze the programs above in terms of space, and we assume h is the height of B, then these programs build a stack of h pairs of recursive calls, and create $2n$ intermediate data structures. However, since the conjunction $HF = [E], append(HF, FR, I)$ in Program 4 could be partially evaluated, resulting in the equality $I = [E|FR]$, Program 10 has a better time complexity than Program 9 by a constant factor, which is not negligible in most cases.

Program 8 in Section 4.1.1, which is an instance of the schema pattern TG for the $infix\_flat$ problem, can be optimized, resulting in Program 11 below:

$$
\begin{aligned}
&infix\_flat(B, F) \leftarrow \\
&\quad infix\_flat\_t([B], F) \\
&infix\_flat\_t(Bs, F) \leftarrow \\
&\quad Bs = [\,], F = [\,] \\
&infix\_flat\_t(Bs, F) \leftarrow \\
&\quad Bs = [B|TBs], \\
&\quad B = void, \\
&\quad infix\_flat\_t(TBs, F) \\
&infix\_flat\_t(Bs, F) \leftarrow \\
&\quad Bs = [B|TBs], \\
&\quad B = bt(L, E, R), \\
&\quad L = void, R = void, \\
&\quad infix\_flat\_t(TBs, TF), \\
&\quad F = [E|TF] \\
&infix\_flat\_t(Bs, F) \leftarrow \\
&\quad Bs = [B|TBs], \\
&\quad B = bt(L, E, R), \\
&\quad L = void,
\end{aligned}
$$

$$
\begin{aligned}
& R = bt(\_, \_, \_), \\
& infix\_flat\_t([R|TBs], TF), \\
& F = [E|TF] \\
infix\_flat\_t(Bs, F) \leftarrow & \\
& Bs = [B|TBs], \\
& B = bt(L, E, R), \\
& L = bt(\_, \_, \_), \\
& R = void, \\
& infix\_flat\_t([L, bt(void, E, R)|TBs], F) \\
infix\_flat\_t(Bs, F) \leftarrow & \\
& Bs = [B|TBs], \\
& B = bt(L, E, R), \\
& (L = bt(\_, \_, \_); R = bt(\_, \_, \_)), \\
& infix\_flat\_t([L, bt(void, E, void), R|TBs], F)
\end{aligned}
$$

Logic Program 11

In Program 11, the calls to *append* have disappeared: the *append* loops have been merged into the $infix\_flat$ loop in the templates $DCLR$ or $DCRL$, and we have a linear time program. However, the space complexity of Program 11 is worse than for the DC programs for the $infix\_flat$ problem: this program builds a stack of $O(n)$ recursive calls, and it creates as many intermediate data structures. Fortunately, this program can be made tail recursive in the mode (*in*,*out*), as the last five clauses are mutually exclusive.

Therefore, for input DC programs like the programs given for $infix\_flat$, which use *append* as the *compose* operator, the tupling generalization schemas result in an efficient TG program, since the optimizability conditions of these tupling generalization schemas are satisfied.

It is also possible that we have a program, which is an instance of the schema pattern TG, where the optimizability conditions O_{t112} (or O_{t212}) are not satisfied, which means that the *compose* calls in some of the clauses of *r_tupling* can not be eliminated. So, this TG program can be worse than the corresponding $DCLR$ program. In these situations, the tupling generalization schemas can be used in the reverse direction (i.e., to transform TG programs into DC programs), and we will have a more efficient DC program as the output program of the transformation.

4.2 Descending Generalization

I explained the idea of descending generalization in Section 2.1.6, and the descending generalization process was illustrated in Example 22 for the list reversing problem. In Section 2.2.2, I presented the automation process of descending generalization of a restricted sub-family of DC programs, achieved by Flener and Deville in [20].

Descending generalization can also be called the *accumulation strategy* (presented in Section 2.2.1 by giving example constructions of this strategy both in functional programming and in logic programming), since an accumulator parameter is introduced by descending generalization, and it is progressively extended to the final result. Descending generalization can also be seen as transformation towards *difference-structure* manipulation. In Section 2.1.6, the pair of parameters R and A in Example 22 can also be thought as representing the difference-list $R \backslash A$, which itself represents the difference between lists R and A, where A is a suffix of R. But descending generalization yields something more general than transformation to difference-list manipulation, since any form of difference-structures can be created by descending generalization.

In Section 4.2.1, I give four descending generalization schemas, and explain how they ensure correct and efficient descending generalization in program transformation. The time and space complexity analyses of the program schemas of these generalization schemas are discussed in Section 4.2.2.

4.2.1 Descending Generalization Schemas

Four descending generalization schemas (two for each DC program schema pattern) are given. Since the conditions of each descending generalization schema are different, the process of choosing the appropriate generalization schema for the input DC program is done only by checking the conditions, and then the eureka [20] (i.e. the specification of the generalized problem) comes for free.

The reason why we call the descendingly generalized (DG) program schema patterns '*DGLR*' and '*DGRL*' is similar to the reason why we call the divide-and-conquer program schema patterns *DCLR* and *DCRL*, respectively. In descending generalization, the composition ordering for extending the accumulator parameter in the template *DGLR* is from *left-to-right* (LR) and the composition ordering for extending the accumulator parameter in the template *DGRL* is from *right-to-left* (RL).

The first two descending generalization schemas are:

DG_1 : $\langle$ *DCLR*, *DGLR*, A_{dg1}, O_{dg112}, O_{dg121} $\rangle$ where

A_{dg1} : - *compose* is associative
- *compose* has e as the left identity element,
where e appears in *DCLR* and *DGLR*

O_{dg112} : - *compose* has e as the right identity element,
where e appears in *DCLR* and *DGLR*
and $\mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction
$process(HX, HY), compose(A_{p-1}, HY, A_p)$
results in the introduction of a non-recursive relation

O_{dg121} : - partial evaluation of the conjunction

$process(HX, HY), compose(I_{p-1}, HY, I_p)$
results in the introduction of a non-recursive relation

DG_4 : $\langle$ $DCRL$, $DGLR$, A_{dg4}, O_{dg412}, $O_{dg421}\rangle$ where
A_{dg4} : - *compose* is associative
- *compose* has e as the left and right identity element,
where e appears in $DCRL$ and $DGLR$
O_{dg412} : - $\mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction
$process(HX, HY), compose(A_{p-1}, HY, A_p)$
results in the introduction of a non-recursive relation
O_{dg421} : - partial evaluation of the conjunction
$process(HX, HY), compose(HY, I_p, I_{p-1})$
results in the introduction of a non-recursive relation

These schemas have the *same* formal specification (i.e. eureka) for the relation $r\_descending_1$:

$$\forall X : \mathcal{X}.\ \forall Y, A : \mathcal{Y}.\ \ \mathcal{I}_r(X) \Rightarrow$$

$$[r\_descending_1(X, Y, A) \ \Leftrightarrow\ \exists S : \mathcal{Y}.\ \mathcal{O}_r(X, S) \wedge \mathcal{O}_c(A, S, Y)]$$

where $\mathcal{O}_c$ is the output condition of *compose*, and $\mathcal{O}_r$ is the output condition of r, the initial problem. Template 4 below is the template of the common schema pattern $DGLR$ of DG_1 and DG_4.

Logic Program Template 4

$r(X, Y) \leftarrow$

$r\_descending_1(X, Y, e)$

$r\_descending_1(X, Y, A) \leftarrow$

$minimal(X),$

$solve(X, S), compose(A, S, Y)$

$r\_descending_1(X, Y, A) \leftarrow$

$nonMinimal(X),$

$decompose(X, HX, TX_1, \ldots, TX_t),$

$compose(A, e, A_0),$

$r\_descending_1(TX_1, A_1, A_0), \ldots, r\_descending_1(TX_{p-1}, A_{p-1}, A_{p-2}),$

$process(HX, HY), compose(A_{p-1}, HY, A_p),$

$r\_descending_1(TX_p, A_{p+1}, A_p), \ldots, r\_descending_1(TX_t, A_{t+1}, A_t),$

$Y = A_{t+1}$

Note that, in the $DGLR$ template, I have only used all the open predicates of $DCLR$ (or $DCRL$), and no other new predicates (other than primitive $= /2$).

For an input program, one of these generalization schemas is applied, if both the applicability and the optimizability conditions of the selected generalization schema are satisfied. The applicability conditions of these two generalization schemas differ, since the composition ordering is also changed from RL to LR in DG_4.

If the input program is a $DCLR$ (respectively, $DCRL$) program for the generalization schema DG_1 (respectively, DG_4) and the applicability conditions are satisfied, then the optimizability conditions O_{dg112} (respectively, O_{dg412}) have to be satisfied to yield a more efficient output $DGLR$ program.

In the minimal case of $r\_descending_1$, the simplification of the conjunction $solve(X, S), compose(A, S, Y)$ can result in $Y = A$, if relation r maps the minimal form of X into e, and e is also the right identity element of *compose*. This equality can be further compiled into the head of the minimal clause. The first optimizability condition of DG_1 (or DG_4) is defined to check whether the *compose* call in the minimal case of $r\_descending_1$ can be eliminated.

In the non-minimal case of $r\_descending_1$, the atom $compose(A, e, A_0)$ can be further simplified to the equality $A = A_0$, if *compose* has e as the right identity element. The conjunction $process(HX, HY), compose(A_{p-1}, HY, A_p)$ can be partially evaluated, resulting in the disappearance of that call to *compose*, and thus in a merging of the *compose* loop into the r loop in the template $DCLR$ (or $DCRL$). The second optimizability condition of DG_1 (or DG_4) is defined to check whether the elimination of the *compose* call in the non-minimal case of $r\_descending_1$ is possible.

I illustrate descending generalization on the $infix\_flat$ problem. The informal specification of the $infix\_flat$ problem, and its $DCLR$ and $DCRL$ programs are in Example 27 of Chapter 3.

Example 31 The specification of a program for the LR descendingly generalized problem of $infix\_flat$ is:

$infix\_flat\_desc_1(B, F, A)$ iff list F is the concatenation of list A and the infix representation of binary tree B.

Program 12 is the program for $infix\_flat$ as an instance of $DGLR$ for $t = 2$ and $p = 2$.

$$infix\_flat(B, F) \leftarrow$$

$$\quad infix\_flat\_desc_1(B, F, [\,])$$

$$infix\_flat\_desc_1(B, F, A) \leftarrow$$

$$\quad B = void,$$

$$\quad S = [\,], append(A, S, F)$$

$$infix\_flat\_desc_1(B, F, A) \leftarrow$$

$$\quad B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$append(A, [\,], A_0),$$

$$infix\_flat\_desc_1(L, A_1, A_0),$$

$$HF = [E], append(A_1, HF, A_2),$$

$$infix\_flat\_desc_1(R, A_3, A_2),$$

$$F = A_3$$

Logic Program 12

Since the applicability conditions of DG_1 (respectively, DG_4) are satisfied for the input $DCLR$ (respectively, $DCRL$) $infix\_flat$ program, the descendingly generalized $infix\_flat$ program can be Program 12. However, for this problem, descending generalization of the $infix\_flat$ programs with the above DG transformation schemas cannot be done, since the optimizability conditions of DG_1 (respectively, DG_4) are not satisfied by the open relations of $infix\_flat$. In the non-minimal case of $infix\_flat\_desc_1$, partial evaluation of the conjunction $HF = [E], append(A_1, HF, A_2)$ does *not* result in the introduction of a non-recursive relation, because of properties of *append* (actually, due to the inductive definition of lists). Moreover, *append* is called each time with the accumulator parameter, which increases in length, as the input induction parameter, which shows that this program is not a good choice as an output descendingly generalized program for this problem. So, the optimizability conditions are really needed. □

The other two descending generalization schemas are:

DG_2 : $\langle$ $DCLR$, $DGRL$, A_{dg2}, O_{dg212}, O_{dg221} $\rangle$ where

A_{dg2} : - *compose* is associative

- *compose* has e as the left and right identity element, where e appears in $DCLR$ and $DGRL$

O_{dg212} : - $\mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction
$process(HX, HY), compose(HY, A_p, A_{p-1})$
results in the introduction of a non-recursive relation

O_{dg221} : - partial evaluation of the conjunction
$process(HX, HY), compose(I_{p-1}, HY, I_p)$
results in the introduction of a non-recursive relation

DG_3 : $\langle$ $DCRL$, $DGRL$, A_{dg3}, O_{dg312}, $O_{dg321}\rangle$ where

A_{dg3} : - *compose* is associative
- *compose* has e as the right identity element,
where e appears in $DCRL$ and $DGRL$

O_{dg312} : - *compose* has e as the left identity element,
where e appears in $DCLR$ and $DGRL$
and $\mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction
$process(HX, HY), compose(HY, A_p, A_{p-1})$
results in the introduction of a non-recursive relation

O_{dg321} : - partial evaluation of the conjunction
$process(HX, HY), compose(HY, I_p, I_{p-1})$
results in the introduction of a non-recursive relation

These schemas have the *same* formal specification (i.e. eureka) for the relation $r\_descending_2$:

$$\forall X : \mathcal{X}.\ \forall Y, A : \mathcal{Y}.\ \ \mathcal{I}_r(X) \Rightarrow$$

$$[r\_descending_2(X, Y, A) \ \Leftrightarrow\ \exists S : \mathcal{Y}.\ \mathcal{O}_r(X, S) \wedge \mathcal{O}_c(S, A, Y)]$$

where $\mathcal{O}_c$ is the output condition of *compose*, and $\mathcal{O}_r$ is the output condition of r, the initial problem. Template 5 below is the template of the common schema pattern $DGRL$ of DG_2 and DG_3.

Logic Program Template 5

$r(X, Y) \leftarrow$

$$r\_descending_2(X, Y, e)$$

$$r\_descending_2(X, Y, A) \leftarrow$$

$$minimal(X),$$

$$solve(X, S), compose(S, A, Y)$$

$$r\_descending_2(X, Y, A) \leftarrow$$

$$nonMinimal(X),$$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$compose(e, A, A_{t+1}),$$

$$r\_descending_2(TX_t, A_t, A_{t+1}), \ldots, r\_descending_2(TX_p, A_p, A_{p+1}),$$

$$process(HX, HY), compose(HY, A_p, A_{p-1}),$$

$$r\_descending_2(TX_{p-1}, A_{p-2}, A_{p-1}), \ldots, r\_descending_2(TX_1, A_0, A_1),$$

$$Y = A_0$$

Again, in the $DGRL$ template, I have only used all the open predicates of $DCLR$ (or $DCRL$), and no other new predicates (other than the primitive $= /2$).

If the input program is a $DCLR$ (respectively, $DCRL$) program for the generalization schema DG_2 (respectively, DG_3) and the applicability conditions are satisfied, then the optimizability conditions O_{dg212} (respectively, O_{dg312}) have to be satisfied to yield a more efficient output $DGRL$ program.

In the minimal case of $r\_descending_2$, the simplification of the conjunction $solve(X, S), compose(S, A, Y)$ can result in $Y = A$, if relation r maps the minimal form of X into e, and e is also the left identity element of *compose*. This equality can be further compiled into the head of the minimal clause. The first optimizability condition of DG_2 (or DG_3) is defined to check whether the

compose call in the minimal case of $r\_descending_2$ can be eliminated.

In the non-minimal case of $r\_descending_2$, the atom $compose(A_0, e, Y)$ can be further simplified to the equality $Y = A_0$, if *compose* has e as the left identity element. The conjunction $process(HX, HY), compose(HY, A_p, A_{p-1})$ can be partially evaluated, resulting in the disappearance of that call to *compose*, and thus in a merging of the *compose* loop into the r loop in the template $DCLR$ (or $DCRL$). The second optimizability condition of DG_2 (or DG_3) is defined to check whether the elimination of the *compose* call in the non-minimal case of $r\_descending_2$ is possible.

Example 32 The specification of a program for the RL descendingly generalized problem of $infix\_flat$ is:

$infix\_flat\_desc_2(B, F, A)$ iff list F is the concatenation of the infix representation of binary tree B and list A.

Program 13 is the program for $infix\_flat$ as an instance of $DGRL$ for $t = 2$ and $p = 2$.

$$infix\_flat(B, F) \leftarrow$$

$$\quad infix\_flat\_desc_2(B, F, [\,])$$

$$infix\_flat\_desc_2(B, F, A) \leftarrow$$

$$\quad B = void,$$

$$\quad S = [\,], append(S, A, F)$$

$$infix\_flat\_desc_2(B, F, A) \leftarrow$$

$$\quad B = bt(\_, \_, \_),$$

$$\quad B = bt(L, E, R),$$

$$append(e, A, A_3),$$

$$infix\_flat\_desc_2(R, A_2, A_3),$$

$$HF = [E], append(HF, A_2, A_1),$$

$$infix\_flat\_desc_2(L, A_0, A_1),$$

$$F = A_0$$

Logic Program 13

Since both the applicability conditions and the optimizability conditions of DG_2 (respectively, DG_3) are satisfied for the input $DCLR$ (respectively, $DCRL$) $infix\_flat$ program, both descending generalizations of the $infix\_flat$ programs result in Program 13. The partial evaluation of the conjunction $HF = [E], append(HF, A_2, A_1)$ in the non-minimal case of $infix\_flat\_desc_2$ results in a call to $= /2$, as $A_1 = [E|A_2]$. □

Although the descending generalization schemas are constructed to descendingly generalize DC programs (i.e. to transform DC programs into DG programs), these schemas can also be used in the reverse direction, such that they can be used to transform DG programs into DC programs, if the optimizability conditions for the corresponding DC program schema pattern are satisfied, since the applicability conditions hold in both directions. If we have Program 12 for the $infix\_flat$ problem, and we want to transform it into a more efficient program, then the DC programs can be the best candidates if we have the descending generalization schemas above. This last sentence will be better understood in Section 4.2.2.

4.2.2 Complexity Analysis

In this section, I present the complexity analysis of the input and output programs of the descending generalization schemas, and I will use this complexity

analysis to discuss the efficiency gain obtained by the descending generalization schemas.

For the complexity analysis of the programs of the descending generalization schemas, I again use the $infix\_flat$ problem, which was also used in Section 4.1.2 for the discussion of the tupling generalization schemas. I use Programs 9 and 10 in Section 4.1.2, which are the optimized versions of the $infix\_flat$ $DCLR$ and $DCRL$ programs.

As I discussed in Section 4.1.2, these programs have $O(n^2)$ time complexity in the worst case, if n is the number of elements in tree B. We analyzed these programs in terms of space in Section 4.1.2 where it was shown that their space complexities are also not very good. However, the RL version is better than the LR version, since the *append* call in the non-minimal case of the RL version can be eliminated.

Program 12 in Section 4.2.1 can be optimized, resulting in Program 14 below.

$$
\begin{aligned}
&infix\_flat(B, F) \leftarrow \\
&\quad infix\_flat\_desc_1(B, F, [\,]) \\
&infix\_flat\_desc_1(B, F, A) \leftarrow \\
&\quad B = void, F = A \\
&infix\_flat\_desc_1(B, F, A) \leftarrow \\
&\quad B = bt(L, E, R), \\
&\quad infix\_flat\_desc_1(L, A_1, A), \\
&\quad append(A_1, [E], A_2), \\
&\quad infix\_flat\_desc_1(R, F, A_2)
\end{aligned}
$$

Logic Program 14

As I discussed in Example 31, in Program 14 the calls to *append* cannot be fully eliminated. Moreover, if we compare the time used by the *append* calls in Program 9 (or 10) and Program 14, the time used by the call $append(A_1, [E], A_2)$ in Program 14 is higher than the time used by the *append* call in Program 9 (or 10) by a (nonnegligible) constant factor. This time increase is caused by the increase in the length of the accumulator list, which is the induction parameter of *append*. So, Program 14 is less efficient than Program 9 (or 10), although its time complexity is also $O(n^2)$ in the worst case.

Program 13 in Section 4.2.1 can be optimized, resulting in Program 15 below.

$$infix\_flat(B, F) \leftarrow$$

$$\quad infix\_flat\_desc_2(B, F, [\,])$$

$$infix\_flat\_desc_2(B, F, A) \leftarrow$$

$$\quad B = void, F = A$$

$$infix\_flat\_desc_2(B, F, A) \leftarrow$$

$$\quad B = bt(L, E, R),$$

$$\quad infix\_flat\_desc_2(R, NewA, A),$$

$$\quad infix\_flat\_desc_2(L, F, [E|NewA])$$

Logic Program 15

In Program 15, the calls to *append* have disappeared, and we have a linear time program. The space complexity of Program 15 is also better than the space complexities of Programs 9 and 10. Since an accumulator parameter is used, this program creates only h intermediate data structures, although it builds a

stack of h pairs of recursive calls. However, the program for $infix\_flat\_desc_2$ can be made semi-tail recursive in the mode (in,out,in).

Therefore, for the input DC programs like the programs given for $infix\_flat$, which use *append* as the *compose* operator, the descending generalization schemas DG_2 and DG_3 result in more efficient programs than the descending generalization schemas DG_1 and DG_4. If the *compose* operator of the input DC program that is an instance of the $DCLR$ template (or $DCRL$) satisfies the optimizability conditions of DG_1 (or DG_4), then obviously the descending generalization schema DG_1 (or DG_4) will result in more efficient programs than the descending generalization schema DG_2 (or DG_3).

If the descending generalization schemas are used in the reverse direction (i.e., to transform DG programs into DC programs), then for instance Programs 9 and 10 are more efficient in time and space than Program 14. So, it is still possible to gain efficiency by using the descending generalization schemas in the reverse direction. However, the DG_1 generalization schema (respectively, the DG_2 generalization schema), for an input program that is an instance of the $DGLR$ schema pattern (respectively, $DGRL$ schema pattern), can be better than the DG_4 generalization schema (respectively, the DG_3 generalization schema), for an input program that is an instance of the $DGLR$ schema pattern (respectively, $DGRL$ schema pattern), or vice versa, depending on the optimizability conditions of the descending generalization schemas for the input programs that are instances of the $DGLR$ schema pattern (respectively, $DGRL$ schema pattern).

4.3 Simultaneous Tupling-and-Descending Generalization

While working on constructing possible generalized generalization schemas for different input program schemas, we also tried to apply descending generalization to a tupling generalized problem, and vice versa. The generalization schemas that we explain in this section are the results of this work. We call

them simultaneous tupling-and-descending generalization schemas, although the reader may notice by looking at the specification of the generalized problem that the process may also be thought of as applying descending generalization to a tupling generalized problem.

As I did while explaining the tupling and descending generalization schemas, I will first give the simultaneous tupling-and-descending generalization schemas in Section 4.3.1. Then, I will discuss the efficiency gain that can be obtained with these generalization schemas by using the time and space complexity analysis of the programs of these generalization schemas in Section 4.3.2.

4.3.1 Simultaneous Tupling-and-Descending Generalization Schemas

Like I did in Section 4.2.1 for descending generalization, four simultaneous tupling-and-descending generalization schemas will be given; two for each DC program schema pattern. The first two simultaneous tupling-and-descending generalization schemas are:

TDG_1 : $\langle\ DCLR, TDGLR, A_{td1}, O_{td112}, O_{td121}\ \rangle$ where

A_{td1} : - *compose* is associative
- *compose* has e as the left and right identity element, where e appears in $DCLR$ and $TDGLR$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \Rightarrow [\neg minimal(X) \Leftrightarrow nonMinimal(X)]$

O_{td112} : partial evaluation of the conjunction $process(HX, HY), compose(A, HY, NewA)$ results in the introduction of a non-recursive relation

O_{td121} : partial evaluation of the conjunction $process(HX, HY), compose(I_{p-1}, HY, I_p)$ results in the introduction of a non-recursive relation

TDG_4 : $\langle\ DCRL, TDGLR, A_{td4}, O_{td412}, O_{td421}\ \rangle$ where

A_{td4} : - *compose* is associative

- *compose* has e as the left and right identity element,
where e appears in $DCRL$ and $TDGLR$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \Rightarrow [\neg minimal(X) \Leftrightarrow nonMinimal(X)]$

O_{td412} : partial evaluation of the conjunction
$process(HX, HY), compose(A, HY, NewA)$
results in the introduction of a non-recursive relation

O_{td421} : partial evaluation of the conjunction
$process(HX, HY), compose(HY, I_p, I_{p-1})$
results in the introduction of a non-recursive relation

They have the *same* formal specification, namely $S_{r\_td_1}$, for the generalized problem:

$$\forall Xs : list(\mathcal{X}).\forall Y, A : \mathcal{Y}.\ (\forall X : \mathcal{X}.\ X \in Xs \Rightarrow \mathcal{I}_r(X)) \Rightarrow$$

$$[r\_td_1(Xs, Y, A) \Leftrightarrow (Xs = [\,] \wedge Y = A)$$

$$\vee (Xs = [X_1, X_2, \ldots, X_q] \wedge \bigwedge_{i=1}^{q} \mathcal{O}_r(X_i, Y_i)\ \wedge I_1 = Y_1 \wedge$$

$$\bigwedge_{i=2}^{q} \mathcal{O}_c(I_{i-1}, Y_i, I_i)\ \wedge \mathcal{O}_c(A, I_q, I_{q+1}) \wedge Y = I_{q+1})]$$

where $\mathcal{O}_c$ is the output condition of *compose*, and $\mathcal{O}_r$ is the output condition of r, and $q \geq 1$. Template 6 below is the template of the common schema pattern $TDGLR$ of TDG_1 and TDG_4.

Logic Program Template 6

$$r(X, Y) \leftarrow$$

$$r\_td_1([X], Y, e)$$

$$r\_td_1(Xs, Y, A) \leftarrow$$

$$Xs = [\,],$$

$$Y = A$$

$r\_td_1(Xs, Y, A) \leftarrow$

$Xs = [X|TXs],$

$minimal(X),$

$solve(X, HY),$

$compose(A, HY, NewA),$

$r\_td_1(TXs, Y, NewA)$

$r\_td_1(Xs, Y, A) \leftarrow$

$Xs = [X|TXs],$

$nonMinimal(X),$

$decompose(X, HX, TX_1, \ldots, TX_t),$

$minimal(TX_1), \ldots, minimal(TX_t),$

$process(HX, HY), compose(A, HY, NewA),$

$r\_td_1(TXs, Y, NewA)$

$r\_td_1(Xs, Y, A) \leftarrow$

$Xs = [X|TXs],$

$nonMinimal(X),$

$decompose(X, HX, TX_1, \ldots, TX_t),$

$minimal(TX_1), \ldots, minimal(TX_{p-1}),$

$(nonMinimal(TX_p); \ldots; nonMinimal(TX_t)),$

$process(HX, HY), compose(A, HY, NewA),$

$$r\_td_1([TX_p, \ldots, TX_t|TXs], Y, NewA)$$

$$r\_td_1(Xs, Y, A) \leftarrow$$

$$Xs = [X|TXs],$$

$$nonMinimal(X),$$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$(nonMinimal(TX_1); \ldots; nonMinimal(TX_{p-1})),$$

$$minimal(TX_p), \ldots, minimal(TX_t),$$

$$minimal(U_1), \ldots, minimal(U_{p-1}),$$

$$decompose(N, HX, U_1, \ldots, U_{p-1}, TX_p, \ldots, TX_t),$$

$$r\_td_1([TX_1, \ldots, TX_{p-1}, N|TXs], Y, A)$$

$$r\_td_1(Xs, Y, A) \leftarrow$$

$$Xs = [X|TXs],$$

$$nonMinimal(X),$$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$(nonMinimal(TX_1); \ldots; nonMinimal(TX_{p-1})),$$

$$(nonMinimal(TX_p); \ldots; nonMinimal(TX_t)),$$

$$minimal(U_1), \ldots, minimal(U_t),$$

$$decompose(N, HX, U_1, \ldots, U_t),$$

$$r\_td_1([TX_1, \ldots, TX_{p-1}, N, TX_p, \ldots, TX_t|TXs], Y, A)$$

Like I did in the tupling and descending generalized program schemas, in the $TDGLR$ template, I have only used all the open predicates of $DCLR$ (or $DCRL$), and no other new predicates (other than primitive $= /2$).

The applicability conditions of TDG_1 (respectively, TDG_2) ensure the equivalence of the $DCLR$ (respectively, $DCRL$) and $TDGLR$ programs for a given problem. The optimizability conditions ensure that the output $TDGLR$ program of these generalization schemas are more efficient than the input DC programs. Like the optimizability conditions of the tupling and descending generalization schemas, the optimizability conditions, together with some of the applicability conditions, check whether the *compose* calls in the template $TDGLR$ can be eliminated.

In this section, the example programs are given for the $infix\_flat$ problem.

Example 33 The specification of the left-to-right (LR) simultaneous tupling-and-descendingly generalized problem of $infix\_flat$ is:

$infix\_flat\_td_1(Bs, F, A)$ iff list F is the concatenation of list A and the infix representations of the elements in binary tree list Bs.

Program 16 below is the program for $infix\_flat$ as an instance of $TDGLR$.

$$
\begin{aligned}
&infix\_flat(B, F) \leftarrow \\
&\quad infix\_flat\_td_1([B], F, [\,]) \\
&infix\_flat\_td_1(Bs, F, A) \leftarrow \\
&\quad Bs = [\,], \\
&\quad F = A \\
&infix\_flat\_td_1(Bs, F, A) \leftarrow \\
&\quad Bs = [B|TBs],
\end{aligned}
$$

$$B = void,$$

$$HF = [\,],$$

$$append(A, HF, NA),$$

$$infix\_flat\_td_1(TBs, F, NA)$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = void, R = void,$$

$$HF = [E], append(A, HF, NA),$$

$$infix\_flat\_td_1(TBs, F, NA)$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = void,$$

$$R = bt(\_, \_, \_),$$

$$HF = [E], append(A, HF, NA),$$

$$infix\_flat\_td_1([R|TBs], F, NA)$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = bt(\_, \_, \_),$$

$$R = void,$$

$$U = void,$$

$$N = bt(U, E, R),$$

$$infix\_flat\_td_1([L, N|TBs], F, A)$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = bt(\_, \_, \_),$$

$$R = bt(\_, \_, \_),$$

$$U_1 = void, U_2 = void,$$

$$N = bt(U_1, E, U_2),$$

$$infix\_flat\_td_1([L, N, R|TBs], F, A)$$

Logic Program 16

Since the applicability conditions of TDG_1 (respectively, TDG_4) are satisfied for the input $DCLR$ (respectively, $DCRL$) $infix\_flat$ program, the simultaneous tupling-and-descendingly generalized $infix\_flat$ program can be

Program 16. For the $infix\_flat$ problem, the generalization schemas TDG_1 (or TDG_4) cannot be applied, because the optimizability condition O_{td112} (or O_{td412}) is not satisfied by *append*, the *compose* relation of $infix\_flat$. □

The other two simultaneous tupling-and-descending generalization schemas are:

TDG_2 : $\langle\ DCLR,\ TDGRL,\ A_{td2}\ O_{td212},\ O_{td221}\ \rangle$ where

A_{td2} : - *compose* is associative
- *compose* has e as the left and right identity element, where e appears in $DCLR$ and $TDGRL$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \Rightarrow [\neg minimal(X) \Leftrightarrow nonMinimal(X)]$

O_{td212} : partial evaluation of the conjunction $process(HX, HY), compose(HY, A, NewA)$ results in the introduction of a non-recursive relation

O_{td221} : partial evaluation of the conjunction $process(HX, HY), compose(I_{p-1}, HY, I_p)$ results in the introduction of a non-recursive relation

TDG_3 : $\langle\ DCRL,\ TDGRL,\ A_{td3}\ O_{td312},\ O_{td321}\ \rangle$ where

A_{td3}: - *compose* is associative
- *compose* has e as the left and right identity element, where e appears in $DCRL$ and $TDGRL$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \Rightarrow [\neg minimal(X) \Leftrightarrow nonMinimal(X)]$

O_{td312} : partial evaluation of the conjunction $process(HX, HY), compose(HY, A, NewA)$ results in the introduction of a non-recursive relation

O_{td321} : partial evaluation of the conjunction $process(HX, HY), compose(HY, I_p, I_{p-1})$ results in the introduction of a non-recursive relation

The specification of the generalized problem, namely $S_{r\_td_2}$ is:

$$\forall Xs : list(\mathcal{X}).\forall Y, A : \mathcal{Y}.\ (\forall X : \mathcal{X}.\ X \in Xs \Rightarrow \mathcal{I}_r(X)) \Rightarrow$$

$$[r\_td_2(Xs, Y, A) \Leftrightarrow (Xs = [\,] \wedge Y = A)$$

$$\vee(Xs = [X_1, X_2, \ldots, X_q] \wedge \bigwedge_{i=1}^{q} \mathcal{O}_r(X_i, Y_i) \ \wedge I_1 = Y_1 \wedge$$

$$\bigwedge_{i=2}^{q} \mathcal{O}_c(I_{i-1}, Y_i, I_i) \ \wedge \mathcal{O}_c(I_q, A, I_{q+1}) \wedge Y = I_{q+1})]$$

where $\mathcal{O}_c$ is the output condition of *compose*, and $\mathcal{O}_r$ is the output condition of r, and $q \geq 1$. Template 7 below is the template of the common schema pattern $TDGRL$ of DG_2 and DG_3.

Logic Program Template 7

$r(X, Y) \leftarrow$

$r\_td_2([X], Y, e)$

$r\_td_2(Xs, Y, A) \leftarrow$

$Xs = [],$

$Y = A$

$r\_td_2(Xs, Y, A) \leftarrow$

$Xs = [X|TXs],$

$minimal(X),$

$r\_td_2(TXs, NewA, A),$

$solve(X, HY),$

$compose(HY, NewA, Y)$

$r\_td_2(Xs, Y, A) \leftarrow$

$Xs = [X|TXs],$

$nonMinimal(X),$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$minimal(TX_1), \ldots, minimal(TX_t),$$

$$r\_td_2(TXs, NewA, A),$$

$$process(HX, HY), compose(HY, NewA, Y)$$

$$r\_td_2(Xs, Y, A) \leftarrow$$

$$Xs = [X|TXs],$$

$$nonMinimal(X),$$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$minimal(TX_1), \ldots, minimal(TX_{p-1}),$$

$$(nonMinimal(TX_p); \ldots; nonMinimal(TX_t)),$$

$$r\_td_2([TX_p, \ldots, TX_t|TXs], NewA, A),$$

$$process(HX, HY), compose(HY, NewA, Y)$$

$$r\_td_2(Xs, Y, A) \leftarrow$$

$$Xs = [X|TXs],$$

$$nonMinimal(X),$$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$(nonMinimal(TX_1); \ldots; nonMinimal(TX_{p-1})),$$

$$minimal(TX_p), \ldots, minimal(TX_t),$$

$$minimal(U_1), \ldots, minimal(U_{p-1}),$$

$$decompose(N, HX, U_1, \ldots, U_{p-1}, TX_p, \ldots, TX_t),$$

$$r\_td_2([TX_1, \ldots, TX_{p-1}, N|TXs], Y, A)$$

$$r\_td_2(Xs, Y, A) \leftarrow$$

$$Xs = [X|TXs],$$

$$nonMinimal(X),$$

$$decompose(X, HX, TX_1, \ldots, TX_t),$$

$$(nonMinimal(TX_1); \ldots; nonMinimal(TX_{p-1})),$$

$$(nonMinimal(TX_p); \ldots; nonMinimal(TX_t)),$$

$$minimal(U_1), \ldots, minimal(U_t),$$

$$decompose(N, HX, U_1, \ldots, U_t),$$

$$r\_td_2([TX_1, \ldots, TX_{p-1}, N, TX_p, \ldots, TX_t|TXs], Y, A)$$

Again, in the $TDGRL$ template, I have only used all the open predicates of $DCLR$ (or $DCRL$), and no other new predicates (other than primitive $= /2$).

The reader is invited to analyze the applicability conditions and the optimizability conditions of TDG_2 and TDG_3, like I did for the previous generalization schemas.

Example 34 The specification of the right-to-left (RL) simultaneous tupling-and-descendingly generalized problem of $infix\_flat$ is:

$infix\_flat\_td_2(Bs, F, A)$ iff list F is the concatenation of the infix representations of the elements in binary tree list Bs and list A.

Program 17 below is the program for $infix\_flat$ as an instance of $TDGRL$.

$$infix\_flat(B, F) \leftarrow$$

$$infix\_flat\_td_2([B], F, [\,])$$

$$infix\_flat\_td_2(Bs, F, A) \leftarrow$$

$$Bs = [\,],$$

$$F = A$$

$$infix\_flat\_td_2(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = void,$$

$$infix\_flat\_td_2(TBs, NA, A),$$

$$HF = [\,],$$

$$append(HF, NA, F)$$

$$infix\_flat\_td_2(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = void, R = void,$$

$$infix\_flat\_td_2(TBs, NA, A),$$

$$HF = [E], append(HF, NA, F)$$

$$infix\_flat\_td_2(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = void,$$

$$R = bt(\_, \_, \_),$$

$$infix\_flat\_td_2([R|TBs], NA, A),$$

$$HF = [E], append(HF, NA, F)$$

$$infix\_flat\_td_2(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = bt(\_, \_, \_),$$

$$R = void,$$

$$U = void,$$

$$N = bt(U, E, R),$$

$$infix\_flat\_td_2([L, N|TBs], F, A)$$

$$infix\_flat\_td_2(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(\_, \_, \_),$$

$$B = bt(L, E, R),$$

$$L = bt(\_, \_, \_),$$

$$R = bt(\_, \_, \_),$$

$$U_1 = void, U_2 = void,$$

$$N = bt(U, E, R),$$

$$infix\_flat\_td_2([L, N, R|TBs], F, A)$$

Logic Program 17

Since both the applicability conditions and the optimizability conditions of TDG_2 (respectively, TDG_3) are satisfied for the input $DCLR$ (respectively, $DCRL$) $infix\_flat$ program, both simultaneous tupling-and-descending generalizations of the $infix\_flat$ programs result in Program 17 above. □

These simultaneous tupling-and-descending generalization schemas can also be used in the reverse direction (i.e., to transform TDG programs into DC programs); the reason for using these schemas in the reverse direction will become clear in Section 4.3.2, where the optimized versions of Programs 16 and 17, and the complexity analyses of these $infix\_flat$ programs, are given as well.

4.3.2 Complexity Analysis

For the complexity analysis of the programs of the simultaneous tupling-and-descending generalization schemas, I again use the $infix\_flat$ problem, which was also used in Sections 4.1.2 and 4.2.2 for the discussions of the tupling and descending generalization schemas. I use Programs 9 and 10 in Section 4.1.2, which are the optimized versions of the $infix\_flat$ $DCLR$ and $DCRL$ programs.

I will again summarize the time and space complexities of these DC programs. They have $O(n^2)$ time complexity in the worst case, and build a stack of h pairs of recursive calls, and create $2n$ intermediate data structures, if n is the number of elements in tree B and h is the height of B.

Program 15 in Section 4.3.1 can be optimized, resulting in Program 18 below.

$$infix\_flat(B, F) \leftarrow$$

$$infix\_flat\_td_1([B], F, [\,])$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [\,], F = A$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = void,$$

$$infix\_flat\_td_1(TBs, F, A)$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(L, E, R),$$

$$L = void, R = void,$$

$$append(A, [E], NA),$$

$$infix\_flat\_td_1(TBs, F, NA)$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(L, E, R),$$

$$L = void,$$

$$R = bt(\_, \_, \_),$$

$$append(A, [E], NA),$$

$$infix\_flat\_td_1([R|TBs], F, NA)$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(L, E, R),$$

$$L = bt(\_, \_, \_),$$

$$R = void,$$

$$infix\_flat\_td_1([L, bt(void, E, R)|TBs], F, A)$$

$$infix\_flat\_td_1(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(L, E, R),$$

$$L = bt(\_, \_, \_),$$

$$R = bt(\_, \_, \_),$$

$$infix\_flat\_td_1([L, bt(void, E, void), R|TBs], F, A)$$

Logic Program 18

Unfortunately, in Program 18, the calls to *append* cannot be fully eliminated, because of properties of *append*. The time used by the *append* calls in r_td_1 is more than the time used by the *append* calls in the $infix\_flat$ DC programs, because the accumulator parameter, which is extended by the partial result, is input as the induction parameter to each *append* call.

Program 16 in Section 4.3.1 can be optimized, resulting in Program 19

below.

$$
\begin{aligned}
&infix\_flat(B, F) \leftarrow \\
&\quad infix\_flat\_td_2([B], F, [\,]) \\
&infix\_flat\_td_2(Bs, F, A) \leftarrow \\
&\quad Bs = [\,], F = A \\
&infix\_flat\_td_2(Bs, F, A) \leftarrow \\
&\quad Bs = [B|TBs], \\
&\quad B = void, \\
&\quad infix\_flat\_td_2(TBs, F, A) \\
&infix\_flat\_td_2(Bs, F, A) \leftarrow \\
&\quad Bs = [B|TBs], \\
&\quad B = bt(L, E, R), \\
&\quad L = void, R = void, \\
&\quad infix\_flat\_td_2(TBs, TF, A), \\
&\quad F = [E|TF] \\
&infix\_flat\_td_2(Bs, F, A) \leftarrow \\
&\quad Bs = [B|TBs], \\
&\quad B = bt(L, E, R), \\
&\quad L = void,
\end{aligned}
$$

$$R = bt(\_, \_, \_),$$

$$infix\_flat\_td_2([R|TBs], TF, A),$$

$$F = [E|TF]$$

$$infix\_flat\_td_2(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(L, E, R),$$

$$L = bt(\_, \_, \_),$$

$$R = void,$$

$$infix\_flat\_td_2([L, bt(void, E, R)|TBs], F, A)$$

$$infix\_flat\_td_2(Bs, F, A) \leftarrow$$

$$Bs = [B|TBs],$$

$$B = bt(L, E, R),$$

$$L = bt(\_, \_, \_),$$

$$R = bt(\_, \_, \_),$$

$$infix\_flat\_td_2([L, bt(void, E, void), R|TBs], F, A)$$

Logic Program 19

In Program 19, the calls to *append* have disappeared, and we have a linear time program. Although the space complexity of Program 19 is worse than for the DC programs for the $infix\_flat$ problem, this program can be made tail recursive in the mode (*in*,*out*, *in*), as the last five clauses are mutually exclusive.

Therefore, for the input DC programs like the programs given for $infix\_flat$,

which use *append* as the *compose* operator, the generalization schemas TDG_2 and TDG_3 result in more efficient programs than the generalization schemas TDG_1 and TDG_4. If the *compose* operator of the input DC program that is an instance of the $DCLR$ template (or $DCRL$) satisfies the optimizability conditions of TDG_1 (or TDG_4), then obviously the generalization schema TDG_1 (or TDG_4) will result in more efficient programs than the generalization schema TDG_2 (or TDG_3).

If the TDG generalization schemas are used in the reverse direction (i.e., to transform TDG programs into DC programs), then for instance Programs 9 and 10 are more efficient in time and space than Program 18. So, it is still possible to gain efficiency by using the TDG generalization schemas in the reverse direction. However, the TDG_1 generalization schema (respectively, the TDG_2 generalization schema) for an input program that is an instance of the $TDGLR$ schema pattern (respectively, $TDGRL$ schema pattern) can be better than the TDG_4 generalization schema (respectively, the TDG_3 generalization schema) for an input program that is an instance of the $TDGLR$ schema pattern (respectively, $TDGRL$ schema pattern), or vice versa, depending on the post-transformation conditions of these generalization schemas for the input programs that are instances of the $TDGLR$ schema pattern (respectively, $TDGRL$ schema pattern).

Chapter 5

Duality Transformation Schemas

In Chapter 3, while discussing the composition ordering in DC program schemas, the reader who is familiar with functional programming will notice the similarities between composition ordering and the *fold* operators in functional programming. A detailed explanation of the fold operators and their laws can be found in [3]. Now, I will only give the definitions of the fold operators, and their first law. The definition of *foldr* is as follows:

$$foldr\ f\ a\ [x_1, x_2, \ldots, x_n] = f\ x_1(f\ x_2(\ldots(f\ x_n\ a)\ldots))$$

An equivalent formulation, possibly easier to read, is:

$$foldr\ (\oplus)\ a\ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\ldots(x_n \oplus a)\ldots))$$

where $\oplus$, like f, is just a variable that can be bound to a function of two arguments.

The *foldl* operator can be defined as:

$$foldl\ f\ a\ [x_1, x_2, \ldots, x_n] = f(\ldots(f(f\ a\ x_1)x_2)\ldots)x_n\ldots))$$

An equivalent formulation, possibly easier to read, is:

$$foldl\ (\oplus)\ a\ [x_1, x_2, \ldots, x_n] = (\ldots((a \oplus x_1) \oplus x_2)\ldots) \oplus x_n$$

Thus, equation 3.1 in Chapter 3 that illustrates the composition of Y in the $DCLR$ template can be rewritten using $foldl$:

$$foldl\ (\oplus)\ e\ [TY_1, \ldots, TY_{p-1}, HY, TY_p, \ldots, TY_t]$$

In a similar way, the $foldr$ operator can be used to rewrite equation 3.2 that illustrates the composition of Y in the $DCRL$ template as follows:

$$foldr\ (\oplus)\ e\ [TY_1, \ldots, TY_{p-1}, HY, TY_p, \ldots, TY_t]$$

The first three laws of the fold operators are called *duality theorems.* The *first duality theorem* states that:

$$foldr\ (\oplus)\ a\ xs = foldl\ (\oplus)\ a\ xs$$

if $\oplus$ is associative and has (left and right) identity element a, and xs is a finite list.

Since *append*, which is *compose* in our $flat$ examples, is associative and has [] as the identity element, Programs 1 and 2 (respectively, Programs 3 and 4, and Programs 5 and 6) are equivalent (resulting in the same Y, which is also stated in the first duality theorem) for $prefix\_flat$ (respectively, $infix\_flat$, and $postfix\_flat$). This shows that the problem families that the two program schemas abstract have an intersection family (resulting in equivalent programs for the problem), if *compose* satisfies the constraints of the first duality theorem.

So a transformation technique can be constructed that takes Program 1 (3, or 5, respectively) as an input, and produces Program 2 (4, or 6, respectively) as an output program, and vice versa.

Since I already have the input and output program schema patterns, and the applicability conditions (i.e., the constraints of the first duality theorem) of a possible transformation schema, I will give transformation *schemas*, rather than constructing a transformation technique.

The transformation schemas given in Section 5.1 are thus called *duality schemas.* The time and space complexity analysis of the input and output

programs of these duality schemas are given in Section 5.2. The correctness proofs of these duality schemas are in [9].

5.1 Duality Schemas

Using the previous discussion, the first duality schema D_{dc} below is given for transforming DC programs.

D_{dc} : $\langle$ $DCLR$, $DCRL$, A_{ddc}, O_{ddc12}, $O_{ddc21}\rangle$ where
A_{ddc} : - *compose* is associative
- *compose* has e as the left and right identity element, where e appears in $DCLR$ and $DCRL$
O_{ddc12} : - partial evaluation of the conjunction $process(HX, HY), compose(HY, I_p, I_{p-1})$ results in the introduction of a non-recursive relation
O_{ddc21} : - partial evaluation of the conjunction $process(HX, HY), compose(I_{p-1}, HY, I_p)$ results in the introduction of a non-recursive relation

where the program schema patterns $DCLR$ and $DCRL$ are the DC schema patterns given in Chapter 3, and A_{ddc} comes from the constraints of the first duality theorem. The optimizability conditions check whether the *compose* operator can be eliminated in the output program.

Taking Program 1 (3, or 5, respectively) in Chapter 3 as an input, and producing Program 2 (4, or 6, respectively) as an output program can be achieved by the duality schema D_{dc}, but not the inverse transformation, since the optimizability condition O_{ddc21} is not satisfied by *append*, which is the *compose* relation of the $infix\_flat$ problem.

Similarly, it is possible to give duality schemas for the DG and TDG program schemas. The duality schema for DG programs, namely D_{dg}, is:

D_{dg} : $\langle$ $DGLR$, $DGRL$, A_{ddg}, O_{ddg12}, $O_{ddg21}\rangle$ where

A_{ddg} : - *compose* is associative
- *compose* has e as the left and right identity element, where e appears in $DGLR$ and $DGRL$

O_{ddg12}: - $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction $process(HX, HY), compose(HY, A_p, A_{p-1})$ results in the introduction of a non-recursive relation

O_{ddg21}: - $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction $process(HX, HY), compose(A_{p-1}, HY, A_p)$ results in the introduction of a non-recursive relation

where the templates of the schema patterns $DGLR$ and $DGRL$ are Logic Program Templates 4 and 5 in Section 4.2.1.

Taking Program 12 in Section 4.2.1 as an input, and producing Program 13 as an output program can be achieved by the duality schema D_{dg}, but not the inverse transformation, because of properties of *append*.

The duality schema for TDG programs, namely D_{tdg}, is:

D_{tdg} : $\langle\ TDGLR,\ TDGRL,\ A_{dtdg},\ O_{dtdg12},\ O_{dtdg21}\rangle$ where

A_{dtdg} : (1) *compose* is associative
(2) *compose* has e as the left and right identity element, where e appears in $TDGLR$ and $TDGRL$

O_{dtdg12}: - $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction $process(HX, HY), compose(HY, NewA, F)$ results in the introduction of a non-recursive relation

O_{dtdg21}: - $\forall X : \mathcal{X}.\ \mathcal{I}_r(X) \wedge minimal(X) \Rightarrow \mathcal{O}_r(X, e)$
- partial evaluation of the conjunction $process(HX, HY), compose(A, HY, NewA)$ results in the introduction of a non-recursive relation

where the templates of the schema patterns $TDGLR$ and $TDGRL$ are Logic Program Templates 6 and 7 in Section 4.3.1.

Taking Program 16 in Section 4.3.1 as an input, and producing Program 17 as an output program can be achieved by the duality schema D_{tdg}, but not the inverse transformation, because of properties of *append*.

5.2 Complexity Analysis

Since the complexity analysis of DC, DG, and TDG programs for the $infix\_flat$ problem have already been given in Sections 4.1.2, 4.2.2, and 4.3.2, in this section I use these results to discuss the efficiency gain obtained by the duality schemas in Section 5.1.

Although Programs 9 and 10, which are the optimized versions of the $DCLR$ and $DCRL$ $infix\_flat$ programs, namely Programs 3 and 4, have time complexity $O(n^2)$, Program 10 has a better time complexity than Program 9 by a constant factor, which is not negligible. If we have Program 3 and we want to transform it into a more efficient program, then D_{dc} will be applied resulting in Program 4, which can be optimized into Program 10, because the applicability and optimizability conditions of D_{dc} are satisfied. Since Program 10 is more efficient than Programs 3 and 9, this shows that we can obtain efficiency gain by the D_{dc} schema. If we want to transform Program 4 into a more efficient program and D_{dc} is selected, then D_{dc} will not be applied, because the optimizability conditions of D_{dc} are not satisfied by the open relations of $infix\_flat$. So, we do not have Programs 3 or 9 as an output program of this duality schema, which shows that the duality schema D_{dc} does not result in a program that has worse time complexity than the input program.

Similarly, for the DG and TDG $infix\_flat$ programs, the RL versions have better time complexity than the LR versions. Because of the optimizability conditions in the D_{dg} and D_{tdg} schemas, the RL versions will always be output by these schemas.

Of course, it is possible to have an LR version of DC, DG, or TDG program that is more efficient than its RL version for some problems. In these cases, the duality schemas will output the LR program, if the input program is RL,

and they will not output the RL program if the input program is LR. So they always output the corresponding efficient version, which is ensured by the optimizability conditions.

Chapter 6

Evaluation of the Transformation Schemas

In this chapter, I evaluate the transformation schemas using performance tests done on the manually optimized input and output programs of each transformation schema. However, the reader may find this evaluation of little value, since the transformation schemas in this thesis are only dealing with the declarative features of the programs. So, I must say that this evaluation is made because I think that these performance tests will help us to see what our theoretical results will be when tested practically, although in an environment with procedural side-effects. The programs are executed and tested using Mercury 0.6 (for an overview of the Mercury logic programming language, refer to [50], and every detail about Mercury can be found in its home-page 'http://munkora.cs.mu.oz.au/mercury/') on a SPARCstation 4. Since the programs are really short, the procedures were called 500 or 1000 times to achieve meaningful timing results. In Table 6.1, the results of the performance tests for seven selected problems are shown, where each column heading represents the schema pattern to which the program written for the problem of that row belongs. The timing results are normalized wrt the DCLR column.

The reason why I chose the problems above is that all the seven programs that are instances of the seven program schema patterns can be written for these problems, because of the properties of the *compose*, *minimal*,

problems	DCLR	DCRL	TG	DGLR	DGRL	TDGLR	TDGRL
$prefix\_flat$	1.00	0.92	0.23	11.88	0.15	12.38	0.15
$infix\_flat$	1.00	0.49	0.02	7.78	0.05	7.59	0.15
$postfix\_flat$	1.00	0.69	0.14	5.48	0.09	5.55	0.09
$reverse$	1.00	1.00	0.04	1.01	0.01	1.02	0.04
$quicksort$	1.00	0.85	0.72	6.02	0.56	6.42	1.01
$sumlist$	1.00	1.00	8.33	0.01	0.33	4.00	8.67
$length$	1.00	1.00	16.33	0.67	1.00	9.00	14.00

Table 6.1. Performance Tests Results

$nonMinimal$, and $solve$ relations of their DC programs. The specification, and the DC and TG (respectively, DG) programs for $quicksort$ (respectively, for $reverse$) were given in Example 21 (respectively, in Example 22) in Section 2.1.6. The specification of a program for relation $sumlist$ is:

$sumlist(L, S)$ iff integer S is the sum of the elements in the integer-list L.

The specification of a program for relation $length$ is:

$length(L, N)$ iff integer N is the number of elements in the list L.

Let us first compare the $DCLR$ and $DCRL$ schema patterns. For $reverse$, $sumlist$, and $length$, the $DCLR$ and $DCRL$ programs are the same, since they are single-recursive, and their *compose* relations are either associative like *append* in $reverse$, or even commutative like + in $sumlist$ and $length$. For the binary tree $flat$ problems and for $quicksort$, the $DCRL$ programs are much better than the $DCLR$ programs, because of the relations like *append* (which is the *compose* relation in all these examples), whose properties are the main reason for properly achieving the optimizations of the $DCRL$ programs of the problems above.

Hence, if the input programs for the binary tree $flat$ problems, and for the $quicksort$ problem to the duality schema, are instances of the $DCLR$ schema pattern, then duality transformations will be performed resulting in the $DCRL$ programs for these relations, since both the applicability and the optimizability conditions are satisfied by these programs. So, the duality transformation of

the $DCLR$ programs for the relations, having the undefined relations in their open programs like the ones of the problems above, results in $DCRL$ programs that are more efficient than the input $DCLR$ programs. If the $DCRL$ programs for the above relations are input to the duality schema, then the duality transformation will *not* be performed, since the optimizability conditions are not satisfied by *append*, which is the *compose* relation of the $DCRL$ programs. Of course, there may exist some other relations where the duality transformation of their $DCRL$ programs into the $DCLR$ programs will provide an efficiency gain. Unfortunately, I did not find a meaningful well-known example of this category.

The next step in evaluating the transformation schemas is to compare the generalized programs of these example relations. If we look at Table 6.1, the most obvious observation is that the $DGRL$ programs for all these example relations are very efficient programs. However, tupling generalization seems to be the second best as a generalization choice, and it even must be the first choice in problems like $infix\_flat$, where the composition place of the head in the result parameter is the middle, and the *minimal* and *nonMinimal* checks can be performed in minimum time. Although a similar situation occurs for the *quicksort* problem, the TG program of *quicksort* is not as efficient as the $DGRL$ program. This is mainly because of *partition*, which is the *decompose* relation of *quicksort*, being a costly operation, although we eliminate most of the *partition* calls by putting extra minimality checks in the TG template. Since *append*, which is the *compose* relation in all the problems except *sumlist* and *length*, cannot be eliminated in the resulting $DGLR$ and $TDGLR$ programs, the $DGLR$ and $TDGLR$ programs for these relations have the worst timing results. The reason for their bad performances is that the percentages of the total running times of the $DGLR$ and $TDGLR$ programs used by *append* are much higher than the percentages of the total running times of the $DCLR$ and $DCRL$ programs used by *append* for these relations. The reason for the increase in the percentages is that the length of the accumulator, which is the input parameter to *append* in the $DGLR$ and $TDGLR$ programs, is bigger than the length of the input parameter of *append* in the $DCLR$ and $DCRL$ programs, since the partial result has to be repeatedly input to the *compose* relation in descending generalization.

The generalization of the input DC programs must be performed if all the applicability conditions are satisfied by the input DC programs (for the problems above, the applicability conditions of each generalization schema given in this thesis are satisfied by the input DC programs). The generalization must also check the optimizability conditions, and then must choose the generalization schema where both the applicability conditions and the optimizability conditions are satisfied. A generalization must be performed if it really results in a program that is much more efficient than the input program. So, the descending generalization of the input $DCLR$ program for $infix\_flat$ resulting in the $DGLR$ program must not be done, even if the applicability conditions are satisfied, since the performance of the $DGLR$ program for $infix\_flat$ is much worse than the input $DCLR$ program. This is the main reason for the existence of the optimizability conditions in the schemas. If we try to descendingly generalize the input $DCLR$ program (respectively, the $DCRL$ program) for any of the $flat$, $reverse$, or $quicksort$ problems, then the DG_2 (respectively, DG_3) schema will be chosen, since the optimizability conditions of DG_2 (respectively, DG_3) are satisfied. Also, if we try to do a simultaneous tupling-and-descending generalization of the input $DCLR$ program (respectively, the $DCRL$ program) for any of the $flat$, $reverse$, or $quicksort$ problems, then the TDG_2 (respectively, TDG_3) schema will be chosen, since the optimizability conditions of TDG_2 (respectively, TDG_3) are satisfied by the input programs. The optimizability conditions of DG_1 or DG_4 (respectively, TDG_1 or TDG_4) are not satisfied by the problems above. So, these schemas are out of the question during generalization of the DC programs of the problems above, which is what the user will want in a transformation system that is not doing a transformation that does not provide efficiency gain.

For the relations $sumlist$ and $length$, the results are completely different in the sense that the TG programs are much worse than the DC programs. The reason for this bad performance seems to be the overhead calls added by doing the generalization of the input parameter, which is already a list, into a list of lists. The other reasons for this efficiency loss may be the properties of +, and the implementation of + in Mercury. Actually, this much of an efficiency loss is not expected, this is the main reason which makes us to think that the implementation of the built-in predicates in Mercury may cause these

results. Of course, the other reason is the performance results of the DG (respectively, STDG) programs, where the $DGLR$ programs (respectively, the $TDGLR$ programs) are found to be more efficient than the $DGRL$ programs (respectively, the $TDGRL$ programs) for *sumlist* and *length*. The only reason that I can come up with is the implementation of + in Mercury. Since + is commutative, different implementations can be done in different languages.

In some of the cases, using generalization schemas to transform the input programs that are already generalized programs of the relations to DC programs will produce an efficiency gain. For example, if the $DGLR$ program for any of the *flat* problems is the input program to descending generalization (namely, DG_1 or DG_4), then the generalization will be performed resulting in the $DCLR$ (or, $DCRL$) program, which is more efficient than the input $DGLR$ program. Similarly, an efficiency gain will be obtained if the programs of the $TDGLR$ schema pattern are input to the generalization process, since the optimizability conditions of the generalization schemas in the reverse directions are satisfied. However, if the input program for any of the above relations to generalization is a $DGRL$ or $TDGRL$ program, then the generalization schemas are still applied in the reverse direction, which means that the reverse engineering will result in a program that is less efficient than the input program. This makes us think of some other ways of defining the optimiz*ability* conditions, namely optimiz*ation* conditions, such that the transformation will always either result in a better program than the input program. However, more performance analyses and complexity analyses are needed to make such a decision.

Therefore, a transformation system that will be developed with a database of the transformation schemas explained in this thesis has to verify the optimizability conditions, since these conditions ensure efficiency gain by these transformations.

Chapter 7

Prototype Transformation System

TRANSYS is a prototypical implementation of the schema-based program transformation approach explained in this thesis. TRANSYS is an automatic (i.e. without any user interaction) program transformation system that is developed to be integrated in a schema-based program development environment. Therefore, the input closed program to the transformation is assumed to be developed by a synthesizer (e.g. a proper extension of DIALOGS [61]) using the database of program schema patterns available in the system, as otherwise the transformation system cannot transform the input program. So the program schema pattern, of which the input closed program is an instance, is a-priori known. Thus, given an input program that is an instance of a program schema pattern in the database, the system will output all the programs that are instances of the program schema patterns in the database, and that are more efficient than the input program. The representation of program schema patterns and transformation schemas makes the system more data-oriented, which means that the actual algorithm of the system has a minimum amount of sub-procedures to define the representation of the schemas and schema patterns. The transformation schemas, and the program schema patterns, which are the input (or output) program schema patterns of these transformation schemas given in this thesis, are all available in the database of the system.

TRANSYS has been developed in SICStus Prolog 3, patch #5. Since TRANSYS is a prototype system, for some parts of the system, instead of implementing them, I integrated other systems:

- For verifying the applicability conditions and some of the optimizability conditions, PTTP is integrated into the system. The *Prolog Technology Theorem Prover* (PTTP) was developed by M.E. Stickel in the Artificial Intelligence Center of SRI International in California (for a detailed explanation of PTTP, the reader can refer to [54, 55]). PTTP is an implementation of the model elimination theorem proving procedure that extends Prolog to the full first-order predicate calculus. TRANSYS uses the version of PTTP that is written in Prolog and compiles clauses into Prolog.

- For verifying the other optimizability conditions, and applying the optimizations to the output programs of the transformation schemas, I integrated Mixtus 0.3.6. Mixtus was developed by D. Sahlin in SICS (Swedish Institute of Computer Science) in Kista (for a detailed explanation of Mixtus, the reader can refer to [48]). Mixtus is an automatic partial evaluator for full Prolog. Given a Prolog program and a query, it will produce a new program specialized for all instances of that query. The partial evaluator is guaranteed to terminate for all input programs and queries.

I explain how the programs, the program schema patterns, and the transformation schemas are defined in the system in Section 7.1. In Section 7.2, I explain the high-level algorithm of the system, and how the above systems PTTP and Mixtus are integrated into TRANSYS. I discuss the features of TRANSYS using a sample run of the system, and I evaluate TRANSYS as a transformation system in Section 7.3.

7.1 Representation Language

In this section, I will explain how the programs, program schema patterns, and transformation schemas are represented in the system. The program schema pattern representation is more complicated than the transformation schema and the program representations, since first-order logic is not enough to represent and manipulate the program schema patterns fully. In Section 7.1.1, I will give the syntax of the schema pattern language for the program schema pattern representation, and I will explain the semantics of the schema pattern language in Section 7.1.2. However, the schema pattern language used in this system cannot be used to represent every program schema pattern because of implementation restrictions, which will be explained in Section 7.1.2. For a more detailed representation of a program schema pattern, though in second-order logic, the reader can refer to [2]. In Section 7.1.3, the representations of the programs and transformation schemas are given.

7.1.1 Schema Pattern Language: Syntax

Currently, in the database of the system, the existing program schema patterns are the DC schema patterns, the TG, DG, and TDG schema patterns, and the *RS reuse schema pattern*, which is the base schema pattern with the steadfastness constraint *true*, which means that every program can be an instance of this schema pattern. So, a program that is an instance of the reuse schema pattern has itself as its extension.

A *program schema pattern* is represented as a relation $lps(NS, L, Temp, PL)$, where

- NS is the name of the program schema pattern;
- L is the list of the actual name of the top-level relation R and the actual names of the undefined relations in the NS schema pattern, which will be substituted in $Temp$ during particularization;
- $Temp$ is the template of the NS schema pattern, which is a list of *template*

clauses (defined below);

- PL is the list of parameters; in the reuse schema pattern, PL is equal to the singleton list $[N]$, but in the DC schema patterns and generalized program schema patterns, PL consists of:
  - E, a specific constant existing in every schema pattern for initiating the composition;
  - N, the number of arguments of the top level relation R (currently, in the database, it is hard-wired to 2);
  - H, the number of heads of the induction parameter when decomposed;
  - T, the number of tails of the induction parameter when decomposed;
  - Ps, the list of numbers denoting the composition places of heads $(head_1, ..., head_H)$, when composing the result parameter (since N is 2, there is 1 result parameter).

The syntax of *template clauses* can be given using the BNF grammar:

$$\begin{aligned} Clause &::= if(Head, Body) \\ Head &::= Atom \\ Body &::= true|SeqAtoms|and(SeqAtoms, Body) \\ SeqAtoms &::= Atom|Conjunction|Disjunction \\ Atom &::= Pred\_name(Args) \\ Args &::= Arg|Arg, Args \\ Arg &::= Term|Variable|Indexed\_Variable|Vector\_of\_Variables \end{aligned}$$

where $Term$ is a term and $Variable$ is a variable, and

- an *Indexed_Variable* is represented by a term of the form $V\#I$, where V is a variable (called the *root*), and I is the index, which can be either an integer, or a variable, or an expression of the forms $(J + X)$ or $(J - X)$, where X is an integer and J is a variable;
- a *Vector_of_Variables* is represented by a term of the form $vec(V, LB, UB)$, where V is a variable (called the *root* of the $Vector\_of\_Variables$), LB

is the lower bound, and UB is the upper bound, where a *bound* is either an integer or a variable;

- a *Conjunction* is represented by a term of the form $conjatoms(Atom, LB, UB)$, where $Atom$ is as defined above, and LB and UB are the lower and upper bounds of the index J, which is in $Atom$,
- a *Disjunction* is represented by a term of the form $disjatoms(Atom, LB, UB)$, where $Atom$ is as defined above, and LB and UB are the lower and upper bounds of the built-in index, represented by a special variable J in $Atom$.

Example 35 The program schema pattern $DCRL$ for a relation R of arity 2 with the first parameter being the induction parameter, which is decomposed into 1 head and N tails, and the head composition place in the result parameter being P, can be represented as:

$$lps(dcrl, [R, M, S, NM, DEC, PROC, COMP], Tmp, [E, 2, 1, T, [P]])$$

iff

$$\begin{aligned} Tmp = [\ & if(R(X,Y),\ and(M(X), S(X,Y))), \\ & if(R(X,Y),\ and(NM(X), and(DEC(X, HX, vec(TX, 1, T)), \\ & \qquad and(conjatoms(R(TX\#J, TY\#J), 1, T), \\ & \qquad and(I\#T1 = E, \\ & \qquad and(conjatoms(COMP(TY\#J, I\#(J+1), I\#J), P, T), \\ & \qquad and(PROC(HX, HY), and(COMP(HY, I\#P, I\#P1), \\ & \qquad and(conjatoms(COMP(TY\#J, I\#J, I\#(J-1)), 1, P1), \\ & \qquad Y = I\#0)))))))))] \end{aligned}$$

where $P1 = P - 1$ and $T1 = T + 1$. □

A full implementation of the program schema patterns in the system will be given in the next section where the semantics of the schema patterns is explained by defining the operations on them. The program schema patterns are stored in a file called *dbase.pl*.

7.1.2 Schema Pattern Language: Semantics

In this section, I explain how a template of a program schema pattern is manipulated to obtain a template (actually an open program) without ellipses.

Definition 34 (Index Replacement) Let $X\#J$ be an $Indexed\_Variable$. The *replacement* of an index I by an integer k applied to $X\#J$, which is denoted as $IRep_{I\leftarrow a}X\#J$, gives either a new variable X_a that will refer to the $Indexed\_Variable$ $X\#J$ throughout the template, where it is used, or remains $X\#J$ if $I \neq J$.

Definition 35 (Vector_of_Variables Expansion) Let $vec(V, LB, UB)$ be a $Vector\_of\_Variables$. The *expansion* of $vec(V, LB, UB)$ is done if LB and UB are both substituted by integers. The expansion of $vec(X, LB, UB)$ is defined as follows:

- $X\#LB$ if $LB = UB$,
- $X\#LB, X\#(LB+1), \ldots, X\#UB$ if $LB < UB$,
- the empty sequence if $LB > UB$.

So, in $vec(X, LB, UB)$, the root X ranges between the lower and upper bound.

After expansion, a Vector_of_Variables having its lower bound greater than its upper bound will fully disappear from the arguments of a relation.

The replacement of an index I by the integer a applied to a *Term* T makes no change in *Term* T. Then, the replacement of an index I by the integer a applied to a relation R of the form $P(T\#1, \ldots, T\#n)$, which is denoted as $ARep_{I\leftarrow a}R$, gives $P(IRep_{I\leftarrow a}T\#1, \ldots, IRep_{I\leftarrow a}T\#n)$.

Definition 36 (Conjunction Expansion) Let $conjatoms(A, LB, UB)$ be a conjunction. Conjunction *expansion* is done after LB and UB are both substituted by integers, and all arguments of A different from the $Indexed\_variables$

with index J, which is the special variable for representing the index of the conjunction, are gone through index replacement. The expansion of the conjunction $conjatoms(A, LB, UB)$ is defined as follows:

- $true$ if $LB > UB$,
- $ARep_{J \leftarrow LB} A$ if $LB = UB$,
- $$and(ARep_{J \leftarrow LB} A, \quad and(ARep_{J \leftarrow (LB+1)} A, \ldots, \\ and(ARep_{J \leftarrow (UB-1)} A, ARep_{J \leftarrow UB} A) \ldots))$$

Definition 37 (Disjunction Expansion) Let $disjatoms(A, LB, UB)$ be a disjunction. Disjunction *expansion* is done after LB and UB are both substituted by integers, and all arguments of A different from the $Indexed\_variables$ with index J, which is the special variable for representing the index of the disjunction, are gone through index replacement. The expansion of the disjunction $disjatoms(A, LB, UB)$ is defined as follows:

- $false$ if $LB > UB$,
- $ARep_{J \leftarrow LB} A$ if $LB = UB$,
- $$or(ARep_{J \leftarrow LB} A, \quad or(ARep_{J \leftarrow (LB+1)} A, \ldots, \\ or(ARep_{J \leftarrow (UB-1)} A, ARep_{J \leftarrow UB} A) \ldots))$$

Restrictions: The $Vector\_of\_Variables$ in my system is restricted to a vector of variables having a variable in the root, which means double indexing is not allowed. The $Conjunction$ and $Disjunction$ representation is also restricted to built-in index J, which also means that double indexing of variables is not allowed. Another restriction is that the undefined relation names are taken as input, so no construction of undefined indexed relations is allowed. Thus, the program schema patterns that can be represented in the prototype system are limited.

Definition 38 (Particularization of a Template) The *particularization of* $Template$ of the program schema pattern

$$lps(NS, [R, M, S, NM, DEC, PROC, COMP], Template, [E, N, H, T, Ps])$$

results in an open program for relation R, by doing the following sequence of operations:

1. *Parameter/Term Bindings*: The parameters (i.e. *ellipses* of the template: N, H, T, Ps) are bound to their actual integer values. Also, at this stage, the variable/term bindings are achieved for the variables

$$E, NS, R, M, S, NM, DEC, PROC, COMP$$

   with their actual values.

2. *Variable/Relation Bindings*: The predicate variables in the template are bound to the actual names of the open relations. This will be better understood in the example below.

3. *Template Manipulation*: The template of the program schema pattern is converted to an open program. Index replacement, Vector_of_Variables expansion, and conjunction and disjunction expansion are the subprocesses of this final process.

The programs of the template manipulation process are in a file called $dedotify.pl$.

Example 36 The representation of the $DCRL$ program schema pattern for a relation R of arity 2 with the first parameter being the induction parameter, which is decomposed into 1 head and N tails, and the head composition place in the result parameter being P in the system is given in Example 35 in Section 7.1.1.

The particularization of $Template$ by the goal

$$lps(DCRL, [r, m, s, nm, dec, proc, comp], Template, [[], 2, 1, 2, [2]])$$

will result in the open program below:

$$\begin{aligned} Template = [&if(r(X,Y),\ and(m(X), s(X,Y))),\\ &if(r(X,Y),\ and(nm(X), and(dec(X, HX, TX_1, TX_2),\end{aligned}$$

$$
\begin{aligned}
&and(and(r(TX_1, TY_1), r(TX_2, TY_2)),\\
&and(I_3 = [],\\
&and(comp(TY_2, I_3, I_2),\\
&and(proc(HX, HY),\\
&and(comp(HY, I_2, I_1),\\
&and(comp(TY_1, I_1, I_0),\\
&Y = I_0)))))))))]
\end{aligned}
$$

Therefore, the actual open program is obtained by conversion from the template. □

7.1.3 Representation of Programs and Transformation Schemas

A *program* for relation R is represented as a term $lp(NS, L, Ext, PL)$, where

- The represented program is an instance of the schema pattern NS;
- L is the list of the name of R and the actual names of the undefined relations in NS;
- Ext (stands for extension) is the list of programs for the undefined relations in NS;
- PL is the list of parameters, which consists, for DC, of:
  - E, a specific constant existing in every schema pattern for initiating the composition;
  - N, the number of arguments of R (currently in the database, it is hard-wired to 2);
  - H, the number of heads of the induction parameter of R when decomposed;
  - T, the number of tails of the induction parameter of R when decomposed;

- Ps, the list of numbers denoting the composition places of the heads $(head_1, ..., head_H)$, when composing the result parameter of R (since N is 2, there is 1 result parameter);
- $Sorts$, the list of constants (e.g. *list* or *btree*) indicating the types of the parameters of R.

Since Ext is a list of programs, where each one is also represented as above, the system has the mechanism to deal with nested programs.

Example 37 Program 4 in Chapter 3 can be represented by the term

$$lp(dcrl,[infix\_flat, f\_min, f\_solve, f\_nonmin, f\_decomp, f\_proc, f\_compose], Ext, [[\,], 2, 1, 2, [2], [btree, list]])$$

where Ext is the list containing the programs for the undefined relations

$$f\_min, f\_solve, f\_nonmin, f\_decomp, f\_proc, f\_compose$$

also represented using the program representation above. □

A *transformation schema* is represented by an atom

$$ts(NTS, NS_1, NS_2, I, E, L, ACs, PCs)$$

where

- NTS is the name of the transformation schema;
- NS_1 and NS_2 are the names of the program schema patterns that satisfy the applicability conditions of the transformation schema;
- I is either 1, indicating that the input program to the transformation is an instance of program schema pattern NS_1, or 2, indicating that the input program to the transformation is an instance of program schema pattern NS_2;
- E is a specific constant existing in every schema pattern for initiating the composition;

- L is the list of the actual name of the top-level relation R and the actual names of the undefined relations in the NS_1 and NS_2 schema patterns;
- ACs is the list of the applicability conditions of the transformation schema, e.g. the first applicability condition of DG_1 in Section 4.2.1 (i.e. *compose* is associative) is represented as a tuple $(a, COMP)$, where constant a indicates associativity, and $COMP$ is a variable referring to the actual name of the *compose* relation;
- PCs is the list of the optimizability conditions of the transformation schema, e.g. in DG_1, if the input program is an instance of the $DCLR$ schema pattern, the first part of the first optimizability condition of DG_1 (i.e. *compose* has the left identity element e) is represented as $(1, ri, COMP, E)$, where 1 indicates that the input program is an instance of the $DCLR$ schema pattern, ri is a constant indicating left identity, $COMP$ is a variable referring to the actual name of the *compose* relation, and E is a variable indicating the special composition constant e in the templates.

Example 38 The generalization schema DG_1 in Section 4.2.1 is represented as a fact:

$$gs(dg1, dclr, dglr, I, E, [R, M, S, NM, DEC, PROC, COMP],$$
$$[(a, COMP), (li, COMP, E)],$$
$$[(1, ri, COMP, E), (1, min, M, R, E), (1, pe, PROC, COMP),$$
$$(2, pe, PROC, COMP)]) \leftarrow$$

□

The transformation schemas are stored as facts in a file called *dbase.pl*.

7.2 Algorithm of the System

The program schema patterns given in Chapters 3 and 4 and the transformation schemas given in Chapters 4 and 5 are all represented in the system as explained

in Section 7.1. These program schema patterns and transformation schemas in the database of the system can be represented using the graph in Figure 7.1 below.

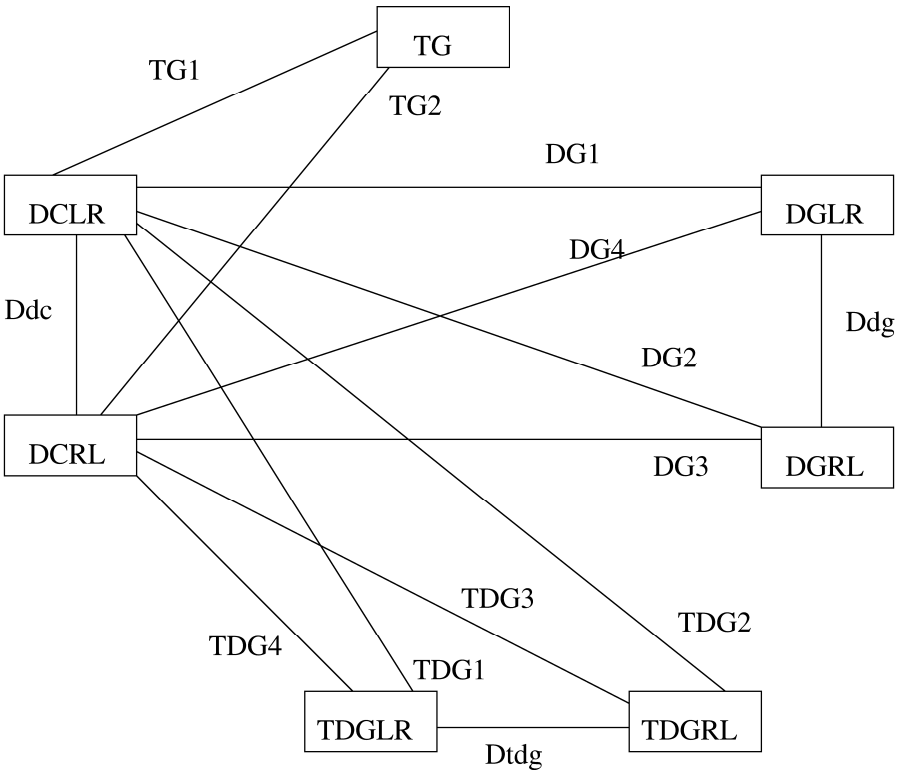

Figure 7.1. An Undirected Graph Representing the Database of the System

Each node in the graph represents a program schema pattern in the database, and each edge represents a transformation schema. Since the transformation schemas are applicable in both directions, the graph is undirected.

Given an input program P_1, the prototype system traverses the graph on the edges where both the applicability and optimizability conditions are satisfied, so as to output all the programs that are ensured to be more efficient than P_1 by the optimizability conditions of the applicable transformation schemas.

When a program P_2 is output, which is an instance of the output program schema of one of the transformation schemas where both the applicability and optimizability conditions are satisfied, then the program P_2 is further input to Mixtus, the partial evaluator that is used in the system and explained in the introductory section of Chapter 7, for optimization. Then Mixtus outputs the

an optimized program P_3 to the user of the system. Therefore, at one instance, the system, also with Mixtus, outputs two programs where the second one is the optimized version of the first one. Then, the output program P_2 is input to the system again to obtain other (possibly more efficient) programs, which will be the output programs of the transformation schemas that are applicable to P_2. So, what the system does for an input program can be seen as edge traversing of the graph in Figure 7.1 from a given start node.

The $transform/3$ relation, whose program is given with the simple Prolog code below, is called by the top-level graph traversing relation in the system to find a transformation schema that is applicable and ensuring an efficiency gain:

$$
\begin{aligned}
&transform(LP\_IN, LP\_OUT, LS) :- \\
&\qquad LP\_IN = lp(NS\_IN, L, Ext, [E, N, H, T, Ps, Sorts]), \\
&\qquad ts(NST, I, NS\_IN, NS\_OUT, E, L, ACs, PCs), \\
&\qquad memberCheck(LS, NS\_OUT), \\
&\qquad satisfied(ACs, Sorts, Ext), \\
&\qquad verified(PCs, L, Sorts, Ext, I), \\
&\qquad LP\_OUT = lp(NS\_OUT, L, Ext, [E, N, H, T, Ps, Sorts]).
\end{aligned}
$$

where LP_IN is the input program, LP_OUT is the program, which is an instance of the output program schema NS_OUT of the transformation schema NST, and LS is the list of the names of the program schema patterns that are not processed by the $transform$ relation yet. NST is selected by the call $ts(NST, I, NS\_IN, NS\_OUT, E, L, ACs, PCs)$ and it is checked by the $memberCheck$ relation whether it was already found to be applicable resulting in a more efficient output program for the input program. If the output program schema of the transformation schema has not been processed before, then the applicability conditions are checked by the $satisfied$ relation. Finally, the optimizability conditions of NST are checked by the $verified$ relation. The $satisfied$ relation calls PTTP, a theorem prover, to prove the applicability conditions. The $verified$ relation also calls PTTP to prove some of the optimizability conditions, e.g., proving E being the left identity of the *compose* relation, and it calls Mixtus to check the optimizability conditions when the

partial evaluation results are needed. Of course, the intermediate operations, which are needed to prepare the inputs for PTTP and Mixtus, and to operate on the outputs of these subsystems, are also taken care of by the low-level relations of the prototype system. For example, I hardwire the PTTP proof to search to depth 100 at most, since this number is less than 10 in all the tests and PTTP's default maximum is 1000000, which requires a lot of time if the theorem is not provable.

7.3 Evaluation of the System

As I explained in the previous sections, I used a theorem prover, PTTP, and a partial evaluator, Mixtus, to check the applicability and optimizability conditions, and to do the optimizations. Since these subsystems are too generic (i.e., they are not written to do only the operations in the system), they require nearly 90 percent of the time used by the system. For example, I hardwired PTTP proofs to search to depth 100 at most, so it will search up to the 100th level if the theorem is not provable, which can really take a lot of time. However, a more application-specific subsystem would require less time, but this would also lower the generality and extendibility of the prototype system. So, the time complexity of the system is mainly dominated by the time used in the verification of the applicability and optimizability conditions. Since the output program of each applicable transformation schema is again input to the system, if it is not already processed by the system, in the worst case the time used by the system can be given as $n * m * T$, where n is the number of program schema patterns processed, which is currently bounded by 7, and m is the number of the selected transformation schemas, which is currently bounded by 13, and T is the time required for checking the applicability and optimizability conditions.

The time complexity of the system can be improved by extending the system such that, for each input program, a dynamic list of the results of the condition checks is maintained during the execution, and before calling PTTP or Mixtus for checking a condition, the condition check will be looked up from that list. This will really improve the time complexity of the system by a constant factor,

which is not negligible, since currently, in the database of the system, the conditions of one of the transformation schemas are equal to, or a superset of, or a subset of the conditions of another transformation schema.

The sample run output of the system where the input program is Program 12, which is the $DGLR$ $infix\_flat$ program, is given in Appendix B. Actually, when Program 12 is input to the system, first the D_{dg} transformation schema is selected. Since the applicability and optimizability conditions of D_{dg} are satisfied, the system stores the $DGRL$ program to be output after all the applicable transformation schemas (i.e., direct edges from $DGLR$ in the graph in Figure 7.1) are checked. Next, DG_1 is selected. Since the optimizability conditions are not satisfied, $DCLR$ will not be output. Finally, for the input $DGLR$ program, DG_4 will be selected. $DCRL$ will be stored as one of the output programs, since both the applicability and optimizability conditions of DG_4 are satisfied. Then, the system outputs $DGRL$, the optimized version of $DGRL$, $DCRL$, and the optimized $DCRL$ programs, in this order. The $DGRL$ program will be input to the system for finding the possible equivalent and optimizable output programs. All the direct edges are checked. Although some transformation schemas are applicable, no programs will be output, since they have already been output for the input $DGLR$ program. Finally, the $DCRL$ program is input to the system. The transformation schemas TG_2 and TDG_3 are applied resulting in the TG and $TDGRL$ programs, since the applicability and the optimizability of these transformation schemas are satisfied. Therefore, the system outputs TG, the optimized version of TG, $TDGRL$, and the optimized $TDGRL$ programs, in this order. The TG and $TDGLR$ programs are further input to the system and some transformation schemas are checked to be applied, but no programs are left that are ensured to be more efficient than the input program, and the ones that are ensured to be more efficient have already been output. So, the system stops.

Chapter 8

Conclusions

I have shown that logic program transformation can be fully automated by using the generalization schemas and duality schemas given in this thesis. The applicability conditions of these transformation schemas ensure the equivalence of the input and output program schemas, but they do not guarantee to have a more efficient output program. The integration of optimizability conditions into the transformation schemas provides the verification of the optimizability of the output program of an applicable transformation schema.

In this research, I have also validated the transformation schemas by using equivalence verification. The correctness proofs of the proposed transformation schemas are in [9].

A prototype transformation system was developed with a database of the program schema patterns and the transformation schemas given in this thesis. I have defined a language for representing the programs, program schema patterns, and transformation schemas. For checking the applicability and some of the optimizability conditions, the theorem prover PTTP was integrated into the system. For verifying the optimizability conditions where the check for partial evaluation is done, and for optimization of the output programs of the transformation schemas, the partial evaluator Mixtus was integrated into the system.

8.1 Contributions of This Research

The generalization schemas that are presented in this thesis are actually extensions of Flener and Deville's generalization schemas [20] by extending the program schema and the transformation schema representations, and the eureka discovery step is fully eliminated by the generalization schemas that we have in this thesis. Therefore, we achieve generalization of programs beyond one tail and prefix composition of the result parameter.

The program schemas, which are proposed in this thesis, are represented in first-order, whereas they were represented in second-order in [20]. The transformation schema representation is also extended from 3-tuples to 5-tuples by integrating the optimizability conditions.

New generalization schemas, namely simultaneous tupling-and-descending generalization schemas, are pre-compiled in this research. Validation of the proposed transformation schemas by equivalence verification [9] is another contribution of this research. The proposed prototype transformation system is also an important contribution of this research, which shows that the proposed transformation schemas can be used in a practical system.

We can also compare the results of this research with Fuchs et al's results [24, 57, 58, 47]. We assume that the schema of the input program is known, which is achieved by matching in their work. Our assumption is reasonable, since our system is developed to be integrated into a schema-based logic program development environment.

We have a different representation for the transformation schemas, which is better than their representation in some respects. For instance, in our work, the transformation schema selection is based on the applicability and optimizability conditions, whereas this process is based on matching and precedence in their work, which means they do not use all the semantic knowledge about the program.

We *now* focus on transforming entire programs, but not *yet* on transforming conjunctions inside programs. They could transform also conjunctions inside

programs. This is one of the important future work directions that I also discuss in the next section.

8.2 Future Work

Although the integration of optimizability conditions into the transformation schemas provides the verification of the optimizability of the output program of an applicable transformation schema, these conditions do not always ensure improved performance (or complexity) of the output program wrt the input program. Therefore, the optimization conditions have to be identified to ensure the efficiency gain by an applicable transformation schema, as I discussed in Chapter 6.

I have only dealt with the declarative semantics of the typed definite programs in closed frameworks for program transformation. Future work can be to extend the program schema patterns for typed normal programs in open frameworks. This may also require extensions in the transformation schemas.

Other future work can be to validate the transformation schemas by using automated complexity analyzers like Le Charlier's GAIA [35], or Debray and Lin's CASLOG [15]. With these analyzers, the transformation schemas can also be better validated in terms of performance.

There exist also some extensions that have to be done on the system to make it work better. As I mentioned in Section 7.1, the representation language must be extended to provide flexibility for representing more generic program schema patterns, e.g., eliminating the special treatment of e, which is actually a second-order variable existing in the current database of the program schema patterns and transformation schema representations, and also representing the indexed relations, like Bauvir's second-order representation language [2]. Also if we think in terms of performance, the performance of the system can be improved by maintaining a dynamic list that keeps track of the results of the applicability and optimizability condition checks.

Of course, consideration of other program schemas, and searching for other

pre-compilable transformation techniques are the extensions that can be done on this research. Pre-compilation of the loop merging strategy seems to be the most important one, since the transformation schemas given in this thesis focus on transforming entire programs, whereas transforming conjunctions inside a program may result in better optimizations of the programs. The loop merging strategy can be pre-compiled by extending the definition of the transformation schemas into recursively defined transformation schemas. Since nested programs were already processed by the prototype system, the transformation schemas can be extended to transform conjunctions inside programs with little work on theory of the transformation schemas.

Therefore, this research is an important step on the way to a complete transformation system that can be integrated in a logic program development environment.

Bibliography

[1] T. Batu. *Schema-Guided Transformations of Logic Algorithms.* Senior Project Report, Bilkent University, Department of Computer Science, 1996.

[2] C. Bauvir. *An Architecture and an Abstract Data Type for an Inductive Schema-Guided Logic Program Synthesizer.* M.Sc. Thesis, University of Namur, Institut d'Informatique, 1996.

[3] R.S. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[4] R.S. Bird. The promotion and accumulation strategies in transformational programming. *ACM TOPLAS* 6(4):487–504, 1984.

[5] D.R. Brough and C.J. Hogger. Compiling associativity into logic programs. *Journal of Logic Programming* 4:345–359, 1987.

[6] D.R. Brough and C.J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing* 9:115–134, 1991.

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

[8] H. Büyükyıldız and P. Flener. Generalized logic program transformation schemas. In: N.E. Fuchs (ed), *Proc. of LOPSTR'97*, LNCS. Springer-Verlag, forthcoming.

[9] H. Büyükyıldız and P. Flener. *Correctness Proofs of Transformation Schemas.* Technical Report BU-CEIS-9713. Bilkent University, Department of Computer Science, 1997.

[10] E. Chasseur and Y. Deville. Logic program schemas, semi-unification and constraints. In: N.E. Fuchs (ed), *Proc. of LOPSTR'97* (this volume).

[11] T.H. Cormen, C.E. Leiserson, and R.R. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[12] A. Cortesi, B. Le Charlier, and S. Rossi. Specification-based automatic verification of Prolog programs. In: J. Gallagher (ed), *Proc. of LOPSTR'96*, pp. 38–57. LNCS 1207. Springer-Verlag, 1997.

[13] S.K. Debray. Optimizing almost-tail-recursive Prolog programs. In: *Proc. of IFIP'85*, pp. 204–219. LNCS 201. Springer-Verlag, 1985.

[14] S.K. Debray. Unfold/fold transformations and loop optimization of logic programs. In: *Proc. of SIGPLAN'88, Conference on Programming Language Design and Implementation. SIGPLAN Notices* 23(7):297–307, 1988.

[15] S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM TOPLAS* 15(5):826–875, 1993.

[16] Y. Deville. *Logic Programming: Systematic Program Development.* Addison Wesley, 1990.

[17] Y. Deville and J. Burnay. Generalization and program schemata: A step towards computer-aided construction of logic programs. In: E.L. Lusk and R.A. Overbeek (eds), *Proc. of NACLP'89*, pp. 409–425. The MIT Press, 1989.

[18] Y. Deville and K.-K. Lau. Logic program synthesis: A survey. *Journal of Logic Programming, Special Issue on 10 Years of Logic Programming* 19–20:321–350, 1994.

[19] P. Flener. *Logic Program Schemata: Synthesis and Analysis.* Technical Report BU-CEIS-9502. Bilkent University, Department of Computer Science, 1995.

[20] P. Flener and Y. Deville. Logic program transformation through generalization schemata. Extended abstract in: M. Proietti (ed), *Proc. of LOPSTR'95*, pp. 171–173. LNCS 1048. Springer-Verlag, 1996. Full version in:

M. Proietti (ed), *Pre-proc. of LOPSTR'95.*

[21] P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In: N.E. Fuchs (ed), *Proc. of LOPSTR'97*, LNCS. Springer-Verlag, forthcoming.

[22] P. Flener, K.-K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. In: M. Lowry and Y. Ledru (eds), *Proc. of ASE'97.* IEEE Computer Society Press, forthcoming.

[23] P. Flener and S. Yılmaz. *Inductive Synthesis of Recursive Logic Programs: Achievements and Prospects.* Submitted to Journal of Logic Programming.

[24] N.E. Fuchs and M.P.J. Fromherz. Schema-based transformation of logic programs. In: T. Clement and K.-K. Lau (eds), *Proc. of LOPSTR'91*, pp. 111–125. Springer-Verlag, 1992.

[25] T.S. Gegg-Harrison. *Basic Prolog Schemata.* Technical Report CS-1989-20, Duke University, Department of Computer Science, 1989.

[26] T.S. Gegg-Harrison. Representing logic program schemata in λProlog. In: L. Sterling (ed), *Proc. of ICLP'95*, pp. 467–481. The MIT Press, 1995.

[27] T.S. Gegg-Harrison. Extensible logic program schemata. In: J. Gallagher (ed), *Proc. of LOPSTR'96*, pp. 256–274. LNCS 1207. Springer-Verlag, 1997.

[28] A. Hamfelt and J. Fischer Nilsson. *Towards a Logic Programming Methodology based on Higher-Order Predicates.* Submitted to New Generation Computing.

[29] A. Hamfelt and J. Fischer Nilsson. Declarative logic programming with primitive recursion relations on lists. In: L. Sterling (ed), *Proc of JICSLP'96.* The MIT Press.

[30] J. Hannan and D. Miller. Uses of higher-order unification for implementing program transformers. In: R.A. Kowalski and K.A. Bowen (eds), *Proc. of ICLP'88*, pp. 942–959. The MIT Press, 1993.

[31] Å. Hansson and S.-Å. Tärnlund. Program transformation by a function that maps simple lists onto d-lists. In: *Proc. of Logic Programming Workshop*, pp. 225–229, 1980.

[32] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica* 11:31–55, 1978.

[33] H.J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In: *Proc. of the Ninth ACM Symposium on Principles of Programming Languages*, pp. 255–267, 1982.

[34] K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. *Steadfast Logic Programs.* Submitted to Journal of Logic Programming.

[35] B. Le Charlier, S. Rossi, and A. Cortesi. Specification-based automatic verification of Prolog programs. In: J. Gallagher (ed), *Proc. of LOPSTR'96.* LNCS 1207. Springer-Verlag, 1997.

[36] J.M. Lever. Program equivalence, program development and integrity checking. In: T. Clement and K.-K. Lau (eds), *Proc. of LOPSTR'91*, pp. 1–12. Springer-Verlag, 1992.

[37] M.J. Maher. Equivalences of logic programs. In: J. Minker (ed), *Foundations of Deductive Databases*, pp. 627–658. Morgan Kaufmann, 1988.

[38] E. Marakakis and J.P. Gallagher. Schema-based top-down design of logic programs using abstract data types. In: L. Fribourg and F. Turini (eds), *Proc. of LOPSTR'94*, pp. 138–153. LNCS 883, 1994.

[39] K. Marriott and H. Søndergaard. Difference-list transformation for Prolog. *New Generation Computing* 11:125–157, 1993.

[40] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM TOPLAS* 4(3):402–454, 1982.

[41] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming* 19(20):261–320, 1994.

[42] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys* 28(2):360–414, 1996.

[43] M. Proietti and A. Pettorossi. Synthesis of eureka precidates for developing logic programs. In: N. Jones (ed), *Proc. of ESOP'90*, pp. 306–325. LNCS 432. Springer-Verlag, 1990.

[44] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. In: J. Maluszynski and M. Wirsing (eds), *Proc. of PLILP'91*, pp. 347–358. LNCS 528. Springer-Verlag, 1991.

[45] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming* 16:123–161, 1993.

[46] M. Proietti and A. Pettorossi. Completeness of some transformation strategies for avoiding unnecessary logical variables. In: P. van Hentrenryck (ed), *Proc. of ICLP'94*, pp. 714–729. The MIT Press, 1994.

[47] J. Richardson and N.E. Fuchs. Development of correct transformation schemata for Prolog programs. In: N.E. Fuchs (ed), *Proc. of LOPSTR'97*, LNCS. Springer-Verlag, forthcoming.

[48] D. Sahlin. *An Automatic Partial Evaluator of Full Prolog.* Ph.D. Thesis, Swedish Institute of Computer Science, 1991.

[49] H. Seki and K. Furukawa. Notes on transformation techniques for generate and test logic programs. In: *Proc. of ISLP'87*, pp. 215–223, 1987.

[50] Z. Somogyi, F. Henderson, and T. Conway. Mercury: An efficient purely declarative logic programming language. In: *Proc. of the Australian Computer Science Conference*, pp. 499–512, 1995.

[51] M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In: J. Lloyd (ed), *Proc. of ISLP'95*, pp. 465–479. The MIT Press, 1995.

[52] L.S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In: J.-M. Jacquet (ed), *Constructing Logic Programs*, pp. 127–140. John Wiley, 1993.

[53] L.S. Sterling and E.Y. Shapiro. *The Art of Prolog, Advanced Programming Techniques.* Second edition, The MIT Press, 1994.

[54] M.E. Stickel. *A Prolog Technology Theorem Prover: A New Exposition and Implementation in Prolog.* Technical Note 464, SRI International, Artificial Intelligence Center, 1989. (a longer version of the reference below that includes annotated code)

[55] M.E. Stickel. A Prolog technology theorem prover: A new exposition and implementation in Prolog. *Theoretical Computer Science* 104:109–128, 1992.

[56] V.F. Turchin. The concept of a supercompiler. *ACM TOPLAS* 8(3):292–325, 1986.

[57] W.W. Vasconcelos and N.E. Fuchs. *Opportunistic Logic Program Analysis and Optimisation: Enhanced Schema-Based Transformations for Logic Programs and their Usage in an Opportunistic Framework for Program Analysis and Optimisation.* Technical Report 95-24. Universität Zürich, Institut fur Informatik, 1995.

[58] W.W. Vasconcelos and N.E. Fuchs. An opportunistic approach for logic program analysis and optimisation using enhanced schema-based transformations. In: M. Proietti (ed), *Proc. of LOPSTR'95*, pp. 174–188. LNCS 1048. Springer-Verlag, 1996.

[59] P. Wadler. Deforestration: Transforming programs to eliminate trees. *Theoretical Computer Science* 73:231–248, 1990.

[60] M. Waldau. Formal validation of transformation schemata. In: T. Clement and K.-K. Lau (eds), *Proc. of LOPSTR'91*, pp. 97–110. Springer-Verlag, 1992.

[61] S. Yılmaz. *Inductive Synthesis of Recursive Logic Programs.* M.Sc. Thesis, Bilkent University, Department of Computer Science, 1997.

[62] J. Zhang and PP.W. Grant. An automatic difference-list transformation algorithm for Prolog. In: *Proc. of ECAI'88*, pp. 320–325, 1988.

A

README File of the Prototype Transformation System

The files of the prototype transformation system TRANSYS:

transys.pl	: top-level relations
dbase.pl	: database of the program schema patterns and transformation schemas
dedotify.pl	: manipulate the templates of the program schema patterns during particularization
prover.pl	: prove the applicability and post-optimizability conditions
mverify.pl	: check the post-optimizability conditions of the transformation schemas
hprint.pl	: print the output programs of the system on the current output stream
utilities.pl	: low-level relations called by the other programs
pttp.pl	: PTTP
pttpq.pl	: PTTP Prolog code for inference counting and timing;
mixtus	: the executable file of the partial evaluator Mixtus

For properly running TRANSYS, first write mixtus in the command line, which calls first the available Sicstus Prolog interpreter, then load transys.pl.

Then you can call the top-level relation *transys*/1 with the input program. The calls of sample example runs are in a file called *run_exs.pl*.

B

Sample Output of the Prototype System

Below are some parts of the output for transforming the *DGLR infix_flat* program:

```
| ?- transys(lp(dglr,[i_flat,minimal,solve,nonminimal,decompose,process,
compose],
[lp(rs,[if(minimal(X),X=void)]),
lp(rs,[if(solve(X,Y),Y=[])]),
lp(rs,[if(nonminimal(X), X=bt(_,_,_))]),
lp(rs,[if(decompose(X,E,T1,T2),X=bt(T1,E,T2))]),
lp(rs,[if(process(E,HF), HF=[E])]),
lp(rs,[if(compose(P,Q,R),and(P=[],Q=R)),if(compose(P,Q,R),and(P = [HP|TP],
and(compose(TP,Q,TR), R = [HP|TR])))])],[[],2,1,2,[2],[btree,list]])).


PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_dsdg

Associativity

Left_Identity

Right_Identity

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_dsdg

Minimality

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).
```

```
% p1(A,B,C):-p(A,B,C)
p1(A, B, [A|B]).

PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_dg1

Associativity

Left_Identity

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_dg1

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).

% p1(A,B,C):-p(A,B,C)
p1(A, B, C) :-
        compose1(B, A, C).

% compose1(A,B,C):-compose(A,[B],C)
compose1([], A, [A]).
compose1([A|B], C, D) :-
        compose1(B, C, E),
        D=[A|E].

PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_dg4

Associativity

Left_Identity

Right_Identity

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_dg4

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).

% p1(A,B,C):-p(A,B,C)
p1(A, B, [A|B]).

*****************************************************
OUTPUT_OF_THE_TRANSFORMATION_AS_AN_INSTANCE_OF dgrl
*****************************************************

i_flat(A,B):-i_flat_d2(A,B,[]).

i_flat_d2(A,B,C):-minimal(A),solve(A,D),compose(D,C,B).

i_flat_d2(A,B,C):-nonminimal(A),decompose(A,E,F,G),compose([],C,H),
          i_flat_d2(G,I,H),process(E,J),compose(J,I,K),
```

```
  i_flat_d2(F,L,K),B=L.

minimal(M):-M=void.

solve(M,N):-N=[].

nonminimal(M):-M=bt(O,P,Q).

decompose(M,R,S,T):-M=bt(S,R,T).

process(R,U):-U=[R].

compose(V,W,X):-V=[],W=X.

compose(V,W,X):-V=[Y|Z],compose(Z,W,AA),X=[Y|AA].

*******************************************
OPTIMIZED_dgrl_PROGRAM
*******************************************
{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/prog}
i_flat(A, B) :-
        i_flat1(A, B).

% i_flat1(A,B):-i_flat(A,B)
i_flat1(A, B) :-
        i_flat_d21(A, B).

% i_flat_d21(A,B):-i_flat_d2(A,B,[])
i_flat_d21(void, []).
i_flat_d21(bt(A,B,C), D) :-
        i_flat_d21(C, E),
        i_flat_d22(A, D, B, E).

% i_flat_d22(A,B,C,D):-i_flat_d2(A,B,[C|D])
i_flat_d22(void, [A|B], A, B).
i_flat_d22(bt(A,B,C), D, E, F) :-
        i_flat_d21(C, E, F, G),
        i_flat_d22(A, D, B, G).

% i_flat_d21(A,B,C,D):-i_flat_d2(A,D,[B|C])
i_flat_d21(void, A, B, [A|B]).
i_flat_d21(bt(A,B,C), D, E, F) :-
        i_flat_d21(C, D, E, G),
        i_flat_d21(A, B, G, H),
        F=H.

*****************************************************
OUTPUT_OF_THE_TRANSFORMATION_AS_AN_INSTANCE_OF dcrl
*****************************************************

i_flat(A,B):-minimal(A),solve(A,B).

i_flat(A,B):-nonminimal(A),decompose(A,C,D,E),i_flat(D,F),i_flat(E,G),
```

```
     H=[],compose(G,H,I),process(C,J),compose(J,I,K),
     compose(F,K,L),B=L.

minimal(M):-M=void.

solve(M,N):-N=[].

nonminimal(M):-M=bt(O,P,Q).

decompose(M,R,S,T):-M=bt(S,R,T).

process(R,U):-U=[R].

compose(V,W,X):-V=[],W=X.

compose(V,W,X):-V=[Y|Z],compose(Z,W,AA),X=[Y|AA].

*****************************************
OPTIMIZED_dcrl_PROGRAM
*****************************************
{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/prog}
i_flat(A, B) :-
        i_flat1(A, B).

% i_flat1(A,B):-i_flat(A,B)
i_flat1(void, []).
i_flat1(bt(A,B,C), D) :-
        i_flat1(A, E),
        i_flat1(C, F),
        compose1(F, G),
        compose2(E, B, G, D).

% compose1(A,B):-compose(A,[],B)
compose1([], []).
compose1([A|B], C) :-
        compose1(B, D),
        C=[A|D].

% compose2(A,B,C,D):-compose(A,[B|C],D)
compose2([], A, B, [A|B]).
compose2([A|B], C, D, [A|E]) :-
        compose2(B, C, D, E).


PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_dg2

Associativity

Left_Identity

Right_Identity

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_dg2
```

```
{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).

% p1(A,B,C):-p(A,B,C)
p1(A, B, C) :-
        compose1(B, A, C).

% compose1(A,B,C):-compose(A,[B],C)
compose1([], A, [A]).
compose1([A|B], C, D) :-
        compose1(B, C, E),
        D=[A|E].

PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_tg2

Associativity

Left_Identity

Right_Identity

Exclusive_OR

Minimality

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_tg2

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).

% p1(A,B,C):-p(A,B,C)
p1(A, B, [A|B]).

PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_tdg3

Associativity

Left_Identity

Right_Identity

Exclusive_OR

Minimality

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_tdg3

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).
```

```
% p1(A,B,C):-p(A,B,C)
p1(A, B, [A|B]).

PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_tdg4

Associativity

Left_Identity

Right_Identity

Exclusive_OR

Minimality

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_tdg4

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).

% p1(A,B,C):-p(A,B,C)
p1(A, B, C) :-
        compose1(B, A, C).

% compose1(A,B,C):-compose(A,[B],C)
compose1([], A, [A]).
compose1([A|B], C, D) :-
        compose1(B, C, E),
        D=[A|E].

PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_dsdc

Associativity

Left_Identity

Right_Identity

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_dsdc

Minimality

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).

% p1(A,B,C):-p(A,B,C)
p1(A, B, C) :-
        compose1(B, A, C).

% compose1(A,B,C):-compose(A,[B],C)
```

```
compose1([], A, [A]).
compose1([A|B], C, D) :-
        compose1(B, C, E),
        D=[A|E].

**************************************************
OUTPUT_OF_THE_TRANSFORMATION_AS_AN_INSTANCE_OF tg
**************************************************

i_flat(A,B):-i_flat_t([A],B).

i_flat_t(C,B):-C=[],B=[].

i_flat_t(C,B):-C=[A|D],minimal(A),i_flat_t(D,E),solve(A,F),compose(F,E,B).

i_flat_t(C,B):-C=[A|D],nonminimal(A),decompose(A,G,H,I),minimal(H),minimal(I),
       i_flat_t(D,E),process(G,F),compose(F,E,B).

i_flat_t(C,B):-C=[A|D],nonminimal(A),decompose(A,G,J,K),minimal(J),
               nonminimal(K),i_flat_t([K|D],E),process(G,F),compose(F,E,B).

i_flat_t(C,B):-C=[A|D],nonminimal(A),decompose(A,G,L,M),nonminimal(L),
               minimal(M),minimal(N),decompose(O,G,N,M),i_flat_t([L,O|D],B).

i_flat_t(C,B):-C=[A|D],nonminimal(A),decompose(A,G,P,Q),nonminimal(P),
               nonminimal(Q),minimal(R),minimal(S),decompose(O,G,R,S),
               i_flat_t([P,O,Q|D],B).

minimal(T):-T=void.

solve(T,U):-U=[].

nonminimal(T):-T=bt(V,W,X).

decompose(T,Y,Z,AA):-T=bt(Z,Y,AA).

process(Y,AB):-AB=[Y].

compose(AC,AD,AE):-AC=[],AD=AE.

compose(AC,AD,AE):-AC=[AF|AG],compose(AG,AD,AH),AE=[AF|AH].

******************************************
OPTIMIZED_tg_PROGRAM
******************************************

i_flat(A, B) :-
        i_flat1(A, B).

% i_flat1(A,B):-i_flat(A,B)
i_flat1(A, B) :-
        'i_flat_t.1'(A, B).
```

```
% 'i_flat_t.1'(A,B):-i_flat_t([A],B)
'i_flat_t.1'(void, []).
'i_flat_t.1'(bt(void,A,void), [A]).
'i_flat_t.1'(bt(void,A,bt(B,C,D)), E) :-
        'i_flat_t.1'(bt(B,C,D), F),
        E=[A|F].
'i_flat_t.1'(bt(bt(A,B,C),D,void), E) :-
        'i_flat_t.bt2'(A, B, C, D, [], E).
'i_flat_t.1'(bt(bt(A,B,C),D,bt(E,F,G)), H) :-
        'i_flat_t.bt2'(A, B, C, D, E, F, G, [], H).

% 'i_flat_t.bt2'(A,B,C,D,[],E):-i_flat_t([bt(A,B,C),bt(void,D,void)],E)
'i_flat_t.bt2'(void, A, void, B, C, [A,B|D]) :-
        i_flat_t2(C, D).
'i_flat_t.bt2'(void, A, bt(B,C,D), E, F, G) :-
        'i_flat_t.bt2'(B, C, D, E, F, H),
        G=[A|H].
'i_flat_t.bt2'(bt(A,B,C), D, void, E, F, G) :-
        'i_flat_t.bt2'(A, B, C, D, [bt(void,E,void)|F], G).
'i_flat_t.bt2'(bt(A,B,C), D, bt(E,F,G), H, I, J) :-
        'i_flat_t.bt2'(A, B, C, D, [bt(E,F,G),bt(void,H,void)|I], J).

% i_flat_t2(A,B):-i_flat_t(A,B)
i_flat_t2([], []).
i_flat_t2([void|A], B) :-
        i_flat_t2(A, B).
i_flat_t2([bt(void,A,void)|B], [A|C]) :-
        i_flat_t2(B, C).
i_flat_t2([bt(void,A,bt(B,C,D))|E], F) :-
        i_flat_t2([bt(B,C,D)|E], G),
        F=[A|G].
i_flat_t2([bt(bt(A,B,C),D,void)|E], F) :-
        i_flat_t2([bt(A,B,C),bt(void,D,void)|E], F).
i_flat_t2([bt(bt(A,B,C),D,bt(E,F,G))|H], I) :-
        i_flat_t2([bt(A,B,C),bt(void,D,void),bt(E,F,G)|H], I).

% 'i_flat_t.bt2'(A,B,C,D,E,F,G,[],H):-i_flat_t([bt(A,B,C),bt(void,D,void),
%bt(E,F,G)],H)
'i_flat_t.bt2'(void, A, void, B, C, D, E, F, [A,B|G]) :-
        'i_flat_t.bt3'(C, D, E, F, G).
'i_flat_t.bt2'(void, A, bt(B,C,D), E, F, G, H, I, J) :-
        'i_flat_t.bt2'(B, C, D, E, F, G, H, I, K),
        J=[A|K].
'i_flat_t.bt2'(bt(A,B,C), D, void, E, F, G, H, I, J) :-
        'i_flat_t.bt2'(A, B, C, D, void, E, void, [bt(F,G,H)|I], J).
'i_flat_t.bt2'(bt(A,B,C), D, bt(E,F,G), H, I, J, K, L, M) :-
        'i_flat_t.bt2'(A, B, C, D, E, F, G, [bt(void,H,void),bt(I,J,K)|L], M).

% 'i_flat_t.bt3'(A,B,C,D,E):-i_flat_t([bt(A,B,C)|D],E)
'i_flat_t.bt3'(void, A, void, B, [A|C]) :-
        i_flat_t2(B, C).
'i_flat_t.bt3'(void, A, bt(B,C,D), E, [A|F]) :-
        'i_flat_t.bt3'(B, C, D, E, F).
```

```
'i_flat_t.bt3'(bt(A,B,C), D, void, E, F) :-
        'i_flat_t.bt3'(A, B, C, [bt(void,D,void)|E], F).
'i_flat_t.bt3'(bt(A,B,C), D, bt(E,F,G), H, I) :-
        'i_flat_t.bt3'(A, B, C, [bt(void,D,void),bt(E,F,G)|H], I).

**************************************************
OUTPUT_OF_THE_TRANSFORMATION_AS_AN_INSTANCE_OF tdgrl
**************************************************

i_flat(A,B):-i_flat_td2([A],B,[]).

i_flat_td2(C,B,D):-C=[],B=D.

i_flat_td2(C,B,D):-C=[A|E],minimal(A),i_flat_td2(E,F,D),solve(A,G),
           compose(G,F,B).

i_flat_td2(C,B,D):-C=[A|E],nonminimal(A),decompose(A,H,I,J),minimal(I),
           minimal(J),i_flat_td2(E,F,D),process(H,G),compose(G,F,B).

i_flat_td2(C,B,D):-C=[A|E],nonminimal(A),decompose(A,H,K,L),minimal(K),
                   nonminimal(L),i_flat_td2([L|E],F,D),
   process(H,G),compose(G,F,B).

i_flat_td2(C,B,D):-C=[A|E],nonminimal(A),decompose(A,H,M,N),nonminimal(M),
           minimal(N),minimal(O),decompose(P,H,O,N),
   i_flat_td2([M,P|E],B,D).

i_flat_td2(C,B,D):-C=[A|E],nonminimal(A),decompose(A,H,Q,R),nonminimal(Q),
           nonminimal(R),minimal(S),minimal(T),decompose(P,H,S,T),
   i_flat_td2([Q,P,R|E],B).

minimal(U):-U=void.

solve(U,V):-V=[].

nonminimal(U):-U=bt(W,X,Y).

decompose(U,Z,AA,AB):-U=bt(AA,Z,AB).

process(Z,AC):-AC=[Z].

compose(AD,AE,AF):-AD=[],AE=AF.

compose(AD,AE,AF):-AD=[AG|AH],compose(AH,AE,AI),AF=[AG|AI].

*****************************************
OPTIMIZED_tdgrl_PROGRAM
*****************************************
{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/prog}
i_flat(A, B) :-
        i_flat1(A, B).

% i_flat1(A,B):-i_flat(A,B)
```

```
i_flat1(A, B) :-
        'i_flat_td2.1'(A, B).

% 'i_flat_td2.1'(A,B):-i_flat_td2([A],B,[])
'i_flat_td2.1'(void, []).
'i_flat_td2.1'(bt(void,A,void), [A]).
'i_flat_td2.1'(bt(void,A,bt(B,C,D)), E) :-
        'i_flat_td2.1'(bt(B,C,D), F),
        E=[A|F].
'i_flat_td2.1'(bt(bt(A,B,C),D,void), E) :-
        'i_flat_td2.bt2'(A, B, C, D, [], E).
'i_flat_td2.1'(bt(bt(A,B,C),D,bt(E,F,G)), H) :-
        i_flat_td2([bt(A,B,C),bt(void,D,void),bt(E,F,G)], H).

% 'i_flat_td2.bt2'(A,B,C,D,[],E):-i_flat_td2([bt(A,B,C),bt(void,D,void)],E,[])
'i_flat_td2.bt2'(void, A, void, B, C, D) :-
        i_flat_td21(C, E),
        D=[A,B|E].
'i_flat_td2.bt2'(void, A, bt(B,C,D), E, F, G) :-
        'i_flat_td2.bt2'(B, C, D, E, F, H),
        G=[A|H].
'i_flat_td2.bt2'(bt(A,B,C), D, void, E, F, G) :-
        'i_flat_td2.bt2'(A, B, C, D, [bt(void,E,void)|F], G).
'i_flat_td2.bt2'(bt(A,B,C), D, bt(E,F,G), H, I, J) :-
        i_flat_td2([bt(A,B,C),bt(void,D,void),bt(E,F,G),bt(void,H,void)|I], J).

% i_flat_td21(A,B):-i_flat_td2(A,B,[])
i_flat_td21([], []).
i_flat_td21([void|A], B) :-
        i_flat_td21(A, C),
        B=C.
i_flat_td21([bt(void,A,void)|B], C) :-
        i_flat_td21(B, D),
        C=[A|D].
i_flat_td21([bt(void,A,bt(B,C,D))|E], F) :-
        i_flat_td21([bt(B,C,D)|E], G),
        F=[A|G].
i_flat_td21([bt(bt(A,B,C),D,void)|E], F) :-
        i_flat_td21([bt(A,B,C),bt(void,D,void)|E], F).
i_flat_td21([bt(bt(A,B,C),D,bt(E,F,G))|H], I) :-
        i_flat_td2([bt(A,B,C),bt(void,D,void),bt(E,F,G)|H], I).


PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_tg1

Associativity

Left_Identity

Right_Identity

Exclusive_OR
```

```
Minimality

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_tg1

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).

% p1(A,B,C):-p(A,B,C)
p1(A, B, C) :-
        compose1(B, A, C).

% compose1(A,B,C):-compose(A,[B],C)
compose1([], A, [A]).
compose1([A|B], C, D) :-
        compose1(B, C, E),
        D=[A|E].

PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_dstdg

Associativity

Left_Identity

Right_Identity

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_dstdg

Minimality

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).

% p1(A,B,C):-p(A,B,C)
p1(A, B, C) :-
        compose1(B, A, C).

% compose1(A,B,C):-compose(A,[B],C)
compose1([], A, [A]).
compose1([A|B], C, D) :-
        compose1(B, C, E),
        D=[A|E].

PTTP_IS_CHECKING_THE_APPLICABILITY_CONDITIONS_OF_tdg2

Associativity

Left_Identity

Right_Identity

Exclusive_OR
```

```
Minimality

PTTP_AND_MIXTUS_CHECKING_THE_OPTIMIZABILITY_CONDITIONS_OF_tdg2

{consulting for mixtus: /csgrad/halime/cs599/thesis/GENSYS/goal}
p(A, B, C) :-
        p1(A, B, C).

% p1(A,B,C):-p(A,B,C)
p1(A, B, C) :-
        compose1(B, A, C).

% compose1(A,B,C):-compose(A,[B],C)
compose1([], A, [A]).
compose1([A|B], C, D) :-
        compose1(B, C, E),
        D=[A|E].

true ?
```