# Partial Evaluation of Multi-Paradigm Declarative Languages:
## Foundations, Control, Algorithms and Efficiency

**Elvira Albert Albiol**
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Memoria presentada para optar al título de:

**Doctora en Informática**

Directores:

**Germán Vidal Oriola**
**María Alpuente Frasnedo**

Tribunal de lectura:

| | | |
|---|---|---|
| **Presidente:** | **Isidro Ramos Salavert** | **U.P. Valencia** |
| **Vocales:** | **Moreno Falaschi** | **U. Udine** |
| | **Michael Hanus** | **C.A.U. Kiel** |
| | **Mario Rodríguez Artalejo** | **U.C. Madrid** |
| **Secretario:** | **Ricardo Peña Marí** | **U.C. Madrid** |

Valencia, 2001

# Partial Evaluation of Multi-Paradigm Declarative Languages:

## Foundations, Control, Algorithms and Efficiency

**Elvira Albert Albiol**
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

A thesis submitted for the the degree of:

**Doctor of Philosophy**

Advisors:

**Germán Vidal Oriola**
**María Alpuente Frasnedo**

Committee:

| | |
|---|---|
| **Isidro Ramos Salavert** | **U.P. Valencia** |
| **Moreno Falaschi** | **U. Udine** |
| **Michael Hanus** | **C.A.U. Kiel** |
| **Mario Rodríguez Artalejo** | **U.C. Madrid** |
| **Ricardo Peña Marí** | **U.C. Madrid** |

Valencia, 2001

# Acknowledgements

The development of this PhD thesis has been made possible by the help of many people. Herewith, I would like to express my gratitude for all their contributions to this work.

I am indebted to Moreno Falaschi for giving me the first opportunity to start work on this thesis in 1998, when he invited me to participate in his research project. I also appreciate his kindness during my stay at the University of Udine.

In Germany, Michael Hanus willingly accepted my collaboration on his project in 1999. Without a doubt, his discussions have been an important contribution to this thesis. I also appreciate his hospitality. I would like to thank his collaborator Frank Steiner, who has always been available during my stays, as well as all the members of Lehrstuhl für Informatik II at RWTH Aachen University for providing a most friendly and relaxed atmosphere.

The Computer Science Department of the Portland State University granted me a research stay of two months in 2000. I want to thank Sergio Antoy for his scientific collaboration and for being so pleasant during this research stay at PSU.

Special thanks to my colleagues of the group "Extensions of Logic Programming" for their valuable help in many situations: to César Ferri, Javier Oliver, Santiago Escobar, Salvador Lucas, María José Ramírez, Alicia Villanueva, Moisés Mascuñán, Carlos Herrero, Pascual Julián, Ginés Moreno, José Hernández, Marisa Llorens, etc. I want to extend thanks to the members of the "Grupo de Programación Lógica e Ingeniería del Software", especially, to its founder Isidro Ramos who first introduced me to the declarative programming paradigm; and, in general, to those colleagues of the Department of "Sistemas Informáticos y Computación", particularly to my office-mate Emilio Vivancos.

I want to especially acknowledge my supervisors:

- To María Alpuente for her interest in our joint research, and overall, for her interest in the development of this thesis. Undoubtly, my career at the university would have been harder without the wholehearted help and support that María has been offering me since my incorporation in the ELP group that she leads.

- To Germán Vidal for his personal dedication to my scientific training, which has gone beyond his obligation as PhD advisor. His fervent interest in science has constituted an inspiring model to follow during these (almost) four years. Our scientific discussions have inspired me (including the tense ones) and have become part of a memorable experience. I appreciate his imperturbable calm and patience which were essential for the progressive advance of our joint research. Above all, I thank him for being a brilliant guide and an excellent friend.

Finally, some affectionate words for my family. To my grandmother for her love. To my parents, for being so noble and kind. To my sister Lourdes, for her affection. To my sister Manoli, for being so close.

I received several grants from the "Universidad Politécnica de Valencia", "Conselleria de Cultura, Educación y Ciencia D.G. de Enseñanzas Universitarias e Investigación", and CICYT TIC 98-0445-C03-01.

Elvira Albert
Valencia, 2001

# Abstract


Partial evaluation is an automatic technique for program optimization which preserves program semantics. Optimization is achieved by specializing programs w.r.t. parts of their input (hence also called program specialization). Informally, a partial evaluator is a mapping which takes a program $P$ and a function call $C$ and derives a more efficient, specialized program $P_C$ which computes the same values and answers for $C$ (and any of its instances) as $P$ does. The range of its potential applications is extremely large, as witnessed by many successful experiences, e.g., in the fields of pattern recognition, neural network training, or scientific computing [Jones *et al.*, 1993]. This technique has been investigated in the context of a wide variety of declarative programming languages, specially in both the functional and logic programming paradigms (see, e.g., [Consel and Danvy, 1993; Danvy *et al.*, 1996; Gallagher, 1993; Jones *et al.*, 1993; Lloyd and Shepherdson, 1991] and references herein). Recently, a unified framework for the partial evaluation of languages which integrate features from functional and logic programming has been introduced in [Alpuente *et al.*, 1998b, 1999c]. This framework is based on the use of *narrowing* (the standard operational mechanism of functional logic languages) to drive the specialization process. However, the narrowing-driven method of [Alpuente *et al.*, 1998b, 1999c] is not still suitable for designing an effective partial evaluator for a realistic multi-paradigm language like Curry [Hanus, 2000a], Escher [Lloyd, 1994] or Toy [López-Fraguas and Sánchez-Hernández, 1999], as there are many language features which are not considered in this framework.

In this thesis, we develop some methods and techniques which make feasible the definition of effective partial evaluators for these languages. In short, the main contributions of the thesis are:

- The definition of a partial evaluation framework for *residuating* functional logic programs, which generalize logic programs based on concurrent computation models with dynamic scheduling to functional logic programs.

- The formulation of effective control issues to handle *primitive* function symbols (e.g., the sequential conjunction and disjunction, the strict equality, if-then-else

constructs, etc.) in functional logic languages.

- The use of an abstract representation for programs (into which higher-level programs can be automatically translated). This allows us to define a simple and concise method for program specialization that covers all the features of modern multi-paradigm declarative languages.
- The development of a partial evaluator for Curry programs written in Curry itself, which is the first purely declarative partial evaluator for a real multi-paradigm language.
- The effectiveness of our partial evaluation method is also crucial for promoting the wide use of our technology. Therefore, we have also defined a formal framework which helps us to assess the gain in efficiency due to the partial evaluation process.

Parts of this work have been previously presented in [Albert *et al.*, 1998a, 1999a, 2000f, 2001a,b].

# Resumen

La evaluación parcial es una técnica automática para la optimización de los programas que garantiza la preservación de su semántica. La optimización se consigue especializando los programas con respecto a parte de sus datos de entrada (y, por ello, esta técnica recibe también el nombre de *especialización de programas*). Informalmente, un evaluador parcial es una función que toma un programa $P$ y una llamada $C$ y deriva un programa especializado más eficiente $P_c$ que computa las mismas respuestas para $C$ (y cualquiera de sus instancias) que $P$. La potencialidad de la evaluación parcial es enorme, como demuestran las numerosas experiencias positivas realizadas hasta la fecha en los campos de reconocimiento de patrones, entrenamiento de redes neuronales, o computación científica [Jones *et al.*, 1993], entre otros. Esta técnica ha sido investigada en el contexto de una gran variedad de lenguajes de programación declarativos, especialmente en los paradigmas de programación funcional y lógica (ver, e.g., [Consel and Danvy, 1993; Danvy *et al.*, 1996; Gallagher, 1993; Jones *et al.*, 1993; Lloyd and Shepherdson, 1991] y las referencias allí citadas). Recientemente, se ha introducido un marco unificado para la evaluación parcial de lenguajes que integran características de programación lógica y funcional [Alpuente *et al.*, 1998b, 1999c]. Este marco se basa en el uso del *estrechamiento* (el mecanismo operacional estándar de los lenguages lógico funcionales) para guiar el proceso de especialización. Sin embargo, el método basado en estrechamiento de Alpuente *et al.* [1998b, 1999c] no es adecuado para diseñar un evaluador parcial efectivo para un lenguage multi-paradigma moderno como Curry [Hanus, 2000a], Escher [Lloyd, 1994] o Toy [López-Fraguas and Sánchez-Hernández, 1999], ya que existen muchas características de dichos lenguajes que no han sido contempladas en el marco.

En la presente tesis desarrollamos métodos y técnicas que nos permiten definir evaluadores parciales efectivos para los programas escritos en lenguajes lógico funcionales modernos. Brevemente, las principales contribuciones de la tesis son:

- La definición de un marco de evaluación parcial para programas lógico funcionales con *residuación* (los cuales generalizan los programas lógicos mediante modelos de computación concurrentes).

- La formulación de operadores de control efectivos para manipular símbolos de función *primitivos* en lenguajes lógico funcionales.

- El uso de una representación abstracta para los programas que se caracteriza porque los programas de alto nivel se pueden traducir automáticamente a dicha representación. Esto nos permite definir un método simple y conciso para la especialización de programas que cubre todas las características de los lenguajes declarativos multi-paradigma.

- El desarrollo de un evaluador parcial para programas Curry, escrito en el propio lenguaje Curry, que constituye el primer evaluador parcial puramente declarativo para un lenguaje multi-paradigma realista.

- La efectividad de nuestro método de evaluación parcial es también crucial para promover el uso de la tecnología desarrollada. Por ello, se define también un marco formal para estimar la ganancia en eficiencia conseguida por el proceso de evaluación parcial.

Algunas partes de este trabajo han sido previamente publicadas en [Albert *et al.*, 1998a, 1999a, 2000f, 2001a,b].

# Contents

# Chapter 1

# Introduction

In this chapter, we offer a general overview of the state of the art in our area of interest: partial evaluation in functional logic programming. First, we briefly recall the main results attained for declarative languages in general as well as for functional logic languages in particular. Then, we concisely expound the main goals of this thesis.

## 1.1 Functional Logic Programming

Let us begin with a quote of a recent reflection by Robinson [2000]:

> *Functional and logic programming are two only superficially different dialects of computational logic. It is inexplicable that the two idioms have been kept apart for so long within the computational logic repertory. We need a single programming language in which both kinds of programming are possible and can be used in combination with each other.*

After almost two decades of research on the integration of declarative programming languages, the field has reached a maturity in understanding which are the essential features of the different paradigms in order to constitute the *kernel* of a powerful and useful multi-paradigm programming language. In particular, the language should combine in a practical, comprehensive and seamless way the most important features of functional programming (e.g., nested functions, efficient demand-driven functional computations, higher-order), logic programming (e.g., logical variables, partial data structures, constraints, encapsulated search) and concurrent programming (e.g., concurrent computations with synchronization on logical variables). Hanus [1994] presented the most complete overview of functional logic programming.

The most powerful computational model for functional logic languages is based on the combination of two different operational principles: *narrowing* and *residuation*. The narrowing mechanism permits the instantiation of variables in input expressions in order to perform reduction steps (by rewriting) on the instantiated expression. Usually, the instantiation is computed by unifying a subterm of the considered expression with the left-hand side of some program rule. Sometimes, it can be determined by inspecting certain data structures used to guide the process (e.g., the definitional trees of Antoy [1992]). Narrowing is complete in the sense of logic programming (computation of *answers*) as well as in the sense of functional programming (computation of *values*). In order to avoid unnecessary computations and to deal with infinite data structures, several lazy narrowing strategies have been designed (see e.g., [Antoy *et al.*, 2000; Loogen *et al.*, 1993; Moreno-Navarro and Rodríguez-Artalejo, 1992]). Due to the optimality properties w.r.t. the length of the derivations and the number of computed solutions, *needed narrowing* is currently the best lazy narrowing strategy for functional logic programs [Antoy *et al.*, 2000]. On the other hand, the residuation principle is based on the idea of delaying, or *suspending*, the evaluation of function calls until they are ready for a deterministic (i.e., purely functional) evaluation. Residuation preserves the deterministic nature of functions and supports concurrent computations in a natural way. However, the residuation principle is not generally complete since the computation can end up into a state in which no function can be selected for execution. This situation is called *deadlock* and, in our case, corresponds to function calls which are not sufficiently instantiated to perform a reduction step. This may lead to a loss of solutions and is clearly undesirable. Luckily, there exist some methods which provide sufficient criteria to ensure completeness in some particular cases, as those presented in [Boye, 1993; Hanus, 1995b]).

Declarative languages, in general, and functional logic programming languages, in particular, are not broadly widespread in the industry. Concerning functional logic languages, we identify two causes of weakness: the lack of a language accepted as a standard by the research community as well as the fact that current implementations of existing functional logic languages are often experimental systems, not mature enough for the development of industrial applications. Usually, the improvements achieved for integrated languages are hidden by inefficient implementations non attractive to the user. In order to promote the industrial use of integrated languages, it is essential to succeed in developing effective optimization, verification and debugging techniques for functional logic programs.

## 1.2 Program Transformation

Program transformation includes many different methods whose common goal is to derive correct and efficient programs. Given a program $\mathcal{P}$, by program transformation we mean the generation of a new program $\mathcal{P}'$ which solves the same problem and is semantically equivalent to $\mathcal{P}$ but has an improved behaviour w.r.t. a concrete evaluation criterion [Bossi and Cocco, 1993].

The *folding/unfolding* techniques, originally introduced by Burstall and Darlington [1977] for functional programs, have been intensively studied in the area of program transformation. Unfolding consists in the replacement of a function call by its definition, by applying the corresponding substitution. Folding is the inverse transformation, i.e., the replacement of certain code fragment by an equivalent function call. The fold/unfold approximation to program transformation was first adapted to logic programming by [Tamaki and Sato, 1984]. In this context, the unfolding of a logic program consists in applying a resolution step to a subgoal in the body of the clause in all possible forms.

In principle, the fold/unfold technique introduced by Burstall and Darlington does not provide automatic methods for its application. This methodology is commonly based on the construction, by means of an strategy, of a sequence of equivalent programs, each obtained from the preceding ones by using an elementary transformation rule. In addition to the typical fold and unfold transformations, other rules are considered: instantiation, abstraction, definition introduction/elimination, etc. There exists a large class of program optimizations which can be achieved by fold/unfold transformations. Typical instances of this class are:

- Composition

  A popular style of programming, usually called *compositional*, consists in decomposing a given task by means of individual functions which are then combined together. The compositional style of programming is often helpful for writing programs which can be easily understood and proved correct w.r.t. their specifications. However, this programming style often produces inefficient programs because the composition of various functions does no take into account the interactions which may occur among the evaluations of these subtasks (e.g., intermediate data structures may be used). Burstall and Darlington [1977] developed the first approach to improve the efficiency of programs written according to the compositional style of programming. The main goal of their *composition* technique is to eliminate multiple traversals of the same data structure. The idea essentially consists in merging nested function calls, in which the inner function call builds up a composite object which is used by the outer call.

  The most popular composition strategy is Wadler's deforestation [Wadler, 1990].

This is an automatic method which achieves the elimination of data structures used to transfer intermediate results between calls. The technique is based on the application of seven transformation rules over a restricted class of programs, namely, programs in *treeless form*. The composition technique for functional logic programs was investigated in [Moreno, 2000; Alpuente *et al.*, 1999b, 2000]. The main drawback of deforestation is that the technique is expensive to apply.

A more practical approach is the short cut to deforestation (also known as *cheap deforestation*) due to Gill *et al.* [1993]. This is an automatic technique to improve the efficiency of compositional functions based on a single, local rule. The extension of this technique to the functional logic paradigm was first presented in [Albert *et al.*, 2000c,d].

- Tupling

Other (semi-automatic) technique which may improve the efficiency of compositional programs is tupling [Burstall and Darlington, 1977; Pettorossi, 1977]. The aim of tupling is to transform a program which computes a function defined by separated (non-nested) function calls with some common arguments into a single call to a (possibly new) recursive function which returns a tuple of the results of the separate calls. By using tupling, one may avoid multiple accesses to the same data structures or common subcomputations, similarly to the idea of *sharing* which is used in graph rewriting to improve the efficiency of computations in time and space [Baader and Nipkow, 1998].

The technique is difficult to automate and complex analyses are usually required [Chin, 1993; Chin *et al.*, 1999].

- Partial Evaluation

Partial evaluation is an automatic technique for program optimization which preserves program semantics. Optimization is achieved by specializing programs w.r.t. parts of their input. Informally, a partial evaluator is a mapping which takes a program $P$ and a call $C$ and derives a more efficient, specialized program $P_C$ which produces the same values and answers for $C$ (and any of its instances) as $P$ does. Currently, there are many techniques whose optimization power is higher than traditional partial evaluation techniques (e.g., Turchin's supercompilation [Turchin, 1986b], generalized partial computation [Futamura and Nogi, 1988], or conjunctive partial deduction [De Schreye *et al.*, 1999]) which are still known as partial evaluation.

Nevertheless, the optimization potential of these techniques is more limited than the fold/unfold approach on its full power, but they are totally automatic.

## 1.3 Partial Evaluation

Partial evaluation is a source-to-source program transformation technique for specializing programs w.r.t. parts of their input (hence also called *program specialization*). Partial evaluation has been intensively applied in the area of functional programming [Consel and Danvy, 1993; Jones *et al.*, 1993; Turchin, 1986b] and logic programming [Gallagher, 1993; Komorowski, 1982; Lloyd and Shepherdson, 1991; Pettorossi and Proietti, 1994], where it is usually known as *partial deduction*. Although the goals are similar, the general methods are frequently different due to the distinct underlying computational models. The techniques used for the partial evaluation of functional programming are usually based on the reduction of expressions and the propagation of constants. The techniques used in the transformation of logic languages achieve a greater propagation of information thanks to the use of logical variables and unification [Glück and Sørensen, 1994].

In logic programming, the idea of partial evaluation basically consists in performing the following steps [Lloyd and Shepherdson, 1991]:

1. Given a logic program $\mathcal{P}$ and a goal $G$, construct a (finite) SLD tree for $\mathcal{P} \cup \{G\}$ which contains, at least, a node different from the root;

2. for each non-failing leaf, collect the subgoals $G_i$ and the corresponding substitutions $\theta_i$ in order to form a set of clauses $\theta_i(G) \leftarrow G_i$, called *resultants*.

The reason why the goal $G$ is required to be atomic is that, otherwise, the computed resultants could not be definite clauses (due to the fact that they might have several atoms in their heads). The set of resultants can be considered as a specialized version of the initial program to solve the goal $G$ (and each instance of $G$). Under certain conditions, the specialized program preserves the semantics of the original program. In particular, the *closedness* and *independence* conditions [Lloyd and Shepherdson, 1991] guarantee the equivalence between both programs. Informally, the closedness condition guarantees that the calls which might occur during the execution of the resulting program are covered by some program rule. Therefore, the closedness condition ensures the completeness of the transformation. On the other hand, the independence condition guarantees that the specialized program does not produce additional answers. Intuitively, this condition is checked by testing that there are not overlappings between the specialized calls.

The seminal work of Martens and Gallagher [1995] introduces an automatic method for the partial deduction of logic programs which ensures the termination of the process. This approximation distinguishes two levels for controlling the termination of the process:

1. The first level corresponds to the construction of the SLD-resolution trees which

generate partial evaluations for single atoms. Control issues involved in this process —also known as the "local control" of partial evaluation— aim to keep the construction of the trees finite.

2. The second level, the so-called "global control" of partial evaluation, ensures that the closedness condition is eventually reached, but still maintaining an appropriate level of specialization. In order to reach the closedness condition, it is usually necessary to augment the set of atoms which appear in the initial goal. This way, the global control can be understood as the process which guarantees that the set of atoms to be specialized is kept finite. For this purpose, the use of an abstraction operator, which permits to "simplify" some elements introduced in the set, becomes usually necessary.

## 1.4 Partial Evaluation of Functional Logic Programs

While the aim of traditional partial evaluation is to specialize programs w.r.t. some known data, several partial evaluation techniques are able to go beyond this goal, achieving more powerful program optimizations. This is the case of a number of partial evaluation methods for functional programs (e.g., positive supercompilation [Sørensen *et al.*, 1996]), logic programs (e.g., partial deduction [Lloyd and Shepherdson, 1991]), and functional logic programs (e.g., narrowing-driven partial evaluation [Alpuente *et al.*, 1998b]). A common pattern of these techniques is that they are able to achieve optimizations regardless of whether some input data are provided or not (e.g., they can eliminate some intermediate data structures, similarly to Wadler's deforestation [Wadler, 1990]). In some sense, these techniques are *theorem provers* stronger than traditional partial evaluation approaches.

Partial evaluation has been intensively studied in the logic as well as in the functional paradigms. In essence, both approaches are similar; however, the general methods are often different due to the distinct underlying models and the different perspectives. In order to avoid this duplication of work, different approximations which generalize distinct partial evaluation techniques have been developed. For instance, Pettorossi and Proietti [1996a] present a general methodology which can be seen as a common pattern for reasoning about logic and functional program specialization. Moreover, they establish precise correspondences between the different techniques. Also, Jones [1994] presents an abstract framework to describe program specialization. The partial evaluation of logic programs [Lloyd, 1994] and the *driving* mechanism [Sørensen and Glück, 1995] are shown to be instances of the new framework.

Recently, [Alpuente *et al.*, 1998b] introduced a general scheme for the specialization of functional logic programs: Narrowing-driven Partial Evaluation (NPE). In contrast to the approximation frequently used for pure functional languages which is based on functional reduction, the narrowing mechanism which is used to execute functional logic programs is also used during specialization. The logic dimension of narrowing allows one to treat expressions which contain partial information naturally by means of logic variables and unification. On the other hand, the functional dimension permits to consider efficient evaluation strategies and also include a deterministic simplification phase. Thus, the integration of both components provides better opportunities for optimization.

The NPE method is formalized within the theoretical framework established by Lloyd and Shepherdson [1991] for the partial evaluation of logic programs. However, several notions have been generalized in order to deal with features such as nested function calls, the use of lazy or eager evaluation strategies, etc. Control issues are handled by using standard techniques such as those developed in [Martens and Gallagher, 1995; Sørensen and Glück, 1995]. Also, Alpuente *et al.* [1998b] distinguish between local and global control. Informally, the local control concerns the construction of (partial) narrowing trees for independent terms, while the global control is devoted to ensure the closedness of partially evaluated programs, guaranteeing also the termination of the process. Roughly speaking, the algorithm starts by partially evaluating the set of calls which appear in the initial goal and, then, recursively specializes the terms which are dynamically introduced during the process. The local control incorporates an appropriate *unfolding* strategy, whereas the global control provides an *abstraction* operator (generalization) which ensures the global termination of the process. Thus, the framework defines an automatic partial evaluation method which is independent of the narrowing strategy and which always terminates (for appropriate instances) and ensures the closedness condition for the residual programs. Using the terminology of Glück and Sørensen [1996], the global control strategy produces *polyvariant* and *polygenetic* specializations, i.e., NPE is able to produce different specializations for the same function definition and also combine distinct original function definitions in a single specialized function.

Julián [2000] presents a partial evaluation method for functional logic languages with a semantics based on (demand-driven) lazy narrowing [Moreno-Navarro and Rodríguez-Artalejo, 1992]. The basic algorithm is formalized as an instance of the partial evaluation method of Alpuente *et al.* [1998b] using lazy narrowing. For inductively sequential programs, Alpuente *et al.* [1999c] introduce an instance of the NPE method based on the needed narrowing strategy [Antoy *et al.*, 2000]. The new instance lifts to the partial evaluation level the idea of evaluating code only when it is necessary. An attractive property of this instance is that it preserves the *structure* of the original program, i.e., the residual program is also inductively sequential. This property does not hold, in general, for other instances of the NPE framework (see [Alpuente *et al.*, 1999c]).

## 1.5 Partial Evaluation in Practice

The partial evaluation techniques introduced so far can be applied over very simple functional logic languages (i.e., "pure" term rewriting systems). The extension of these techniques to deal with realistic functional logic programs (including features such as higher-order, constraints, input/output, etc.) requires an operational semantics covering all these facilities. However, as the operational semantics becomes more elaborated, the associated partial evaluation techniques become (more powerful but) also increasingly more complex. In order to achieve *practical* program transformation methods (i.e., methods which are applicable to realistic programming languages), it becomes useful to translate the source program to a *maximally* simplified programming language; in other words, one would need a representation in which the diverse features of the language such as constraints, partial applications, external function calls, etc., can be expressed by means of a uniform notation. Also, the associated semantics should be established in purely declarative terms. A general requirement for such representation is that programs written in a higher level language can be automatically translated into the simplified representation. This approach was first investigated by Bondorf [1989]. The basic idea is the use of an intermediate language which allows one to express pattern matching at an appropriate level of abstraction. An advantage of this approximation is that *self-application* is easier, i.e., the possibility of defining a partial evaluator able to specialize itself. Another example can be found in Gurr's thesis [Gurr, 1994], where the idea of using an intermediate representation is used as well. In particular, he describes a partial evaluator for meta-programs written in the logic programming language Gödel. Thanks to the use of a simplified intermediate representation, the extension of existing partial evaluation techniques to deal with the different features of Gödel becomes possible. Along the same lines, Nemytykh *et al.* [1996] propose a supercompiler which uses *flat Refal* as the language for object programs.

## 1.6 Efficiency of Partial Evaluation

The common purpose of partial evaluation techniques is to improve the efficiency of programs. Rather surprisingly, little attention has been paid to the development of formal methods to reason about the effectiveness achieved by these programs transformations —usually, only experimental tests for particular languages and compilers are undertaken. In the following, we briefly review some of the works which study the efficiency achieved by different program transformation techniques.

In the context of logic programming, Amtoft [1991] established several properties about some fold/unfold program transformations. In particular, he demonstrated that the partial evaluation process cannot achieve improvements which are not linear (this result can be also found in [Andersen and Gomard, 1992]). For a program with binding-time annotations, Andersen and Gomard [1992] developed an *efficiency analysis* that computes a relative efficiency interval such that the specialization will result in an improvement within the predicted interval. Nielson [1988] introduced a type system in order to express the bindings of variables and parameters, and defined several relations to formally express when a partial evaluator is better than another.

There are also interesting works centered on the study of *cost analyses* for logic and functional programs. These analyses can be useful to determine the effectiveness achieved by program transformations. For instance, Debray and Lin [1993] developed a method for the (semi-)automatic analysis of the cost, for a significant class of logic programs (treating indeterminism and the generation of multiple solutions via backtracking). Such cost analyses can be applied to the study of the effectiveness achieved by partial deduction, although the authors did not address this issue. In the same line, Sands [1995] introduced a theory of cost equivalence that can be used to reason about the computational cost of lazy functional programs. The problem is considered difficult, since the cost of lazy computations cannot be expressed in terms of a compositional function [Sands, 1995] (i.e., the cost of executing an expression like $f(g(x))$ cannot be obtained by composing the costs of executing $f$ and $g$ independently).

The aforementioned works mainly consider the number of evaluation steps as the basis to estimate the cost of executing a program. However, very simple experiments show that the number of evaluation steps and execution time are not easily correlated in general (see Chapter 9). In traditional *profiling* approaches (e.g., [Sansom and Peyton-Jones, 1997]), several cost criteria which are different from the number of evaluation steps are considered, although only experimentally. For the development of practical partial evaluation tools, it is essential to study formal criteria to estimate the efficiency of the process.

## 1.7 Objective of the Thesis

This thesis constitutes a natural extension of the works developed during the last five years within the ELP (Extensions of Logic Programming) group at the Technical University of Valencia. In Vidal's thesis [Vidal, 1996], the NPE approach to the specialization of functional logic languages was presented. Recently, [Julián, 2000] has introduced an instance of the NPE framework for lazy languages. As for the present thesis, the main objective is the development of a *practical* and *efficient* partial evaluation method for a modern multi-paradigm language, covering all the features which have not been considered in the previous framework (e.g., residuation, local declarations, external functions, constraints, higher-order functions, etc.). Such extension will revert on a significant increment of the expressivity of the language that the partial evaluation method is able to optimize. Also, it is essential to design a useful partial evaluation tool appropriate for software development and able to cooperate with other tools. For this purpose, we undertake our research in four main lines:

1. <u>Foundations of Partial Evaluation</u>

   Within the operational semantics of modern functional logic languages based on a combination of narrowing and residuation, the precise mechanism for each function is specified by evaluation annotations. In this thesis, we present a partial evaluation framework which is able to deal with programs containing evaluation annotations. The task is difficult due to several reasons. Firstly, a naïve adaptation of the NPE method, in which suspended computations are simply *residualized* (i.e., frozen) during partial evaluation, is not adequate since is not safe in general and, moreover, we would obtain a "poor" specialization in many cases. Therefore, we will introduce an extension of the standard computational model which allows us to ignore evaluation annotations at partial evaluation time while still guaranteeing the correctness of the process. Consequently, our method is less restrictive than other existing methods for logic programs (with constraints and) dynamic selection in which suspended expressions cannot be unfolded (e.g., [Etalle *et al.*, 1997]). Secondly, the inference of safe evaluation annotations for partially evaluated programs is far from trivial. In particular, in some cases it is necessary to split resultants (giving rise to several intermediate functions) in order to preserve the computed answers as well as the behaviour regarding suspensions. Moreover, we will also provide correctness results for the transformation, including the equivalence between the original and specialized programs w.r.t. the floundering behaviour.

2. <u>Control Issues in Partial Evaluation</u>

   In the original NPE method, control issues are handled by using standard techniques such as those developed in [Martens and Gallagher, 1995; Sørensen and Glück, 1995]. Originally, these algorithms do not incorporate a specific treatment for some *primitive* function symbols of the language (e.g., the sequential conjunction and disjunction, the strict equality, if-then-else constructs, etc.). This can lead to a loss of specialization potential and, in many cases, the use of (some form of) *tupling* is needed (as defined by Pettorossi and Proietti [1996a] for logic programs) in order to specialize conjunctive expressions or equations. The original NPE algorithm has been extended in [Julián, 2000; Albert *et al.*, 1998a,b] by formulating concrete control options able to deal with primitive function symbols in the context of lazy functional languages. As for local control, the reduction of elements within a conjunction can be performed in a "don't care non-deterministic way". However, the order in which the elements are chosen during the construction of the partial narrowing trees is crucial to achieve a good specialization. To deal with this, a dynamic selection strategy which keeps track of the calls evaluated in the same derivation is introduced. As for global control, the generic NPE algorithm needs particular techniques to split complex terms while keeping together, at the same time, the terms which share data structures. We generalize this method so that it can be parametric w.r.t. the narrowing strategy. This allows us to integrate the method within the partial evaluation framework discussed in the above point. By using a needed narrowing strategy, we experimentally evaluate the effect of incorporating our control strategies in the INDY system [Albert *et al.*, 1998c].

3. Partial Evaluation in Practice

   Going more deeply into the practical aspects, there are still a number features of modern multi-paradigm languages which are not considered by the previous framework. For instance, it is essential to design strategies to deal with local declarations, external functions, constraints, higher-order functions, etc., which are frequently present in current functional logic languages like Curry [Hanus, 2000a] or Toy [López-Fraguas and Sánchez-Hernández, 1999]. In this thesis, we introduce a novel approach to the partial evaluation of realistic multi-paradigm programs, i.e., programs with the features mentioned above. Inspired by the techniques developed within other paradigms [Bondorf, 1989; Gurr, 1994; Nemytykh *et al.*, 1996], we consider the translation of the source language into a maximally simplified intermediate language. The intermediate representation is essential to express the different features of the language at an appropriate level of abstraction. This allows us to design a simple and concise partial evaluation method which covers all the features of modern languages. Moreover, thanks to the simplicity of the intermediate language, it is actually possible to implement the partial evaluator in the same language that it partially evaluates (without any extra-logical feature). This amounts to say that self-application is theoretically possible (i.e., the ability to specialize the partial evaluator itself).

4. Effectiveness of Partial Evaluation

   The common goal of partial evaluation techniques is to improve the efficiency of programs. However, only experimental tests on particular languages and compilers are usually undertaken. Clearly, a machine-independent way of measuring the effectiveness of a partial evaluation method would be urgently necessary for both users and developers of partial evaluators. Hence, we will introduce a formal approach to reasoning about the effectiveness of partial evaluators of functional logic programs. In contrast to the traditional profiling approaches to this problem, our framework is based on properties of the rewrite system that models a functional logic program. Consequently, our assessments are independent of any specific language implementation or computing environment. Our framework complements more traditional profiling approaches to reasoning about the effectiveness of partial evaluators and, in some cases, predicts or explains counterintuitive experimental results. Firstly, we define several criteria for the cost of a computation: number of steps, number of function applications, and effort for pattern matching or unification. Then, we express the cost of each criterion by means of recurrence equations over algebraic data types, which can be automatically inferred from the partial evaluation process itself. A simple inspection of these equations already provides useful information. In some cases, the equations can be solved by transforming their arguments from arbitrary data types to natural numbers. In other cases, it is possible to estimate the improvement of a partial evaluator by analyzing the recurrence equations.

## 1.8 Structure of the Thesis

This thesis is organized in four parts:

- Part I: Foundations

  This part summarizes the foundations of partial evaluation as they will be used in the subsequent chapters. In Chapter 3, we recall the main notions concerning the partial evaluation of functional logic programs based on needed narrowing. Chapter 4 extends the method in order to deal with *residuating* functional logic programs, which generalize logic programs based on concurrent computational models with delays to functional logic programs. The main results included in this chapter are published in [Albert *et al.*, 1999a,b].

- Part II: Control Issues

  This part studies the definition of partial evaluation algorithms for functional logic languages containing *primitive* function symbols. First, we recall the generic partial evaluation algorithm for functional logic languages in Chapter 5. Then, Chapter 6 extends the generic approach by formulating concrete control options which effectively handle these primitive symbols. Some of the results which appear in this chapter are published in [Albert *et al.*, 1998a,b]. These control issues have been implemented in the Indy system, which we describe in Appendix A.

- Part III: Partial Evaluation in Practice

  In Chapter 7, we present a *practical* and *effective* approach for the partial evaluation of modern multi-paradigm languages. The new scheme is carefully designed for an abstract representation in which high-level programs can be automatically translated. A short version of this chapter appeared in [Albert *et al.*, 2000e,f]. Our novel approach allows us to define a partial evaluator for the language Curry, which we present in Chapter 8. The partial evaluator is written in Curry, the same language that it partially evaluates; hence, it is amenable to self-application. A brief description of the partial evaluator appeared in [Albert *et al.*, 2001b].

- Part IV: Effectiveness

  In the last part of the thesis, we develop a formal approach to measuring the effectiveness achieved by partial evaluation in the context of a functional logic language. In Chapter 9, we first define several criteria for the cost of a functional logic computation. Then, we provide an automatic technique for deriving recurrence equations which allow us to relate the cost of executing the original and the residual programs. An abstract of this chapter appeared in [Albert *et al.*, 2001a, 2000a,b].

The thesis concludes in Chapter 10, where we summarize the main contributions of the work and outline several possible directions for further research.

# Chapter 2

# Preliminaries

In this chapter, we summarize some basic notions to be used in the developments of the present thesis. First, we introduce some concepts and notations such as the signature, terms and substitutions and provide a precise terminology. Then, we briefly present some notions and results regarding term rewriting systems. Most of them are taken from [Klop, 1992; Baader and Nipkow, 1998; Dershowitz and Jouannaud, 1990]. Finally, we conclude the chapter with an introduction to functional logic programming and its operational semantics. The properties related to functional logic programming can be also found in [Antoy *et al.*, 2000; Alpuente *et al.*, 1999c; Antoy, 1992; Hanus, 1994].

## 2.1 Signature and Terms

We consider a (*many-sorted*) *signature* $\Sigma$ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for $n$-ary constructor and operation symbols, respectively. There is at least one sort *Bool* containing the 0-ary Boolean constructors $true$ and $false$. The set of *terms* and *constructor terms* (possibly with *variables* from $\mathcal{X}$; e.g., $x, y, \ldots$) are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. The set of variables occurring in a term $t$ is denoted by $\mathcal{V}ar(t)$. A term $t$ is *ground* if $\mathcal{V}ar(t) = \varnothing$. A term is *linear* if it does not contain multiple occurrences of any variable. We write $\overline{o_n}$ for the *list of objects* $o_1, \ldots, o_n$.

A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A term is *operation-rooted* if it has an operation symbol at the root. $root(t)$ denotes the symbol at the root of the term $t$.

A *position* $p$ in a term $t$ is represented by a sequence of natural numbers ($\Lambda$ denotes the empty sequence, i.e., the root position). Positions are ordered by the *prefix* ordering: $u \leq v$, if there exists $w$ such that $u.w = v$. Positions $u, v$ are *disjoint*, denoted $u \perp v$, if neither $u \leq v$ nor $v \leq u$. Given a term $t$, we let $\mathcal{P}os(t)$ denote the set of positions of $t$, which can be defined inductively as follows:

$$\mathcal{P}os(t) = \begin{cases} \{\Lambda\} & \text{if } t \in \mathcal{X} \\ \{\Lambda\} \cup \{i.u \mid 1 \leq i \leq n \ \wedge \ u \in \mathcal{P}os(t_i)\} & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Similarly, $\mathcal{FP}os(t)$ denotes the set of non-variable positions of $t$.

We denote by $t|_p$ the *subterm* of $t$ at position $p$:

$$t|_p = \begin{cases} t & \text{if } p = \Lambda \\ t_i|_u & \text{if } p = i.u \text{ and } t = f(t_1, \ldots, t_k), \text{ with } 1 \leq i \leq k \end{cases}$$

$t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term $s$ (see [Dershowitz and Jouannaud, 1990] for details). For a set of (pairwise distinct, ordered) positions $P = \{p_1, \ldots, p_n\}$, we let $t[s_1, \ldots, s_n]_P = (((t[s_1]_{p_1})[s_2]_{p_2}) \ldots [s_n]_{p_n})$.

## 2.2 Substitutions

We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* $\sigma$ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables $x$. The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is called the *domain* of $\sigma$. The codomain of a substitution $\sigma$ is the set $\mathcal{R}an(\sigma) = \{\sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. A substitution $\sigma$ is *(ground) constructor*, if $\sigma(x)$ is (ground) constructor for all $x \in \mathcal{D}om(\sigma)$. The identity substitution is denoted by $id$. Substitutions are extended to morphisms on terms by $\sigma(f(\overline{t_n})) = f(\overline{\sigma(t_n)})$ for every term $f(\overline{t_n})$. Given a substitution $\theta$ and a set of variables $V \subseteq \mathcal{X}$, we denote by $\theta_{\upharpoonright V}$ the substitution obtained from $\theta$ by restricting its domain to $V$. We write $\theta = \sigma\ [V]$ if $\theta_{\upharpoonright V} = \sigma_{\upharpoonright V}$, and $\theta \leq \sigma\ [V]$ denotes the existence of a substitution $\gamma$ such that $\gamma \circ \theta = \sigma\ [V]$.

A term $t'$ is an *instance* of $t$ if there is a substitution $\sigma$ with $t' = \sigma(t)$. This implies a *subsumption ordering* on terms which is defined by $t \leq t'$ iff $t'$ is an instance of $t$. A *unifier* of two terms $s$ and $t$ is a substitution $\sigma$ with $\sigma(s) = \sigma(t)$. A unifier $\sigma$ is the *most general* unifier ($mgu$) if $\sigma \leq \sigma'\ [\mathcal{X}]$ for each other unifier $\sigma'$. A $mgu$ is unique up to variable renaming.

## 2.3 Term Rewriting Systems

A set of rewrite rules $l \to r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms $l$ and $r$ are called the *left-hand side* and the *right-hand side* of the rule, respectively. A TRS $\mathcal{R}$ is *left-linear* if $l$ is linear for all $l \to r \in \mathcal{R}$. A TRS is constructor-based (CB) if each left-hand side $l$ is a pattern. Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap*, if there is a non-variable position $p \in \mathcal{FP}os(l)$ and a most general unifier $\sigma$ such that $\sigma(l|_p) = \sigma(l')$. The pair $\langle \sigma(l)[\sigma(r')]_p, \sigma(r) \rangle$ is called a *critical pair* (and is also called an *overlay* if $p = \Lambda$). A critical pair $\langle t, s \rangle$ is trivial if $t = s$ (up to variable renaming). A left-linear TRS without critical pairs is called *orthogonal*. It is called *almost orthogonal* if its critical pairs are trivial overlays. If it only has trivial critical pairs it is called *weakly orthogonal*. Note that, in CB-TRSs, almost orthogonality and weak orthogonality coincide.

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position $p$ in $t$, a rewrite rule $R = l \to r$ and a substitution $\sigma$ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$ ($p$ and $R$ will often be omitted in the notation of a computation step). The instantiated left-hand side $\sigma(l)$ is called a *redex* (reducible expression).

A term $t$ is root-stable (often called a *head-normal form*) if it cannot be rewritten to a redex. A *constructor root-stable* term is either a variable or a *constructor-rooted* term, that is, a term rooted by a constructor symbol. By abuse, when it is clear from the context, we use the notion "head-normal form" to refer to constructor root-stable terms. A term $t$ is called *irreducible* or in *normal form* if there is no term $s$ with $t \to s$. Given a binary relation $\to$, $\to^+$ denotes the transitive closure of $\to$ and $\to^*$ denotes the reflexive and transitive closure of $\to$.

## 2.4 Functional Logic Programming

Term rewriting systems provide an adequate computational model for functional languages which allow the definition of functions by means of patterns (e.g., Haskell, Hope or Miranda) [Baader and Nipkow, 1998; Klop, 1992; Plasmeijer and van Eekelen, 1993].

In the sequel, a *functional logic program* is a finite left-linear constructor-based TRS. Conditions in program rules can be handled by using the predefined functions `and`, `if_then_else`, `case_of` which are reduced by standard defining rules [Moreno-Navarro and Rodríguez-Artalejo, 1992]. Within this class of TRSs, the class of *inductively sequential* rewrite systems has been defined, studied, and used for the implementation of programming languages which provide for optimal computations both in functional and functional logic programming [Antoy, 1992; Antoy *et al.*, 2000; Hanus, 1997b; Hanus *et al.*, 1998; Loogen *et al.*, 1993]. Inductively sequential rewrite systems can be thought of as constructor-based TRSs with discriminating left-hand sides, i.e., typical functional programs. Thus, in the remainder, we follow the standard term rewriting framework of Dershowitz and Jouannaud [1990] for developing our results.

### 2.4.1 Narrowing

The operational semantics of functional logic languages is usually based on narrowing, a generalization of term rewriting which combines reduction and variable instantiation. To evaluate terms containing variables, narrowing non-deterministically instantiates the variables such that a rewrite step is possible (usually by computing most general unifiers between a subterm and the left-hand side of some rule [Hanus, 1994], although this requirement is relaxed in needed narrowing steps to obtain an optimal evaluation strategy [Antoy *et al.*, 2000]). Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if $p$ is a non-variable position in $t$ and $\sigma(t) \to_{p,R} t'$.[1] We denote by $t_0 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ ($\sigma = id$ if $n = 0$). Since we are interested in computing *values* (constructor terms) as well as *answers* (substitutions) in functional logic programming, we say that the narrowing derivation $t \leadsto^*_\sigma c$ *computes the result* $c$ *with answer* $\sigma$ if $c$ is a constructor term.

Narrowing derivations can be represented by a (possibly infinite) finitely branching tree. Following [Lloyd and Shepherdson, 1991], we adopt the convention that any derivation is potentially incomplete (a branch thus can be failed, incomplete, successful, or infinite). A *failing leaf* contains an expression which is not a constructor term and which cannot be further narrowed.

### 2.4.2 Needed Narrowing

A challenge in the design of functional logic languages is the definition of an appropriate strategy to solve equations. Such strategy should be at least *sound* (i.e., only correct solutions are computed) and *complete* (i.e., all solutions or more general representations of the solutions are computed). The narrowing relation can be used to define such a strategy. However, since simple narrowing can have a huge search space, great effort has been made in order to develop a narrowing strategy which limits this space (e.g., basic [Hullot, 1980], innermost [Fribourg, 1985], outermost [Echahed, 1990], standard [Darlington and Guo, 1989; Ida and Nakahara, 1997; Middeldorp and Okui, 1998; You, 1991], lazy [Giovannetti *et al.*, 1991; Hans *et al.*, 1992; Moreno-Navarro *et al.*, 1990; Moreno-Navarro and Rodríguez-Artalejo, 1992; Reddy, 1985] or narrowing with redundancy tests [Bockmayr *et al.*, 1993]). Each strategy requires certain conditions on the rewrite system in order to ensure completeness.

*Needed narrowing* [Antoy *et al.*, 2000] is a strategy that, for *inductively sequential* programs, preserves the completeness of narrowing. Moreover, it is currently the best known narrowing strategy due to its optimality properties w.r.t. the length of successful derivations and the number of computed solutions. The optimality is achieved

[1] In this definition, the substitution $\sigma$ is usually restricted to the variables in $t$. This is correct because the definition of narrowing requires the rewrite step to be possible.

by performing only "unavoidable" steps for solving equations. The notion of "unavoidable" is well-known from rewriting. In *orthogonal* systems, there exists a redex called *needed* that must "eventually" be reduced to compute the normal form of a term [Huet and Lévy, 1992; Klop and Middeldorp, 1991; O'Donnell, 1977]. Furthermore, by rewriting the needed redexes of a term, its normal form can be computed (if it exists). Thus, only needed redexes really matter for rewriting in orthogonal systems. This fact has been extended to narrowing in inductively sequential systems —a subclass of orthogonal systems— in [Antoy *et al.*, 2000], achieving a strategy which performs derivations of minimal length.

Furthermore, there is a second optimality result concerned with the substitution computed by a narrowing derivation: different needed narrowing derivations compute *disjoint* sets of substitutions. This property complements the neededness of a step in the sense that the derivations computed by the strategy are needed in their entirety as well. Roughly speaking, any solution computed by a derivation is not computed by any other derivation; hence, every derivation leading to a solution is needed as well as any step of the derivation.

The needed narrowing strategy performs steps which are *needed* for computing solutions:

**Definition 1** *[Antoy et al., 2000]*

1. *A narrowing step $t \rightsquigarrow_{p,R,\sigma} t'$ is called needed (or outermost-needed) iff, for every $\eta \geq \sigma$, $p$ is the position of a needed (or outermost-needed) redex of $\eta(t)$, respectively.*

2. *A narrowing derivation is called needed (or outermost-needed) iff every step of the derivation is needed (or outermost-needed, respectively).*

In comparison to needed reduction, the definition of needed narrowing adds a new level of difficulty for computing the needed steps. In addition to consider the normal forms, one must take into account any possible instantiation of the input term. In inductively sequential systems, the problem has an efficient solution which is achieved by giving up the requirement that the unifier of a narrowing step be most general. The instantiation must ensure the need of the position regardless of future unifiers which is not possible when using the most general unifier. It is simply defined so that this extra instantiation would eventually be performed later in the derivation. Thus, one is just "anticipating" it and the completeness of narrowing is preserved. This approach, though, complicates the notion of narrowing strategy.

Traditionally, a narrowing strategy is a function from terms to non-variable positions in these terms so that exactly one position is selected for the next narrowing step [Echahed, 1990; Padawitz, 1988]. However, this notion of narrowing strategy is inadequate for narrowing with arbitrary unifiers which are essential to capture the need of a narrowing step.

**Definition 2** *[Antoy et al., 2000] A narrowing strategy is a function from terms into sets of triples. If $\mathcal{S}$ is a narrowing strategy, $t$ is a term, and $(p, l \to r, \sigma) \in \mathcal{S}(t)$, then $p$ is a position of $t$, $l \to r$ is a rewrite rule, and $\sigma$ a substitution such that*

$$t \leadsto_{p,l \to r,\sigma} \sigma(t[r]_p)$$

*is a narrowing step.*

Now, we look at the class of inductively sequential systems for which there exists an efficiently computable needed narrowing strategy. These systems have the property that the rules defining any operation can be organized in a hierarchical structure called definitional tree [Antoy, 1992], which is used to implement needed narrowing.

#### Definitional Trees.

A definitional tree can be seen as a partially ordered set of patterns with some additional constraints. The symbols *branch* and *leaf*, used in the next definition, are uninterpreted functions used to classify the nodes of the tree.

**Definition 3** *[Antoy, 1992] $\mathcal{T}$ is a partial definitional tree, or pdt, with pattern $\pi$ iff one of the following cases hold:*

$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$ *where $\pi$ is a pattern, $o$ is the occurrence of a variable of $\pi$, the sort of $\pi|_o$ has constructors $c_1, \ldots, c_k$ for some $k > 0$, and for all $i$ in $\{1, \ldots, k\}$, $\mathcal{T}_i$ is a pdt with pattern $\pi[c_i(x_1, \ldots, x_n)]_o$, where $n$ is the arity of $c_i$ and $x_1, \ldots, x_n$ are new distinct variables.*

$\mathcal{T} = leaf(\pi)$ *where $\pi$ is a pattern.*

We denote by $\mathcal{P}(\Sigma)$ the set of *pdt*s over the signature $\Sigma$. Let $\mathcal{R}$ be a rewrite system. $\mathcal{T}$ is a *definitional tree* of an operation $f$ iff $\mathcal{T}$ is a *pdt* whose pattern argument is $f(x_1, \ldots, x_n)$, where $n$ is the arity of $f$ and $x_1, \ldots, x_n$ are new distinct variables, and for every rule $l \to r$ of $\mathcal{R}$ with $l = f(x_1, \ldots, x_n)$, there exists a leaf $leaf(\pi)$ of $\mathcal{T}$ such that $l$ is a *variant* (i.e., equal modulo variable renaming) of $\pi$, and we say that the node $leaf(\pi)$ represents the rule $l \to r$. A definitional tree $\mathcal{T}$ of an operation $f$ is called *minimal* iff below any *branch* node of $\mathcal{T}$ there is a *leaf* representing a rule defining $f$.

An operation $f$ of a rewrite system $\mathcal{R}$ is called *inductively sequential* iff there exists a definitional tree $\mathcal{T}$ of $f$ such that each *leaf* node of $\mathcal{T}$ represents at most one rule of $\mathcal{R}$ and all rules are represented. A rewrite system $\mathcal{R}$ is called *inductively sequential* if all its defined functions are inductively sequential.

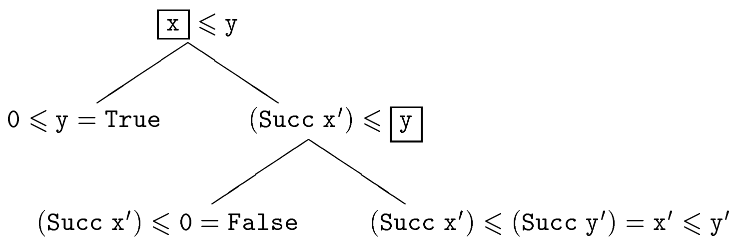


Figure 2.1: Definitional tree for the function "$\leqslant$"

It is often convenient and simplifies understanding to provide a graphic representation of definitional trees, where each inner node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding rules.

Let us note that, even if we consider a first-order language, we use a curried notation in the examples (as is usual in functional languages). Likewise, we use lower case letter for variables and upper case letter for constructor symbols, following the Curry syntax. Analogously, in our examples, rules have the form $l = r$ instead of $l \to r$, as in our formal notation.

**Example 1** Consider the rules defining the "$\leqslant$" function:

$$\begin{array}{lcl} \texttt{0} \leqslant \texttt{y} & = & \texttt{True} \\ (\texttt{Succ x}') \leqslant \texttt{0} & = & \texttt{False} \\ (\texttt{Succ x}') \leqslant (\texttt{Succ y}') & = & \texttt{x}' \leqslant \texttt{y}' \end{array}$$

The definitional tree for the function "$\leqslant$" is illustrated in Figure 2.1. In the picture, each branch node contains the pattern of the corresponding node of the definitional tree. Every leaf node represents a rule. The inductive argument of a branch node is pointed by surrounding the corresponding subterm within a box.

For the definition of the needed narrowing strategy, it is useful to distinguish whether or not a $leaf$ node of a definitional tree of some operation $f$ represents a rule defining $f$. We use $exempt(\pi)$ to point out that $leaf(\pi)$ is a *pdt* which does not represent any rule. Similarly, $rule(\pi, \sigma(l) \to \sigma(r))$ is used to abbreviate the fact that $leaf(\pi)$ is a *pdt* representing some rule $l \to r$ of the considered rewrite system, where $\sigma$ is the renaming substitution such that $\sigma(l) = \pi$. The patterns of the definitional tree form a finite set which is partially ordered by the subsumption preordering. The set of patterns occurring within the leaves of a definitional tree is complete w.r.t. the set of constructors in the sense of [Huet and Hullot, 1982]. Consequently, the defined functions of an inductively sequential term rewriting system are completely defined over their application domains [Guttag and Horning, 1978; Thiel, 1984] (i.e., the normal form of any ground term is a constructor term) if the considered rewriting system is terminating and the possible definitional trees do not contain *exempt* nodes.

The following proposition shows that functions defined by a single rule are always inductively sequential. This will become useful in forthcoming chapters.

**Proposition 4** *[Alpuente et al., 1999c] If $f(\overline{t_n})$ is a linear pattern, then there exists a definitional tree for the function $f$ with pattern $f(\overline{x_n})$.*

**The Outermost-Needed Strategy.**

We now provide the informal description of the needed narrowing strategy presented in [Antoy *et al.*, 2000]. Let $t = f(t_1, \ldots, t_n)$ be a term to be narrowed. We unify $t$ with some maximal element of the set of patterns of a definitional tree of $f$. Let $\pi$ denote such a pattern, $\tau$ be the most general unifier of $t$ and $\pi$, and $\mathcal{T}$ be the *pdt* in which $\pi$ occurs.

- If $\mathcal{T}$ is a *rule pdt*, then we narrow $\tau(t)$ at the root position with the rule represented by $\mathcal{T}$.
- If $\mathcal{T}$ is an *exempt pdt*, then $\tau(t)$ cannot be narrowed to a constructor-rooted term.
- If $\mathcal{T}$ is a *branch pdt*, then we recur on $\tau(t|_o)$, where $o$ is the occurrence contained in $\mathcal{T}$ and $\tau$ is the substitution which ensures that the step will be actually needed (i.e, the *anticipated* substitution [Antoy *et al.*, 2000]). The result of the recursive invocation is properly composed with $\tau$ and $o$.

The outermost-needed strategy is a mapping, $\lambda$, that implements the above computation. $\lambda$ takes an operation-rooted term, $t$, and a definitional tree, $\mathcal{T}$, of the root of $t$, and non-deterministically returns a triple, $(p, R, \sigma)$, where $p$ is a position of $t$, $R$ is either a rule $l \to r$ of $\mathcal{R}$ or the distinguished symbol "?", and $\sigma$ is a substitution.

- If $R = l \to r$, then the strategy performs the narrowing step

$$t \leadsto_{p, l \to r, \sigma} \sigma(t[r]_p)$$

- If $R = ?$, then the strategy gives up, since it is impossible to narrow $t$ to a constructor-rooted term.

In the following, *pattern*$(\mathcal{T})$ denotes the pattern argument of $\mathcal{T}$, and $\prec$ denotes the Noetherian ordering on $\mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{P}(\Sigma)$ defined by: $(t_1, \mathcal{T}_1) \prec (t_2, \mathcal{T}_2)$ iff:

- Either $t_1$ has fewer occurrences of defined operation symbols than $t_2$;
- or $t_1 = t_2$ and $\mathcal{T}_1$ is a proper subtree of $\mathcal{T}_2$.

**Definition 5** *[Antoy et al., 2000] The function $\lambda$ takes two arguments: an operation-rooted term, $t$, and a pdt, $\mathcal{T}$, such that $pattern(\mathcal{T})$ and $t$ unify. The function $\lambda$ yields a set of triples of the form $(p, R, \sigma)$, where $p$ is a position of $t$, $R$ is either a rule $l \to r$ of $\mathcal{R}$ or the distinguished symbol "?", and $\sigma$ is a unifier of $pattern(\mathcal{T})$ and $t$. Thus, let $t$ be a term and $\mathcal{T}$ a pdt in the domain of $\lambda$. The function $\lambda$ is defined by induction on $\prec$ as follows.*

$$\lambda(t,\mathcal{T}) \ni \begin{cases} (\Lambda, R, mgu(t,\pi)) & \textit{if } \mathcal{T} = rule(\pi, R); \\ (\Lambda, ?, mgu(t,\pi)) & \textit{if } \mathcal{T} = exempt(\pi); \\ (p, R, \sigma) & \textit{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad t \textit{ and } pattern(\mathcal{T}_i) \textit{ unify, for some } i, \textit{ and} \\ & \quad (p, R, \sigma) \in \lambda(t, \mathcal{T}_i); \\ (o.p, R, \sigma \circ \tau) & \textit{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k) \\ & \quad t \textit{ and } pattern(\mathcal{T}_i) \textit{ do not unify, for any } i, \textit{ and} \\ & \quad \tau = mgu(t,\pi) \\ & \quad \mathcal{T}' \textit{ is a definitional tree of the root of } \tau(t|_o), \textit{ and} \\ & \quad (p, R, \sigma) \in \lambda(\tau(t|_o), \mathcal{T}_i); \end{cases}$$

The function $\lambda$ is trivially well-defined in the third case. By the definition of *pdt*, there exists a proper sub*pdt* $\mathcal{T}_i$ of $\mathcal{T}$ such that $pattern(\mathcal{T}_i)$ and $t$ unify if $t|_o$ is constructor-rooted or a variable. Similarly, $\lambda$ is well-defined in the fourth case since this case can only occur if $t|_o$ is operation-rooted. In this case, $\tau|_{\mathcal{V}ar(t)}$ is a constructor substitution since $\pi$ is a linear pattern. Since $t$ is operation-rooted and $o \neq \Lambda$, $\tau(t|_o)$ has fewer occurrences of defined operation symbols than $t$. Since $t|_o$ is operation-rooted, so is $\tau(t|_o)$. By the definition of *pdt*, $pattern(\mathcal{T}') \leq \tau(t|_o)$, i.e., $pattern(\mathcal{T}')$ and $\tau(t|_o)$ unify. This implies that $\lambda$ is well-defined in this case as well.

As in proof procedures for logic programming, we assume that the definitional trees always contain new variables if they are used in a narrowing step. This implies that all computed substitutions are idempotent (we will implicitly assume this property in the following).

The computation of $\lambda(t, \mathcal{T})$ may entail a non-deterministic choice when $\mathcal{T}$ is a *branch pdt* —the integer $i$ when $\tau(t|_o)$ is a variable. The substitution $\tau$, when $t|_o$ is operation-rooted, is the *anticipated* substitution guaranteeing the need of the computed position. It is pushed down in the recursive call to $\lambda$ to ensure the consistency of the computation when $t$ is non-linear. The anticipated substitution is neglected when $t|_o$ is not operation-rooted since the pattern in $\mathcal{T}_i$ is an instance of $\pi$. Hence, $\sigma$ extends the anticipated substitution.

The evaluation to ground constructor terms (and not to arbitrary expressions) is the intended semantics of functional languages and also the semantics of most functional logic languages. In particular, the equality predicate $\approx$ is defined as the *strict* *equality* on terms (note that terminating rewrite systems are not required and, thus, reflexivity is not desired), i.e., the equation $t_1 \approx t_2$ is satisfied if $t_1$ and $t_2$ are reducible to the same ground constructor term, as in functional languages. Furthermore, a substitution $\sigma$ is a *solution* for an equation $t_1 \approx t_2$ if $\sigma(t_1) \approx \sigma(t_2)$ is satisfied. Thus, equations can be interpreted as terms by defining the symbol $\approx$ as a binary operation symbol, more precisely, one operation symbol for each sort, and, all notions for terms (such as substitution, rewriting, narrowing, etc.) will be used for equations. The strict equality can be defined as a binary Boolean function by the following set of orthogonal rewrite rules (see, e.g., [Antoy *et al.*, 2000]):

$$\begin{array}{rcl}
\mathtt{C} \approx \mathtt{C} & = & \mathtt{True} \\
\mathtt{C}\ \mathtt{x_1}\ \ldots\ \mathtt{x_n} \approx \mathtt{C}\ \mathtt{y_1}\ \ldots\ \mathtt{y_n} & = & (\mathtt{x_1} \approx \mathtt{y_1}) \& \ldots \& (\mathtt{x_n} \approx \mathtt{y_n}) \\
\mathtt{True}\ \&\ \mathtt{True} & = & \mathtt{True}
\end{array}$$

where & is assumed to be a right-associative infix operator and $c$ is a constructor of arity 0 in the first rule and arity $n > 0$ in the second rule. Thus, we do not treat the strict equality in any special way, and it is sufficient to consider it as a Boolean function which must be reduced to the constant `True`. We say that $\sigma$ is a *computed answer substitution* for an equation $e$ if there is a narrowing derivation $e \leadsto_{\sigma}^{*} true$. The equivalence between the reducibility to a same ground constructor term and the reducibility to `True` using the strict equality rules is addressed in [Antoy *et al.*, 2000]. The orthogonality of a rewrite system is not changed by these rules. The same holds for the inductive sequentiality [Antoy *et al.*, 2000]. More details about strict equality can be found in [Antoy *et al.*, 2000; Giovannetti *et al.*, 1991; Moreno-Navarro and Rodríguez-Artalejo, 1992].

In Definition 5, it is not considered the evaluation of constructor-rooted terms. This limitation suffices for solving equations. Nevertheless, the results could be easily extended to constructor-rooted terms: *to compute an outermost-needed narrowing step for a constructor-rooted term, it suffices to compute an outermost-needed narrowing step for any of its maximal operation-rooted subterms [Antoy et al., 1997].*

**Properties of Outermost-Needed Narrowing.**

In [Antoy *et al.*, 2000], outermost-needed narrowing is proved to be a sound and complete procedure to solve equations when the equality rules are added to narrow equations to *true*. Moreover, higher-order narrowing with definitional trees has been also proven sound and complete for arbitrary terms rather than equations in [Hanus and Prehofer, 1999]. The following theorem states the soundness of outermost-needed narrowing for inductively sequential rewrite systems:

**Theorem 6** [Antoy *et al.*, 2000] *Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules. If $t \approx t' \leadsto_{\sigma}^{*} true$ is an outermost-needed narrowing* *derivation, then* $\sigma$ *is a solution for* $t \approx t'$.

Outermost-needed narrowing instantiates variables to constructor terms. Thus, in [Antoy *et al.*, 2000], it is shown that outermost-needed narrowing is complete for constructor substitutions as solutions of equations within the class of inductively sequential programs. Moreover, needed narrowing is complete for the entire class of orthogonal rewrite systems w.r.t. irreducible substitutions [Antoy, 2000], although neither needed rewriting nor needed narrowing steps are computable for this class of rewrite systems. The following theorem states the completeness of outermost-needed narrowing for inductively sequential rewrite systems:

**Theorem 7** [Antoy *et al.*, 2000] *Let* $\mathcal{R}$ *be an inductively sequential rewrite system extended by the equality rules. Let* $\sigma$ *be a constructor substitution that is a solution of an equation* $t \approx t'$ *and* $V$ *be a finite set of variables containing* $\mathcal{V}ar(t) \cup \mathcal{V}ar(t')$. *Then there exists a derivation* $t \approx t' \leadsto^*_{\sigma'} true$ *computed by* $\lambda$ *such that* $\sigma' \leq \sigma\ [V]$.

The following theorem states the *strong completeness* of needed narrowing, i.e., given a conjunctive expression of the form $t_1$ & $t_2$ one can arbitrarily evaluate first $t_1$ or $t_2$ and the resulting step is still a needed narrowing step. Theorem 8 is not identically stated in [Antoy *et al.*, 2000], however, it can be easily derived from the results presented there. Nevertheless, we present an sketch of the facts needed for the formal proof.

**Theorem 8** *Let* $\mathcal{R}$ *be an inductively sequential rewrite system extended by the rule* $R = (\texttt{True}\ \&\ \texttt{True} = \texttt{True})$. *Then, the following derivation* $t_1$ & $t_2 \leadsto^*_\sigma true$ *is a needed narrowing derivation using an arbitrary definitional tree for* $R$.

*Proof.* (Sketch)
In [Antoy *et al.*, 2000], the definition of needed narrowing allows one to consider an arbitrary definitional tree at each narrowing step. Moreover, the above definition of the conjunction is inductively sequential and has two definitional trees

$$
\begin{array}{l}
branch(\texttt{x}\,\&\,\texttt{y}, 1, rigid, \\
\qquad\qquad branch(\texttt{True}\,\&\,\texttt{y}, 2, rigid, \\
\qquad\qquad\qquad\qquad rule(\texttt{True}\,\&\,\texttt{True} = \texttt{True})))
\end{array}
$$

and

$$
\begin{array}{l}
branch(\texttt{x}\,\&\,\texttt{y}, 2, rigid, \\
\qquad\qquad branch(\texttt{x}\,\&\,\texttt{True}, 1, rigid, \\
\qquad\qquad\qquad\qquad rule(\texttt{True}\,\&\,\texttt{True} = \texttt{True})))
\end{array}
$$

Therefore, each conjunct $t_i$, $i \in \{1, 2\}$ can be solved by using the corresponding definitional tree and this is trivially correct. □

The following theorem strengthens the characterization of the needed narrowing strategy by showing that no redundant solutions are computed by $\lambda$. To identify such solutions, the notion of *disjointness* is used. We say that two substitutions are disjoint when they do not "unify", more precisely, given two substitutions $\sigma$ and $\sigma'$, they are *disjoint* (on a set of variables $V$) iff there exists some $x \in V$ such that $\sigma(x)$ and $\sigma'(x)$ are not unifiable. This notion does not correspond to the incomparability of substitutions used in unification theory [Plotkin, 1972] to characterize minimal sets of solutions. In general, incomparable substitutions might have a common instance and, thus, they do not describe disjoint regions of the solution space. For instance, substitutions $\{x \mapsto 0\}$ and $\{y \mapsto 0\}$ are incomparable on $x, y$ but substitution $\{x \mapsto 0, y \mapsto 0\}$ is described by both substitutions. In contrast, disjoint substitutions describe independent regions of the solution space. Therefore, disjointness is a stronger notion than incomparability.

**Theorem 9** [Antoy *et al.*, 2000] *Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules, $e$ an equation to solve and $V = \mathcal{V}ar(e)$. Let $e \leadsto^{+}_{\sigma}$ true and $e \leadsto^{+}_{\sigma'}$ true be two distinct derivations computed by $\lambda$. Then, $\sigma$ and $\sigma'$ are disjoint on $V$.*

In [Antoy *et al.*, 2000], it is also discussed that, when *sharing* between "blood related" subterms is considered, the cost and length of a derivation computed by the needed narrowing strategy is minimal and, thus, the strategy is optimal. For this purpose, it is necessary the extension to the case of narrowing of the notions of: narrowing *multistep*, *family* of redexes, and *complete* step. We do not formalize this property here, but refer to [Antoy *et al.*, 2000] for further details.

Finally, an important advantage of functional logic languages in comparison to pure logic languages is their improved operational behavior in avoiding non deterministic computation steps. This can be achieved by using a demand-driven computation strategy which can avoid the evaluation of potential non-deterministic expressions.

**Example 2** For instance, consider the rules in Example 1 and the following rules defining the addition on natural numbers:

$$\begin{array}{rcl} \mathtt{0} + \mathtt{n} & \rightarrow & \mathtt{n} \\ (\mathtt{S\ m}) + \mathtt{n} & \rightarrow & \mathtt{S\ (m + n)} \end{array}$$

Given the term $\mathtt{0} \leqslant \mathtt{x} + \mathtt{x}$, needed narrowing evaluates it by one deterministic step to `True`. In an equivalent logic program, this nested term must be flattened into a conjunction of two predicate calls, like $+(\mathtt{x}, \mathtt{x}, \mathtt{z}) \wedge \leqslant(\mathtt{0}, \mathtt{z}, \mathtt{b})$, which causes a non-deterministic computation due to the predicate call $+(\mathtt{x}, \mathtt{x}, \mathtt{z})$.[2]

[2]Such non-deterministic computations could be avoided using Prolog systems with coroutining, but then we are faced with the problem of floundering and incompleteness.

Another reason for the improved operational behavior of functional logic languages is the ability of particular evaluation strategies (like needed narrowing or parallel narrowing [Antoy *et al.*, 1997]) to evaluate ground terms in a completely deterministic way, which is important to ensure an efficient implementation of purely functional evaluations. This property, which is obvious by the definition of needed narrowing, is formally stated in the following proposition. For this purpose, a term $t$ is called *deterministically evaluable* (w.r.t. needed narrowing) if each step in a narrowing derivation issuing from $t$ is deterministic. A term $t$ *deterministically normalizes* to a constructor term $c$ (w.r.t. needed narrowing) if $t$ is deterministically evaluable and there is a needed narrowing derivation $t \leadsto_{id}^{*} c$ (i.e., $c$ is the normal form of $t$).

**Proposition 10** *[Alpuente et al., 1999c] Let $\mathcal{R}$ be an inductively sequential program and $t$ be a term.*

1. *If $t \leadsto_{id}^{*} c$ is a needed narrowing derivation, then $t$ deterministically normalizes to $c$.*
2. *If $t$ is ground, then $t$ is deterministically evaluable.*

# Part I

# Foundations

# Chapter 3

# Partial Evaluation based on Needed Narrowing

In this chapter, we first recall the main notions associated to the partial evaluation of functional logic programs based on needed narrowing. Then, the most important properties of the programs specialized using needed narrowing are summarized. All the results and missing proofs can be found in [Alpuente *et al.*, 1999c].

## 3.1 Introduction

In contrast to the approach usually taken in pure functional languages, the algorithm for the specialization of functional logic languages uses the unification-based computation mechanism of narrowing for the specialization of the program as well as for its execution. The logic dimension of narrowing permits to treat expressions containing partial information in a natural way, by means of logical variables and unification, while the functional dimension allows one to consider efficient evaluation strategies as well as to include a deterministic simplification phase which provides better opportunities for optimization and improves both, the overall specialization and the efficiency of the method. Narrowing-driven partial evaluation (NPE) is formalized within the theoretical framework established in [Lloyd and Shepherdson, 1991] for the partial deduction of logic programs, although a number of concepts have been generalized for dealing with functional features, such as user-defined functions, nested function calls, eager and lazy evaluation strategies, or deterministic reduction steps. The generic method is parametric with respect to the narrowing strategy which is used for the automatic construction of (finite) narrowing trees [Alpuente *et al.*, 1998b]. Here we present an instance of the framework based on needed narrowing which preserves the semantics of inductively sequential programs.

## 3.2 Narrowing-driven Partial Evaluation

The construction follows mainly the partial deduction framework of [Lloyd and Shepherdson, 1991] for logic programs. In order to deal with (interpreted) nested function calls, a number of concepts and results have been generalized. Since functional logic programs subsume functional and logic programs, some notions get more elaborated in comparison to pure functional or logic languages. What it is gain in return is a more general framework, which subsumes some of the basic developments in both fields. The conditions for the correctness of the transformation cover many practical cases. For pure languages, these conditions essentially boil down to conditions for the correctness of the partial evaluation of pure functional or logic programs.

Let us first recall the main ideas of partial deduction [Lloyd and Shepherdson, 1991]. Given a program $P$ and a set of atoms $S$, the aim of partial deduction is to derive a new program $P'$ which computes the same answers as $P$ for any input goal whose atoms are instances of the atoms in $S$. The program $P'$ is obtained by gathering together the set of *resultants*, which are constructed as follows:

- For each atom $A$ of $S$, first construct a finite SLD-tree, $\tau(A)$, for $P \cup \{\leftarrow A\}$.
- Then, consider the leaves of the nonfailing branches of $\tau(A)$, say $G_1, \ldots, G_r$, and the computed substitutions along these branches, say $\theta_1, \ldots, \theta_r$.
- Finally, construct the clauses:

$$\theta_1(A) \leftarrow G_1, \ldots, \theta_r(A) \leftarrow G_r$$

The restriction to specialize single atoms $A \in S$ is motivated by the fact that resultants are required to be Horn clauses. The correctness of the transformation is ensured whenever $P' \cup \{G\}$ is *S-closed*, i.e., every atom in $P' \cup \{G\}$ containing a predicate symbol occurring in an atom in $S$ is an instance of an atom in $S$. An *independence* condition, which holds if no two atoms in $S$ have a common instance, is needed to guarantee that the residual program $P'$ does not produce additional answers.

In the setting of functional logic programs, we proceed similarly to the partial deduction of logic programs. First, we formalize how specialized program rules (which correspond to the clauses of the residual logic program) are constructed from narrowing derivations (which correspond to the branches of the SLD-trees used in partial deduction). For this purpose, the notion of resultant is introduced.

**Definition 11 (resultant)** *Let $\mathcal{R}$ be a program and $s$ be a term. Given a narrowing derivation $s \leadsto_{\sigma}^{+} t$, its associated resultant is the rewrite rule $\sigma(s) \to t$.*

Note that, whenever the specialized call $s$ is not a linear pattern, left-hand sides of resultants may not be linear patterns either and hence resultants may not be program rules. In order to produce program rules, a post-processing renaming transformation is introduced. Additionally, this transformation not only eliminates redundant structures but also obtains independent specializations and is necessary for the correctness of the partial evaluation transformation, similarly to partial deduction. Roughly speaking, independence ensures that the different specializations for the same function definition are correctly distinguished, which is crucial for polyvariant specialization.

The set of narrowing derivations originating from a term, $t$, can be represented by a (possibly infinite) finitely branching tree rooted by $t$. Following [Lloyd and Shepherdson, 1991], we adopt the convention that any derivation is potentially incomplete (thus, a derivation is either infinite, successful, failed —i.e., ends in an expression that is neither a constructor term nor can be further narrowed— or incomplete). Given the narrowing tree $\mathcal{N}$ for a term $t$ in program $\mathcal{R}$, a *partial* (or *incomplete*) narrowing tree $\mathcal{N}'$ for $s$ in $\mathcal{R}$ is obtained by considering only the narrowing derivations from $t$ down to the terms $t_1, \ldots, t_n$ such that $t_1, \ldots, t_n$ appear in $\mathcal{N}$ and each non-failing branch of $\mathcal{N}$ contains exactly one of them.

The *(pre–)partial evaluation* of an operation-rooted term $s$ is obtained by constructing a (possibly incomplete) narrowing tree for $s$ and then extracting the specialized definitions (the resultants) from the non–failing, root–to–leaf paths of the tree.

**Definition 12 (pre–partial evaluation)** *Let $\mathcal{R}$ be a program and $s$ an operation-rooted term. Let $\mathcal{N}$ be a finite (possibly incomplete) narrowing tree for $s$ in $\mathcal{R}$ such that no constructor root-stable term in the tree has been narrowed. Let $\overline{t_n}$ be the terms in the non-failing leaves of $\mathcal{N}$. Then, the set of resultants for the narrowing sequences $\{s \leadsto^+_{\sigma_i} t_i \mid i = 1, \ldots, n\}$ is called a pre–partial evaluation of $s$ in $\mathcal{R}$.*

*The pre–partial evaluation of a set of operation-rooted terms $S$ in $\mathcal{R}$ is defined as the union of the pre–partial evaluations for the terms of $S$ in $\mathcal{R}$.*

Let us note that it is not necessary to evaluate root-stable terms during partial evaluation. However, since root-stability is undecidable, we impose the restriction to not surpass head normal forms (i.e., constructor root-stable forms).

Roughly speaking, the reason for requiring that the partial evaluation of a term $s$ does not surpass its head normal form is that, at runtime, the evaluation of a (nested) call $C[s]_p$ containing the partially evaluated term $s$ at some position $p$ might not *demand* evaluating $s$ beyond its head normal form. Since the "contexts" $C[\,]$ in which $s$ will appear are not known at partial evaluation time, we avoid to interfering with the "lazy nature" of computations in the specialized program by imposing this condition.

The following example illustrates that this restriction cannot be dropped. In the examples, we will usually underline the redex selected to be evaluated in the next step of the current derivation. For simplicity, substitutions are restricted to goal variables in the sequel.

**Example 3** Consider the following program $\mathcal{R}$:

$$\begin{array}{rcl} \mathtt{f\ 0} & = & \mathtt{0} \\ \mathtt{g\ x} & = & \mathtt{Succ\ (f\ x)} \\ \mathtt{h\ (Succ\ x)} & = & \mathtt{Succ\ 0} \end{array}$$

with the set of calls $S = \{\mathtt{g\ x},\ \mathtt{h\ x}\}$ and the following narrowing sequences for the terms in $S$:

$$\begin{array}{l} \underline{\mathtt{g\ x}} \leadsto_{\mathtt{id}} \mathtt{Succ}\ (\underline{\mathtt{f\ x}}) \leadsto_{\{\mathtt{x}\mapsto\mathtt{0}\}} \mathtt{Succ\ 0} \\ \underline{\mathtt{h\ x}} \leadsto_{\{\mathtt{x}\mapsto(\mathtt{Succ\ x'})\}} \mathtt{Succ\ 0} \end{array}$$

Then, a pre–partial evaluation of $S$ in $\mathcal{R}$ is the program $\mathcal{R}'$:

$$\begin{array}{rcl} \mathtt{g\ 0} & = & \mathtt{Succ\ 0} \\ \mathtt{h\ (Succ\ x)} & = & \mathtt{Succ\ 0} \end{array}$$

Now, the equation $\mathtt{h\ (g\ (Succ\ 0))} \approx \mathtt{x}$ has the following successful needed narrowing derivation in $\mathcal{R}$:

$$\begin{array}{rl} \mathtt{h}\ \underline{(\mathtt{g\ (Succ\ 0)})} \approx \mathtt{x} & \leadsto\ \underline{\mathtt{h\ (Succ\ (f\ (Succ\ 0)))}} \approx \mathtt{x}\ \leadsto\ (\mathtt{Succ\ 0}) \approx \mathtt{x} \\ & \leadsto^{*}_{\{\mathtt{x}\mapsto\mathtt{Succ\ 0}\}}\ \mathtt{True} \end{array}$$

whereas it fails in the specialized program $\mathcal{R}'$.

In logic programming, the partial deduction of a program with respect to $S$ is required to be $S$-closed, and only $S$-closed goals are considered. This condition permits to prove the completeness of the transformation. Roughly speaking, the notion of closedness guarantees that all calls which might occur during the execution of the resulting program are covered by some program rule. A *closedness* condition in the style of [Lloyd and Shepherdson, 1991] would basically say that a term $t$ is $S$-closed if each outer subterm $t'$ of $t$ headed by a defined function symbol is a constructor instance of some term in $S$. However, this notion of closedness is too restrictive for functional (logic) programs, where nested defined function symbols appear quite frequently. Therefore, we will use the generalized notion of closedness of Alpuente *et al.* [1998b], which copes with nested function calls which cannot be obtained through instantiation of the specialized calls by constructor substitutions. This is interesting because it brings more powerful correctness results which apply to many practical cases and which produce better specializations, as we will experimentally show in the forthcoming chapters. Intuitively, the use of a general notion of closedness implies a less frequent need for *generalization* (i.e., the operation which is needed to ensure the finiteness of the process), which diminishes the loss of information and improves the potential for specialization. Also, this recursive notion of closedness allows one to safely execute a wider class of input goals which are recursively covered by the set of specialized functions.

The recursive closedness condition is formalized by inductively checking that the different calls in the rules are sufficiently covered by the specialized functions. Informally, a term $t$ rooted by a defined function symbol is closed w.r.t. a set of calls $S$, if it is an instance of a term in $S$ and the terms in the matching substitution are recursively closed by $S$. In the following, we denote by $\mathcal{P}$ the set of *primitive* function symbols (e.g., the conjunctive symbol, the equality, etc.). Note that $\mathcal{P} \subseteq \mathcal{F}$.

**Definition 13 (closedness)** *Let $S$ be a finite set of operation-rooted terms. We say that a term $t$ is $S$-closed if $closed(S,t)$ holds, where the predicate closed is defined inductively as follows:*

$$closed(S,t) \;\Leftrightarrow \begin{cases} true & \text{if } t \in \mathcal{X} \\ closed(S,t_1) \wedge \ldots \wedge closed(S,t_n) & \text{if } t = c(\overline{t_n}), \\ & \quad c \in (\mathcal{C} \cup \mathcal{P}),\ n \geq 0 \\ \bigwedge_{x \mapsto t' \in \theta} closed(S,t') & \text{if } \exists \theta, \exists s \in S \text{ such that } \theta(s) = t \end{cases}$$

*We say that a set of terms $T$ is $S$-closed, written $closed(S,T)$, if $closed(S,t)$ holds for all $t \in T$, and we say that a TRS $\mathcal{R}$ is $S$-closed if $closed(S, \mathcal{R}_{calls})$ holds. Here we denote by $\mathcal{R}_{calls}$ the set of the right-hand sides of the rules in $\mathcal{R}$.*

According to this definition, variables are always closed, while an operation-rooted term is $S$-closed if it is an instance of some term in $S$ and the terms in the matching substitution are recursively $S$-closed. On the other hand, for constructor-rooted terms we proceed by checking the closedness of their arguments. Finally, for expression rooted by a "primitive" function symbol, we have two non-deterministic ways to proceed: either by checking the closedness of their arguments (as in the case of constructor-rooted terms) or by proceeding as in the case of an operation-rooted term. For instance, a conjunction $t_1 \wedge t_2$ or an equation $t_1 \approx t_2$, can be proven closed w.r.t. $S$ either by checking that it is an instance of a call in $S$ (followed by an inductive test of the subterms), or by splitting it into two conjuncts and then trying to match with the "simpler" terms $t_1$ and $t_2$ in $S$. The possibility of treating primitive symbols as constructors is useful when we are not interested in specializing complex expressions (like conjunctions or strict equations) but we still want to run them after specialization. Note that this is safe, since we consider that the rules which define the primitive functions are automatically added to each program, hence calls to these symbols are steadily covered in the specialized program. In Chapter 6, a general technique for

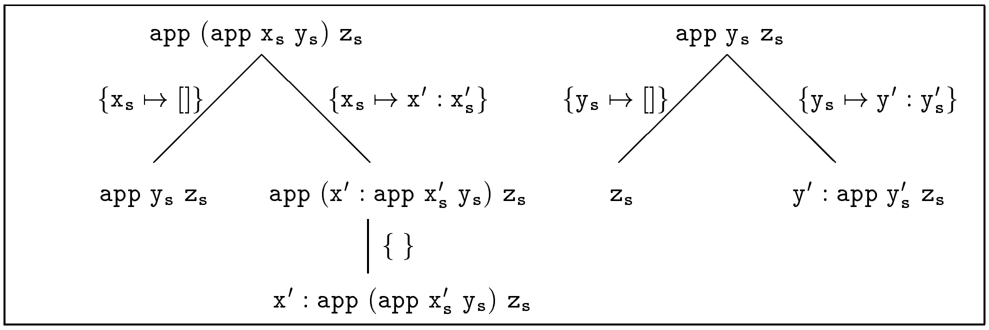


Figure 3.1: Partial needed narrowing trees for $\{$app (app $x_s$ $y_s$) $z_s$), app $x_s$ $y_s\}$

dealing with primitive symbols is presented.

In general, given a call $s$ and a program $\mathcal{R}$, there exists an infinite number of different pre–partial evaluations of $s$ in $\mathcal{R}$. A fixed rule for generating resultants called an *unfolding rule* is assumed, which determines the expressions to be narrowed and which decides how to stop the construction of narrowing trees (see, e.g., Chapter 6 for the definition of concrete unfolding rules).

**Example 4** Consider the well-known function append to concatenate two lists:

$$\begin{array}{l} \texttt{append}\ []\ \texttt{y}_\texttt{s} \to \texttt{y}_\texttt{s} \\ \texttt{append}\ (\texttt{x} : \texttt{x}_\texttt{s})\ \texttt{y}_\texttt{s} \to \texttt{x} : (\texttt{append}\ \texttt{x}_\texttt{s}\ \texttt{y}_\texttt{s}) \end{array}$$

with the set of calls $S = \{$append (append $x_s$ $y_s$) $z_s$, append $x_s$ $y_s\}$, and the partial needed narrowing trees depicted in Figure 3.1 for the terms in $S$. A pre–partial evaluation of $S$ in $\mathcal{R}$ using the partial needed narrowing trees of Figure 3.1 is the $S$-closed program:

$$\begin{array}{l} \texttt{append}\ (\texttt{append}\ []\ \texttt{y}_\texttt{s})\ \texttt{z}_\texttt{s} \to \texttt{append}\ \texttt{y}_\texttt{s}\ \texttt{z}_\texttt{s} \\ \texttt{append}\ (\texttt{append}\ \texttt{x} : \texttt{x}_\texttt{s}\ \texttt{y}_\texttt{s})\ \texttt{z}_\texttt{s} \to \texttt{x} : \texttt{append}\ (\texttt{append}\ \texttt{x}_\texttt{s}\ \texttt{y}_\texttt{s})\ \texttt{z}_\texttt{s} \\ \texttt{append}\ []\ \texttt{z}_\texttt{s} \to \texttt{z}_\texttt{s} \\ \texttt{append}\ (\texttt{y} : \texttt{y}_\texttt{s})\ \texttt{z}_\texttt{s} \to \texttt{y} : (\texttt{append}\ \texttt{y}_\texttt{s}\ \texttt{z}_\texttt{s}) \end{array}$$

In the following, we denote by pre–NN–PE the set of resultants computed for $S$ in $\mathcal{R}$ by considering an unfolding rule which constructs finite needed narrowing trees. We will use the acronym NN–PE for the renamed rules which will result from the correspondent *post-processing renaming* transformation. The idea behind this transformation is that, for any $S$-closed call $t$, the answers computed for $t$ in $\mathcal{R}$ and the answers computed for the renamed call in the specialized, renamed program coincide. In particular, in order to apply a partial evaluator based on needed narrowing and to ensure that the resulting program is inductively sequential whenever the source program is, we have to make sure that the set of specialized terms (after renaming) contains only linear patterns with distinct root symbols. This can be ensured by introducing a new function symbol for each specialized term and then replacing each call in the specialized program by a call to the corresponding renamed function. In particular, the left-hand sides of the specialized program (which are constructor instances of the specialized terms) are replaced by instances of the corresponding new linear patterns through renaming.

**Definition 14 (independent renaming)** *An independent renaming $\rho$ for a set of operation-rooted terms $S$ is a mapping from terms to terms defined as follows: for $s \in S$, $\rho(s) = f_s(\overline{x_n})$, where $\overline{x_n}$ are the distinct variables in $s$ in the order of their first occurrence and $f_s$ is a new function symbol, which does not occur in $\mathcal{R}$ or $S$ and is different from the root symbol of any other $\rho(s')$, with $s' \in S$ and $s' \neq s$. By abuse, we let $\rho(S)$ denote the set $S' = \{\rho(s) \mid s \in S\}$.*

Let us illustrate the above definition with an example.

**Example 5** Consider again the program `append` and the set of terms $S$ of Example 4. An independent renaming $\rho$ for $S$ is the mapping:

$$\{\ \texttt{append}\ \texttt{x}_\texttt{s}\ \texttt{y}_\texttt{s} \mapsto \texttt{app}\ \texttt{x}_\texttt{s}\ \texttt{y}_\texttt{s},$$
$$\texttt{append}\ (\texttt{append}\ \texttt{x}_\texttt{s}\ \texttt{y}_\texttt{s})\ \texttt{z}_\texttt{s} \mapsto \texttt{dapp}\ \texttt{x}_\texttt{s}\ \texttt{y}_\texttt{s}\ \texttt{z}_\texttt{s}\ \}.$$

While the independent renaming suffices to rename the left-hand sides of resultants (since they are constructor instances of the specialized calls), the right-hand sides are renamed by means of the auxiliary function $ren_\rho$, which *recursively* replaces each call in the given expression by a call to the corresponding renamed function (according to $\rho$).

**Definition 15** *Let $S$ be a finite set of operation-rooted terms and $\rho$ an independent renaming for $S$. We define the mapping $ren_\rho$ as follows:*

$$ren_\rho(t) = \begin{cases} t & \text{if } t \in \mathcal{X} \\ c(\overline{ren_\rho(t_n)}) & \text{if } t = c(\overline{t_n}),\ c \in (\mathcal{C} \cup \mathcal{P}),\ n \geq 0 \\ \theta'(\rho(s)) & \text{if } \exists\theta, \exists s \in S \text{ such that } t = \theta(s) \text{ and} \\ & \quad \theta' = \{x \mapsto ren_\rho(\theta(x)) \mid x \in \mathcal{D}om(\theta)\} \\ t & \text{otherwise} \end{cases}$$

Similarly to the test for closedness, an equation $s \approx t$ can be (non-deterministically) renamed either by independently renaming $s$ and $t$ or by replacing the considered equation by a call to the corresponding new, renamed function (when the equation is an instance of some specialized call in $S$). Note also that, whenever an operation-rooted term $t$ is not an instance of any term in $S$ (which can occur if $t$ is not $S$-closed), $ren_\rho(t)$ returns $t$ unchanged, i.e., the term $t$ is not renamed, which implies that it will not be closed w.r.t. the renamed calls $\rho(S)$. This case is interesting because it permits to define mapping $ren_\rho$ as a total function. On the other hand, it is not problematic since whenever $ren_\rho(t) = t$, the conditions for the correctness theorem (Theorem 23) are not fulfilled. The notion of partial evaluation can be formally defined as follows.

**Definition 16 (partial evaluation)** *Let $\mathcal{R}$ be a program, $S$ a finite set of operation-rooted terms and $\mathcal{R}'$ a pre–partial evaluation of $\mathcal{R}$ w.r.t. $S$. Let $\rho$ be an independent renaming of $S$. We define the partial evaluation $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) as follows:*

$$\mathcal{R}'' = \bigcup_{s \in S} \{\theta(\rho(s)) \to ren_\rho(r) \mid \theta(s) \to r \in \mathcal{R}' \textit{ is a resultant for } s \textit{ in } \mathcal{R}\}$$

We now illustrate the definition with an example.

**Example 6** Consider again the program `append` and the set $S$ of Example 4, and the independent renaming $\rho$ for $S$ of Example 5. A partial evaluation $\mathcal{R}'$ of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) is:

$$\begin{array}{rcl}
\mathtt{dapp}\ []\ \mathtt{y_s}\ \mathtt{z_s} & = & \mathtt{app}\ \mathtt{y_s}\ \mathtt{z_s} \\
\mathtt{dapp}\ (\mathtt{x} : \mathtt{x_s})\ \mathtt{y_s}\ \mathtt{z_s} & = & \mathtt{x} : (\mathtt{dapp}\ \mathtt{x_s}\ \mathtt{y_s}\ \mathtt{z_s}) \\
\mathtt{app}\ []\ \mathtt{y_s} & = & \mathtt{y_s} \\
\mathtt{app}\ (\mathtt{x} : \mathtt{x_s})\ \mathtt{y_s} & = & \mathtt{x} : (\mathtt{app}\ \mathtt{x_s}\ \mathtt{y_s})
\end{array}$$

We note that, for a given renaming $\rho$, the filtered form of a program $\mathcal{R}$ may depend on the strategy which selects the term from $\rho(S)$ which is used to rename a given call $t$ in $\mathcal{R}$ since there may exist, in general, more than one $s$ in $S$ that covers the call $t$. For instance, consider the set of terms $S = \{\mathtt{append}\ (\mathtt{append}\ \mathtt{x_s}\ \mathtt{y_s})\ \mathtt{z_s},\ \mathtt{append}\ \mathtt{x_s}\ \mathtt{y_s}\}$ and the independent renaming $\rho$ of Example 5. By using function $ren_\rho$, we can rename the expression $\mathtt{x_s} :\ \mathtt{append}\ (\mathtt{append}\ \mathtt{x_s}\ \mathtt{y_s})\ \mathtt{z_s}$ using the term $\mathtt{append}\ (\mathtt{append}\ \mathtt{x_s}\ \mathtt{y_s})\ \mathtt{z_s}$ as follows:

$$ren_\rho(\mathtt{x_s} :\ \mathtt{append}\ (\mathtt{append}\ \mathtt{x_s}\ \mathtt{y_s})\ \mathtt{z_s}) = \mathtt{x_s} :\ \mathtt{dapp}\ \mathtt{x_s}\ \mathtt{y_s}\ \mathtt{z_s}$$

Note also that the following renaming:

$$ren_\rho(\mathtt{x_s} :\ \mathtt{append}\ (\mathtt{append}\ \mathtt{x_s}\ \mathtt{y_s})\ \mathtt{z_s}) = \mathtt{x_s} :\ \mathtt{app}\ (\mathtt{app}\ \mathtt{x_s}\ \mathtt{y_s})\ \mathtt{z_s}$$

can be performed by using the second element of $S$ to rename the expression. Some potential specialization might be lost due to an inconvenient choice. Plausible heuristics able to produce the better potential specialization have been defined, for instance, in the Indy system (see Appendix A).

The following theorem states an important property of partial evaluation w.r.t. needed narrowing: if the input program is inductively sequential, then the specialized program is also inductively sequential so that we can apply the optimal needed narrowing strategy to the specialized program too.

**Theorem 17** *[Alpuente et al., 1999c] Let $\mathcal{R}$ be an inductively sequential program and $S$ a finite set of operation-rooted terms. Then each NN-PE of $\mathcal{R}$ w.r.t. $S$ is inductively sequential.*

Another nice property of NN-PE is that a term which is deterministically normalizable w.r.t. the original program cannot cause a non-deterministic evaluation w.r.t. the specialized program. The following proposition guarantees this property.

**Proposition 18** *[Alpuente et al., 1999c] Let $\mathcal{R}$ be an inductively sequential program, $S$ a finite set of operation-rooted terms, $\rho$ an independent renaming of $S$, and $e$ an equation. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If $e$ deterministically normalizes to true w.r.t. $\mathcal{R}$, then $e'$ deterministically normalizes to true w.r.t. $\mathcal{R}'$.*

This property of specialized programs is desirable and important from an implementation point of view, since the implementation of non-deterministic steps is an expensive operation in logic-oriented languages. Moreover, additional non-determinism in the specialized program might have the effect that solutions are no longer computable in a sequential implementation based on backtracking. Therefore, this property is also desirable in partial deduction of logic programs but, as far as we know, no similar results are known for partial deduction of logic programs.

Finally, we state the strong correctness of NN-PE (Theorem 23), which amounts to the full computational equivalence between the original and the specialized programs (i.e., the fact that the two programs compute exactly the same values and answers) for the considered goals. The proof proceeds essentially as follows. Firstly, the soundness (resp. completeness) of the transformation is proven at the level of rewriting, i.e., one proves that each reduction sequence in the original (resp. specialized) program can be performed in the specialized (resp. original) program for the considered queries:[1]

**Proposition 19** *[Alpuente et al., 1999c] Let $\mathcal{R}$ be an inductively sequential program. Let $e$ be an equation, $S$ a finite set of operation-rooted terms, $\rho$ an independent renaming of $S$, and $\mathcal{R}'$ a NN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If $e' \to^* true$ in $\mathcal{R}'$ then $e \to^* true$ in $\mathcal{R}$.*

**Proposition 20** *[Alpuente et al., 1999c] Let $\mathcal{R}$ be an inductively sequential program. Let $e$ be an equation, $S$ a finite set of operation-rooted terms, $\rho$ an independent renaming of $S$, and $\mathcal{R}'$ a NN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If $e \to^* true$ in $\mathcal{R}$ then $e' \to^* true$ in $\mathcal{R}'$.*

[1] For simplicity, the theorems are proved for equations since the soundness and completeness of needed narrowing have been also proven for equations instead of terms (see Theorems 6 and 7).

Now, the soundness (resp. completeness) of the transformation is obtained as a direct consequence of Proposition 19 (resp. Proposition 20) and the soundness and completeness of needed narrowing:

**Theorem 21 (soundness)** *[Alpuente et al., 1999c] Let $\mathcal{R}$ be an inductively sequential program. Let $e$ be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If $e' \leadsto^*_{\sigma'}$ true is a needed narrowing derivation for $e'$ in $\mathcal{R}'$, then there exists a needed narrowing derivation $e \leadsto^*_\sigma$ true in $\mathcal{R}$ such that $\sigma \leq \sigma'$ $[V]$.*

**Theorem 22 (completeness)** *[Alpuente et al., 1999c] Let $\mathcal{R}$ be an inductively sequential program. Let $e$ be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. If $e \leadsto^*_\sigma$ true is a needed narrowing derivation for $e$ in $\mathcal{R}$, then there exists a needed narrowing derivation $e' \leadsto^*_{\sigma'}$ true in $\mathcal{R}'$ such that $\sigma' \leq \sigma$ $[V]$.*

Finally, by using the minimality of needed narrowing, the strong correctness of NN-PE is provided, i.e., the answers computed in the original and the partially evaluated programs coincide (up to renaming).

**Theorem 23 (strong correctness)** *[Alpuente et al., 1999c] Let $\mathcal{R}$ be an inductively sequential program. Let $e$ be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{R}'$ be a NN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. Then, $e \leadsto^*_\sigma$ true is a needed narrowing derivation for $e$ in $\mathcal{R}$ iff there exists a needed narrowing derivation $e' \leadsto^*_{\sigma'}$ true in $\mathcal{R}'$ such that $\sigma' = \sigma$ $[V]$ (up to renaming).*

## 3.3 Related Work

Very little work has been done in the area of functional logic program specialization. Levi and Sirovich [1975] defined a partial evaluation procedure for the functional programming language TEL that uses a unification-based symbolic execution mechanism, which can be understood as (a form of lazy) narrowing. Darlington and Pull [1988] showed how unification can enable instantiation and unfolding steps to be combined to get the ability (of narrowing) to deal with logical variables. A partial evaluator for the functional language HOPE (extended with unification) was also outlined. No actual procedure was included, and no control issues were considered. The problems of ensuring termination and preserving semantics were not addressed in these papers.

Now we recall from [Alpuente *et al.*, 1998b] the relation to some works on partial evaluation of functional programs, term-rewriting systems, and partial deduction of logic programs.

### 3.3.1 Supercompilation

The work on supercompilation [Turchin, 1986a] is the closest to NPE. Supercompilation (supervised compilation) is a transformation technique for functional programs which consists of three core constituents: *driving*, *generalization*, and *generation of residual programs*. Driving can be understood as a unification-based function transformation mechanism, which uses some kind of evaluation machinery similar to (lazy) narrowing to build (possibly infinite) "computation trees" for a program with a given call. Turchin's papers use the following terminology [Romanenko, 1991]: "generalized pattern matching" stands for unification; "contractions" stands for narrowing substitutions and, very often in the texts, "driving" stands for the narrowing itself, i.e., the operation of instantiation of a function call for all possible value cases of the arguments, followed by unfolding of the different branches. By virtue of driving, the supercompiler is able to get the same amount of (unification-based) information propagation as the partial deduction of logic programs [Glück and Sørensen, 1994].

Turchin's work describes the supercompiler for Refal (Recursive Function Algorithmic Language), a pattern-matching functional language with a nonstandard notion of patterns. The semantics of Refal is given in terms of a rewriting interpreter (with the inside-out order of evaluation), but driving is embedded into the supercompiler—an outside-in metaevaluator over Refal which subsumes deforestation [Wadler, 1990], partial evaluation, and other standard transformations of functional programming languages [Glück and Sørensen, 1996; Sørensen *et al.*, 1994]. For example, supercompilation is able to support certain forms of theorem proving, program synthesis, and program inversion. Recently, Jones [1994] put Turchin's driving methodology on a solid semantic foundation which is not tied to any particular programming language or data structure.

The driving process does not preserve the semantics, as it can extend the domain of functions (as noted by Glück and Sørensen [1994], Jones *et al.* [1993], and Sørensen *et al.* [1994]). Techniques to ensure termination of driving are studied in [Sørensen and Glück, 1995] and [Turchin, 1988]. The idea of Turchin [1988] is to supervise the construction of the tree and, at certain moments, loop back, i.e., fold a configuration to one of the previous states, and in this way construct a finite graph. The *generalization* operation which makes it possible to loop back the current configuration is often necessary. In [Sørensen and Glück, 1995], termination is guaranteed following a method which is comparable to the general approach of Martens and Gallagher [1995] for ensuring global termination of partial deduction.

The positive supercompiler of Sørensen *et al.* [1996] is a reformulation of Turchin's supercompiler for a simpler call-by-name language with tree-structured patterns. In this language, all arguments of a function are input parameters, and there is only pattern matching for nonnested linear patterns. The formulation is generalizable to less restrictive languages at the cost of more complex driving algorithms. Local and global control are not (explicitly) distinguished, as only one-step unfolding is performed. A large evaluation structure is built which comprises something similar to both the local narrowing trees and the global configurations of Martens and Gallagher [1995]. Each call $t$ in the driving process graph produces a residual function. The body of the definition of the new function is derived from the descendants of $t$ in the graph.

We note that the "root-to-leaf" notion of resultant of Definition 11 (similar to partial deduction) gives rise to fewer rules than in the case of positive supercompilation, where rules are extracted from each single computation step, but has the disadvantage that there are less opportunities to find appropriate "regularities" (in the sense of Pettorossi and Proietti [1996a]), just because fewer specialized functions are produced. On the other hand, the recursive notion of closedness of Definition 13 substantially enhances the specialization power of the method, since it subsumes the *perfect* ("$\alpha$-identical") closedness test of Sørensen *et al.* [1996]. This recursive closedness is essentially equivalent to the folding operation introduced in [Sørensen and Glück, 1995], although the NPE method allows one to also fold back a function call which is closed by a different branch of the process tree.

Glück and Sørensen [1994] focused on the correspondence between partial deduction and driving, stating the similarities between the driving of a functional program and the construction of a SLD-tree for a similar Prolog program. The notions presented in this chapter can also be seen as a new formulation of the essential principle of driving in simpler and more familiar terms to the logic programming community. Regarding the correctness, there is a precise correspondence between the answer substitutions computed in the original and the specialized programs obtained by the partial evaluation algorithm. This is different from [Scherlis, 1981] and [Sands, 1995], where a specialization method for equations with complex left-hand sides is also defined (in the context of strict and nonstrict functional languages, respectively), but they can only compare the ground equational consequences semantics of the programs.

### 3.3.2 Partial Evaluation for Term-Rewriting Systems

Earlier work on partial evaluation of term-rewriting systems has focused on self-application rather than on the termination or the correctness of the transformation [Bondorf, 1988, 1989]. In [Bondorf, 1989], a self-applicable, call-by-value partial evaluator for (an intermediate language for) TRS's is presented, called TreeMix. During unfolding, variables are "backward" instantiated by a mechanism, introduced in [Bondorf, 1988], which involves unification and is very similar to driving. Rather than specializing a program in the form of a TRS, programs are first translated by TreeMix into intermediate code which is then specialized, and the residual program is finally translated back into a TRS. The intermediate language, called Tree, is able to express pattern matching at a level of abstraction which makes it feasible to achieve self-application. Bondorf's TreeMix works off-line, guided by program annotations generated before transformation. No correctness result for the partial evaluation transformation is given. Regarding termination, it is up to the user to decide, by manual annotation, when to unfold or when to generalize, in order to achieve accurate specialization. There is no guarantee against infinite specialization (or infinite unfolding). A severe restriction of the system is its fixed innermost reduction strategy: Tree is a nontyped, first-order, strict (innermost reduction order) functional language, and TreeMix heavily depends on the operational properties of the language. According to Bondorf [1989], call-by-name evaluation would require a complete revision of the partial evaluator.

The work by Miniussi and Sherman [1996] aims to improve the efficiency of a term-rewriting implementation of equational programs by tackling the problems associated to the partial evaluation of an imperative, intermediate code that the implementation uses for the equations. The method avoids some intermediate rewrite steps and achieves the fusing of intermediate data structures (in the style of Wadler's deforestation) without risking nontermination.

In [Bonacina, 1988], the author describes a (not much operational) partial evaluation algorithm based on (Knuth-Bendix) equational completion[2] (also called superposition). Completion is a unification-based rule commonly used in automatic reasoning to generate critical pairs (which are equational consequences of a set of rules coming from overlapping left-hand sides) as a means to derive canonical programs. Termination of the transformation is not guaranteed. Another related approach is that of Dershowitz and Reddy [1993], which also formulates a completion-based technique for the synthesis of functional programs, but requires a kind of heuristic ("eureka" rule to be discovered) and, thus, it is closer to fold/unfold transformation than to partial evaluation. Another *partial completion* procedure is defined by Bellegarde [1995], which is able to perform tupling and deforestation.

A different partial evaluation method for a rewriting-based, functional logic language is presented in [Lafave and Gallagher, 1997] which relies on a slight adaptation of the recursive notion of closedness of [Alpuente *et al.*, 1998b]. The method is specially designed for the language Escher [Lloyd, 1995], whose operational semantics is based on the residuation principle. Lafave and Gallagher consider the specialization of

[2] Narrowing can be seen as a "linear" restriction on completion, similar to "linear resolution."

arbitrary expressions just as in the NPE framework and, thus, a finer control strategy for specializing complex goals (in the style of the strategies presented in Chapter 6 and [Leuschel *et al.*, 1996]) is needed to achieve good specialization for some involved problems such as tupling.

### 3.3.3 Partial Deduction

Within our integrated paradigm, more general results than those already obtained in partial deduction can be achieved. In fact, if one considers the straightforward embedding of logic programs into functional logic programming, by means of boolean-valued functions (with constructor function calls as arguments) representing predicates, the NPE method essentially boils down to standard partial deduction. Namely, by using a simple, nonrecursive definition of closedness, the NPE results are essentially equivalent to those for partial deduction of pure positive logic programs.

On the other hand, standard partial deduction computes partial derivations for each atom independently. Recently, a technique for the partial deduction of conjunctions of atoms has been introduced in [Glück *et al.*, 1996], also known as *conjunctive* partial deduction. This technique achieves program optimizations such as (a limited form of) tupling and deforestation, which cannot be obtained by classical partial deduction. Within our framework, when one considers the generalized closedness which recurses over the structure of the terms, what it is obtained in turn is a framework which is essentially as powerful as conjunctive partial deduction. These frameworks can be speeded up by a fine treatment of predefined symbols, similar to the partitioning techniques of Glück *et al.* [1996], as we will also do in Chapter 6.

Experiments with the prototypical implementation described in Appendix A show that NPE combines some good features of deforestation [Wadler, 1990], partial evaluation [Consel and Danvy, 1993; Jones *et al.*, 1993], and partial deduction [Lloyd and Shepherdson, 1991; Martens and Gallagher, 1995], similarly to conjunctive partial deduction [Glück *et al.*, 1996; Leuschel *et al.*, 1996] and supercompilation [Turchin, 1986a; Glück and Sørensen, 1996].

# Chapter 4

# Partial Evaluation of Residuating Programs

In this chapter, we develop a partial evaluation technique for *residuating functional logic* programs, which generalize logic programs based on concurrent computation models with delays to functional logic programs. We show how to lift the nondeterministic choices from run time to specialization time. We ascertain the conditions under which the original and the transformed programs have the same answer expressions for the considered class of queries as well as the same floundering behavior. Preliminary empirical evaluation demonstrates that our technique also works well in practice and leads to substantial performance improvements. A short version of this chapter appeared in [Albert *et al.*, 1999a,b].

## 4.1 Introduction

The approach of Chapter 3 to the partial evaluation of functional logic programs is appropriate for a functional logic language whose operational semantics is based solely on narrowing. However, it is not powerful enough to develop a partial evaluation tool for a modern functional logic language (such as, e.g., Curry [Hanus, 2000a] and Escher [Lloyd, 1994]), since NN-PE does not consider the *residuation* principle.

The *residuation* principle is based on the idea of delaying function calls until they are ready for deterministic evaluation. Residuation preserves the deterministic nature of functions and naturally supports concurrent computations. Unfortunately, it is unable to compute solutions if arguments of functions are not sufficiently instantiated during the computation, although program analysis methods exist which provide sufficient criteria for the completeness of residuation [Boye, 1993; Hanus, 1995a].

Residuating functional logic languages employ dynamic scheduling to run efficiently, similarly to modern (constraint) logic programming languages, where some calls are dynamically delayed until their arguments are sufficiently instantiated to evaluate the call. Residuation is the basis for implementing many concurrent (constraint) programming languages such as Oz [Smolka, 1995] and is also used in other multi-paradigm declarative languages such as Escher [Lloyd, 1994, 1995], Le Fun [Aït-Kaci *et al.*, 1987], Life [Aït-Kaci, 1990], and NUE-Prolog [Naish, 1991].

Modern multi-paradigm declarative languages combine functional, logic and concurrent programming styles by unifying (needed) narrowing and residuation into a single model [Hanus, 1997b]. To support coroutining, the model usually provides for *suspension* of function calls if a *demanded* argument is not sufficiently instantiated. The precise mechanism (narrowing or residuation) for each function is specified by *evaluation annotations*. Deterministic functions are declared *rigid* (which forces delayed evaluation by rewriting), while non-deterministic functions are declared *flex* (which enables narrowing steps). The computation domain considers disjunctions of (*answer* $\|$ *expression*) pairs in order to reflect not only the computed values but also the different variable bindings. The following example illustrates the integrated model (the computation steps are denoted by $\xrightarrow{\texttt{RN}}$ as in [Hanus, 1997b]).

**Example 7** Consider the rules defining the less-or-equal function "$\leqslant$" and the addition "+" on natural numbers:

$$
\begin{array}{rcllcrcl}
\texttt{0} & \leqslant & \texttt{n} & = \texttt{True} & \qquad & \texttt{0} + \texttt{x} & = & \texttt{x} \\
(\texttt{Succ m}) & \leqslant & \texttt{0} & = \texttt{false} & & (\texttt{Succ x}) + \texttt{y} & = & \texttt{Succ} \ (\texttt{x} + \texttt{y}) \\
(\texttt{Succ m}) & \leqslant & (\texttt{Succ n}) & = \texttt{m} \leqslant \texttt{n} & & & &
\end{array}
$$

where "$\leqslant$" is rigid and "+" is flexible. Then, the following goal is evaluated by freezing and awakening the function call to "$\leqslant$":[1]

$$
\begin{array}{l}
\texttt{id} \ \| \ (\texttt{x} \leqslant \texttt{y}) \ \& \ (\underline{\texttt{x} + \texttt{0}} \approx \texttt{0}) \\
\quad \xrightarrow{\texttt{RN}} \ \{\texttt{x} = \texttt{0}\} \ \| \ \underline{\texttt{0} \leqslant \texttt{y}} \ \& \ \texttt{0} \approx \texttt{0} \\
\qquad \vee \ \{\texttt{x} = (\texttt{Succ z})\} \ \| \ ((\texttt{Succ z}) \leqslant \texttt{y}) \ \& \ (\texttt{Succ} \ (\texttt{z} + \texttt{0}) \approx \texttt{0}) \\
\quad \xrightarrow{\texttt{RN}} \ \{\texttt{x} = \texttt{0}\} \ \| \ \texttt{True} \ \& \ \texttt{0} \approx \texttt{0} \\
\qquad \vee \ \{\texttt{x} = (\texttt{Succ z})\} \ \| \ ((\texttt{Succ z}) \leqslant \texttt{y}) \ \& \ (\underline{\texttt{Succ} \ (\texttt{z} + \texttt{0}) \approx \texttt{0}}) \\
\quad \xrightarrow{\texttt{RN}} \ \{\texttt{x} = \texttt{0}\} \ \| \ \texttt{True} \ \& \ \underline{\texttt{0} \approx \texttt{0}} \\
\quad \xrightarrow{\texttt{RN}} \ \{\texttt{x} = \texttt{0}\} \ \| \ \underline{\texttt{True} \ \& \ \texttt{True}} \\
\quad \xrightarrow{\texttt{RN}} \ \{\texttt{x} = \texttt{0}\} \ \| \ \texttt{True} \ .
\end{array}
$$

Note that the second disjunction fails since $\texttt{Succ} \ (\texttt{z} + \texttt{0}) \approx \texttt{0}$ is unsolvable.

[1] Here & is the *concurrent conjunction operator*, i.e., the expression $e_1$ & $e_2$ is reduced by reducing either $e_1$ or $e_2$.

Hence, we generalize the basic framework of Chapter 3 in order to deal with (inductively sequential) programs containing evaluation annotations for program functions. This task is difficult for several reasons. Firstly, a naïve adaptation of the method introduced in Chapter 3 in which floundering computations are simply stopped during partial evaluation is not adequate, since it is unsafe in our setting and, also, a poor specialization would be obtained in most cases (see Example 9). Thus, we use an extension of the standard computation model which allows us to ignore evaluation annotations during partial evaluation while still guaranteeing correctness. As a consequence, our method is less restrictive than many existing methods for (constraint) logic programs with delays, in which suspended expressions cannot be unfolded (e.g., [Etalle *et al.*, 1997]). Secondly, the inference of safe evaluation annotations for the partially evaluated programs is far from trivial (see Example 10). In particular, we are forced to split resultants into several auxiliary (intermediate) functions in some cases to correctly preserve the answer expressions as well as the floundering behaviour.

We also provide correctness results for the transformation, including the equivalence between the original and specialized programs w.r.t. floundering-freeness. This can be used for proving completeness of residuation for the considered class of goals in the original program by analyzing the floundering behavior of the resulting program. In particular, proving floundering-freeness for the specialized program is in many cases trivial (or easier than in the original program) because partial evaluation can transform a rigid function into a flexible one (whenever the specialized call is already sufficiently instantiated), but not vice versa. Moreover, we also prove that the transformation preserves the (inductively sequential) structure of programs.

## 4.2 A Unified Computational Model

The definition of needed narrowing (Section 2.4.2) and its extension to concurrent programming [Hanus, 1997b] is based on definitional trees which have been introduced in Section 2.4.2 for the specification of efficient rewrite strategies. For our purposes, we reformulate this formalization of definitional trees by adding a new parameter $r$ which allows us to determine the precise strategy in order to evaluate a function call:

$\mathcal{T} = rule(\pi \to \lambda)$, where $\pi \to \lambda$ is a variant of a rule.

$\mathcal{T} = branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where $o$ is an occurrence of a variable in $\pi$, $r \in \{rigid, flex\}$, $c_1, \ldots, c_k$ are different constructors of the sort of $\pi|_o$, for some $k > 0$, and, for all $i = 1, \ldots, k$, $\mathcal{T}_i$ is a definitional tree with pattern $\pi[c_i(x_1, \ldots, x_n)]_o$, where $n$ is the arity of $c_i$ and $x_1, \ldots, x_n$ are new variables.

We call a function *flexible* or *rigid* if all the branch nodes in its definitional tree are $flex$ or $rigid$, respectively.

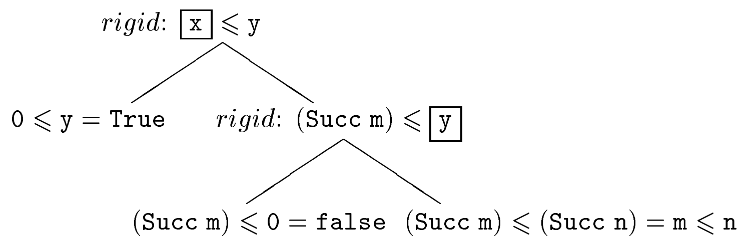


Figure 4.1: Definitional tree for the function "$\leqslant$"

**Example 8** Consider the rules defining function "$\leqslant$" in Example 7. Then

$$
\begin{aligned}
branch(&\texttt{x} \leqslant \texttt{y}, 1, rigid, rule(\texttt{0} \leqslant \texttt{y} = \texttt{True}),\\
&branch((\texttt{Succ m}) \leqslant \texttt{y}, 2, rigid,\\
&\quad rule((\texttt{Succ m}) \leqslant \texttt{0} = \texttt{false}),\\
&\quad rule((\texttt{Succ m}) \leqslant (\texttt{Succ n}) = \texttt{m} \leqslant \texttt{n})))
\end{aligned}
$$

is a definitional tree of $\leqslant$ which allows us to specify that function $\leqslant$ must be evaluated by residuation (due to the $rigid$ annotation). As discussed in Section 2.4.2, it is often convenient to provide a graphic representation of definitional trees. Moreover, here we decorate each inner node with the $flex/rigid$ annotation. For instance, the adorned definitional tree for the function "$\leqslant$" is illustrated in Figure 4.1.

To provide concurrent computation threads, expressions can be combined by the *concurrent conjunction operator* &, i.e., the expression $e_1$ & $e_2$ can be reduced by reducing either $e_1$ or $e_2$. Note that we obtain the behavior of the needed narrowing strategy defined in Section 2.4.2 if all functions are flexible. Moreover, functional logic languages which are based on residuation, like Life or Escher, where functions are always deterministically evaluated or suspended and non–determinism is encoded by predicates, can be modeled with programs where all (non–Boolean) functions are rigid and all predicates (Boolean functions) are flexible.

For a precise definition of this operational semantics, it is convenient to distinguish between *complete* computation steps where one reduction has been performed and *incomplete* computation steps which are suspended due to some rigid branch.[2] Incomplete steps are called *degenerate* in [Antoy, 1997] in the sense that some variables could have been instantiated but no subsequent reduction has been performed. We mark a substitution in an answer expression by the superscript $s$, i.e., $\sigma^s \circ \sigma' \;[\!]\; t$ denotes a *suspended answer expression* where the reduction part of the step has not

[2] In [Hanus, 1997b], this distinction is made by a special constant in the domain of disjunctive expressions while here we use a special mark at substitutions in answer expressions. We find this more convenient to formulate the partial evaluation method for residuating programs as will become apparent in Section 4.4.

been performed due to a suspension in a rigid branch. For convenience, we denote by $\sigma^i$ a composed substitution with $\sigma = \sigma_n^s \circ \cdots \circ \sigma_1$, and by $\sigma^c$ a composed substitution with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ where $\sigma_n$ does not have the form $\varphi^s$. Marks in substitutions are only a technical artifice to simplify our formulation and are simply ignored when composing and applying substitutions.

Given a narrowing derivation $t \leadsto_\sigma^* e$, we define an *answer expression* as a pair $\sigma \,[\!]\, e$ consisting of the substitution $\sigma$ and the expression $e$. An answer expression $\sigma \,[\!]\, e$ is *solved* if $e$ is a constructor term, otherwise it is *unsolved*. Since more than one answer may exist for expressions containing free variables, expressions are reduced to disjunctions of answer expressions. A *disjunctive expression* is a (multi-)set of answer expressions $\{\sigma_1 \,[\!]\, e_1, \ldots, \sigma_n \,[\!]\, e_n\}$, sometimes written as $(\sigma_1 \,[\!]\, e_1) \vee \ldots \vee (\sigma_n \,[\!]\, e_n)$. The set of all disjunctive expressions is denoted by $\mathcal{D}$. $\mathcal{D}^s$ denotes the set of all disjunctive expressions where each disjunct could also be a suspended answer expression. Then, the operational semantics is specified by the functions (see Figure 4.2):

$$cs : \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X}) \to \mathcal{D}^s \quad \text{and} \quad cst : \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X}) \times DT \to \mathcal{D}^s$$

where $DT$ stands for the set of all definitional trees. Moreover, the composition of substitutions and the replacement of subterms is extended to disjunctive expressions as follows:

$$\{\sigma_1 \,[\!]\, t_1, \ldots, \sigma_n \,[\!]\, t_n\} \circ \sigma = \{\sigma_1 \circ \sigma \,[\!]\, t_1, \ldots, \sigma_n \circ \sigma \,[\!]\, t_n\}$$
$$t[\{\sigma_1 \,[\!]\, t_1, \ldots, \sigma_n \,[\!]\, t_n\}]_o = \{\sigma_1 \,[\!]\, \sigma_1(t)[t_1]_o, \ldots, \sigma_n \,[\!]\, \sigma_n(t)[t_n]_o\}$$

As remarked in Section 2.4.2, we assume that the definitional trees always contain new variables if they are used in a narrowing step.

The overall computation strategy is a transformation $\overset{\mathtt{RN}}{\longrightarrow}$ on disjunctive expressions. It takes an operation-rooted term $t$ of a non-suspended disjunct. Then the computation step $cs(t)$ stemming from $t$ is performed, and the selected disjunct is replaced by the computed disjunction composed with the answer computed to that point. A single computation step $cs(t)$ applies a rule, if possible (first case of $cst$), or checks the subterm corresponding to the inductive position of the branch (second case of $cst$): if it is a constructor, we proceed with the corresponding subtree (if possible); if it is a function, we evaluate it by recursively applying the strategy to this subterm; if it is a variable, we suspend (in the case of a rigid branch) or non-deterministically instantiate the variable to the constructors of all children and proceed. Hence, a concurrent conjunction of two expressions proceeds by evaluating the conjunct which does not suspend. We say that a computation $D \overset{\mathtt{RN}}{\longrightarrow}^* D'$ *flounders* if every answer expression $\sigma^i \,[\!]\, t \in D'$ is suspended. A goal $e$ *flounders* iff the computation starting from $e$ flounders. Note that, in each recursive step during the computation of $cst$, we compose the current substitution with the local substitution

**Computation step for a single operation-rooted term $t$:**

$$cs(f(t_1,\ldots,t_n)) = cst(f(t_1,\ldots,t_n),\mathcal{T}) \quad \text{if } \mathcal{T} \text{ is a definitional tree for } f$$

$$cs(t_1 \,\&\, t_2) = \begin{cases} \texttt{True} & \text{if } t_1 = t_2 = \texttt{True} \\ (t_1 \,\&\, t_2)[cs(t_1)]_1 & \text{if } t_1 \neq \texttt{True} \text{ and } cs(t_1) \text{ does not suspend} \\ (t_1 \,\&\, t_2)[cs(t_2)]_2 & \text{if } t_2 \neq \texttt{True},\ cs(t_2) \text{ does not suspend,} \\ & \quad \text{and } cs(t_1) \text{ suspends} \\ id^s \,\llbracket\, t_1 \,\&\, t_2 & \text{otherwise} \end{cases}$$

$$cst(t, rule(l \to r)) = id \,\llbracket\, \sigma(r) \quad \text{if } \sigma \text{ is a substitution with } \sigma(l) = t$$

$$cst(t, branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k))$$

$$= \begin{cases} cst(t, \mathcal{T}_i) \circ id & \text{if } t|_o = c(t_1,\ldots,t_n) \text{ and } pattern(\mathcal{T}_i)|_o = c(x_1,\ldots,x_n) \\ \varnothing & \text{if } t|_o = c(\ldots) \text{ and } pattern(\mathcal{T}_i)|_o \neq c(\ldots), i = 1,\ldots,k \\ id^s \,\llbracket\, t & \text{if } t|_o = x \text{ and } r = rigid \\ \cup_{i=1}^{k} cst(\sigma_i(t), \mathcal{T}_i) \circ \sigma_i & \text{if } t|_o = x, r = flex, \text{ and } \sigma_i = \{x \mapsto pattern(\mathcal{T}_i)|_o\} \\ t[cs(t|_o)]_o \circ id & \text{if } t|_o = f(t_1,\ldots,t_n) \end{cases}$$

**Derivation step for a disjunctive expression:**

$$(\sigma^c \,\llbracket\, t) \vee D \xrightarrow{\texttt{RN}} (\sigma_1 \circ \sigma^c \,\llbracket\, t_1) \vee \ldots \vee (\sigma_n \circ \sigma^c \,\llbracket\, t_n) \vee D$$
if $t$ is operation-rooted and $cs(t) = \sigma_1 \,\llbracket\, t_1 \vee \ldots \vee \sigma_n \,\llbracket\, t_n$

Figure 4.2: Operational semantics of concurrent functional logic programming

of this step (which can be the identity). Thus, each computation step can be represented as $cs(t) = \bigvee_{i=1}^{n} \sigma_{i,k_i} \circ \cdots \circ \sigma_{i,1} \,\llbracket\, t_i$, where each $\sigma_{i,j}$ is either the identity or the replacement of a single variable computed in each recursive step. This is also called the *canonical representation* of a computation step.

This strategy was first introduced by Hanus [1997b] and differs from lazy functional languages only in the possible instantiation of free variables and from logic languages in the lazy evaluation of nested function calls. Moreover, logic programs with coroutining (i.e., delayed predicates waiting for the instantiation of some argument) can be modeled by the use of the concurrent conjunction operator &.

The following theorems relate the results computed by the operational semantics of concurrent functional logic programming ( $\xrightarrow{\texttt{RN}}$ ) to a rewriting semantics. In particular, the next theorem shows the soundness of the unified computational model.

**Theorem 24** *[Hanus, 1997a] If there is a derivation sequence*

$$id \,\llbracket\, e \xrightarrow{\texttt{RN}}{}^{*} \{\sigma_1 \,\llbracket\, e_1 \vee \ldots \vee \sigma_n \,\llbracket\, e_n\}$$

*then* $\sigma_i(e) \to_{\mathcal{R}}^{*} e_i$ *for* $i = 1,\ldots,n$.

For the completeness, one cannot expect a result similar to logic programming (i.e., every correct answer is subsumed by a computed one) since expressions may be suspended when the arguments are not sufficiently instantiated. Moreover, the inversion of the previous theorem cannot be proved since $\stackrel{\mathtt{RN}}{\longrightarrow}$ evaluates only in a lazy manner so that some rewrite steps w.r.t. $\to_{\mathcal{R}}$ do not correspond to steps computed by $\stackrel{\mathtt{RN}}{\longrightarrow}$. However, it holds that part of the solution is computed by $\stackrel{\mathtt{RN}}{\longrightarrow}$. The following theorem states this result.

**Theorem 25** *[Hanus, 1997a] Let $\sigma$ be a substitution and $c$ a constructor such that $\sigma(e) \to^*_{\mathcal{R}} c$ and*

$$id \,\|\, e \stackrel{\mathtt{RN}}{\longrightarrow}{}^{*} \{\sigma_1 \,\|\, e_1 \vee \ldots \vee \sigma_n \,\|\, e_n\}$$

*be a derivation sequence. Then there exists a substitution $\gamma$ with $\sigma = \gamma \circ \sigma_i$, for some $i \in \{1, \ldots, n\}$, and $\gamma(e_i) \to^*_{\mathcal{R}} c$.*

The suspension of function calls is the only reason why solutions cannot be fully computed. To demonstrate this, one can consider the subclass of functional logic programs where $rigid$ branches do not occur. This result corresponds to completeness of lazy narrowing and requires a *fair* selection of unsolved disjuncts $\sigma \,\|\, e$ in the computation steps of $\stackrel{\mathtt{RN}}{\longrightarrow}$ (i.e., every valuable disjunct is evaluated at some time).

**Theorem 26** *[Hanus, 1997a] Let $\mathcal{R}$ be a program where all branch nodes in definitional trees are $flexible$, $\sigma$ a substitution and $c$ a constructor such that $\sigma(e) \to^*_{\mathcal{R}} c$. Then there exists a derivation sequence*

$$id \,\|\, e \stackrel{\mathtt{RN}}{\longrightarrow}{}^{*} \{\sigma_1 \,\|\, e_1 \vee \ldots \vee \sigma_n \,\|\, e_n\}$$

*with $e_i = c$ and $\sigma = \gamma \circ \sigma_i$, for some $i \in \{1, \ldots, n\}$, and substitution $\gamma$.*

In contrast to the classical definition of narrowing, the definition of $\stackrel{\mathtt{RN}}{\longrightarrow}$ provides all ("don't know" non-deterministic) derivations at once by deriving an expression into a disjunctive expression. In order to relate $\stackrel{\mathtt{RN}}{\longrightarrow}$ to the classical nondeterministic narrowing relation, we also write $t \stackrel{\mathtt{RN}}{\leadsto}_{\sigma} t'$ if $\sigma \,\|\, t' \in cs(t)$.

The main difference with the needed narrowing strategy, as introduced in Section 2.4.2, is the possibility that function calls may suspend and the special treatment of the concurrent conjunction & to deal with suspended evaluations. Therefore, we will denote by $\stackrel{\mathtt{NN}}{\longrightarrow}$ and $\stackrel{\mathtt{NN}}{\leadsto}$ the relations defined similarly to $\stackrel{\mathtt{RN}}{\longrightarrow}$ and $\stackrel{\mathtt{RN}}{\leadsto}$ above but where the definition of $cs(t_1 \,\&\, t_2)$ is omitted and the case "$id^s \,\|\, t$ if $t|_o = x$ and $r = rigid$" is replaced by

$$\begin{array}{rcl} cst(t, branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)) & = & \cup_{i=1}^{k} cst(\sigma_i(t), \mathcal{T}_i) \circ \sigma_i^s \\ & & \text{if } t|_o = x, r = rigid, \text{ and } \sigma_i = \{x \mapsto pattern(\mathcal{T}_i)|_o\}. \end{array}$$

The fact that $\stackrel{\mathtt{NN}}{\longrightarrow}$ also decorates suspended bindings with the superscript $s$ instead of simply omitting the case "$id^s \,\|\, t$ if $t|_o = x$ and $r = rigid$" and the condition "$r =$

*flex*" in the definition of *cst* (giving rise to the narrowing strategy of Section 2.4.2) will become useful in Section 4.4.

Note that the meaning of the concurrent conjunction & can be defined by the single rewrite rule `True & True` $\rightarrow$ `True` which we assume to be implicitly added to the rewrite system when we consider needed narrowing steps. This function is inductively sequential and has the two definitional trees

$$branch(\mathtt{x} \,\&\, \mathtt{y}, 1, rigid, branch(\mathtt{True} \,\&\, \mathtt{y}, 2, rigid, rule(\mathtt{True} \,\&\, \mathtt{True} \rightarrow \mathtt{True})))$$

and

$$branch(\mathtt{x} \,\&\, \mathtt{y}, 2, rigid, branch(\mathtt{x} \,\&\, \mathtt{True}, 1, rigid, rule(\mathtt{True} \,\&\, \mathtt{True} \rightarrow \mathtt{True}))).$$

Now consider a term like $t_1 \,\&\, t_2$. It is obvious that a $\stackrel{\mathtt{RN}}{\longrightarrow}$ step where $t_1$ is evaluated corresponds to a needed narrowing step where the first definitional tree is taken for the root function &. Similarly, a $\stackrel{\mathtt{RN}}{\longrightarrow}$ step where $t_2$ is evaluated corresponds to a needed narrowing step with the second definitional tree for &. Now, we formalize the relation between the two calculi. In order to prove the relation between the $\stackrel{\mathtt{RN}}{\longrightarrow}$ calculus and needed narrowing (Theorem 28), we need the following proposition. Informally, it states that for each $\stackrel{\mathtt{RN}}{\longrightarrow}$ step, there is a $\stackrel{\mathtt{NN}}{\longrightarrow}$ step which leads to the same value and computes the same answer.

**Proposition 27** *Let $\mathcal{R}$ be an inductively sequential program. Let $t$ be a term. If $t \stackrel{\mathtt{RN}}{\leadsto}_\sigma t'$ is a complete computation step, then there exists a needed narrowing step $t \stackrel{\mathtt{NN}}{\leadsto}_\sigma t'$.*

*Proof.* We distinguish two different cases:

- Let $t$ be a term which does not contain any conjunction symbol.

  In this case, it is immediate to see that the two calculi are perfectly equivalent.

- Let $t$ be a conjunctive expression of the form $t_1 \,\&\, t_2$.

  In [Antoy *et al.*, 2000], the definition of needed narrowing allows one to consider an arbitrary definitional tree at each narrowing step. Moreover, the definition of the conjunction is inductively sequential and has two definitional trees (see above). Therefore, each conjunct $t_i$, $i \in \{1, 2\}$ can be solved by using the corresponding definitional tree and this is correct, as it stated in Theorem 8.

$\square$

Finally, by using Proposition 27 as well as the correctness of the $\stackrel{\mathtt{RN}}{\longrightarrow}$ calculus and the correctness of needed narrowing, we proceed to formally state the relation between both calculi.

**Theorem 28** *Let $\mathcal{R}$ be an inductively sequential program and $t$ an operation-rooted term.*

1. *If all steps in the derivation $t \overset{\mathtt{RN}^*}{\rightsquigarrow}_\sigma t'$ are complete, then there exists a needed narrowing derivation $t \overset{\mathtt{NN}^*}{\rightsquigarrow}_\sigma t'$ in $\mathcal{R}$.*

2. *If $t \overset{\mathtt{NN}^*}{\rightsquigarrow}_\sigma t'$ is a needed narrowing derivation for $t$ in $\mathcal{R}$, then there exists a derivation $t \overset{\mathtt{RN}^*}{\rightsquigarrow}_\theta t''$ such that $\exists\varphi.\ \varphi(t'') \to^* t'$ and $\sigma = \varphi \circ \theta$.*

*Proof.*

($\subseteq$) Immediate by Proposition 27.

($\supseteq$) Let $t \overset{\mathtt{NN}}{\rightsquigarrow}^*{}_\sigma\, t'$ be a needed narrowing derivation in $\mathcal{R}$. By the soundness of needed narrowing, we have $\sigma(t) \to^* t'$ in $\mathcal{R}$. Now, by the completeness of $\overset{\mathtt{RN}}{\longrightarrow}$, [3] we have $e \overset{\mathtt{RN}}{\rightsquigarrow}^*{}_\theta\, t''$ such that $\exists\varphi.\ \varphi(t'') \to^* t'$ and $\sigma = \varphi \circ \theta$.

□

## 4.3 A Naïve Extension

In this section, we extend the NN-PE framework of Chapter 3 in order to take into account delayed function calls during partial evaluation. In principle, we could simply use the $\overset{\mathtt{RN}}{\longrightarrow}$ calculus to perform partial computations. Therefore, specialized definitions would be basically produced by constructing a set of resultants of the form

$$\begin{array}{lcl} \sigma_1(s) & \to & t_1 \\ \ldots & & \\ \sigma_n(s) & \to & t_n \end{array}$$

associated to a given (partial) computation:

$$id \,\|\, s \overset{\mathtt{RN}}{\longrightarrow}^+ \{\sigma_1 \,\|\, t_1 \ \vee\ \ldots\ \vee\ \sigma_n \,\|\, t_n\}\ .$$

After that, the renaming transformation of Definition 14 could be performed in order to ensure that the specialized function is inductively sequential and also to guarantee independence (in the sense of [Lloyd and Shepherdson, 1991]).

[3] Theorem 25 states the completeness of $\overset{\mathtt{RN}}{\longrightarrow}$ for derivations ending up in a constructor term. Here, we need an analogous result for derivations to arbitrary terms; nevertheless, it can be easily derived from Theorem 25.

However, the framework of partial evaluation above cannot be simply transferred to residuating programs, since a naïve treatment of suspended calls can give rise to resultants which do not preserve program behavior, as illustrated in the following examples.

**Example 9** Consider again the rules defining the functions "$\leqslant$" and "+" of Example 7, and assume now that "$\leqslant$" is flexible and "+" is rigid. Given the expression $\mathtt{x} \leqslant \mathtt{y} + \mathtt{0}$, we have the partial computation

$$\mathtt{id} \,\|\, \mathtt{x} \leqslant \mathtt{y} + \mathtt{0} \;\overset{\mathtt{RN}}{\longrightarrow}\; \{\mathtt{x} = \mathtt{0}\} \,\|\, \mathtt{True} \;\vee\; \{\mathtt{x} = \mathtt{Succ\ m}\}^{\mathtt{s}} \,\|\, (\mathtt{Succ\ m}) \leqslant \mathtt{y} + \mathtt{0}$$

in which the second disjunct corresponds to an incomplete step. If we consider the following renaming function:

$$\{\mathtt{x} \leqslant \mathtt{y} + \mathtt{0} \mapsto \mathtt{sumleq\ x\ y}\}$$

then the associated resultants are the following:

$$\begin{array}{rcl}
\mathtt{sumleq\ 0\ y} & = & \mathtt{True} \\
\mathtt{sumleq\ (Succ\ m)\ y} & = & \mathtt{sumleq\ (Succ\ m)\ y}
\end{array}$$

Obviously, any specialization containing the second rule does not preserve the semantics of the original program (for the intended goals). Unfortunately, getting rid of this trivial resultant does not preserve the semantics either.

The above example reveals the need to relax the standard computation model during partial evaluation in order to "complete" the suspended steps in some suitable way. For instance, we could avoid suspensions by simply replacing $\overset{\mathtt{RN}}{\longrightarrow}$ with $\overset{\mathtt{NN}}{\longrightarrow}$ during partial evaluation. This raises the question of whether it is possible to infer safe evaluation annotations for the specialized definitions, i.e., annotations such that correctness is entailed. The following example answers this question negatively.

**Example 10** Reconsider the program and goal of Example 9, but use $\overset{\mathtt{NN}}{\longrightarrow}$ to construct the partial computation:

$$\begin{array}{rl}
\mathtt{id} \,\|\, \mathtt{x} \leqslant \mathtt{y} + \mathtt{0} \;\overset{\mathtt{NN}}{\longrightarrow} & \{\mathtt{x} = \mathtt{0}\} \,\|\, \mathtt{True} \\
& \vee\ \{\mathtt{x} = \mathtt{Succ\ m}, \mathtt{y} = \mathtt{0}\} \,\|\, (\mathtt{Succ\ m}) \leqslant \mathtt{0} \\
& \vee\ \{\mathtt{x} = \mathtt{Succ\ m}, \mathtt{y} = \mathtt{Succ\ z}\} \,\|\, \mathtt{Succ\ m} \leqslant \mathtt{Succ\ (z + 0)}
\end{array}$$

by using the renaming function of Example 9, we get the following resultants:

$$\begin{array}{rcl}
\mathtt{sumleq\ 0\ y} & = & \mathtt{True} \\
\mathtt{sumleq\ (Succ\ m)\ 0} & = & (\mathtt{Succ\ m}) \leqslant \mathtt{0} \\
\mathtt{sumleq\ (Succ\ m)\ (Succ\ m)} & = & (\mathtt{Succ\ m}) \leqslant \mathtt{Succ\ (z + 0)}
\end{array}$$

Then, neither *flex* nor *rigid* is a correct annotation for the specialized rules. If we assume that they are flexible, then a goal of the form "$(\mathtt{Succ\ x}) \leqslant \mathtt{y} + \mathtt{0}$" (whose renamed version is "$\mathtt{sumleq\ (Succ\ x)\ y}$") would succeed using the specialized rules:

$$\begin{array}{lcl} \mathtt{id} \,\|\, \mathtt{sumleq\ (Succ\ x)\ y} \xrightarrow{\mathtt{RN}} & & \{\mathtt{y} = \mathtt{0}\} \,\|\, (\mathtt{Succ\ x}) \leqslant \mathtt{0} \\ & \vee & \{\mathtt{y} = \mathtt{Succ\ m}\} \,\|\, (\mathtt{Succ\ x}) \leqslant \mathtt{Succ}\ (\mathtt{m} + \mathtt{0}) \end{array}$$

whereas it suspends in the original program:

$$\mathtt{id} \,\|\, (\mathtt{Succ\ x}) \leqslant \mathtt{y} + \mathtt{0} \xrightarrow{\mathtt{RN}} \mathtt{id}^{\mathtt{s}} \,\|\, (\mathtt{Succ\ x}) \leqslant \mathtt{y} + \mathtt{0}$$

On the other hand, declaring the new definition as *rigid* does not work either, since a goal $\mathtt{x} \leqslant \mathtt{y} + \mathtt{0}$ (whose renaming is $\mathtt{sumleq\ x\ y}$) succeeds in the original program:

$$\mathtt{id} \,\|\, \mathtt{sumleq\ x\ y} \xrightarrow{\mathtt{RN}} \{\mathtt{x} = \mathtt{0}\} \,\|\, \mathtt{True} \vee \ldots$$

whereas it suspends using the specialized rules:

$$\mathtt{id} \,\|\, \mathtt{x} \leqslant \mathtt{y} + \mathtt{0} \xrightarrow{\mathtt{RN}} \mathtt{id}^{\mathtt{s}} \,\|\, \mathtt{x} \leqslant \mathtt{y} + \mathtt{0}$$

Informally, the annotation *flex* for the specialized function is not safe since the bindings for variable $\mathtt{y}$ in the left-hand side of the second and third resultants have been brought by the evaluation of the rigid function "+". Similarly, the annotation *rigid* does not work since (at runtime) it prevents the considered call from matching the left-hand side of the first resultant because variable $\mathtt{x}$ was instantiated by evaluating (at partial evaluation time) the flexible function "$\leqslant$".

## 4.4 Auxiliary Functions for Partial Evaluation

Our proposed solution is essentially as follows. We distinguish between two kinds of computations: those in which the initial step for the considered expression is incomplete and those which involve no kind of suspension (because they are eventually stopped before). In the latter case, we simply use the $\xrightarrow{\mathtt{RN}}$ computation model whereas, in the former case, we proceed to complete the degenerate step by using the relaxed relation $\xrightarrow{\mathtt{NN}}$ . This allows us to infer safe evaluation annotations for specialized definitions as follows:

- We annotate as *flex* the specialized definitions which result from computations with no suspension. This is justified by the fact that all variable bindings propagated to the left-hand sides of specialized rules come from the evaluation of flexible functions (since evaluation of rigid functions causes no binding for goal variables). Thus, the handling of these specialized functions as flexible cannot introduce (at runtime) undesirable bindings.

- In the case of a suspension, we are constrained to split resultants by introducing several intermediate functions with befitting evaluation annotations. This is necessary because the $\stackrel{\mathtt{NN}}{\longrightarrow}$ step can introduce bindings which come both from flexible and rigid functions (as shown in Example 10) and the splitting avoids the mixing of bindings of different nature (*flex* and *rigid*).

Formally, a partial evaluation based on the $\stackrel{\mathtt{RN}}{\longrightarrow}$ calculus (RN-PE for short) is constructed from a set of terms $S$ together with a set of (partial) computations for the terms in $S$. In the following, we denote by $\Sigma_{\mathtt{inter}}$ a set of fresh function symbols. These are the symbols which are used to construct the intermediate functions associated to the partial evaluation of suspended expressions.

**Definition 29 (partial evaluation)** *Let $\mathcal{R}$ be a program, $S = \{s_1, \ldots, s_n\}$ a finite set of terms, and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ finite (possibly partial) $\stackrel{\mathtt{RN}}{\longrightarrow}$ computations for $s_1, \ldots, s_n$ in $\mathcal{R}$ of the form:*

$$\mathcal{A}_k = id \,[\![\, s_k \stackrel{\mathtt{RN}}{\longrightarrow}^{+} D_k, \quad k = 1, \ldots, n$$

*where all steps are complete, except (possibly) for the initial one. Let $\rho$ be an independent renaming of $S$. Then, the set of rewrite rules $\mathcal{R}' =$*

$$\{\sigma^c(\rho(s_k)) \to ren_\rho(r) \mid \sigma^c \,[\![\, r \in D_k\}_{k=1}^n \qquad \textit{(non-suspension)}$$
$$\bigcup$$
$$\{split(\rho(s_k), r, slist(\theta \circ \sigma)) \mid \theta^i \,[\![\, \theta^i(s_k) \in D_k,\ \theta^i(s_k) \stackrel{\mathtt{NN}}{\leadsto}_\sigma r\}_{k=1}^n \quad \textit{(suspension)}$$

*is a* partial evaluation of $S$ in $\mathcal{R}$ (under $\rho$). *The evaluation annotation for the derivations involving no suspension is $flex$, whereas the resultants (and their evaluation annotations) for the suspended derivations are computed by means of the auxiliary functions shown in Figure 4.3.* [4]

Roughly speaking, the resultants associated to (one-step) $\stackrel{\mathtt{NN}}{\longrightarrow}$ computations are split into a set of "intermediate" rules, one rule associated to each sequence of consecutive bindings with the same superscript mark (suspended or non-suspended). This way, the specialized rules mimic the behaviour of the original functions perfectly. Note that the intermediate rules play no particular role in the evaluation of expressions, but are only necessary to preserve the flex or rigid nature of the functions in the initial program. The following example shows the construction of a RN-PE for a suspended expression.

**Example 11** Let us consider the following rules:

```
f A B  =  C    % flex
g B C  =  B    % rigid
  h C  =  C    % flex
```

[4] In the definition of $slist(\sigma)$ we consider that $\sigma$ is expressed in its canonical representation.

$$
\begin{array}{lcl}
slist(id) & = & [\,] \\
slist(\varphi_k \circ \cdots \circ \varphi_1) & = & [\theta^m | slist(\varphi_k \circ \cdots \circ \varphi_{j+1})] \\
 & & \textbf{where } \theta = \varphi_j \circ \cdots \circ \varphi_1, m = eval(\varphi_1), \text{ and } j \text{ is the} \\
 & & \text{maximum } i \in \{1, \ldots, k\} \text{ such that } \forall p \in \{1, \ldots, i\} \\
 & & eval(\varphi_p) = eval(\varphi_1) \text{ and } eval(\varphi_{j+1}) \neq eval(\varphi_1)
\end{array}
$$

$$
eval(\varphi) = \begin{cases} rigid & \text{if } \varphi \text{ is marked with the superscript } s \\ flex & \text{otherwise} \end{cases}
$$

$$
\begin{array}{l}
split(l, r, [\varphi^a]) = \{\varphi^a(l) \to ren_\rho(r), \text{ with evaluation annotation } a\} \\
split(l, r, [\varphi^a, \theta^b | tail]) = \{\varphi^a(l) \to l', \text{ with eval. annotation } a\} \cup split(l', r, [\theta^b | tail]) \\
\qquad \textbf{where } l' = f(x_1, \ldots, x_n),\ f \in \Sigma_{\texttt{inter}} \text{ is a fresh function} \\
\qquad \text{symbol, and } \mathcal{V}ar(\varphi^a(l)) = \{x_1, \ldots, x_n\}.
\end{array}
$$

Figure 4.3: Auxiliary functions for partial evaluation

A partial evaluation for `f x (g y (h z))` constructed from the (suspended) derivation

$$\texttt{f x (g y (h z))} \overset{\texttt{RN}}{\leadsto}_{\{\texttt{x}\mapsto\texttt{A}\}^s} \texttt{f A (g y (h z))}$$

proceeds as follows (here we assume that $\rho(\texttt{f x (g y (h z))}) = \texttt{f' x y z}$):

1. First, the $\overset{\texttt{NN}}{\leadsto}$ step

   $$\texttt{f A (g y (h z))} \overset{\texttt{NN}}{\leadsto}_{\{\texttt{y}\mapsto\texttt{B},\texttt{z}\mapsto\texttt{C}\}} \texttt{f A (g B C)}$$

   is computed.

2. Then, the call $slist(\{\texttt{x} \mapsto \texttt{A}, \texttt{y} \mapsto \texttt{B}, \texttt{z} \mapsto \texttt{C}\})$ is undertaken, which returns the set of substitutions $[\{\texttt{x} \mapsto \texttt{A}\}^{\texttt{flex}}, \{\texttt{y} \mapsto \texttt{B}\}^{\texttt{rigid}}, \{\texttt{z} \mapsto \texttt{C}\}^{\texttt{flex}}]$.

3. Finally, the computation of *split* proceeds as follows:

   $$
   \begin{array}{l}
   split(\texttt{f' x y z}, \texttt{f A (g B C)}, [\{\texttt{x} \mapsto \texttt{A}\}^{\texttt{flex}}, \{\texttt{y} \mapsto \texttt{B}\}^{\texttt{rigid}}, \{\texttt{z} \mapsto \texttt{C}\}^{\texttt{flex}}]) \\
   \quad = \{\texttt{f' A y z} = \texttt{f}'_1 \texttt{ y z}\} \\
   \qquad \cup\, split(\texttt{f}'_1 \texttt{ y z}, \texttt{f A (g B C)}, [\{\texttt{y} \mapsto \texttt{B}\}^{\texttt{rigid}}, \{\texttt{z} \mapsto \texttt{c}\}^{\texttt{flex}}]) \\
   \quad = \{\texttt{f' A y z} = \texttt{f}'_1 \texttt{ y z}, \\
   \qquad \texttt{f}'_1 \texttt{ B z} = \texttt{f}'_2 \texttt{ z}\} \\
   \qquad \cup\, split(\texttt{f}'_2 \texttt{ z}, \texttt{f A (g B C)}, [\{\texttt{z} \mapsto \texttt{C}\}^{\texttt{flex}}]) \\
   \quad = \{\texttt{f' A y z} = \texttt{f}'_1 \texttt{ y z}, \\
   \qquad \texttt{f}'_1 \texttt{ B z} = \texttt{f}'_2 \texttt{ z}, \\
   \qquad \texttt{f}'_2 \texttt{ C} = \texttt{ren}_\rho(\texttt{f A (g B C)})\}
   \end{array}
   $$

   where "$\texttt{f}'$" and "$\texttt{f}'_2$" are flexible, and "$\texttt{f}'_1$" is rigid.

As discussed in Chapter 3, a general requirement in the partial evaluation of lazy functional logic programs is that no constructor-rooted expression can be evaluated during partial evaluation. This is also true in our context. We refer to Chapter 3 for a detailed explanation on this limitation; also, Example 3 illustrates why this restriction cannot be dropped.

## 4.5 Preservation of Inductive Sequentiality

In this section, we demonstrate an important property of RN-PE: if the input program is inductively sequential, then the specialized program is also inductively sequential. The proof relies on several properties of definitional trees [Antoy, 1992], which we enumerate below. In contrast to the original definition (see Definition 3), here we use the "declarative" definition [Antoy, 1997] since it is more appropriate for proving the results about RN-PE. In particular, a definitional tree is seen now as a set of linear patterns $\mathcal{P}$ associated to the set of linear patterns $S$ corresponding to the left-hand sides of the rules defining a function.

A definitional tree of a finite set of linear patterns $S$ is a non-empty set $\mathcal{P}$ of linear patterns partially ordered by subsumption having the following properties:

*Root property:* There is a minimum element $pattern(\mathcal{P})$, also called the *pattern* of the definitional tree.

*Leaves property:* The maximal elements, called the *leaves*, are the elements of $S$. Non-maximal elements are also called *branches*.

*Parent property:* If $\pi \in \mathcal{P}$, $\pi \neq pattern(\mathcal{P})$, there exists a unique $\pi' \in \mathcal{P}$, called the *parent* of $\pi$ (and $\pi$ is called a *child* of $\pi'$), such that $\pi' < \pi$ and there is no other pattern $\pi'' \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ with $\pi' < \pi'' < \pi$.

*Induction property:* Given $\pi \in \mathcal{P} \backslash S$, there is a position $o$ in $\pi$ with $\pi|_o \in \mathcal{X}$ (called the *inductive position*), and constructors $c_1/k_1, \ldots, c_n/k_n \in \mathcal{C}$ with $c_i \neq c_j$ for $i \neq j$, such that, for all $\pi_1, \ldots, \pi_n$ which have the parent $\pi$, $\pi_i = \pi[c_i(\overline{x_{k_i}})]_o$ (where $\overline{x_{k_i}}$ are new distinct variables) for all $1 \leq i \leq n$.

An inductively sequential TRS can be viewed as a set of definitional trees, each defining a function symbol. There can be more than one definitional for an inductively sequential function. In the following, a fixed definitional tree for each defined function is assumed.

First, let us state the following proposition, which essentially shows that each substitution in a $\xrightarrow{\texttt{RN}}$ step instantiates only variables occurring in the initial term.

**Proposition 30** *If $\varphi_k \circ \cdots \circ \varphi_1 \,[\!]\, t' \in cst(t, \mathcal{T})$ is the canonical representation of a $\stackrel{\mathtt{RN}}{\longrightarrow}$ computation step, then, for $i = 1, \ldots, k$, $\varphi_i = id$ or $\varphi_i = \{x \mapsto c(\overline{x_n})\}$ (where $\overline{x_n}$ are pairwise different variables) with $x \in \mathcal{V}ar(\varphi_{i-1} \circ \cdots \circ \varphi_1(t))$.*

*Proof.* By induction on $k$. □

The following lemma shows that, for different $\stackrel{\mathtt{RN}}{\longrightarrow}$ steps (computing different substitutions), there is always a variable which is instantiated to different constructors.

**Lemma 31** *Let $t$ be an operation-rooted term, $\mathcal{T}$ an associated definitional tree and $\{(\varphi_k \circ \cdots \circ \varphi_1 \,[\!]\, t_1),\ (\varphi'_{k'} \circ \cdots \circ \varphi'_1 \,[\!]\, t_2)\} \subseteq cst(t, \mathcal{T})$, $k \leq k'$. Then, for all $i \in \{1, \ldots, k\}$,*

- *either $\varphi_i \circ \cdots \circ \varphi_1 = \varphi'_i \circ \cdots \circ \varphi'_1$, or*
- *there exists some $j < i$ with*
  1. *$\varphi_j \circ \cdots \circ \varphi_1 = \varphi'_j \circ \cdots \circ \varphi'_1$, and*
  2. *$\varphi_{j+1} = \{x \mapsto c(\cdots)\}$ and $\varphi'_{j+1} = \{x \mapsto c'(\cdots)\}$ with $c \neq c'$.*

*Proof.* By induction on $k$ (the number of recursive steps performed by $cst$ to compute $\varphi_k \circ \cdots \circ \varphi_1 \,[\!]\, t_1$):

**$k = 1$:** Then $\mathcal{T} = rule(l \to r)$ and $cst(t, \mathcal{T}) = \{id \,[\!]\, \sigma(r)\}$. Thus, the lemma trivially holds by vacuity.

**$k > 1$:** Then $\mathcal{T} = branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_n)$. We prove the induction step by a case distinction on the form of the subterm $t|_o$:

**$t|_o = x \in \mathcal{X}$:** Here we distinguish two cases:

1. $r = rigid$. Then $cst(t, \mathcal{T}) = \{id^s \,[\!]\, t\}$ and the lemma trivially holds by vacuity.
2. $r = flex$. Then $\varphi_1 = \{x \mapsto c_i(\overline{x_n})\}$ and $\varphi_k \circ \cdots \circ \varphi_2 \,[\!]\, \varphi_1(t) \in cst(\varphi_1(t), \mathcal{T}_i)$ for some $i$. If $\varphi'_1 = \{x \mapsto c(\cdots)\}$ with $c \neq c_i$, then the proposition directly holds. Otherwise, if $\varphi_1 = \varphi'_1$, the proposition follows from the induction hypothesis applied to

$$cst(\varphi_1(t), \mathcal{T}_i) = (\varphi_k \circ \cdots \circ \varphi_2 \,[\!]\, \varphi_1(t)) \vee (\varphi'_{k'} \circ \cdots \circ \varphi'_2 \,[\!]\, \varphi_1(t)) \vee D'$$

where $D'$ is a (possibly empty) set of disjunctive expressions.

**$t|_o = c_i(\overline{t_n})$:** Then $\varphi_1 = id$ and $\varphi_k \circ \cdots \circ \varphi_2 \,[\!]\, t \in cst(t, \mathcal{T}_i)$. Clearly, $\varphi'_1 = id$ by definition of $cst$. Hence the proposition follows from the induction hypothesis applied to

$$cst(\varphi_1(t), \mathcal{T}_i) = (\varphi_k \circ \cdots \circ \varphi_2 \,[\!]\, \varphi_1(t)) \vee (\varphi'_{k'} \circ \cdots \circ \varphi'_2 \,[\!]\, \varphi_1(t)) \vee D'$$

where $D'$ is a (possibly empty) set of disjunctive expressions.

$t|_o = f(\overline{t_n})$**:** Then $\varphi_1 = id$ and $\varphi_k \circ \cdots \circ \varphi_2 \, [\![ \, t \in cst(t, \mathcal{T}')$ where $\mathcal{T}'$ is a definitional tree for $f$. By definition of $cst$, $\varphi'_1 = id$. Then the proposition follows from the induction hypothesis applied to

$$cst(\varphi_1(t), \mathcal{T}_i) = (\varphi_k \circ \cdots \circ \varphi_2 \, [\![ \, \varphi_1(t)) \vee (\varphi'_{k'} \circ \cdots \circ \varphi'_2 \, [\![ \, \varphi_1(t)) \vee D'$$

where $D'$ is a (possibly empty) set of disjunctive expressions.

□

The following theorem states the preservation of the (inductively sequential) structure of programs during the partial evaluation process. Firstly, this is only proved for partial evaluations w.r.t. linear patterns but, then, we extend this result to arbitrary sets of terms.

**Theorem 32** *Let $\mathcal{R}$ be an inductively sequential program and $t$ be a linear pattern. If $t \overset{\text{RN}}{\rightsquigarrow}^{+}_{\sigma_i} t_i$, $i = 1, \ldots, n$, are all the $\overset{\text{RN}}{\rightsquigarrow}$ (partial) derivations for $t$ ending in a non-failing leaf in $\mathcal{R}$, then the following set of rules $\mathcal{R}'$ which belong to the RN-PE of $\mathcal{R}$ w.r.t. $t$ (before renaming):* [5]

$$\begin{array}{l} \sigma_1(t) \to r_1 \\ \vdots \\ \sigma_n(t) \to r_2 \end{array}$$

*are inductively sequential, where $r_i = t_i$ or $r_i = f(x_1, \ldots, x_n)$ and $f \in \Sigma_{\mathtt{inter}}$, $\mathcal{V}ar(\sigma_i(t)) = \{x_1, \ldots, x_n\}$.*

*Proof.* To show the inductive sequentiality of $\mathcal{R}'$, it suffices to show that there exists a definitional tree for the set $S = \{\sigma_1(t), \ldots, \sigma_n(t)\}$ with pattern $f(\overline{x_p})$ if $t$ has the $p$-ary function $f$ at the root. We prove this property by induction on the number of inner nodes of the narrowing tree for $t$.

**Base case:** If the number of inner nodes is 1, we first construct a definitional tree for the set $S = \{t\}$ containing only the pattern at the root of the narrowing tree. This is always possible by Proposition 4. Now we construct a definitional tree for the sons of the root by extending this initial definitional tree. This construction is identical to the induction step.

**Induction step:** Assume that $s$ is a leaf in the narrowing tree, $\sigma$ is the accumulated substitution from the root to this leaf, and $\mathcal{P}$ is a definitional tree for the set

$$S = \{\theta(t) \mid t \overset{\text{RN}}{\rightsquigarrow}^{+}_{\theta} s' \text{ is a } \overset{\text{RN}}{\longrightarrow} \text{ derivation with a non-failing leaf } s'\}.$$

[5] By abuse, we consider the RN-PE before performing the post-processing renaming. In Theorem 34, we extend this result to consider a RN-PE as defined in Definition 29.

Now we apply a $\overset{\text{RN}}{\longrightarrow}$ step to $s$, i.e., let

$$
\begin{array}{l}
s \overset{\text{RN}}{\leadsto}_{\varphi_1} s_1 \\
\vdots \\
s \overset{\text{RN}}{\leadsto}_{\varphi_m} s_m
\end{array}
$$

be all $\overset{\text{RN}}{\leadsto}$ steps for $s$. For the induction step, it is sufficient to show that there exists a definitional tree for

$$S' = (S \backslash \{\sigma(t)\}) \cup \{\varphi_1(\sigma(t)), \ldots, \varphi_m(\sigma(t))\} \ .$$

Consider for each step $s \overset{\text{RN}}{\leadsto}_{\varphi_i} s_i$ the associated canonical representation $(p, R, \varphi_{ik_i} \circ \cdots \circ \varphi_{i1}) \in cst(s, \mathcal{P}_s)$ (where $\mathcal{P}_s$ is a definitional tree for the root of $s$). Let

$$\mathcal{P}' = \mathcal{P} \cup \{\varphi_{ij} \circ \cdots \circ \varphi_{i1} \circ \sigma(t) \mid 1 \leq i \leq m, 1 \leq j \leq k_i\}$$

We prove that $\mathcal{P}'$ is a definitional tree for $S'$ by showing that each of the four properties of a definitional tree holds for $\mathcal{P}'$.

*Root property:* The minimum elements are identical for both definitional trees, i.e., $pattern(\mathcal{P}) = pattern(\mathcal{P}')$, since only instances of a leaf of $\mathcal{P}$ are added in $\mathcal{P}'$.

*Leaves property:* First, we know that the maximal elements of $\mathcal{P}$ are $S$. Since all substitutions computed by the $\overset{\text{RN}}{\longrightarrow}$ calculus along different derivations are independent (Lemma 31), $\sigma$ is independent to all other substitutions occurring in $S$ and the substitutions $\varphi_1, \ldots, \varphi_m$ are pairwise independent. Thus, the replacement of the element $\sigma(t)$ in $S$ by the set $\{\varphi_1(\sigma(t)), \ldots, \varphi_m(\sigma(t))\}$ does not introduce any comparable (w.r.t. the subsumption ordering) terms. This implies that $S'$ is the set of maximal elements of $\mathcal{P}'$.

*Parent property:* Let $\pi \in \mathcal{P}' \backslash \{pattern(\mathcal{P}')\}$. We consider two cases for $\pi$:

1. $\pi \in \mathcal{P}$: Then the parent property trivially holds since only instances of a leaf of $\mathcal{P}$ are added in $\mathcal{P}'$.
2. $\pi \notin \mathcal{P}$: By definition of $\mathcal{P}'$, $\pi = \varphi_{ij} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$ for some $1 \leq i \leq m$ and $1 \leq j \leq k_i$. We show by induction on $j$ that the parent property holds for $\pi$.
   Base case ($j = 1$): Then $\pi = \varphi_{i1}(\sigma(t))$. We have $\varphi_{i1} \neq id$ (otherwise $\pi = \sigma(t) \in \mathcal{P}$). Thus, by Proposition 30, $\varphi_{i1} = \{x \mapsto c(\overline{x_n})\}$ with $x \in \mathcal{V}ar(s) \subseteq \mathcal{V}ar(\sigma(t))$. Due to the linearity of the initial pattern and all substituted terms (cf. Proposition 30), $\sigma(t)$ has a single occurrence $o$ of variable $x$ and, therefore, $\pi = \sigma(t)[c(\overline{x_n})]_o$, i.e., $\sigma(t)$ is the unique parent of $\pi$.

Induction step ($j > 1$): We assume that the parent property holds for $\pi' = \varphi_{i,j-1} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$. Let $\varphi_{ij} \neq id$ (otherwise the induction step is trivial). By Proposition 30, $\varphi_{ij} = \{x \mapsto c(\overline{x_n})\}$ with $x \in \mathcal{V}ar(\varphi_{i,j-1} \circ \cdots \circ \varphi_{i1} \circ \sigma(t))$ (since $\mathcal{V}ar(s) \subseteq \mathcal{V}ar(\sigma(t))$). Now we proceed as in the base case to show that $\pi'$ is the unique parent of $\pi$.

*Induction property:* Let $\pi \in \mathcal{P}' \backslash S'$. We consider two cases for $\pi$:

1. $\pi \in \mathcal{P} \backslash \{\sigma(t)\}$: Then the induction property holds for $\pi$ since it already holds in $\mathcal{P}$ and only instances of $\sigma(t)$ are added in $\mathcal{P}'$.
2. $\pi = \varphi_{ij} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$ for some $1 \leq i \leq m$ and $0 \leq j < k_i$. Assume $\varphi_{i,j+1} \neq id$ (otherwise, do the identical proof with the representation $\pi = \varphi_{i,j+1} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$). By Proposition 30, $\varphi_{i,j+1} = \{x \mapsto c(\overline{x_n})\}$ and $\pi$ has a single occurrence of variable $x$ (due to the linearity of the initial pattern and all substituted terms). Therefore, $\pi' = \varphi_{i,j+1} \circ \cdots \circ \varphi_{i1} \circ \sigma(t)$ is a child of $\pi$. Consider another child $\pi'' = \varphi_{i'j'} \circ \cdots \circ \varphi_{i'1} \circ \sigma(t)$ of $\pi$ (other patterns in $\mathcal{P}'$ cannot be children of $\pi$ due to the induction property for $\mathcal{P}$). Assume $\varphi_{i'j'} \circ \cdots \circ \varphi_{i'1} \neq \varphi_{i,j+1} \circ \cdots \circ \varphi_{i1}$ (otherwise, both children are identical). By Lemma 31, there exists some $l$ with $\varphi_{i'l} \circ \cdots \circ \varphi_{i'1} = \varphi_{il} \circ \cdots \circ \varphi_{i1}$, $\varphi_{i',l+1} = \{x' \mapsto c'(\cdots)\}$, and $\varphi_{i,l+1} = \{x' \mapsto c''(\cdots)\}$ with $c' \neq c''$. Since $\pi''$ and $\pi'$ are children of $\pi$ (i.e., immediate successors w.r.t. the subsumption ordering) it must be $x' = x$ (otherwise, $\pi'$ differs from $\pi$ at more than one position) and $\varphi_{i',j'} = \cdots = \varphi_{i',l+2} = id$ (otherwise, $\pi''$ differs from $\pi$ at more than one position). Thus, $\pi'$ and $\pi''$ differ only in the instantiation of variable $x$ which has exactly one occurrence in their common parent $\pi$, i.e., there is a position $o$ of $\pi$ with $\pi|_o = x$ and $\pi' = \pi[c'(\overline{x_{n'_i}})]_o$ and $\pi'' = \pi[c''(\overline{x_{n''_i}})]_o$. Since $\pi''$ was an arbitrary child of $\pi$, the induction property holds.

□

Since partial evaluation is usually initiated with more than one term, we extend the previous theorem to this more general case.

**Corollary 33** *Let $\mathcal{R}$ be an inductively sequential program and $S$ be a finite set of linear patterns with pairwise different root symbols. Let us consider the following set of rules associated to the $\overset{\mathtt{RN}}{\leadsto}$ (partial) derivations for $s \in S$ in $\mathcal{R}$ ending in a non-failing leaf $s \overset{\mathtt{RN}}{\leadsto}^{+}_{\sigma_i} t_i$, $i = 1, \ldots, n$, which belong to the RN-PE of $\mathcal{R}$ w.r.t. $s$ (before renaming):*

$$\sigma_1(s) \to r_1$$
$$\vdots$$
$$\sigma_n(s) \to r_2$$

*where* $r_i = t_i$ *or* $r_i = f(x_1, \ldots, x_n)$ *and* $f \in \Sigma_{\mathtt{inter}}$, $\mathcal{V}ar(\sigma_i(s)) = \{x_1, \ldots, x_n\}$. *Then, the union of the sets of rules associated to each term* $s \in S$ *is inductively sequential.*

*Proof.* This is a consequence of Theorem 32 since we can construct a definitional tree for each partial evaluation of a pattern of $S$. Since all patterns have different root symbols, the roots of these definitional trees do not overlap. □

Now we are able to show that the partial evaluation of an arbitrary set of terms w.r.t. an inductively sequential program always produces an inductively sequential program.

**Theorem 34** *Let* $\mathcal{R}$ *be an inductively sequential program,* $S$ *a finite set of operation-rooted terms,* $\rho$ *an independent renaming of* $S$*, and* $s \in S$ *an operation-rooted term. Let us consider the following set of rules associated to the* $\overset{\mathtt{RN}}{\leadsto}$ *(partial) derivations for* $s \in S$ *in* $\mathcal{R}$ *ending in a non-failing leaf* $s \overset{\mathtt{RN}}{\leadsto}{}^{+}_{\sigma_i} t_i$, $i = 1, \ldots, n$ *which belong to the RN-PE of* $\mathcal{R}$ *w.r.t.* $s$*:*

$$\sigma_1(\rho(s)) \to r_1$$
$$\vdots$$
$$\sigma_n(\rho(s)) \to r_2$$

*where* $r_i = ren_\rho(t_i)$ *or* $r_i = f(\overline{x_n})$ *and* $f \in \Sigma_{\mathtt{inter}}$, $\mathcal{V}ar(\sigma_i(\rho(s))) = \{x_1, \ldots, x_n\}$. *Then, the union of the sets of rules associated to each term* $s \in S$ *is inductively sequential.*

*Proof.* Let $\mathcal{R}'$ be the above RN-PE of $\mathcal{R}$ w.r.t. $S$ (before renaming) and let $\rho$ be an independent renaming of $S$. Then each rule of a RN-PE $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. $S$ has the form:

$$\rho(\sigma_i(s)) \to ren_\rho(r_i) \qquad \textit{or} \qquad \rho(\sigma_i(s)) \to f(x_1, \ldots, x_n)$$

where $f \in \Sigma_{\mathtt{inter}}$, $\mathcal{V}ar(\sigma_i(\rho(s))) = \{x_1, \ldots, x_n\}$ for some rule in $\mathcal{R}'$

$$\sigma_i(s) \to r_i \qquad \textit{or} \qquad \sigma_i(s) \to f(x_1, \ldots, x_n)$$

respectively. Consider the extended rewrite system

$$\mathcal{R}_\rho = \mathcal{R} \cup \{\rho(s) \to s \mid s \in S\}$$

where the renaming $\rho$ is encoded by a set of rewrite rules. Note that $\mathcal{R}_\rho$ is inductively sequential since the new left-hand sides $\rho(s)$ are of the form $f_s(\overline{x_n})$ with new function symbols $f_s$.

Let $\mathcal{R}'_\rho$ be an arbitrary set of rules which belong the RN-PE of $\mathcal{R}_\rho$ w.r.t. $\rho(S)$ (before renaming). Since $\rho(S)$ is a set of linear patterns with pairwise different root symbols, $\mathcal{R}'_\rho$ is inductively sequential by Corollary 33. It is obvious that each subset of an inductively sequential program is also inductively sequential (since only the left-hand sides of the rules are relevant for this property). Therefore, to complete the proof it is sufficient to show that all left-hand sides of rules from $\mathcal{R}''$ can also occur as left-hand sides in some $\mathcal{R}'_\rho$.

Each rule of $\mathcal{R}''$ has the form $\theta(\rho(s)) \to ren_\rho(r)$ for some rule $\theta(s) \to r \in \mathcal{R}'$. By definition of $\mathcal{R}'$, there exists a $\stackrel{\mathtt{RN}}{\longrightarrow}$ derivation $s \stackrel{\mathtt{RN}}{\leadsto}{}^{+}_{\theta}\ r$ w.r.t. $\mathcal{R}$. Hence,

$$\rho(s)\ \stackrel{\mathtt{RN}}{\leadsto}_{id}\ s\ \stackrel{\mathtt{RN}}{\leadsto}{}^{+}_{\theta}\ r$$

is a $\stackrel{\mathtt{RN}}{\longrightarrow}$ derivation w.r.t. $\mathcal{R}_\rho$. Thus, $\theta(\rho(s)) \to r$ is a resultant which can occur in some $\mathcal{R}'_\rho$. □

Now, we proceed with the proof of the inductive sequentiality of specialized programs.

**Theorem 35** *Let $\mathcal{R}$ be an inductively sequential program, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming for the terms in $S$. Then each RN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) is inductively sequential.*

*Proof.* Let us consider an arbitrary RN-PE $\mathcal{R}'$ of $\mathcal{R}$ w.r.t. $S$ under $\rho$. Since the specialized rules of $\mathcal{R}'$ associated to each term in $S$ define different (renamed) function symbols, we only prove the claim for the specialized rules produced from an arbitrary term $s \in S$. Let us consider that the specialized rules associated to the term $s \in S$ have been constructed from the derivations

$$s \stackrel{\mathtt{RN}}{\leadsto}{}^{+}_{\sigma_i}\ t_i \stackrel{\mathtt{NN}}{\leadsto}{}^{*}\ t'_i, \quad i = 1, \ldots, n$$

By definition of RN-PE, the rules defining in $\mathcal{R}'$ the outermost function symbol of $\rho(s)$ are the resultants associated to the $\stackrel{\mathtt{RN}}{\leadsto}$ derivations $s \stackrel{\mathtt{RN}}{\leadsto}{}^{+}_{\sigma_i}\ t_i$, $i = 1, \ldots, n$:

$$\begin{array}{rcl} \sigma_1(\rho(s)) & \to & r_1 \\ \sigma_2(\rho(s)) & \to & r_2 \\ & \ldots & \\ \sigma_n(\rho(s)) & \to & r_n \end{array}$$

where each $r_i$ is of the form:

- $r_i = ren_\rho(t_i)$, if the $\stackrel{\mathtt{RN}}{\leadsto}$ steps of $s \stackrel{\mathtt{RN}}{\leadsto}{}^{+}_{\sigma_i}\ t_i$ are all complete, or
- $r_i = f(x_1, \ldots, x_k)$, with $f \in \Sigma_{\mathtt{inter}}$ and $\mathcal{V}ar(t_i) = \{x_1, \ldots, x_k\}$, if the derivation $s \stackrel{\mathtt{RN}}{\leadsto}{}^{+}_{\sigma_i}\ t_i$ consists of a single, incomplete $\stackrel{\mathtt{RN}}{\leadsto}$ step.

Now, by Theorem 32, the above rules defining the root symbol of $\rho(s)$ are inductively sequential.

In the remaining, we prove that the rules defining the intermediate function symbols of $\Sigma_{\mathtt{inter}}$ are also inductively sequential. Let us consider a derivation

$$s \overset{\mathtt{RN}}{\leadsto}_{\sigma_j} t_j \overset{\mathtt{NN}}{\leadsto}_{\theta_j} t'_j, \quad 1 \leq j \leq n$$

such that the $\overset{\mathtt{RN}}{\leadsto}$ step is incomplete (and thus $t_j = \sigma_j(s)$). By definition of RN-PE, $\mathcal{R}'$ contains a set of intermediate rules of the form

$$\begin{array}{rcl} \sigma_j(\rho(s)) & \to & f_1(x_{1,1}, \ldots, x_{1,k_1}) \\ \varphi_1(f_1(x_{1,1}, \ldots, x_{1,k_1})) & \to & f_2(x_{2,1}, \ldots, x_{2,k_2}) \\ & \ldots & \\ \varphi_m(f_m(x_{m-1,1}, \ldots, x_{m-1,k_{m-1}})) & \to & ren_\rho(t'_j) \end{array}$$

where $slist(\theta_j \circ \sigma_j) = [\sigma_j^a, \varphi_1^{a_1}, \ldots, \varphi_m^{a_m}]$ and $f_1, \ldots, f_m \in \Sigma_{\mathtt{inter}}$ are different fresh function symbols. By construction, the terms $f_i(x_{i,1}, \ldots, x_{i,k_i})$, $1 \leq i \leq m$, are linear patterns. Trivially, since $\varphi_i$ are constructor substitutions, each $\varphi_i(f_i(x_{i,1}, \ldots, x_{i,k_i}))$ is also a linear pattern, $1 \leq i \leq m$. Finally, since each intermediate rule is defined by a single rule, its inductive sequentiality follows from Proposition 4, which completes the proof. $\square$

## 4.6 Correctness

For the correctness of our partial evaluation framework, we first extend the *closedness* condition of Section 3.2 to the case of residuating programs.

**Definition 36 (closedness)** *Let $S$ be a finite set of terms. We say that a term $t$ is $S$-closed if closed$(S,t)$ holds, where the predicate closed is defined inductively as follows:*

$$closed(S,t) \Leftrightarrow \begin{cases} true & \text{if } t \in \mathcal{X} \\ \bigwedge_{i=1,\ldots,n} closed(S,t_i) & \text{if } t = c(\overline{t_n}),\ c \in (\mathcal{C} \cup \mathcal{P} \cup \Sigma_{\mathtt{inter}}) \\ \bigwedge_{x \mapsto t' \in \theta} closed(S,t') & \text{if } \exists \theta, \exists s \in S \text{ such that } \theta(s) = t \end{cases}$$

*We say that a set of terms $T$ is $S$-closed, written closed$(S,T)$, if closed$(S,t)$ holds for all $t \in T$, and we say that a TRS $\mathcal{R}$ is $S$-closed if closed$(S, \mathcal{R}_{calls})$ holds.*

Note that expressions rooted by an "intermediate" function symbol in $\Sigma_{\mathtt{inter}}$ are $S$-closed by definition, independently of the considered set $S$. This is motivated by the fact that intermediate functions are not "visible" in the specialized program (i.e., they do not belong to the set of specialized calls), but are only intended as a mechanism to preserve the floundering behaviour.

The following result establishes the precise relation between partial evaluations based on needed narrowing, as defined in Chapter 3, and partial evaluations with $\stackrel{\text{RN}}{\longrightarrow}$, as defined here. Intuitively, any RN-PE $\mathcal{R}'$ can be transformed into an equivalent program $\mathcal{R}''$ (w.r.t. needed narrowing) by replacing each set of rules

$$\begin{array}{rcl} \sigma(\rho(s)) & \rightarrow & f_1(\overline{x_{m_1}}) \\ \varphi_1(f_1(\overline{x_{m_1}})) & \rightarrow & f_2(\overline{x_{m_2}}) \\ & \ldots & \\ \varphi_k(f_k(\overline{x_{m_k}})) & \rightarrow & ren_\rho(r) \end{array}$$

associated to a suspended expression, by the new rule

$$\theta(\rho(s)) \rightarrow ren_\rho(r), \quad \text{with } \theta = \varphi_k \circ \cdots \circ \varphi_1 \circ \sigma$$

and ignoring all the evaluation annotations. The program constructed in this way is a correct NN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$), as formalized in the following.

**Theorem 37** *Let $\mathcal{R}$ be an inductively sequential program. Let $S$ be a finite set of operation-rooted terms and $\rho$ an independent renaming of $S$. If $\mathcal{R}'$ is a RN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$), then there exists a NN-PE $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that, for all goals $e$ which do not contain fresh function symbols from $\Sigma_{\mathtt{inter}}$, we have $e \stackrel{\text{NN}^*}{\leadsto}_\sigma true$ in $\mathcal{R}'$ iff $e \stackrel{\text{NN}^*}{\leadsto}_\sigma true$ in $\mathcal{R}''$.*

*Proof.* First, let us consider an arbitrary RN-PE $\mathcal{R}'$ of $\mathcal{R}$ w.r.t. $S$ (under $\rho$). Now, we show how a NN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) can be constructed from $\mathcal{R}'$ which has the same needed narrowing semantics. Moreover, we could consider an arbitrary NN-PE, and obtain the associated RN-PE such that the proof in the other direction follows analogously. For simplicity, we proceed only in the left-to-right direction. We distinguish two cases depending on the way that the residual rules are constructed:

- Let $\sigma(\rho(s)) \rightarrow ren_\rho(r) \in \mathcal{R}'$ be a resultant which has been constructed from a computation $s \stackrel{\text{RN}^*}{\leadsto}_\sigma r$ in which all the steps are complete. By Theorem 28, we have $s \stackrel{\text{NN}^*}{\leadsto}_\sigma r$ in $\mathcal{R}$. Hence, we can consider that $\sigma(\rho(s)) \rightarrow ren_\rho(r)$ belongs to $\mathcal{R}''$ as well.

- Let

$$\begin{array}{rcl} \sigma(\rho(s)) & \rightarrow & f_1(\overline{x_{m_1}}) \\ \varphi_1(f_1(\overline{x_{m_1}})) & \rightarrow & f_2(\overline{x_{m_2}}) \\ & \ldots & \\ \varphi_k(f_k(\overline{x_{m_k}})) & \rightarrow & ren_\rho(r) \end{array}$$

be the set of rules in $\mathcal{R}'$ associated to the (incomplete) $\stackrel{\text{RN}}{\leadsto}$ step $s \stackrel{\text{RN}}{\leadsto}_\sigma \sigma(s)$. By definition of RN-PE, there exists a computation

$$s \stackrel{\text{RN}}{\leadsto}_\sigma \sigma(s) \stackrel{\text{NN}}{\leadsto}_\varphi r$$

with $\varphi = \varphi_k \circ \cdots \circ \varphi_1$. By definition of $\stackrel{\mathtt{RN}}{\longrightarrow}$, it is immediate that $s \stackrel{\mathtt{NN}}{\rightsquigarrow}_\theta r$, with $\theta = \varphi_k \circ \cdots \circ \varphi_1 \circ \sigma$. In this case, we consider that $\theta(\rho(s)) \rightarrow ren_\rho(r)$ belongs to $\mathcal{R}''$.

Now we prove that each derivation $e \stackrel{\mathtt{NN}}{\rightsquigarrow}^*_\theta true$ in $\mathcal{R}'$ can also be proven in $\mathcal{R}''$. We prove this claim by induction on the length $n$ of the former derivation.

$n = 1$. This case is trivial, since the rules for the strict equality belong to both programs.

$n > 1$. If the first step of the needed narrowing derivation $e \stackrel{\mathtt{NN}}{\rightsquigarrow}^*_\theta true$ in $\mathcal{R}'$ does not use a resultant constructed from an incomplete $\stackrel{\mathtt{RN}}{\rightsquigarrow}$ step, then the claim follows trivially by the inductive hypothesis, since the applied rule also belongs to $\mathcal{R}''$. Otherwise, we consider that the above derivation has the form

$$e \stackrel{\mathtt{NN}}{\rightsquigarrow}_{\sigma_1} e_1 \stackrel{\mathtt{NN}}{\rightsquigarrow}_{\sigma_2} \ldots \stackrel{\mathtt{NN}}{\rightsquigarrow}_{\sigma_k} e_k \stackrel{\mathtt{NN}}{\rightsquigarrow}^* true$$

where the first $k$ steps are given with the resultants:

$$\begin{array}{rcl} \sigma(\rho(s)) & \rightarrow & f_1(\overline{x_{m_1}}) \\ \varphi_1(f_1(\overline{x_{m_1}})) & \rightarrow & f_2(\overline{x_{m_2}}) \\ & \ldots & \\ \varphi_k(f_k(\overline{x_{m_k}})) & \rightarrow & ren_\rho(r) \end{array}$$

which are associated to an incomplete derivation of the form:

$$s \stackrel{\mathtt{RN}}{\rightsquigarrow}_\sigma \sigma(s) \stackrel{\mathtt{NN}}{\rightsquigarrow}_{\varphi_k \circ \cdots \circ \varphi_1} r$$

Note that the existence of these $k$ steps (and the fact that they are given consecutively) is an easy consequence of the fact that $f_1, \ldots, f_k$ are fresh function symbols defined by a single rule and that the left-hand sides of $\mathcal{R}'$ are linear patterns. Now, it is immediate that the derivation step $e \stackrel{\mathtt{NN}}{\rightsquigarrow}_\delta e_k$ can be proven in $\mathcal{R}''$ by using the corresponding resultant

$$\varphi_k \circ \ldots \circ \varphi_1 \circ \sigma(\rho(s)) \rightarrow ren_\rho(r)$$

where $\delta = \sigma_k \circ \cdots \circ \sigma_1$. Hence, the claim follows by the inductive hypothesis.

□

Now, we state the partial correctness of RN-PE, which amounts to the full computational equivalence between the original and specialized programs when the considered goal does not flounder.

**Theorem 38 (partial correctness)** *Let $\mathcal{R}$ be an inductively sequential program. Let $e$ be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{R}'$ be a RN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$.*

1. *If $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma'}$ true in $\mathcal{R}'$, then $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma} t$ and $\varphi(t) \to^* true$ in $\mathcal{R}$ with $\sigma' = \varphi \circ \sigma\ [V]$.*
2. *If $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma}$ true in $\mathcal{R}$, then $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma'} t$ and $\varphi(t) \to^* true$ in $\mathcal{R}'$ with $\sigma = \varphi \circ \sigma'\ [V]$.*

*Proof.* Immediate by Theorem 37 and the correctness of NN-PE (Theorem 23). □

Loosely speaking, the previous result establishes that, if evaluation annotations are not considered (that is, no function calls are delayed), then the specialized program $\mathcal{R}'$ *is able* to produce the same answers (computed by needed narrowing) as the original one $\mathcal{R}$ (and vice versa). The preservation of floundering-freeness (i.e., absence of floundering) for the intended goals is needed to establish the total correctness of the transformation. On the other hand, it ensures that the transformation does not introduce additional floundering points, which is of crucial importance when we are using the transformation for optimizing a program. Moreover, this feature may allow us to use the transformation as a tool for proving floundering-freeness of the original program (Example 14 will illustrate this point). In fact, if after the transformation we can state that $\mathcal{R}' \cup \{e'\}$ does not flounder, then we are also sure that $\mathcal{R} \cup \{e\}$ does not flounder either, where $e' = ren_\rho(e)$.

Unfortunately, the recursive notion of closedness introduced in Definition 13 is too weak (generous) to preserve the floundering behaviour, as illustrated by the following example.

**Example 12** Let us consider the following set of rules:

```
f x A  =  g x    % flex        h A  =  B    % rigid
  g B  =  C      % flex
```

A RN-PE of $\{\mathtt{f\ x\ y},\ \mathtt{h\ x}\}$ under $\rho = \{\mathtt{f\ x\ y} \mapsto \mathtt{f'\ x\ y},\ \mathtt{h\ x} \mapsto \mathtt{h'\ x}\}$ is:

```
f' B A  =  C    % flex
  h' A  =  B    % rigid
```

Now, the $S$-closed expression $\mathtt{f\ (h\ x)\ x}$ has the following successful computation in the original program:

$$\mathtt{id} \,[\![\, \mathtt{f(h\ x)\ x} \overset{\mathtt{RN}}{\longrightarrow} \{\mathtt{x} \mapsto \mathtt{A}\} \,[\![\, \mathtt{g\ (h\ A)} \overset{\mathtt{RN}}{\longrightarrow} \{\mathtt{x} \mapsto \mathtt{A}\} \,[\![\, \mathtt{g\ B} \overset{\mathtt{RN}}{\longrightarrow} \{\mathtt{x} \mapsto \mathtt{A}\} \,[\![\, \mathtt{C}$$

by using the following definitional tree for function $f$:

$$\begin{array}{l} branch(\mathtt{f\ x\ y}, 2, flex, \\ \qquad\qquad rule(\mathtt{f\ x\ A} = \mathtt{g\ x})) \end{array}$$

whereas $\rho(\mathtt{f\ (h\ x)\ x}) = \mathtt{f'\ (h'\ x)\ x}$ may suspend in the specialized program:

$$\mathtt{id} \,[\![\, \mathtt{f'\ (h'\ x)\ x} \stackrel{\mathtt{RN}}{\longrightarrow} \mathtt{id^s} \,[\![\, \mathtt{f'\ (h'\ x)\ x}$$

by considering the following definitional tree:

$$\begin{array}{l} branch(\mathtt{f'\ x\ y}, 1, flex, \\ \qquad\qquad branch(\mathtt{f'\ B\ y}, 2, flex, \\ \qquad\qquad\qquad\qquad rule(\mathtt{f'\ B\ A} = \mathtt{C}))) \end{array}$$

for the specialized function $\mathtt{f'}$.

The above example points out a problem which arises when proving the total correctness of our transformation: *the floundering behaviour of an expression in the specialized program may vary depending on the definitional trees used for its evaluation.* For instance, in the above example, if we prove the $\stackrel{\mathtt{RN}}{\longrightarrow}$ derivation using the following definitional tree for function $\mathtt{f'}$:

$$\begin{array}{l} branch(\mathtt{f'\ x\ y}, 2, flex, \\ \qquad\qquad branch(\mathtt{f'\ x\ A}, 1, flex, \\ \qquad\qquad\qquad\qquad rule(\mathtt{f'\ B\ A} = \mathtt{C}))) \end{array}$$

then the goal $\mathtt{f'\ (h'\ x)\ x}$ does not suspend, and the following derivation for the term is obtained:

$$\mathtt{id} \,[\![\, \mathtt{f'\ (h'\ x)\ x} \stackrel{\mathtt{RN}}{\longrightarrow} \{\mathtt{x} \mapsto \mathtt{A}\} \,[\![\, \mathtt{f'\ (h'\ A)\ A} \stackrel{\mathtt{RN}}{\longrightarrow} \{\mathtt{x} \mapsto \mathtt{A}\} \,[\![\, \mathtt{f'\ B\ A} \stackrel{\mathtt{RN}}{\longrightarrow} \{\mathtt{x} \mapsto \mathtt{A}\} \,[\![\, \mathtt{C}$$

Indeed, the problem is not caused by our partial evaluation scheme but also happens in the evaluation of expressions by using the $\stackrel{\mathtt{RN}}{\longrightarrow}$ calculus. The following example illustrates this point.

**Example 13** Let us consider the following program:

```
f B A  =  C    % flex
  h A  =  B    % rigid
```

and two different definitional trees for function $\mathtt{f}$:

$$\begin{array}{l} branch(\mathtt{f\ x\ y}, 1, flex, \\ \qquad\qquad branch(\mathtt{f\ B\ y}, 2, flex, \\ \qquad\qquad\qquad\qquad rule(\mathtt{f\ B\ A} = \mathtt{C}))) \end{array}$$

in which the first inductive position corresponds to the first argument of $\mathtt{f}$ and,

$$\begin{array}{l} branch(\mathtt{f\ x\ y}, 2, flex, \\ \qquad\qquad branch(\mathtt{f\ x\ A}, 1, flex, \\ \qquad\qquad\qquad\qquad rule(\mathtt{f\ B\ A} = \mathtt{C}))) \end{array}$$

where the first inductive position is the second argument of function `f`. Now, the expression `f (h x) x` has the following successful computation by using the latter definitional tree

$$\mathtt{id} \,[\![\, \mathtt{f(h\ x)\ x} \xrightarrow{\mathtt{RN}} \{\mathtt{x} \mapsto \mathtt{A}\} \,[\![\, \mathtt{f(h\ A)\ A} \xrightarrow{\mathtt{RN}} \{\mathtt{x} \mapsto \mathtt{A}\} \,[\![\, \mathtt{f\ B\ A} \xrightarrow{\mathtt{RN}} \{\mathtt{x} \mapsto \mathtt{A}\} \,[\![\, \mathtt{C}$$

whereas it suspends by using the first definitional tree.

We have identified that this problem appears when the expression to be partially evaluated contains nested function symbols with several occurrences of a variable. In these situations, depending on the argument which is first evaluated (which is determined by the definitional tree), the repeated variable may be instantiated or not, leading to different floundering behaviours. Thus, we consider in the following a restricted notion of closedness (called *basic* closedness in [Alpuente *et al.*, 1998b], in symbols $closed^-$) in which nested expressions are not allowed. It is defined as the recursive closedness of Definition 36 except for the case

$$closed(S,t) = \bigwedge_{x \mapsto t' \in \theta} closed(S,t') \quad \text{if } \exists \theta, \exists s \in S \text{ such that } \theta(s) = t$$

which is replaced by the more simple condition

$$closed^-(S,t) = true \quad \text{if } \exists \theta, \exists s \in S \text{ such that } \theta(s) = t \text{ and } \theta \text{ is constructor.}$$

In the following, we will use this simpler closedness condition to demonstrate the equivalence between the original and residual programs w.r.t. the floundering behavior.

The first lemma states a precise relation between the renaming of a term and that of a constructor instance of the same term.

**Lemma 39** *Let $S$ be a finite set of terms and $\rho$ an independent renaming for $S$. Given a constructor substitution $\theta$, then $\{\theta(s') \mid s' \in ren_\rho(s)\} \subseteq ren_\rho(\theta(s))$ for any term $s$.*

*Proof.* We prove the claim by structural induction on $s$:

**Let $s \in \mathcal{X}$:** Since $\theta$ is constructor, by definition of $ren_\rho$ it is immediate that $\{\theta(s') \mid s' \in ren_\rho(s)\} = \{\theta(s)\} = ren_\rho(\theta(s))$ and the claim follows (note that constructors and variables are not renamed by $ren_\rho$).

**Let $s = c(s_1,\ldots,s_n)$ with $n \geq 0$ and $c \in \mathcal{C}$:** Then $\theta(s) = c(\theta(s_1),\ldots,\theta(s_n))$ and, by definition of $ren_\rho$, the following equivalences hold:

$$\begin{array}{rcl}
\{\theta(s') \mid s' \in ren_\rho(s)\} & = & \{\theta(s') \mid s' \in ren_\rho(c(s_1,\ldots,s_n))\} \\
& = & \{\theta(c(s'_1,\ldots,s'_n)) \mid s'_i \in ren_\rho(s_i),\ i = 1,\ldots,n\} \\
& = & \{c(\theta(s'_1),\ldots,\theta(s'_n)) \mid s'_i \in ren_\rho(s_i),\ i = 1,\ldots,n\}.
\end{array}$$

Now, by the inductive hypothesis, we have $\{\theta(s'_i) \mid s'_i \in ren_\rho(s_i)\} \subseteq ren_\rho(\theta(s_i))$ for all $i \in \{1, \ldots, n\}$. Therefore,

$$\begin{array}{rcl} \{c(\theta(s'_1), \ldots, \theta(s'_n)) \mid s'_i & \in & ren_\rho(s_i),\ i = 1, \ldots, n\} \\ & \subseteq & \{c(s''_1, \ldots, s''_n) \mid s''_i \in ren_\rho(\theta(s_i)),\ i = 1, \ldots, n\} \\ & = & ren_\rho(c(\theta(s_1), \ldots, \theta(s_n))) \\ & = & ren_\rho(\theta(s))\ . \end{array}$$

**Let** $s = f(t_1, \ldots, t_n)$ **with** $n \geq 0$ **and** $f \in \mathcal{F}$**:** By definition of $ren_\rho$, we have

$$\begin{array}{rcl} \{\theta(s') \mid s' \in ren_\rho(s)\} & = & \{\theta(\sigma(\rho(s''))) \mid s = \sigma(s''),\ s'' \in S, \text{ and} \\ & & \sigma \text{ is a constructor substitution }\} \end{array}$$

If the above set is empty (e.g., because $s$ is not closed w.r.t. $S$), then the claim is trivial. Otherwise, consider the set

$$S' = \{s'' \mid s = \sigma(s''),\ s'' \in S, \text{ and } \sigma \text{ is a constructor substitution }\}\ .$$

Trivially, $\theta(s) = \theta(\sigma(s''))$ with $\sigma \circ \theta$ constructor for all $s'' \in S'$. Therefore, by definition of $ren_\rho$, we have:

$$\{\theta(\sigma(\rho(s''))) \mid s'' \in S'\} \subseteq ren_\rho(\theta(s))$$

which concludes the proof.

□

Note that the converse property $\{\theta(s') \mid s' \in ren_\rho(s)\} \supseteq ren_\rho(\theta(s))$ does not hold, since $\theta(s)$ has generally more possible renamings. For instance, one can consider the set $S = \{f(x), f(s(y))\}$ and the independent renaming $\rho = \{f(x) \mapsto f_1(x),\ f(s(y)) \mapsto f_2(y)\}$. Given the term $f(z)$ and the constructor substitution $\theta = \{z \mapsto s(0)\}$, we can easily see that $ren_\rho(\theta(f(z))) = \{f_1(s(0)), f_2(0)\}$ while $\theta(ren_\rho(f(z))) = \{f_1(s(0))\}$.

**Lemma 40** *Let $S$ be a finite set of terms and $\rho$ an independent renaming for $S$. Let $s$ and $t$ be operation-rooted terms which are closed w.r.t. $S$. Then, $ren_\rho(s) \cap ren_\rho(t) \neq \varnothing$ iff $s = t$.*

*Proof.* The claim $s = t \Rightarrow ren_\rho(s) \cap ren_\rho(t) \neq \varnothing$ is immediate by considering the fact that $s$ and $t$ are closed w.r.t. $S$ and, thus, at least one possible renaming exists.

Now we prove $ren_\rho(s) \cap ren_\rho(t) \neq \varnothing \Rightarrow s = t$. Since $s$ and $t$ are operation-rooted, by definition of $ren_\rho$, we have:

$$S_1 = ren_\rho(s) = \{\sigma(\rho(s')) \mid s' \in S,\ s = \sigma(s'), \text{ and } \sigma \text{ is a constructor substitution}\}$$

and

$S_2 = ren_\rho(t) = \{\sigma(\rho(s')) \mid s' \in S,\ t = \sigma(s'),$ and $\sigma$ is a constructor substitution$\}$

Now, for each element $t' \in S_1 \cap S_2$, there exist $s_1, s_2 \in S$ such that $s = \sigma_1(s_1)$, $t = \sigma_2(t_2)$, $t' = \sigma_1(\rho(s_1))$, and $t' = \sigma_2(\rho(s_2))$, where $\sigma_1$ and $\sigma_2$ are constructor substitutions. Since the independent renaming $\rho$ renames different terms to different new function symbols, the only possibility is that $s_1 = s_2$ and, then, $\sigma_1 = \sigma_2\ [\mathcal{V}ar(s_1)]$. Hence, we can define $S_1 \cap S_2$ as follows:

$S_1 \cap S_2 = \{\sigma(\rho(s')) \mid s' \in S,\ s = \sigma(s'),\ t = \sigma(s'),$ and $\sigma$ is a constructor substitution$\}$

Since $S_1 \cap S_2 \neq \varnothing$, there exists at least one $s' \in S$ such that $s = \sigma(s')$ and $t = \sigma(s')$. Therefore, $s = \sigma(s') = t$ and the claim follows. □

Now we prove that whenever a renamed term $s'$ matches another renamed term $t'$, then $s$ matches $t$ too.

**Lemma 41** *Let $S$ be a finite set of terms and $\rho$ an independent renaming for $S$. Let $s$ and $t$ be operation-rooted terms which are closed w.r.t. $S$. Let $s' \in ren_\rho(s)$ and $t' \in ren_\rho(t)$ be their renamings. If $s' = \theta(t')$ with $\theta$ constructor, then $s = \theta(t)$.*

*Proof.* Since $s' = \theta(t')$ and $\theta(t') \in \{\theta(t'') \mid t'' \in ren_\rho(t)\}$, then

$$s' \in \{\theta(t'') \mid t'' \in ren_\rho(t)\}\ .$$

By Lemma 39, we have $s' \in ren_\rho(\theta(t))$. Since it also holds that $s' \in ren_\rho(s)$, then $s' \in ren_\rho(s) \cap ren_\rho(\theta(t))$. Then, by Lemma 40, $s = \theta(t)$, which concludes the proof. □

Again, the converse property is not true. Consider, for instance, the set of terms $S = \{f(x), f(s(y))\}$ and the independent renaming $\rho = \{f(x) \mapsto f_1(x),\ f(s(y)) \mapsto f_2(y)\}$. Given the terms $f(x)$ and $f(s(0))$, $f(x)$ is a constructor instance of $f(s(0))$ but $f_1(x) \in ren_\rho(f(x))$ is not an instance of $f_2(0) \in ren_\rho(f(s(0)))$.

In order to prove the equivalence between the original and specialized programs w.r.t. floundering-freeness, we need a proof scheme similar to the demonstration of correctness of [Alpuente *et al.*, 1998b, Theorem 3.10], where the simpler notion of closedness is also used. The key idea is to prove that, given a resultant $R$ obtained from a narrowing derivation $\mathcal{D}$ in the original program, if there exists a narrowing derivation which uses the same rules and in the same order as $\mathcal{D}$, then the same derivation can be proven in the residual program using the resultant $R$. To avoid duplication of work, we will omit the identical part of the proof scheme and refer to [Alpuente *et al.*, 1998b] for further details. Rather, we focus on the demonstration of floundering-freeness, which follows below.

**Theorem 42 (floundering-freeness)** *Let $\mathcal{R}$ be an inductively sequential program, $e$ an equation, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{R}'$ be a RN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed$^-$, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. Then, $e$ suspends in $\mathcal{R}$ iff $e'$ suspends in $\mathcal{R}'$.*

*Proof.*
($\subseteq$) Since $e'$ is $S'$-closed, then $e$ is $S$-closed (by Lemma 41). Hence, there exists a constructor substitution $\gamma$ and a term $s \in S$ such that $e = \gamma(s)$. Since $e \leq s$, then $s$ also flounders in one step. Then, by definition of RN-PE, there exists a derivation

$$s \overset{\mathtt{RN}}{\rightsquigarrow}_\sigma \sigma(s) \overset{\mathtt{NN}}{\rightsquigarrow}_\theta r$$

such that $slist(\theta \circ \sigma) = [\sigma^{a_0}, \varphi_1^{a_1}, \ldots, \varphi_k^{a_k}]$, and the specialized rules in $\mathcal{R}'$ have the form

$$\begin{array}{rcll} \sigma(s') & \to & f_1(\overline{x_{m_1}}) & (R_0) \\ \varphi_1(f_1(\overline{x_{m_1}})) & \to & f_2(\overline{x_{m_2}}) & (R_1) \\ & \cdots & & \\ \varphi_k(f_k(\overline{x_{m_k}})) & \to & ren_\rho(r) & (R_k) \end{array}$$

where the evaluation annotations for the outermost symbols in the left-hand sides of $R_0, \ldots, R_k$ are, respectively, $a_0, \ldots, a_k$. Since $e$ suspends, we have that the computation $cs(e)$ reaches a call of the form

$$cst(\delta(e), branch(\pi, o, rigid, \mathcal{T}_1, \ldots, \mathcal{T}_j))$$

where $\delta(e)|_o \in \mathcal{X}$. Then, there exists some $i \in \{1, \ldots, k\}$ such that $\varphi_i \circ \cdots \circ \varphi_1 \circ \sigma \not\leq \delta \circ \gamma\ [\mathcal{D}om(\varphi_i \circ \cdots \circ \varphi_1 \circ \sigma)]$ and $a_i = rigid$. Therefore, the following computation for $e' = \gamma(\rho(s))$ can be proven

$$e' \overset{\mathtt{RN}}{\rightsquigarrow}_{R_0} \ldots \overset{\mathtt{RN}}{\rightsquigarrow}_{R_{i-1}} e_{i-1}$$

where $e_{i-1} = \delta(\gamma(f_i(\overline{x_{m_i}})))$. Finally, we conclude that $e_{i-1}$ suspends, because the evaluation annotation for $f_i$ is $rigid$ (since $a_i = rigid$) and $\varphi_i \circ \cdots \circ \varphi_1 \circ \sigma \not\leq \delta \circ \gamma\ [\mathcal{D}om(\varphi_i \circ \cdots \circ \varphi_1 \circ \sigma)]$.
($\supseteq$) By the $S'$-closedness of $e'$, we have that $e' = \gamma(s')$ for some constructor substitution $\gamma$ and term $s' \in S'$, where $s' = \rho(s)$ and $s \in S$. Since $e'$ suspends, then the evaluation annotation for $\rho(s')$ in $\mathcal{R}$ must be $rigid$. Hence, according to Definition 29, we have that the associated $\overset{\mathtt{NN}}{\rightsquigarrow}$ derivation for $s$ in $\mathcal{R}$ has the form

$$s \overset{\mathtt{NN}}{\rightsquigarrow}_\theta r \text{ with } split(\theta) = [\varphi_0^{rigid}, \ldots, \varphi_k]$$

which produces the following set of resultants in $\mathcal{R}'$:

$$\begin{array}{rcl} \varphi_0(s') & \to & f_1(\overline{x_{m_1}}) \\ \varphi_1(f_1(\overline{x_{m_1}})) & \to & f_2(\overline{x_{m_2}}) \\ & \cdots & \\ \varphi_k(f_k(\overline{x_{m_k}})) & \to & ren_\rho(r) \end{array}$$

Moreover, we have that $\varphi_0(s') \not\leq e' = \gamma(s')$ and thus $\varphi_0 \not\leq \gamma\ [Var(s')]$. Since the evaluation annotation for the outermost function symbol of $s' = \rho(s)$ is $rigid$, by definition of RN-PE, we have that $cs(s) = id^s \, [\!]\, s$. Finally, since $\varphi_0 \not\leq \gamma\ [Var(s')]$, the computation of $cs(\gamma(s))$ reaches a call of the form $cst(\gamma(s), branch(\pi, o, rigid, \mathcal{T}_1, \ldots, \mathcal{T}_j))$, with $\gamma(s)|_o \in \mathcal{X}$ and thus the expression $e = \gamma(s)$ suspends as well. □

As a corollary of Theorems 38 and 42, we can establish the total correctness of the transformation.

**Theorem 43 (total correctness)** *Let $\mathcal{R}$ be an inductively sequential program. Let $e$ be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{R}'$ be a RN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed$^-$, where $e' = ren_\rho(e)$ and $S' = \rho(S)$.*

1. *If $e' \overset{\mathtt{RN}^*}{\rightsquigarrow}_{\sigma'} true$ in $\mathcal{R}'$, then $e \overset{\mathtt{RN}^*}{\rightsquigarrow}_{\sigma} true$ in $\mathcal{R}$ where $\sigma' = \sigma\ [V]$ (soundness)*
2. *If $e \overset{\mathtt{RN}^*}{\rightsquigarrow}_{\sigma} true$ in $\mathcal{R}$, then $e' \overset{\mathtt{RN}^*}{\rightsquigarrow}_{\sigma'} true$ in $\mathcal{R}'$ where $\sigma' = \sigma\ [V]$ (completeness)*

*Proof.*

(soundness) Since $e' \overset{\mathtt{RN}}{\rightsquigarrow}{}^*_{\sigma'}\ true$ in $\mathcal{R}'$, all steps are complete. Thus, by claim 1 of Theorem 28, there exists a needed narrowing derivation $e' \overset{\mathtt{NN}}{\rightsquigarrow}{}^*_{\sigma'}\ true$ in $\mathcal{R}'$. By Theorem 37, there exists a NN-PE $\mathcal{R}''$ of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $e' \overset{\mathtt{NN}}{\rightsquigarrow}{}^*_{\sigma'}\ true$ in $\mathcal{R}''$. Hence, we can apply the (strong) soundness of NN-PE (Theorem 21), which proves that there exists a needed narrowing derivation $e \overset{\mathtt{NN}}{\rightsquigarrow}{}^*_{\sigma}\ true$ in $\mathcal{R}$, where $\sigma = \sigma'\ [V]$. Since $\mathcal{R}$ is inductively sequential, by claim 2 of Theorem 28, there exists a derivation $e \overset{\mathtt{RN}}{\rightsquigarrow}{}^*_{\theta}\ e''$ such that $\exists\varphi.\ \varphi(e'') \to^* true$ and $\varphi \circ \theta = \sigma$. The preservation of floundering-freeness (Theorem 42) ensures that the goal $e$ does not flounder in $\mathcal{R}$, thus $\varphi = id$ and $e'' = true$. Hence, we conclude that $e \overset{\mathtt{RN}}{\rightsquigarrow}{}^*_{\sigma}\ true$ can be proven in $\mathcal{R}$.

(completeness) This is proven by following a proof structure perfectly analogous to the proof of soundness, but in the opposite sense of direction.

□

We conjecture that the results presented in this section could be extended to cover the more general, recursive notion of closedness if the considered definitional trees are fixed through the partial evaluation process.

**Conjecture 44** *Let $\mathcal{R}$ be an inductively sequential program. Let $e$ be an equation, $V \supseteq \mathcal{V}ar(e)$ a finite set of variables, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{R}'$ be a RN-PE of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$.*

1. *If* $e' \overset{\mathtt{RN}^*}{\leadsto}_{\sigma'}$ *true in* $\mathcal{R}'$*, then there exists a set* $\mathcal{T}$ *of definitional trees associated to the functions of the set* $S$ *such that* $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma}$ *true in* $\mathcal{R}$ *using the definitional trees in* $\mathcal{T}$*, where* $\sigma' = \sigma\ [V]$ *(soundness)*

2. *If* $e \overset{\mathtt{RN}^*}{\leadsto}_{\sigma}$ *true in* $\mathcal{R}$*, then there exists a set* $\mathcal{T}'$ *of definitional trees associated to the functions of the set* $S'$ *such that* $e' \overset{\mathtt{RN}^*}{\leadsto}_{\sigma'}$ *true in* $\mathcal{R}'$ *using the definitional trees in* $\mathcal{T}'$*, where* $\sigma' = \sigma\ [V]$ *(completeness)*

In Chapter 7, we present a more general framework which also considers residuation, and where correctness results are provided using the recursive notion of closedness. The key idea there is that, there, definitional trees are implicitly fixed, since we use an intermediate representation of programs (cf. Section 7.2.1) in which definitional trees are compiled into program rules.

## 4.7 Practicality

In this section, we illustrate the advantages of the RN-PE method in the context of residuating functional logic programs as well as the practicality of our approach.

Let us introduce an example which shows that RN-PE can be used for proving floundering-freeness of a wide class of goals in a given program.

**Example 14** Consider the following program which defines the arithmetic addition and the predicate `isNat`, which returns `True` when the argument is a natural number:

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{0} + \mathtt{y} & = & \mathtt{y} & \mathtt{isNat\ 0} & = & \mathtt{True} \\ (\mathtt{Succ\ x}) + \mathtt{y} & = & \mathtt{Succ\ (x + y)} & \mathtt{isNat\ (Succ\ x)} & = & \mathtt{isNat\ x} \end{array}$$

where "+" is rigid and "`isNat`" is flexible. Consider

$$S = \{(\mathtt{x} + \mathtt{y} \approx \mathtt{z})\ \&\ (\mathtt{isNat\ x}),\ (\mathtt{y} \approx \mathtt{z})\ \&\ \mathtt{True}\}$$

and the independent renaming

$$\rho = \{(\mathtt{x} + \mathtt{y} \approx \mathtt{z})\ \&\ (\mathtt{isNat\ x}) \mapsto \mathtt{and3\ x\ y\ z},\ (\mathtt{y} \approx \mathtt{z})\ \&\ \mathtt{True} \mapsto \mathtt{and2\ y\ z}\}$$

Now, by considering the partial computations depicted in Figures 4.4 and 4.5,[6] the following RN-PE of the program w.r.t. $S$ (under $\rho$) is constructed:

$$\begin{array}{rcl} \mathtt{and3\ 0\ 0\ 0} & = & \mathtt{True} \\ \mathtt{and3\ 0\ (Succ\ y)\ (Succ\ z)} & = & \mathtt{and2\ y\ z} \\ \mathtt{and3\ (Succ\ x)\ y\ (Succ\ z)} & = & \mathtt{and3\ x\ y\ z} \\ \mathtt{and2\ 0\ 0} & = & \mathtt{True} \\ \mathtt{and2\ (Succ\ x)\ (Succ\ y)} & = & \mathtt{and2\ x\ y} \end{array}$$

[6] Here we assume that strict equality $\approx$ is flexible.

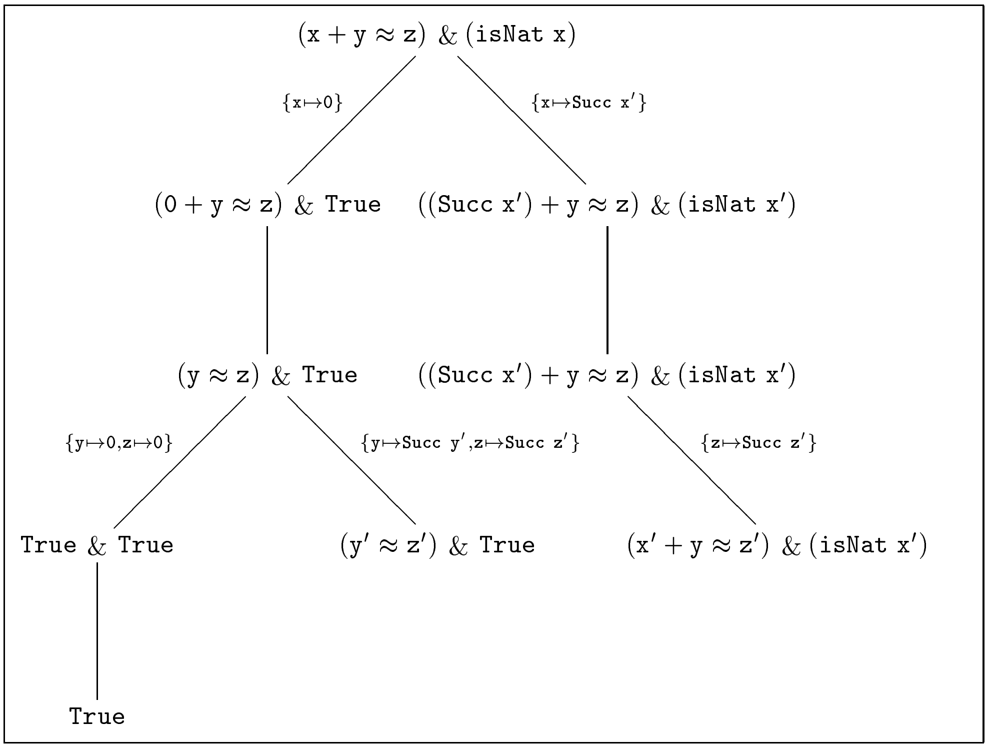


Figure 4.4: Partial computations for $(\mathtt{x} + \mathtt{y} \approx \mathtt{z})\ \&\ (\mathtt{isNat}\ \mathtt{x})$.

where both `and3` and `and2` are flexible functions. Then, for proving floundering-freeness it is sufficient to check that no operation symbol of the resulting partially evaluated program has a *rigid* annotation. For instance, one can easily see that the goal $(\mathtt{x} + \mathtt{y} = \mathtt{z})\ \&\ (\mathtt{isNat}\ \mathtt{x})$ is floundering-free in the residual program (hence in the original), since the program has no rigid functions, while in the original program this is not immediate.

The last example in this section is aimed at showing that RN-PE can be also used to simplify the dynamic behavior of a program, which can result in a significant optimization.

**Example 15** Consider the classical map coloring program which assigns a color to

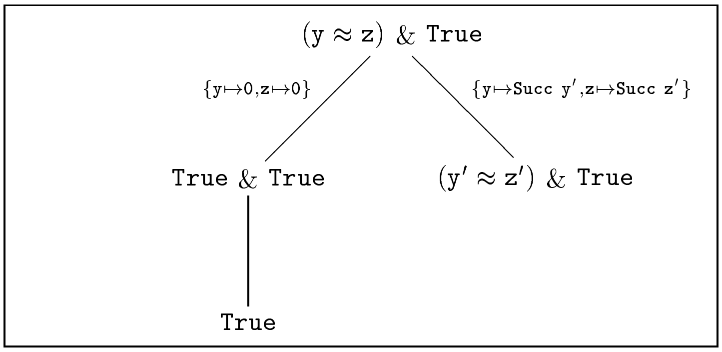


Figure 4.5: Partial computations for (y ≈ z) & True.

each of four countries such that countries with a common border have different colors:

```
isColor Red           =  True
isColor Yellow        =  True
isColor Green         =  True
coloring l1 l2 l3 l4  =  isColor l1 & isColor l2
                             & isColor l3 & isColor l4
coloring l1 l2 l3 l4  =  diff l1 l2 & diff l1 l3
                             & diff l2 l4 & diff l3 l4
```

where the predefined function `diff`, which checks whether its arguments are different, is the only *rigid* function

```
diff x y = (x == y) =:= False
```

As predefined in Curry, function == denotes the rigid strict equality whereas function =:= is used as the flexible strict equality. Now, we consider the specialization of the expression:

```
correct l1 l2 l3 l4 & coloring l1 l2 l3 l4
```

which gives the following specialized program:

```
and4 Red Yellow Green Red       =  True
and4 Red Green Yellow Red       =  True
and4 Yellow Red Green Yellow    =  True
and4 Yellow Green Red Yellow    =  True
and4 Green Red Yellow Green     =  True
and4 Green Yellow Red Green     =  True
```

where some potential colorings have been discarded, thus simplifying the dynamic behavior of the program and achieving a significant speedup (actually it runs 23 times faster).

## 4.8 Conclusions

We have presented a general partial evaluation framework which derives from that of Chapter 3, but extends it to consider the combination of needed narrowing and residuation as the core operational mechanism. The extended framework allows us to safely deal with evaluation annotations, which is crucial for controlling unfolding during partial evaluation as well as for correctly synthesizing evaluation annotations for the specialized functions.

Despite the practical importance of logic programs with dynamic scheduling, there has been surprisingly little work devoted to their specialization. The only transformation framework that we are aware of for logic languages with delays is that of Etalle *et al.* [1997], which is based on the fold/unfold approach to program transformation. It differs from our methodology, since our framework is based on the (automatic) partial evaluation approach and applies to logic languages with lazy functions. Moreover, we allow unfolding of suspended expressions at partial evaluation time, which is not the case of [Etalle *et al.*, 1997].

Control problems have not been tackled through Part I. Within our context, control issues involve the definition of a partial evaluation procedure based on the narrowing relation which ensures termination and closedness. They will be addressed in Parts II and III.

# Part II

# Control Issues

# Chapter 5

# A Generic Partial Evaluation Scheme

In this section, we recall the partial evaluation scheme for functional logic languages of Alpuente *et al.* [1998b]. The method is based on an automatic unfolding algorithm which builds narrowing trees. After presenting the algorithm in Section 5.2, we will address its termination in Section 5.3.

## 5.1 Introduction

In the literature on program specialization, control problems have been tackled from two different approaches. *On-line* partial evaluators perform all the control decisions *during* the actual specialization phase, whereas *off-line* specializers perform an analysis phase that generates annotations on the program *before* the actual specialization begins [Jones *et al.*, 1993]. Off-line methods can be faster than on-line methods, and they are easier to stop [Consel and Danvy, 1993]. An interesting application of partial evaluation is the generation of compilers and compiler generators [Jones *et al.*, 1993]. For this purpose, the partial evaluator must be *self-applicable*, i.e., it enjoys the possibility of specializing the partial evaluator itself [Jones *et al.*, 1993]. In order to achieve *effective* self-application, partial evaluation of functional programs has mainly stressed the off-line approach, while supercompilation and partial deduction have concentrated on the on-line approach, which usually provides more accurate specializations. In this chapter, we are concerned with the questions of precision and termination, but not self-application, and thus we focus on the on-line approach.

Now, we look at a simple on-line partial evaluation method for functional logic programs which essentially proceeds as the partial evaluation algorithm by Gallagher

[1993]. For a given program $\mathcal{R}$ and input term $t$, a partial evaluation for $S$ in $\mathcal{R}$ is computed, with $S$ initialized to the set of calls (terms) appearing in $t$. Then, this process is repeated for every term which occurs in the right-hand side of the resulting rules and which is not closed with respect to the set of terms already evaluated. Assuming that it terminates, this procedure computes a set of partially evaluated terms $S'$ and a set of resultants $\mathcal{R}'$ (the partial evaluation of $S'$ in $\mathcal{R}$) such that the closedness condition for $\mathcal{R}' \cup \{t\}$ is satisfied. The following example illustrates the method.

**Example 16** Consider the following program which defines the addition on natural numbers:

$$\begin{array}{rcl} \mathtt{0} + \mathtt{n} & \to & \mathtt{n} \\ (\mathtt{Succ}\ \mathtt{m}) + \mathtt{n} & \to & \mathtt{Succ}\ (\mathtt{m} + \mathtt{n}) \end{array}$$

and the initial term:

$$(\mathtt{Succ}\ (\mathtt{Succ}\ \mathtt{0})) + \mathtt{x}$$

In the first iteration, we start with $S = \{(\mathtt{Succ}\ (\mathtt{Succ}\ \mathtt{0})) + \mathtt{x}\}$. Now, we compute the following one-step derivation:

$$(\mathtt{Succ}\ (\mathtt{Succ}\ \mathtt{0})) + \mathtt{x} \leadsto_{\{\mathtt{m} \mapsto \mathtt{Succ}\ \mathtt{0}\}} \mathtt{Succ}\ ((\mathtt{Succ}\ \mathtt{0}) + \mathtt{x})$$

Then, we repeat the process for the term $(\mathtt{Succ}\ \mathtt{0}) + \mathtt{x}$ which appears in the derived expression and which is not closed w.r.t. $S$:

$$(\mathtt{Succ}\ \mathtt{0}) + \mathtt{x} \leadsto_{\{\mathtt{m}' \mapsto \mathtt{0}\}} \mathtt{Succ}\ (\mathtt{0} + \mathtt{x})$$

Finally, the process is performed for the term $\mathtt{0} + \mathtt{x}$, which appeared in the above derivation:

$$\mathtt{0} + \mathtt{x} \leadsto_{\mathtt{id}} \mathtt{x}$$

To summarize, by using the procedure described above, we have computed the following one-step derivations:

$$\begin{array}{l} (\mathtt{Succ}\ (\mathtt{Succ}\ \mathtt{0})) + \mathtt{x} \leadsto_{\{\mathtt{m} \mapsto \mathtt{Succ}\ \mathtt{0}\}} \mathtt{Succ}\ ((\mathtt{Succ}\ \mathtt{0}) + \mathtt{x}) \\ (\mathtt{Succ}\ \mathtt{0}) + \mathtt{x} \leadsto_{\{\mathtt{m}' \mapsto \mathtt{0}\}} \mathtt{Succ}\ (\mathtt{0} + \mathtt{x}) \\ \mathtt{0} + \mathtt{x} \leadsto_{\mathtt{id}} \mathtt{x} \end{array}$$

for the set of terms $S' = \{(\mathtt{Succ}\ (\mathtt{Succ}\ \mathtt{0})) + \mathtt{x},\ (\mathtt{Succ}\ \mathtt{0}) + \mathtt{x},\ \mathtt{0} + \mathtt{x}\}$. Thus, the partial evaluation of $S'$ in $\mathcal{R}$ is the following residual program $\mathcal{R}'$:

$$\begin{array}{rcl} (\mathtt{Succ}\ (\mathtt{Succ}\ \mathtt{0})) + \mathtt{x} & = & \mathtt{Succ}\ ((\mathtt{Succ}\ \mathtt{0}) + \mathtt{x}) \\ (\mathtt{Succ}\ \mathtt{0}) + \mathtt{x} & = & \mathtt{Succ}\ (\mathtt{0} + \mathtt{x}) \\ \mathtt{0} + \mathtt{x} & = & \mathtt{x} \end{array}$$

Note that $\mathcal{R}' \cup \{(\mathtt{Succ}\ (\mathtt{Succ}\ \mathtt{0}) + \mathtt{x})\}$ is $S'$-closed.

There are two main issues concerning the correctness for a partial evaluation procedure: termination, i.e., given any input goal, execution should always reach a stage for which there is no way to continue; and (partial) correctness, i.e., (if execution terminates, then) the operational semantics of the goal with respect to the residual and original programs should coincide (this latter issue has been dealt in the previous part of the thesis).

As for termination, the partial evaluation procedure outlined above involves two classical termination problems. The first problem—the so-called *local* termination problem—is the termination of unfolding, or how to control and keep the expansion of the narrowing trees (which provide partial evaluations for individual calls) finite. The *global* level of control concerns the termination of recursive unfolding, or how to stop recursively constructing narrowing trees while still guaranteeing that the desired amount of specialization is retained and that the closedness condition is reached. As we mentioned before, the set of terms $S$ appearing in the goal with which the specialization is performed usually needs to be augmented in order to fulfill the closedness condition. This brings up the problem of how to keep finite the process of constructing this set, by means of some appropriate abstraction operator which guarantees termination. In the following, we follow the approach of Alpuente *et al.* [1998b] which establishes a clear distinction between local and global control. This contrasts with the standard approach followed in the literature on (positive) supercompilation [Glück and Sørensen, 1994; Sørensen and Glück, 1995; Turchin, 1986a], where these two issues are not (explicitly) distinguished, as only one-step unfolding is performed and a large evaluation structure is built which comprises something similar to both the local narrowing trees and the global configurations of Martens and Gallagher [1995].

As for local termination, depth-bounds, loop-checks, and well-founded orderings are some commonly used tools for controlling the unfolding during the construction of the search trees [Bruynooghe *et al.*, 1992]. As for global control, it is common to consider proper (well-founded) abstraction operators which are based on the notion of $msg$ (*most specific generalization*, see Definition 57). As it is well known, using the $msg$ can induce a loss of precision. In the following section, we will present a simple partial evaluation algorithm which is parametric w.r.t. these local and global operators.

## 5.2 The Partial Evaluation Procedure

This PE procedure of Alpuente *et al.* [1998b] originates from the framework for ensuring global termination of partial deduction given by Martens and Gallagher [1995], which was reformulated to deal with the use of functions in combination with logical variables. An important issue of partial evaluators is their *control restructuring* ability, which is concerned with the relationship between program points (function definitions) in the original and the residual programs [Glück and Sørensen, 1996]:

- *Monovariant:* any program point in the original program gives rise to zero or one program point in the residual program.
- *Polyvariant:* any program point in the original program can give rise to one or more program points in the residual program.
- *Monogenetic:* any program point in the residual program is produced from a single program point of the original program.
- *Polygenetic:* any program point in the residual program may be produced from one or more program points of the original program.

NPE is able to produce both polyvariant as well as polygenetic specializations. Polyvariance is achieved thanks to the use of the unification-based information propagation mechanism of narrowing, which allows us to compute a number of independent specializations for a given call with respect to different instantiations of its variables. The possibility of specializing complex, nested terms also allows the partial evaluator to produce polygenetic specializations, since these terms are treated as single units during the partial evaluation process (thus producing a single specialized definition constructed by using the definitions of all functions in the expression).

In the following, we formally introduce the generic algorithm for partial evaluation of functional logic programs based on narrowing of Alpuente *et al.* [1998b], which ensures the closedness of the partially evaluated program. The resulting NPE algorithm is generic with respect to:

1. the *narrowing relation* that constructs search trees,
2. the *unfolding rule* which determines when and how to terminate the construction of the trees, and
3. the *abstraction operator* used to guarantee that the construction of the set of terms from which the partial evaluation will be obtained is finite.

Let us note that we could also use the $\stackrel{\mathtt{RN}}{\longrightarrow}$ calculus introduced in Section 4.2 as operational mechanism instead of simply using ordinary narrowing. Actually, a partial evaluator algorithm based on the $\stackrel{\mathtt{RN}}{\longrightarrow}$ calculus has been implemented as an extension of the Indy system (see Appendix A). However, we prefer to start from the original, more generic algorithm of Alpuente *et al.* [1998b].

In the remainder of this chapter, we let $\stackrel{\varphi}{\leadsto}$ denote a generic narrowing relation. A narrowing strategy $\varphi$ from a term $s$, denoted by $\varphi(s)$, returns a set of triples:

$\{(p_1, R_1, \sigma_1), \ldots, (p_n, R_n, \sigma_n)\}$ such that:

$$s \rightsquigarrow_{(p_1,R_1,\sigma_1)} s_1$$
$$\ldots$$
$$s \rightsquigarrow_{(p_n,R_n,\sigma_n)} s_n$$

are all possible narrowing steps from $s$ using the strategy $\varphi$.

Now, we formalize the notion of *unfolding rule* $U_\varphi$ which constructs a (possibly incomplete) root-to-leaf $\overset{\varphi}{\rightsquigarrow}$-narrowing tree and then extracts the resultants of the derivations of the tree.

**Definition 45 (unfolding rule)** *An unfolding rule $U_\varphi(s, \mathcal{R})$ (or simply $U_\varphi$) is a mapping which returns a partial evaluation for the term $s$ in the program $\mathcal{R}$ using the narrowing relation $\overset{\varphi}{\rightsquigarrow}$ (i.e., a set of resultants).*

*Given a set of terms $S$, by $U_\varphi(S, \mathcal{R})$ we denote the union of the sets $U_\varphi(s, \mathcal{R})$, for all $s$ in $S$.*

In the partial evaluation procedure, we use an abstraction operator *abstract* which keeps finite the process of constructing the set of partially evaluated terms (global control). In our context, this is guaranteed by ensuring that the set of specialized calls is kept finite, as we only replace some call of such a set by new terms which are lesser than or equal to (according to a suitable ordering) the calls which already exist in the set. The function *abstract* takes a sequence[1] of terms $q$ and a set of terms $S$, which represent the sequence of terms previously specialized and the new set to be considered for subsequent specializations, respectively. In the following, let us denote by $S_q$ the set of terms contained in the sequence $q$. Given $q$ and $T$ as inputs, the abstraction operator returns a *safe* approximation of $S_q \cup T$. Informally, by "safe" we mean that each term in $S_q \cup T$ is closed w.r.t. the sequence which results from $abstract(q, T)$.

**Definition 46 (abstraction operator)** *Let $q$ be a sequence of terms and let $S$ be a finite set of terms. An abstraction operator abstract is a mapping which returns a new sequence $q' = abstract(q, S)$ such that*

1. *if $s \in S_{q'}$ then there exists $t \in (S_q \cup S)$ such that $t|_p = \theta(s)$ for some nonvariable position $p$ and substitution $\theta$, and*
2. *for all $t \in (S_q \cup S)$, $t$ is closed with respect to $S_{q'}$.*

[1] In [Alpuente *et al.*, 1998b], the method is formalized via a transition system $(State, \longmapsto_{\mathcal{P}})$ whose transition relation $\longmapsto_{\mathcal{P}} \subseteq State \times State$ formalizes the computation steps. The set $State$ of partial evaluation configurations (states) is a parameter of the definition. The notion of state has been instantiated to sequences of terms in our framework for the sake of simplicity. It is also the instance for configurations which is considered in [Alpuente *et al.*, 1998b].

Roughly speaking, condition (1) ensures that the abstraction operator does not "create" new function symbols (i.e., symbols not present in the input arguments), whereas condition (2) ensures the correct propagation of closedness information, which is essential for guaranteeing that the residual program indeed satisfies the closedness condition.

Now we present the transition relation $\longmapsto_{\mathcal{PE}}$, which is the core mechanism of the partial evaluation algorithm.

**Definition 47 (partial evaluation transition relation)** *We define the partial evaluation relation $\longmapsto_{\mathcal{PE}}$ as the smallest relation satisfying*

$$\frac{\mathcal{R}' = U_\varphi(S_q, \mathcal{R})}{q \longmapsto_{\mathcal{PE}} abstract(q, \mathcal{R}'_{calls})}.$$

The transition relation is computed in two steps. First $\mathcal{R}'$ is computed from $S_q$ and $\mathcal{R}$. Then, by applying *abstract* to $q$ and $\mathcal{R}'_{calls}$, the next sequence of terms is computed. Here $\mathcal{R}$ is a global parameter of the definition, and at each step an auxiliary system $\mathcal{R}'$ is computed. Similarly to Martens and Gallagher [1995], applying *abstract* in every iteration permits to tune the control of polyvariance as much as needed. Also, it is within the *abstract* operation that the progress toward termination resides.

Now we are ready to formulate the generic algorithm for the narrowing-driven partial evaluation of functional logic programs with respect to a given goal.

**Definition 48 (partial evaluation algorithm)** *Let $\mathcal{R}$ be a program and $S$ be a finite set of expressions. We define the partial evaluation function $\mathcal{PE}$ as follows:*

$$\mathcal{PE}(\mathcal{R}, S) = S_q \quad \textit{if } abstract(nil, S) \longmapsto^*_{\mathcal{PE}} q \quad \textit{and} \quad q \longmapsto_{\mathcal{PE}} q$$

The partial evaluation algorithm essentially proceeds as follows. First, it starts the algorithm with the initial set of terms $S$. Then, it tries to derive new sequences of terms until a "fixed point" is reached, i.e., until the application of $\longmapsto_{\mathcal{PE}}$ to the current sequence returns the same sequence.

We note that the partial evaluation algorithm does not compute a partially evaluated program, but a set of partially evaluated terms $S_q$ which unambiguously determines its associated pre–partial evaluation $\mathcal{R}'$ in $\mathcal{R}$ (using $U_\varphi$). Then, the renaming transformation introduced in Chapter 3 allows us to obtain the final partial evaluation. The following theorem establishes that the closedness condition is reached independently from the narrowing strategy, unfolding rule, and abstraction operator.

**Theorem 49** *[Alpuente et al., 1998b] Let $\mathcal{R}$ be a program and $s$ be a term. If $\mathcal{PE}(\mathcal{R}, s)$ terminates computing the set of terms $S$, then $\mathcal{R}'$ and $s$ are $S$-closed, where the specialized program is given by $\mathcal{R}' = U_\varphi(S, \mathcal{R})$.*

As a consequence of the above theorem, it is trivial that the partially evaluated program $\mathcal{R}''$ (i.e., $\mathcal{R}'$ after applying a post-processing of renaming) will be also $S'$-closed, where $S'$ is the set of terms obtained by applying an independent renaming function to $S$.

The partial evaluation algorithm incorporates only the scheme of a complete narrowing-driven partial evaluator. The resulting partial evaluations might be further optimized by eliminating redundant functors and unnecessary repetition of variables, trying to adapt standard techniques presented in [Benkerimi and Hill, 1993], [Benkerimi and Lloyd, 1990], [Gallagher, 1993], [Gallagher and Bruynooghe, 1990]. This is especially interesting in our setting, where functions appearing as arguments of calls are in no way "dead" structures (as in partial deduction), but can also generate new calls to function definitions. As discussed in Chapter 3, a post-processing renaming transformation (see Definition 14) is then necessary to restore the *constructor discipline* and to eliminate redundant functions. As already mentioned, the renaming phase may also serve to remove any remaining lack of independence in the set of specialized terms.

In the following, we present a solution to the termination problem by introducing some simple and powerful unfolding and abstraction operators.

## 5.3 Ensuring Termination

The techniques to control termination of the NPE algorithm can be seen as an adaptation of existing techniques to control termination of partial deduction and (positive) supercompilation. At the local level, they mainly follow the approach of Sørensen and Glück [1995], which relies on *well-quasi orderings* to ensuring termination when constructing the so-called *partial process trees*. At the global level, the construction borrows some ideas from the method to ensure global termination of partial deduction of Martens and Gallagher [1995]. The main differences with respect to Martens and Gallagher [1995] are the following:

1. the use of a well-quasi ordering instead of a well-founded order to control termination, and

2. not to register descendency relationships among expressions at the global level: states are represented by means of sequences of terms rather than adapting the tree-like structures by Martens and Gallagher [1995]. This simplifies the formulation and permits to focus on the singularities of the method.

A commonly used tool for proving termination properties is based on the intuitive notion of orderings in which a term that is "syntactically simpler" than another term is smaller than the other. The following definition extends the homeomorphic embedding ("syntactically simpler") relation [Dershowitz and Jouannaud, 1990] to nonground terms. Variants of this relation are used in termination proofs for term-rewriting systems [Dershowitz and Jouannaud, 1991] and for ensuring local termination of partial deduction [Bol, 1993]. After it was taken up in Sørensen and Glück [1995], many recent works have also adopted the use of (different variants) of the embedding ordering to avoid infinite computations during partial evaluation and supercompilation (e.g., see [Alpuente *et al.*, 1998b, 1997], [Glück *et al.*, 1996], [Leuschel and Martens, 1996], [Leuschel *et al.*, 1998], [Turchin, 1996]).

**Definition 50 (embedding relation)** *The homeomorphic embedding relation $\trianglelefteq$ on terms in $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ is defined as the smallest relation satisfying $x \trianglelefteq y$ for all $x, y \in \mathcal{X}$, and $s = f(s_1, \ldots, s_m) \trianglelefteq g(t_1, \ldots, t_n) = t$, if and only if*

1. *$f = g$ (with $m = n$) and $s_i \trianglelefteq t_i$ for all $i = 1, \ldots, n$ or*
2. *$s \trianglelefteq t_j$, for some $j$, $1 \leq j \leq n$.*

Roughly speaking, we say that $s \trianglelefteq t$ if $s$ may be obtained from $t$ by deletion of operators. For example, $\surd\surd(u \times (u + v)) \trianglelefteq (w \times \surd\underline{\surd\surd}((\surd\underline{u} + \surd u)\underline{\times}(\surd\underline{u+}\surd\underline{v})))$. The following result is an easy consequence of Kruskal's Tree Theorem.

**Theorem 51** *[Alpuente et al., 1998b] The embedding relation $\trianglelefteq$ is a well-quasi ordering of the set $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ for finite $\Sigma = \mathcal{C} \cup \mathcal{F}$, that is, $\trianglelefteq$ is a quasi-order (i.e., a transitive and reflexive binary relation) and, for any infinite sequence of terms $t_1, t_2, \ldots$ with a finite number of operators, there exist $j, k$ with $j < k$ and $t_j \trianglelefteq t_k$ (i.e., the sequence is self-embedding).*

The embedding relation $\trianglelefteq$ will be used in Section 5.3.1 to define an unfolding rule which guarantees local termination, i.e., a condition that ensures that narrowing trees are not expanded infinitely in depth. In Section 5.3.2, an abstraction operator that guarantees global termination and which is also based on the ordering $\trianglelefteq$ is presented.

### 5.3.1 Local Termination

In this section, we present an unfolding rule which attempts to maximize unfolding while retaining termination. The strategy is based on the use of the embedding ordering to stop narrowing derivations. This is simple but less crude than imposing ad-hoc depth-bounds and still guarantees finite unfolding in all cases.

The following criterion makes use of the embedding relation in a constructive way to produce finite narrowing trees. In order to avoid an infinite sequence of "diverging" calls, each narrowing redex of the current goal is compared with the selected redexes in the ancestor goals of the same derivation, and the narrowing tree is expanded under the constraints imposed by the comparison. When the compared calls are in the embedding relation, the derivation is stopped. The following notations are needed:

**Definition 52 (comparable terms)** *Let $s$ and $t$ be terms. We say that $s$ and $t$ are comparable, written $comparable(s,t)$, if and only if the outermost function symbol of $s$ and $t$ coincide.*

**Definition 53 (admissible derivation)** *Let $\mathcal{D}$ be a narrowing derivation $t_0 \stackrel{\varphi}{\leadsto}_{\theta} \ldots \stackrel{\varphi}{\leadsto}_{\theta_{n-1}} t_n$ in $\mathcal{R}$. We say that $\mathcal{D}$ is admissible if and only if it does not contain a pair of comparable redexes included in the embedding relation $\trianglelefteq$. Formally,*

$$\begin{array}{rl} admissible(\mathcal{D}) \Leftrightarrow & \forall i = 1,\ldots,n,\ \forall (u,R,\sigma) \in \varphi(t_i),\ \forall j = 0,\ldots,i-1. \\ & (comparable(t_{j|u_j}, t_{i|u}) \Rightarrow t_{j|u_j} \not\trianglelefteq t_{i|u}). \end{array}$$

To formulate the concrete unfolding rule, we also introduce the following preparatory definition.

**Definition 54 (nonembedding narrowing tree)** *Given a term $t_0$ and a program $\mathcal{R}$, we define the* nonembedding narrowing tree $\tau_{\varphi}^{\trianglelefteq}(t_0,\mathcal{R})$ *for $t_0$ in $\mathcal{R}$ as follows:*

$$\mathcal{D} = t_0 \stackrel{\varphi}{\leadsto}_{\theta_0} \ldots \stackrel{\varphi}{\leadsto}_{\theta_{n-1}} t_n \stackrel{\varphi}{\leadsto}_{\theta_n} t_{n+1} \in \tau_{\varphi}^{\trianglelefteq}(t_0,\mathcal{R})$$

*if the following conditions hold:*

1. *the derivation $(t_0 \stackrel{\varphi}{\leadsto}_{\theta_0} \ldots \stackrel{\varphi}{\leadsto}_{\theta_{n-1}} t_n)$ is admissible and*
2. *(a) the leaf $t_{n+1}$ is a constructor term ($\mathcal{D}$ is a successful derivation) or*
   *(b) the leaf $t_{n+1}$ is failed ($\mathcal{D}$ is a failing derivation) or*
   *(c) there exists a triple $(u,R,\sigma) \in \varphi(t_{n+1})$ and a number $i \in \{1,\ldots,n\}$ such that $t_{i|u_i}$ and $t_{n+1|u}$ are comparable, and $t_{i|u_i} \trianglelefteq t_{n+1|u}$ ($\mathcal{D}$ is incomplete and is cut off because there exists a risk of nontermination).*

Hence, derivations are stopped when they either fail, succeed, or the considered redexes satisfy the embedding ordering. Before illustrating the previous definition by means of a simple example, we retrieve the following result, which establishes the usefulness of nonembedding narrowing trees in ensuring local termination.

**Theorem 55** *[Alpuente et al., 1998b] For a program $\mathcal{R}$ and term $t$, $\tau_{\varphi}^{\trianglelefteq}(t,\mathcal{R})$ is a finite (possibly incomplete) narrowing tree for $t$ in $\mathcal{R}$ using $\stackrel{\varphi}{\leadsto}$.*

**Example 17** Consider the program `append`:

```
append [] ys      = ys
append (x : xs) ys = x : (append xs ys)
```

with initial term $\mathtt{append}\ (\mathtt{1{:}2{:}x_s})\ \mathtt{y_s}$. There exists the following infinite branch in the (unrestricted) narrowing tree:

$$\begin{array}{ll}\underline{\mathtt{append}\ (\mathtt{1{:}2{:}x_s})\ \mathtt{y_s}} & \leadsto_{id}\ \mathtt{1{:}}\underline{\mathtt{append}\ (\mathtt{2{:}x_s})\ \mathtt{y_s})} \\ & \leadsto_{id}\ \mathtt{1{:}2{:}}\underline{(\mathtt{append\ x_s\ y_s})} \\ & \leadsto_{\{\mathtt{x_s} \mapsto \mathtt{x'{:}x'_s}\}}\ \mathtt{1{:}2{:}x'{:}}\underline{(\mathtt{append\ x'_s\ y_s})} \\ & \leadsto_{\{\mathtt{x'_s} \mapsto \mathtt{x''{:}x''_s}\}}\ \cdots \end{array}$$

According to the definition of nonembedding narrowing tree, the development of this branch is stopped at the fourth goal, since the derivation

$$\mathtt{append}\ (\mathtt{1{:}2{:}x_s})\ \mathtt{y_s} \leadsto_{id} \mathtt{1{:}(append}\ (\mathtt{2{:}x_s})\ \mathtt{y_s}) \leadsto_{id} \mathtt{1{:}2{:}(append\ x_s\ y_s)}$$

is admissible, and the step:

$$\mathtt{1{:}2{:}}\underline{(\mathtt{append\ x_s\ y_s})} \leadsto_{\{\mathtt{x_s} \mapsto \mathtt{x'{:}x'_s}\}}\ \mathtt{1{:}2{:}x'{:}}\underline{(\mathtt{append\ x'_s\ y_s})}$$

fulfills the ordering, because $\mathtt{append\ x_s\ y_s} \trianglelefteq \mathtt{append\ x'_s\ y_s}$.

Finally, we provide an unfolding rule induced by the notion of nonembedding narrowing tree.

**Definition 56 (nonembedding unfolding rule)** *We define $U_{\varphi}^{\trianglelefteq}(s, \mathcal{R})$ as the set of resultants associated to the derivations in $\tau_{\varphi}^{\trianglelefteq}(s, \mathcal{R})$.*

Nontermination of the partial evaluation algorithm can be caused not only by the creation of an infinite narrowing tree but also by never reaching the closedness condition. In the following section we consider the issue of ensuring global termination.

### 5.3.2 Global Termination

In this section, we define a concrete abstraction operator which is a parameter of the generic partial evaluation algorithm. The operator uses the simple kind of structure that we introduced before (sequences of terms) which is manipulated in such a way that termination of the specialized algorithm is guaranteed and still provides the right amount of polyvariance which avoids losing too much precision despite the use of *msg*'s. For a more sophisticated and more expensive kind of tree-like structure which could improve the amount of specialization in some cases, see [Martens and Gallagher, 1995].

Upon each iteration of the algorithm, the current sequence of terms $q = (t_1, \ldots, t_n)$ is transformed in order to "cover" the terms which result from the partial evaluation of $q$ in $\mathcal{R}$. This transformation is done by means of the following abstraction operation *abstract*$^*$, which makes use of the notion of "most specific generalization."

**Definition 57 (most specific generalization)** *A* generalization *of a nonempty sequence of terms* $(t_1, \ldots, t_n)$ *is a pair* $\langle t, \{\theta_1, \ldots, \theta_n\}\rangle$ *such that, for all* $i = 1, \ldots, n$, $\theta_i(t) = t_i$. *The pair* $\langle t, \Theta\rangle$ *is the* most specific generalization (msg) *of a sequence of terms* $S$, *written* $msg(S)$, *if:*

- $\langle t, \Theta\rangle$ *is a generalization of* $S$ *and,*
- *for every other generalization* $\langle t', \Theta'\rangle$ *of* $S$, $t'$ *is more general than* $t$.

The $msg$ of a set of terms is unique up to variable renaming [Lassez *et al.*, 1988].

**Definition 58 (nonembedding abstraction operator)** *Let* $q$ *be a finite sequence of terms and* $t$ *be a term. We define* $abstract^*(q,t)$ *inductively as follows:*

$$abstract^*(q,t) \;=\; \begin{cases} q & \textit{if } t \in \mathcal{X} \\ abstract^*(q, \{t_1, \ldots, t_n\}) & \textit{if } t = c(t_1, \ldots, t_n),\ c \in \mathcal{C},\ n \geq 0 \\ abs\_call(q,t) & \textit{if } t = f(t_1, \ldots, t_n),\ f \in \mathcal{F},\ n \geq 0 \end{cases}$$

*where applying* $abstract^*$ *to a sequence of terms is simply defined by*

$$abstract^*(q, \{t_1, \ldots, t_n\}) = abstract^*(\ldots abstract^*(q, t_1), \ldots, t_n)$$

*with* $n \geq 1$, *and* $abstract^*(q, \emptyset) = q$. *The auxiliary function* $abs\_call$ *is defined by*

- $abs\_call(nil, t) = t$
- $abs\_call(q, t) =$

$$\begin{cases} (q,t) & \textit{if } \not\exists s \in S.\ (comparable(s,t) \wedge s \trianglelefteq t) \\ abstract^*((q_1, \ldots, q_n), S) & \\ & \textit{if } i \textit{ is the maximum } j \in \{1, \ldots, n\} \textit{ such that} \\ & (comparable(q_j,t) \wedge q_j \trianglelefteq t\}),\ \textit{and} \\ & \theta(q_j) = t \textit{ for some } \theta,\ \textit{where } S = \{s \mid x \mapsto s \in \theta\} \\ & \\ abstract^*((q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n), S) & \\ & \textit{if } i \textit{ is the maximum } j \in \{1, \ldots, n\} \textit{ such that} \\ & (comparable(q_j,t) \wedge q_j \trianglelefteq t\}),\ \textit{and} \\ & q_i \not\leq t,\ \textit{where } msg(q_i,t) = \langle w, \{\theta_1, \theta_2\}\rangle,\ \textit{and} \\ & S = \{w\} \cup \{s \mid x \mapsto s \in \theta_1 \textit{ or } x \mapsto s \in \theta_2\}. \end{cases}$$

Let us informally explain this definition. Given a sequence of terms $q$, in order to add a new term $t$ (headed by a defined function symbol), the function $abstract^*$ proceeds as follows:

- If $t$ does not embed any comparable term in $q$, then $t$ is simply added to $q$.

- If $t$ embeds several comparable terms of $q$, then it considers the rightmost one in the sequence, say $t'$, and distinguishes two cases:
  - if $t$ is an instance of $t'$ with substitution $\theta$, then it recursively attempts to add the terms in $\theta$ to $q$;
  - otherwise, the *msg* of $t'$ and $t$ is computed, say $\langle w, \{\theta_1, \theta_2\}\rangle$, and then it attempts to add $w$ as well as the terms in $\theta_1$ and $\theta_2$ to the set resulting from removing $t'$ from $q$.

We note that, in the latter case, if $t'$ were not removed from $q$, the abstraction operator could run into an infinite loop, as shown in the following example.

**Example 18** Let us consider the sequence of terms $q = (\texttt{f x x},\ \texttt{g x})$ and assume that *abs_call* is modified by not removing the rightmost term $q_i$ of the sequence $q$ after computing the *msg*. Then, a call of the form $abs\_call(q, \texttt{f y z}))$ would proceed as follows:

1. the *msg* of `f x x` and `f y z` is computed, since they are comparable, `f y z` is not an instance of `f x x`, and `f y z` embeds `f x x`;
2. then the call to $abstract^*(q, \{\texttt{f y z},\ \texttt{x}\})$ is done, where $\langle \texttt{f y z}, \{\{\texttt{y} \mapsto \texttt{x}, \texttt{z} \mapsto \texttt{x}\}, \epsilon\}\rangle$ is the *msg* of `f x x` and `f y z`;
3. finally, the call to $abstract^*$ reproduces the original call: $abs\_call(q, \texttt{f y z})$, thus entering into an infinite loop.

A similar example can be found in [Leuschel *et al.*, 1998], where an alternative solution to this problem is proposed; namely, the so-called *strict* homeomorphic embedding $\trianglelefteq^*$ is used which requires $s$ not to be a strict instance of $s'$ for $s \trianglelefteq^* s'$. Thus `f x y` does not strictly embed `f x x`, and then it is simply added to the structure which is used to store the specialized calls.

The loss of precision in the abstraction operator caused by the use of the *msg* is quite reasonable and is compensated by the simplicity of the resulting method. Also, although the list-like configurations can be considered poorer than the tree-like states of Martens and Gallagher [1995] and Sørensen and Glück [1995], they are commonly used and, in some cases, can avoid duplication of function definitions. The following example illustrates how the method presented in this chapter achieves both termination and specialization. The positive supercompiler of Glück and Sørensen [1994] and Sørensen *et al.* [1994] does not terminate on this example, due to the infinite generation of "fresh" calls which, because of the growing accumulating parameter, are not instances of any call that was obtained previously. The partial deduction procedure of Benkerimi and Hill [1993] and Benkerimi and Lloyd [1990] results in the same nontermination pattern for a logic programming version of this program. Most current methods (e.g., [Gallagher, 1993], [Glück *et al.*, 1996], Leuschel and Martens [1996], [Martens and Gallagher, 1995], and Sørensen and Glück [1995]) would instead terminate on this example.

**Example 19** Consider the following program $\mathcal{R}$, which checks whether a sequence is a palindrome by using a reversing function with accumulating parameter:

$$\begin{array}{rcl} \texttt{palindrome x} & = & \texttt{reverse x} \approx \texttt{x} \\ \texttt{reverse x} & = & \texttt{rev x []} \\ \texttt{rev [] y}_\texttt{s} & = & \texttt{y}_\texttt{s} \\ \texttt{rev (x : x}_\texttt{s}\texttt{) y}_\texttt{s} & = & \texttt{rev x}_\texttt{s}\texttt{ (x : y}_\texttt{s}\texttt{)} \end{array}$$

and consider the goal `palindrome (1:2:x)`. We follow the generic partial evaluation algorithm with the nonembedding unfolding rule of Definition 56 based on unrestricted narrowing and the abstraction operator of Definition 58. The initial set is

$$q_1 \;=\; \mathit{abstract}^*(\mathit{nil}, \{\texttt{palindrome (1:2:x}_\texttt{s}\texttt{)}\}) = (\texttt{palindrome (1:2:x}_\texttt{s}\texttt{)}).$$

The partial evaluation of $\texttt{palindrome (1:2:x}_\texttt{s}\texttt{)}$ in $\mathcal{R}$ is the program $\mathcal{R}^1$:

$$\texttt{palindrome (1:2:x:x}_\texttt{s}\texttt{)} = \texttt{rev x}_\texttt{s}\texttt{ (x:2:1:[])} \approx \texttt{(1:2:x:x}_\texttt{s}\texttt{)}$$

with $\mathcal{R}^1_{calls} = \{\texttt{rev x}_\texttt{s}\texttt{ (x:2:1:[])} \approx \texttt{(1:2:x:x}_\texttt{s}\texttt{)}\}$. Then we obtain

$$\begin{array}{rcl} q_2 & = & \mathit{abstract}^*((\texttt{palindrome (1:2:x}_\texttt{s}\texttt{)}), \mathcal{R}^1_{\texttt{calls}}) \\ & = & (\texttt{palindrome (1:2:x}_\texttt{s}\texttt{)},\ \texttt{rev x}_\texttt{s}\texttt{ (x:2:1:[])}). \end{array}$$

The partial evaluation of $\texttt{rev x}_\texttt{s}\texttt{ (x:2:1:[])}$ in $\mathcal{R}$ is the program $\mathcal{R}^2$:

$$\begin{array}{rcl} \texttt{rev [] (x:2:1:[])} & = & \texttt{(x:2:1:[])} \\ \texttt{rev (x}'\texttt{:x}'_\texttt{s}\texttt{) (x:2:1:[])} & = & \texttt{rev x}'_\texttt{s}\texttt{ (x}'\texttt{:x:2:1:[])} \end{array}$$

with $\mathcal{R}^2_{calls} = \{\texttt{x:2:1:[]},\ \texttt{rev x}'_\texttt{s}\texttt{ (x}'\texttt{:x:2:1:[])}\}$. Then we obtain

$$\begin{array}{rcl} q_3 & = & \mathit{abstract}^*((\texttt{palindrome (1:2:x}_\texttt{s}\texttt{)},\ \texttt{rev x}_\texttt{s}\texttt{ (x:2:1:[])}), \mathcal{R}^2_{calls}) \\ & = & (\texttt{palindrome (1:2:x}_\texttt{s}\texttt{)},\ \texttt{rev x}_\texttt{s}\texttt{ (x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}\texttt{)}). \end{array}$$

The partial evaluation of $\texttt{rev x}_\texttt{s}\texttt{ (x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}\texttt{)}$ in $\mathcal{R}$ is the program $\mathcal{R}^3$:

$$\begin{array}{rcl} \texttt{rev [] (x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}\texttt{)} & = & \texttt{x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s} \\ \texttt{rev (x:x}_\texttt{s}\texttt{) (x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}\texttt{)} & = & \texttt{rev x}_\texttt{s}\texttt{ (x:x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}\texttt{)} \end{array}$$

with $\mathcal{R}^3_{calls} = \{\texttt{x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s},\ \texttt{rev x}_\texttt{s}\texttt{ (x:x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}\texttt{)}\}$. Then we obtain

$$\begin{array}{rcl} q_4 & = & \mathit{abstract}^*((\texttt{palindrome (1:2:x}_\texttt{s}\texttt{)},\ \texttt{rev x}_\texttt{s}\texttt{ (x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}\texttt{)}), \mathcal{R}^3_{calls}) \\ & = & (\texttt{palindrome (1:2:x}_\texttt{s}\texttt{)}, \texttt{rev x}_\texttt{s}\texttt{ (x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}\texttt{)}) \;=\; q_3. \end{array}$$

Thus, given the sequence of terms $S' = (\texttt{palindrome}\,(\texttt{1:2:x}_\texttt{s}),\ \texttt{rev}\,\texttt{x}_\texttt{s}\,\texttt{y}_\texttt{s})$, the specialized program $\mathcal{R}'$ resulting from the partial evaluation of $\mathcal{R}$ with respect to $S'$ is:

$$\begin{array}{rcl}
\texttt{palindrome}\,(\texttt{1:2:x:x}_\texttt{s}) & = & \texttt{rev}\,\texttt{x}_\texttt{s}\,(\texttt{x:2:1:[]}) \approx \texttt{1:2:x:x}_\texttt{s} \\
\texttt{rev}\,[]\,(\texttt{x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}) & = & \texttt{x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s} \\
\texttt{rev}\,(\texttt{x :x}_\texttt{s})\,(\texttt{x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s}) & = & \texttt{rev}\,\texttt{x}_\texttt{s}\,(\texttt{x:x}_1\texttt{:x}_2\texttt{:x}_3\texttt{:y}_\texttt{s})
\end{array}$$

where we have saved some infeasible branches which end with $fail$ at specialization time. Note that all computations on the partially static structure have been performed. In the new partially evaluated program, the known elements of the list in the argument of `palindrome` are "passed on" to the list in the second argument of `rev`. The logical inferences needed to perform this, whose number $n$ is linear in the number of the known elements of the list, is done at partial evaluation time, which is thus more efficient than performing the $n$ logical inferences at runtime.

The following lemma states that $abstract^*$ is a correct instance of the generic notion of abstraction operator.

**Lemma 59** *[Alpuente et al., 1998b] The function abstract$^*$ is an abstraction operator.*

The following theorem establishes the (local as well as global) termination of the considered instance of the generic partial evaluation algorithm.

**Theorem 60** *[Alpuente et al., 1998b] The narrowing-driven partial evaluation algorithm terminates for the nonembedding unfolding rule (Definition 56) and abstraction operator (Definition 58).*

The last example in this section illustrates the fact that the method presented so far can also eliminate intermediate data structures and turn multiple-pass programs into one-pass programs, as the deforestation method of Wadler [1990] and the positive supercompiler of Sørensen [1994] do. Standard partial evaluation does not succeed on this example [Glück and Sørensen, 1996]. To achieve a similar effect in partial deduction, a special treatment of conjunctions as in conjunctive partial deduction is needed (see [Leuschel *et al.*, 1996]). We introduce a similar extension in Chapter 6.

**Example 20** Consider again the program `append` of Example 17 with initial goal

$\texttt{append}\,(\texttt{append}\,\texttt{x}_\texttt{s}\,\texttt{y}_\texttt{s})\,\texttt{z}_\texttt{s}$

This goal appends three lists by appending the first two, yielding an intermediate list, and then appending to this auxiliary list the last one. We evaluate the goal by using needed narrowing. Starting with the sequence:

$q = (\texttt{append}\,(\texttt{append}\,\texttt{x}_\texttt{s}\,\texttt{y}_\texttt{s})\,\texttt{z}_\texttt{s})$

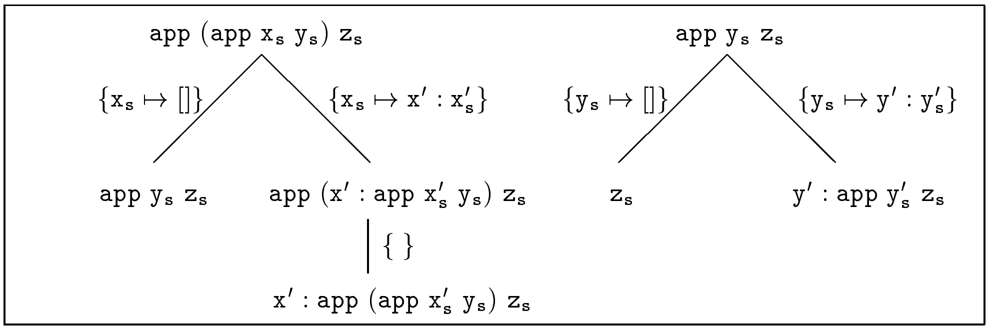


Figure 5.1: Partial needed narrowing trees for $\{\mathtt{app}\ (\mathtt{app}\ x_s\ y_s)\ z_s),\ \mathtt{app}\ x_s\ y_s\}$

we compute the trees depicted in Figure 5.1 for the sequence of terms:

$$q' = (\mathtt{append}\ (\mathtt{append}\ x_s\ y_s)\ z_s,\ \mathtt{append}\ x_s\ y_s)$$

Note that `append` has been abbreviated to `app` in the picture. Then we get the following residual program $\mathcal{R}'$:

$$\begin{array}{l}
\mathtt{append}\ (\mathtt{append}\ []\ y_s)\ z_s = \mathtt{append}\ y_s\ z_s \\
\mathtt{append}\ (\mathtt{append}\ (x : x_s)\ y_s)\ z_s = x : (\mathtt{append}\ (\mathtt{append}\ x_s\ y_s)\ z_s) \\
\mathtt{append}\ []\ z_s = z_s \\
\mathtt{append}\ (y : y_s)\ z_s = y : \mathtt{append}\ y_s\ z_s
\end{array}$$

It is worth noting that, in the previous example, the resulting sequence of terms

$$(\mathtt{append}\ (\mathtt{append}\ x_s\ y_s)\ z_s, \mathtt{append}\ x_s\ y_s)$$

in $q'$ is not independent. This example illustrates the need for an extra renaming phase able to produce an independent sequence of terms such as $(\mathtt{app\_3}\ x_s\ y_s\ z_s,\ \mathtt{app\_2}\ x_s\ y_s)$ and associated specialized program

$$\begin{array}{l}
\mathtt{app_3}\ []\ y_s\ z_s = \mathtt{app_2}\ y_s\ z_s \\
\mathtt{app_3}\ (x : x_s)\ y_s\ z_s = x : (\mathtt{app_3}\ x_s\ y_s\ z_s) \\
\mathtt{app_2}\ []\ z_s = z_s \\
\mathtt{app_2}\ (y : y_s)\ z_s = y : (\mathtt{app_2}\ y_s\ z_s)
\end{array}$$

which has the same computed answers as the original program `append` for the query $\mathtt{app\_3}\ x_s\ y_s\ z_s$. Here we would like to remark that the postprocessing renaming transformation of Chapter 3 automatically obtains the desired optimal program.

## 5.4 Conclusions

The NPE framework summarized in this chapter constitutes a semantics-based partial evaluation scheme for functional logic programs which is able to automatically improve program performance without changing the computational meaning and which still guarantees termination.

The generic method is able to produce *polygenetic* specializations [Albert *et al.*, 1998a], i.e., it is able to extract specialized definitions which combine several function definitions of the original program (see, e.g., [Glück and Sørensen, 1996]). That means that NPE has the same potential for specialization as conjunctive partial deduction or positive supercompilation within the considered paradigm (see Section 3.3).

Unfortunately, the general method presented in this chapter is not endowed with the capability to deal with the additional features of modern declarative languages (e.g. conjunctions, equations, higher-order functions, constraints, program annotations, calls to external functions, etc.). Indeed, the use of these additional features may encumber the nature of the specialization problems and it often turns out that a specific treatment is required for specializing these kind of expressions.

In the next chapter, we extend the NPE algorithm by incorporating a specific treatment for equations and conjunctions. This gives rise to the partial evaluator INDY, described in Appendix A. Concerning the remaining features of modern multi-paradigm languages, in Part III, we develop a novel technique based on the use of an abstract representation (cf. Chapter 7). This abstract representation is designed such that high-level programs can be automatically translated to it. This technique is then smoothly integrated into the general scheme presented in this chapter. The new scheme allows us to define a partial evaluator for a truly lazy functional logic language (which will be described in Chapter 8).

# Chapter 6

# Improving Control

This chapter extends the partial evaluation scheme presented in the former chapter by formulating concrete control options that effectively handle *primitive* function symbols in functional logic languages. In particular, we introduce a well-balanced, dynamic unfolding rule and a novel abstraction operator which highly improve the specialization of the NPE method. Our method is parametric w.r.t. the narrowing strategy, although we use a needed narrowing strategy to experimentally evaluate the new control options. An instance of the general formulation for a lazy narrowing strategy appeared in [Albert *et al.*, 1998b; Julián, 2000].

## 6.1 Introduction

The inclusion of *primitive* function symbols (i.e., functions which have not been defined in the user's code such as those functions defined in the prelude, etc.) in input calls poses specific control problems that have not been previously addressed. Of course, the generic method presented in Chapter 5 may allow one to deal with equations and conjunctions during specialization by simply considering the equality and conjunction operators as constructor symbols of the language. Unfortunately, the use of primitive functions may encumber the nature of the specialization problems and it often turns out that some form of *tupling* (as defined in [Pettorossi and Proietti, 1996b] for logic programs) is required for specializing expressions which contain conjunctions. The NPE algorithm of Chapter 5 does not incorporate a specific treatment for such primitive symbols, which depletes many opportunities for reaching the closedness condition and forces the method to dramatically generalize calls, thus giving up the intended specialization and resulting in monogenetic specializations in many interesting cases (see forthcoming Example 21).

At the local level of control, the reduction of the elements in a conjunction can

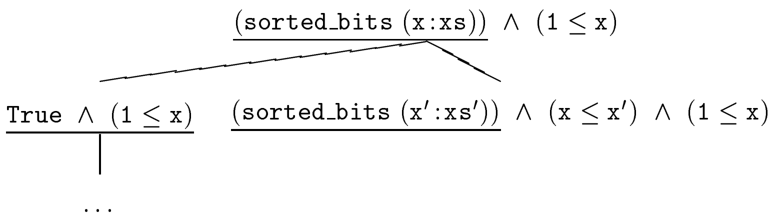


Figure 6.1: Incomplete narrowing tree for (sorted_bits (x:xs)) $\wedge$ (1 $\leq$ x)

be made "don't care" nondeterministically, and the order in which elements are chosen during the construction of the local narrowing trees is crucial to achieving good specialization. Indeed, a naïve selection may lose all specialization. Some kind of dynamic selection strategy which keeps track of the ancestors reduced in the same derivation is necessary to significantly speed-up execution. At the global level of control, enhancing the general NPE algorithm requires particular techniques for carefully splitting complex terms, while still being able to keep the terms which may communicate data structures as close together as possible. This is not always possible because ensuring termination sometimes forces splitting at undesirable positions. In order to improve the specialization of the NPE method, we introduce a well-balanced, dynamic unfolding rule and a novel abstraction operator that do not depend on the narrowing strategy. These control operators allow us to tune the specialization algorithm to handle conjunctions (and other expressions containing primitive functions, such as conditionals and strict equalities) in a natural way, which provides for polygenetic specialization without any ad-hoc artifice. Our method is applicable to modern functional logic languages such as Babel [Moreno-Navarro and Rodríguez-Artalejo, 1992], Curry [Hanus *et al.*, 1995] and Toy [López-Fraguas and Sánchez-Hernández, 1999].

## 6.2 Motivation

In the original NPE framework, no distinction is made between primitive and defined function symbols during specialization. For instance, a conjunction $b_1 \wedge b_2$ is considered as a single entity when checking whether it is covered by the set of specialized calls. This commonly implies a drastic generalization of the involved calls, which may cause losing all specialization. The following example illustrates this point and motivates the remainder of this chapter.

**Example 21** Let us consider the program excerpt:

$$\begin{array}{l}
\texttt{sorted_bits}\ (\texttt{x}:[\,]) \ = \ \texttt{True} \\
\texttt{sorted_bits}\ (\texttt{x}_1:\texttt{x}_2:\texttt{xs}) \ = \ (\texttt{sorted_bits}\ \texttt{x}_2:\texttt{xs}) \ \wedge \ (\texttt{x}_1 \leq \texttt{x}_2) \\
\texttt{0} \leq \texttt{0} \ = \ \texttt{True} \\
\texttt{0} \leq \texttt{1} \ = \ \texttt{True} \\
\texttt{1} \leq \texttt{1} \ = \ \texttt{True}
\end{array}$$

and the call $(\texttt{sorted\_bits}\ (\texttt{x}:\texttt{xs})) \wedge (\texttt{1} \leq \texttt{x})$. The narrowing tree depicted[1] in Fig. 6.1 is built up by using the *nonembedding* unfolding rule of Chapter 5, which expands derivations while new redexes are not "greater" (with the *homeomorphic embedding ordering*) than previous, *comparable* redexes in the branch. From this tree, we can identify two main weaknesses of the plain NPE algorithm:

- The rightmost branch stops because the leftmost redex $\texttt{sorted\_bits}\ (\texttt{x}':\texttt{xs}')$ of the leaf "embeds" the redex $\texttt{sorted\_bits}\ (\texttt{x} : \texttt{xs})$ selected in the previous step, even if no reductions have been performed on the other elements of the conjunction, which does not seem very equitable.

- According to the distinction between local and global control, when local trees are cut off and control passes back to the global level, the algorithm then attempts to prove that the call

  $$(\texttt{sorted\_bits}\ (\texttt{x}':\texttt{xs}')) \ \wedge \ (\texttt{x} \leq \texttt{x}') \ \wedge \ (\texttt{1} \leq \texttt{x})$$

  in the leaf of the tree is closed w.r.t. the initial call (folding):

  $$(\texttt{sorted\_bits}\ (\texttt{x}:\texttt{xs})) \ \wedge \ (\texttt{1} \leq \texttt{x})$$

  Otherwise, a generalization of comparable calls is performed. Since the involved calls are comparable and the call

  $$(\texttt{sorted\_bits}\ (\texttt{x}':\texttt{xs}')) \ \wedge \ (\texttt{x} \leq \texttt{x}') \ \wedge \ (\texttt{1} \leq \texttt{x})$$

  in the leaf of the tree embeds (but it is not closed w.r.t.) the specialized call

  $$(\texttt{sorted\_bits}\ (\texttt{x}:\texttt{xs})) \ \wedge \ (\texttt{1} \leq \texttt{x})$$

  the msg $(\texttt{sorted\_bits}\ (\texttt{x} : \texttt{xs})) \ \wedge \ \texttt{z}$ is computed, which gives up the intended specialization.

[1] We assume a needed narrowing strategy with a fixed left-to-right selection of components within conjunctions.

The first drawback pointed out in this example motivates the definition of more sophisticated unfolding rules which are able to achieve a *balanced* evaluation of the given expression by narrowing appropriate redexes (e.g., by using some kind of dynamic scheduling strategy which takes into account the ancestors narrowed in the same branch). The second drawback suggests the definition of a more flexible abstraction operator which is able to automatically split complex terms before attempting folding or generalization. In the following, we refine the framework of Chapter 5 in order to overcome these problems.

## 6.3 Improved Control Issues

We consider the two levels of control separately. As for local control, in Section 6.3.1, we formalize an unfolding strategy which is specifically designed for "conjunctive specialization". As for global control, a specific treatment of the following primitive function symbols:

"$\approx$" : equality

"$\wedge$" : conjunction

"$\Rightarrow$" : conditional

is introduced in Section 6.3.2 which produces more effective and powerful, polygenetic specializations, as compared to classical NPE.

### 6.3.1 Local Control

The unfolding rule described in Chapter 5 simply exploits the redexes selected by the narrowing strategy $\varphi$ (using a narrowing strategy with a fixed, static selection rule which determines the next conjunct to be reduced) whenever none of them embeds a previous (comparable) redex of the same branch. The following example reveals that this strategy is not elaborated enough for specializing calls which may contain primitive symbols like conjunctions.

**Example 22** Consider again function app to concatenate two lists:

$$\begin{array}{rcl} \mathtt{app}\,[\,]\,\mathtt{y} & = & \mathtt{y} \\ \mathtt{app}\,(\mathtt{x} : \mathtt{xs})\,\mathtt{y} & = & \mathtt{x} :\ (\mathtt{app}\,\mathtt{xs}\,\mathtt{y}) \end{array}$$

with the input goal "$\mathtt{app\ x\ y} \approx \mathtt{w} \wedge \mathtt{app\ w\ z} \approx \mathtt{r}$". Using the nonembedding unfolding rule of Chapter 5, we obtain the tree depicted[2] in Fig. 6.2 (using a fixed left-to-right selection rule for conjunctions). From this local tree, no appropriate specialized

[2]We adopt the standard optimization which makes use of the built-in unification to solve strict equalities $s \approx t$ (provided that only constructor bindings are produced), which avoids the creation of superfluous choice points.

$$\underline{(\mathtt{app\ x\ y}} \approx \mathtt{w})\ \wedge\ (\mathtt{app\ w\ z} \approx \mathtt{r})$$

$$\vdots \qquad \underline{(\mathtt{x}' : (\mathtt{app\ xs}'\ \mathtt{y}) \approx \mathtt{w})}\ \wedge\ (\mathtt{app\ w\ z} \approx \mathtt{r})$$

$$\underline{(\mathtt{x}' \approx \mathtt{w}')}\ \wedge\ (\mathtt{app\ xs}'\ \mathtt{y} \approx \mathtt{ws}')\ \wedge\ (\mathtt{app}\ (\mathtt{w}' : \mathtt{ws}')\ \mathtt{z} \approx \mathtt{r})$$

$$\underline{\mathtt{True}\ \wedge\ (\mathtt{app\ xs}'\ \mathtt{y} \approx \mathtt{ws}')\ \wedge\ (\mathtt{app}\ (\mathtt{w}' : \mathtt{ws}')\ \mathtt{z} \approx \mathtt{r})}$$

$$\underline{(\mathtt{app\ xs}'\ \mathtt{y}} \approx \mathtt{ws}')\ \wedge\ (\mathtt{app}\ (\mathtt{w}' : \mathtt{ws}')\ \mathtt{z} \approx \mathtt{r})$$

Figure 6.2: Naïve local control for $\mathtt{app}(\mathtt{x},\mathtt{y}) \approx \mathtt{w}\ \wedge\ \mathtt{app}(\mathtt{w},\mathtt{z}) \approx \mathtt{r}$

definition for the initial goal can be obtained, since the leaf cannot be folded into the input call of the root of the tree and generalization is required (which causes losing all specialization, as in Example 21).

We note that the NPE method of Chapter 5 succeeds with this example when the specialized call is written as a nested expression `app (app x y) z`, as illustrated in Example 20. This is because exploiting the nesting capability of the functional syntax allows us to transform the original tupling[3] problem illustrated by Example 22 into a simpler, deforestation problem, which is easily solved by the original NPE method. Actually, the original method was proved to be specially well-suited for performing deforestation, while it was not able to perform all tupling automatically (see [Alpuente *et al.*, 1998b]).

Now we are ready to introduce a refined, dynamic unfolding rule which attempts to achieve a fair, balanced evaluation of the complete input term, rather than a deeper evaluation of some given subterm. This can avoid premature cutting of derivations and, in some cases, produces better specialization than using a blind selection rule. This novel concrete unfolding rule dynamically selects the positions to be reduced within a given conjunction, by exploiting some dependency information between redexes gathered along the derivation. Our definition is similar to that in [Glück *et al.*, 1996] but is more general, since it is aimed at solving the general problem of specializing complex terms containing (different kinds of) possibly nested operators. The following notion of *dependent positions* is used to trace the *functional dependencies* between redexes of the local narrowing tree being constructed by partial evaluation.

**Definition 61 (dependent positions)** *Let $D = (s \leadsto_{p,l\to r,\sigma} t)$ be a narrowing step.*

[3]Here we refer to tupling of logic programs, which subsumes both deforestation and tupling of functional programs [Pettorossi and Proietti, 1996b].

*The set of* dependent positions *of a position $q$ of $s$ by $D$, written $q\backslash\backslash D$, is:*

$$q\backslash\backslash D = \begin{cases} \{q.u \mid u \in \mathcal{FP}os(r) \wedge root(r|_u) \notin \mathcal{C}\} & \textit{if } q = p \\ \{q\} & \textit{if } q \not\leq p \\ \{p.u'.v \mid r|_{u'} = x\} & \textit{if } q = p.u.v \textit{ and } l|_u = x \in \mathcal{X} \end{cases}$$

*This notion can be naturally lifted to narrowing derivations.*

The notion of dependency for terms stems directly from the corresponding notion for positions. Note that the above definition is a strict extension of the standard notion of *descendant* in functional programming. Intuitively, the descendants of the subterm $s|_q$ are computed as follows: underline the root symbol of $s|_q$ and perform the narrowing step $s \leadsto t$. Then, every subterm of $t$ with and underlined root symbol is a descendant of $s|_q$ [Klop and Middeldorp, 1991]. Then, a position $q'$ of $t$ depends on a position $q$ of $s$ (by $D$) if $q'$ is a descendant of $q$ (second and third cases), or if the position $q'$ has been introduced by the right-hand side of the rule applied in the reduction of the former position $q$ and it addresses a subterm headed by a defined function symbol (first case). The first case reflects the functional dependencies between positions which are brought by the application of program rules, while the second and third cases specify how positions which are not narrowed are carried up along computation steps. Note that this notion is an extension of the standard partial deduction concept of (covering) ancestor to the functional logic framework. By abuse, we also say that the term addressed by $q$ is an *ancestor* of the term addressed by $q'$ in $D$. If $s$ is an ancestor of $t$ and $root(s) = root(t)$, we say that $s$ is a *comparable ancestor* of $t$ in $D$.

**Example 23** Consider the needed narrowing tree depicted in Fig. 6.2. The call `app x y` in the initial goal is an ancestor of the call `app xs' y` in the leaf (since the call to `app` has been brought by the right-hand side of the second rule of `app`), whereas it is not an ancestor of the rightmost call `app (w' : ws') z` of the leaf (whose ancestor in the initial goal is the call `app w z`).

Now we describe the way in which the dynamic selection rule is defined. We use an auxiliary function *select* which returns the set of positions to be unfolded in the next iteration. The symbol $\perp$ is used to denote that the evaluation must stop. At each iteration, a generic (strongly complete) narrowing strategy $\varphi$ computes a position which addresses the next subterm to be inspected. We distinguish the following cases depending on the selected subterm:

1. variables are not unfolded;
2. if the subterm is a conjunction, we recursively select positions within both conjuncts; then, we non-deterministically return the set of positions of the branch

which does not contain the symbol $\perp$ and which is not empty; if both sets contain the symbol $\perp$, then we return $\{\perp\}$; and if both sets are empty, we return the empty set;

3. otherwise, we proceed to select positions of all the arguments of the expression and return the set of all selected positions; if one of the sets contains the symbol $\perp$, then we return $\{\perp\}$;

**Definition 62 (dynamic narrowing selection strategy)**
*Let $D = (t_0 \leadsto t_1 \leadsto \ldots \leadsto t_n)$, $n \geq 0$ be a narrowing derivation. We define the dynamic selection rule $\varphi_{dynamic}$ as follows:*

$$\varphi_{dynamic}(t_n, D) = select(t_n, \Lambda, D)$$

*where the auxiliary function select is:*

$$\begin{array}{ll} select(t,p,D) = & \textit{if } (p,R,\sigma) \in \varphi(t) \\ & \quad \textit{then if } dependency\_clash(t|_p, D) \textit{ then } \{\perp\} \textit{ else } \{p\} \\ & \quad \textit{else case } t|_p \textit{ of} \\ & \qquad \begin{array}{ll} x \in \mathcal{X}: & \emptyset \\ s_1 \wedge s_2: & \textit{let } O_i = select(t,p.i,D),\ i \in \{1,2\}, \textit{ in} \\ & [\textit{if } \exists i.\ (\perp \notin O_i \wedge O_i \neq \emptyset) \textit{ then } O_i \\ & \textit{else if } (O_1 = O_2 = \emptyset) \textit{ then } \emptyset \textit{ else } \{\perp\}] \\ \textit{otherwise:} & \textit{let } t|_p = f(s_1,\ldots,s_n) \textit{ and} \\ & O_{args} = \bigcup_{i=1}^{n} select(t,p.i,D) \textit{ in} \\ & [\textit{if } \perp \in O_{args} \textit{ then } \{\perp\} \textit{ else } O_{args}] \end{array} \end{array}$$

*where $dependency\_clash(t, D)$ is a generic Boolean function that looks at the ancestors of $t$ in $D$ according to some specific criterium to determine whether there is a risk of nontermination.*

Let us note that, in the remainder of this section, we will investigate two different $dependency\_clash(t, D)$ tests: one which holds whenever there is a comparable ancestor of the selected redex $t$ in $D$; the other consists on additionally checking homeomorphic embedding on comparable ancestors.

Roughly speaking, the dynamic narrowing selection strategy recurs over the structure of the goal and determines the set of positions to be unfolded by a "don't care" selection within each conjunction of exactly one of the components (among those that do not incur into a *dependency_clash*). We introduce a dynamic unfolding rule $U_{dynamic}(t, \mathcal{R})$ which simply expands narrowing trees according to the dynamic narrowing strategy $\varphi_{dynamic}$.[4] The "mark" $\perp$ of Definition 62 is used as a whistle to

[4] All scheduling policies are admissible for an interpreter implementing needed narrowing.

$$(\underline{\texttt{app x y}} \approx \texttt{w}) \;\wedge\; (\texttt{app w z} \approx \texttt{r})$$

$$\vdots$$

$$\underline{\texttt{x}' : (\texttt{app xs}'\ \texttt{y}) \approx \texttt{w}} \;\wedge\; (\texttt{app x z} \approx \texttt{r})$$

$$(\texttt{x}' \approx \texttt{w}') \;\wedge\; (\texttt{app xs}'\ \texttt{y} \approx \texttt{ws}') \;\wedge\; (\underline{\texttt{app}\ (\texttt{w}' : \texttt{ws}')\ \texttt{z}} \approx \texttt{r})$$

$$(\texttt{x}' \approx \texttt{w}') \;\wedge\; (\texttt{app xs}'\ \texttt{y} \approx \texttt{ws}') \;\wedge\; (\underline{\texttt{w}' : (\texttt{app ws}'\ \texttt{z}) \approx \texttt{r})}$$

$$(\texttt{x}' \approx \texttt{w}') \;\wedge\; (\texttt{app xs}'\ \texttt{y} \approx \texttt{ws}') \;\wedge\; (\texttt{w}' \approx \texttt{r}') \;\wedge\; (\texttt{app ws}'\ \texttt{z} \approx \texttt{rs}')$$

Figure 6.3: Improved local control for $(\texttt{app x y} \approx \texttt{w}) \wedge (\texttt{app w z} \approx \texttt{r})$

warn us that the derivation must be cut off since it runs into a *dependency_clash*. That is, each branch $D$ of the tree is stopped whenever $\varphi_{dynamic}(t, D) = \{\bot\}$ or the term $t$ (of the leaf) is in head normal form.

**Example 24** Consider again the program and goal of Example 22. Using the dynamic unfolding rule $U_{dynamic}$, we get the tree depicted in Fig. 6.3. From this tree, an optimal (recursive) specialized definition for the initial call can be derived, provided there is a suitable splitting mechanism to extract, from the leaf of the tree, an appropriate conjunct such as $(\texttt{app xs}'\ \texttt{y} \approx \texttt{ws}') \wedge (\texttt{app ws}'\ \texttt{z} \approx \texttt{rs}')$, which is covered by the initial call in the root of the tree (see Example 26).

The following result holds for those narrowing strategies which are *strong complete*, i.e., given a conjunctive expression of the form $t_1$ & $t_2$ one can "don't care" nondeterministically narrow $t_1$ or $t_2$. For instance, the termination of the expansion of needed narrowing trees using $U_{dynamic}$ is ensured, since needed narrowing is strong complete (see Theorem 8).

The proof ensures termination for two concrete unfolding rules (based on a strong complete narrowing strategy) which use one of these distinct kinds of *dependency_clash* tests:

*emb_redex* : this unfolding rule uses the dynamic narrowing selection strategy of Definition 62 to determine the set of positions to be unfolded by a "don't care" selection within a conjunction. It implements the *dependency_clash* using homeomorphic embedding on comparable ancestors of selected redexes to ensure finiteness;

*comp_redex* : it differs from *emb_redex* in that *comp_redex* uses a simpler definition of *dependency_clash* based on comparable ancestors of redexes as a whistle.

**Lemma 63** *Let $\mathcal{R}$ be a program and $t$ be a goal. Then $U_{dynamic}(t, \mathcal{R})$ produces a finite (possibly incomplete) narrowing tree for $t$ in $\mathcal{R}$ using a strong complete narrowing strategy.*

*Proof.* (sketch)
The unfolding rule $U_{dynamic}(t, \mathcal{R})$ expands local trees according to the dynamic narrowing strategy. $\varphi_{dynamic}(t, \mathcal{R})$ produces a narrowing tree since, at each step, all "don't know" choices are considered and only "don't care" choices are disregarded (except for the selected one).

Finiteness of the tree is ensured by the *dependency_clash*$(t|_p, \mathcal{D})$ test. We show that the two different kinds of tests, considered by the unfolding rules *comp_redex* and *emb_redex* are *dependency_clash* tests.

1. There exists a comparable ancestor of the selected redex in $\mathcal{D}$.

   We proceed by contradiction. Assume that we can build an infinite narrowing tree for $t$ in $\mathcal{R}$ using $U_{dynamic}$. Since the term t is finite, the number of redexes of $t$ is also finite and the narrowing tree for $t$ in $\mathcal{R}$ is finitely branched. Then, by Köning's lemma, there is an infinite branch. Now the result follows trivially, since there is no infinite branch which contains dependent redexes headed by different function symbols unless the signature is infinite.

2. Homeomorphic embedding test on comparable ancestors of the selected redex.

   By contradiction. Similarly to the above case, assume that there is an infinite branch in the tree. Since both the signature and the term $t$ are finite, there is a finite number of subsequences of depending comparable subterms in the branch. Therefore, at least one of these subsequences is infinite. Since the subsequences of depending comparable subterms constructed by *emb_redex* are nonembeding subsequences (see Definition 72), we have an infinite subsequence of terms $t_1, t_2, \ldots$ such that no term $t_j$ embeds a preceding term $t_i$, $i < j$, which contradicts Kruskal's theorem.

□

### 6.3.2 Global Control

In the presence of primitive functions like "$\wedge$" or "$\approx$", using an abstraction operator which respects the structure of the terms (as in Chapter 5) is not very effective, since the generalization of two conjunctions (resp. equations) might be a term of the form $x \wedge y$ (resp. $x \approx y$) where $x$ and $y$ are fresh variables in most cases, as we showed before (see Examples 21 and 22). The drastical solution of decomposing the term into subterms containing just one function call can avoid the problem, but has the negative consequence of losing nearly all specialization (resulting in a monogenetic specialization method). In this section, we introduce a more concerned abstraction operator which is inspired by the partitioning techniques of conjunctive partial deduction [Glück *et al.*, 1996; Leuschel *et al.*, 1996]. Global control in [Glück *et al.*, 1996; Leuschel *et al.*, 1996] is based on *global trees*, as introduced in [Leuschel and Martens, 1996]. Although this technique can improve specialization in many respects, here we prefer to follow a less sophisticated but inexpensive control strategy which is also based on sequences.

First, let us introduce the notion of *best matching term*. During the abstraction process, terms may require being split in order to find the best way of continuing the specialization process without risking nontermination. The following notion of best matching terms, which is aimed at avoiding loss of specialization due to generalization, is a proper generalization of the notion of *best matching conjunction* in [Glück *et al.*, 1996].

**Definition 64 (best matching terms)** *Let $S = \{s_1, \ldots, s_n\}$ be a set of terms, $t$ a term, and consider the set of terms $W = \{w_i \mid \langle w_i, \{\theta_{i1}, \theta_{i2}\}\rangle = msg(\{s_i, t\}), i = 1, \ldots, n\}$. The* best matching terms *$BMT(S,t)$ for $t$ in $S$ are those terms $s_j \in S$ such that the corresponding $w_j$ in $W$ is a minimally general element.*[5]

The following example illustrates the above definition.

**Example 25** Let $S = \{\texttt{f (g x)},\ \texttt{f (g A)},\ \texttt{f z}\}$ and $t = \texttt{f (g B)}$. To compute the best matching terms for $t$ in $S$, we first consider the set:

$$\begin{array}{rcl} W & = & \{msg(\{\texttt{f (g x)}, \texttt{f (g B)}\}), msg(\{\texttt{f (g A)}, \texttt{f (g B)}\}), msg(\{\texttt{f z}, \texttt{f (g B)}\})\} \\ & = & \{\texttt{f (g x)}, \texttt{f (g y)}, \texttt{f z}\}. \end{array}$$

Now, the minimally general elements of $W$ are $\texttt{f (g x)}$ and $\texttt{f (g y)}$, and thus we have: $BMT(S,t) = \{\texttt{f (g x)}, \texttt{f (g A)}\}$.

The notion of BMT is used in the abstraction process at two stages: i) when selecting the more appropriate term in $S$ which covers a new call $t$, and ii) when determining whether a call $t$ headed by a primitive function symbol could be (safely) added to the current sequence of specialized calls or should be split.

[5] An element $s$ of a set $S$ is a minimally general element iff there is no different element $s'$ in $S$ such that $s'$ is a strict instance of $s$.

**Definition 65 (refined abstraction operator)** *Let $q$ be a sequence of terms and $S$ be a set of terms. Let $S_q$ be the set of terms in the sequence $q$. We define $abstract_{\trianglelefteq}(q,S)$ inductively as follows: $abstract_{\trianglelefteq}(q,S)$ =*

$$\begin{cases} q & \text{if } S = \emptyset \text{ or } S = \{t\},\ t \in \mathcal{X} \\ abstract_{\trianglelefteq}(\ldots abstract_{\trianglelefteq}(q,t_1),\ldots,t_n) & \text{if } S = \{t_1,\ldots,t_n\}, n > 0 \\ abstract_{\trianglelefteq}(q,\{t_1,\ldots,t_n\}) & \text{if } S = \{t\},\ t = c(t_1,\ldots,t_n),\ c \in \mathcal{C} \\ abs\_def((q_1,\ldots,q_n),S',t) & \text{if } S = \{t\},\ root(t) \in \mathcal{F} - \mathcal{P} \\ abs\_prim((q_1,\ldots,q_n),S',t) & \text{if } S = \{t\},\ root(t) \in \mathcal{P} \end{cases}$$

*where $S' = \{q_i \mid root(q_i) = root(t) \wedge q_i \trianglelefteq t,\ i = 1,\ldots,n\}$. The functions abs_def and abs_prim are defined as follows:*

$$abs\_def(q,\emptyset,t) = abs\_prim(q,\emptyset,t) = (q,\ t)$$
$$abs\_def((q_1,\ldots,q_n),S,t) =$$
$$abstract_{\trianglelefteq}((q_1,\ldots,q_{i-1},q_{i+1},\ldots,q_n),\{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2))$$
$$\text{if } \langle w,\{\theta_1,\theta_2\}\rangle = msg(\{q_i,t\}),\ \text{with } q_i \in BMT(S,t)$$

$$abs\_prim(q,S,t) = \begin{cases} abs\_def(q,S,t) & \text{if } \exists s \in BMT(S,t) \text{ s.t. } def(t) = def(s) \\ abstract_{\trianglelefteq}(q,S,\{t_1,t_2\}) & \text{otherwise, where } t \stackrel{\triangle}{=} p(t_1,t_2) \end{cases}$$

*where $def(t)$ denotes a sequence with the defined function symbols of $t$ in lexicographical order, and $\stackrel{\triangle}{=}$ is equality up to reordering of elements in a conjunction.*

Essentially, the way in which the abstraction operator proceeds is simple. We distinguish the cases when the considered term i) is a variable, ii) is headed by a constructor symbol, iii) by a defined function symbol, or iv) by a primitive function symbol. The actions that the abstraction operator takes, respectively, are: i) to ignore it, ii) to recursively inspect the subterms, iii) to generalize the given term w.r.t. some of its best matching terms (recursively inspecting the $msg$ $w$ and the subterms of $\theta_1, \theta_2$ not covered by the generalization), and iv) the same as in iii), but considering the possibility of splitting the given expression before generalizing it when $def(t) \neq def(s)$ (which essentially states that some defined function symbols would be lost due to the application of $msg$).

In the following, we proceed to demonstrate the correctness and termination of the global specialization process using the abstraction operator of Definition 65.

The following notations and concepts on multiset orderings will become necessary for the proofs developed in this section and in Chapter 7.

**Definition 66** *The maximum number of nested symbols in a term $t$, denoted by $depth(t)$, is defined inductively as follows:*

$$depth(t) = \begin{cases} 1 & \textit{if } t \in \mathcal{X} \\ 1 + max(\{depth(t_1), \ldots, depth(t_n)\}) & \textit{if } t = f(t_1, \ldots, t_n), f \in (\mathcal{F} \cup \mathcal{C}) \end{cases}$$

*where the function $max$ computes the maximum of a set of numbers.*

A *multiset* is a "set" where the multiplicity of occurrences is taken into account. We denote multisets by brackets, e.g., if $1, 2 \in I\!N$ then $\{1, 1, 2\}$ and $\{1, 2\}$ are different multisets of natural numbers. The set of finite multisets over $I\!N$ is denoted by $M(I\!N)$. The *multiset ordering*, $<_{mul}$, is an ordering over $M(I\!N)$.

**Definition 67 (multiset ordering)** *Let $M, M' \in M(I\!N)$, we say that $M <_{mul} M'$ if and only if exists $X \subseteq M$, and $X' \subseteq M'$ such that $M = (M' \setminus X') \cup X$ and $\forall n \in X,\ \exists n' \in X'.\ n < n'$, where $<$ is the usual strict partial order over $I\!N$.*

The multiset ordering is well-founded (i.e. there are not infinite decreasing chains) and complete induction can be applied over $M(I\!N)$.

To facilitate the proofs, reasoning about sets of partially evaluated terms, we introduce the concept of complexity $\mathcal{M}_S$ of a set of terms $S$.

**Definition 68 (complexity)** *Let $S$ be a set of terms. The complexity $\mathcal{M}_S$ of the set $S$ is the finite multiset of natural numbers corresponding to the depth of the elements of $S$: $\mathcal{M}_S = \{depth(s) \mid s \in S\}$.*

First, the following lemma establishes the transitivity of the closedness relation (Definition 13) and it is necessary in order to prove Proposition 70.

**Lemma 69** *[Alpuente et al., 1998b] If term $t$ is $S_1$-closed, and $S_1$ is $S_2$-closed, then $t$ is $S_2$-closed.*

Now, we proceed to demonstrate the correctness of our abstraction operator.

**Proposition 70** *The function $abstract_{\trianglelefteq}$ is an abstraction operator in the sense of Definition 90.*

*Proof.* Following the definition of abstraction operator, we need to prove that:

1. if $s \in S_{q'}$, where $q' = abstract_{\trianglelefteq}(q, T)$, then there exists $t \in (S_q \cup T)$ such that $t|_p = \theta(s)$ for some position $p$ and substitution $\theta$, and
2. for all $t \in (S_q \cup T)$, $t$ is closed w.r.t. the set of terms in $abstract_{\trianglelefteq}(q, T)$.

Condition (1) is trivially satisfied because the abstraction operator is based on using $msg$'s, which do not introduce new function symbols not appearing in $q$ or $T$.

Now we prove condition (2) by well-founded induction on $S_q \cup T$. If $S_q \cup T = \emptyset$, then $abstract_{\trianglelefteq}(nil, \emptyset) = nil$, and the proof is done. Let us consider the inductive case $S_q \cup T \neq \emptyset$, and assume that $T$ is not empty (otherwise the proof is trivial). Then, $T = T_0 \cup \{t\}$, with $T_0 = \{t_1, \ldots, t_{n-1}\}$. By the inductive hypothesis, the property holds for all $S_\unlhd$ and for all $T_\unlhd$ such that $\mathcal{M}_{S_\unlhd \cup T_\unlhd} <_{mul} \mathcal{M}_{S_q \cup T}$. Following the definition of $abstract_\unlhd$, we now consider four cases:

1. If $t$ is a variable symbol, then

$$\begin{array}{rcl} q' & = & abstract_\unlhd(q, T) \\ & = & abstract_\unlhd(q, T_0 \cup \{t\}) \\ & = & abstract_\unlhd(abstract_\unlhd(\ldots abstract_\unlhd(q, \{t_1\}), \ldots, \{t_{n-1}\}), \{t\}) \\ & = & abstract_\unlhd(abstract_\unlhd(q, T_0), \{t\}) \\ & = & abstract_\unlhd(q, T_0). \end{array}$$

   Since $\mathcal{M}_{S_q \cup T_0} <_{mul} \mathcal{M}_{S_q \cup T_0 \cup \{t\}} = \mathcal{M}_{S_q \cup T}$ then, by the inductive hypothesis, $S_q \cup T_0$ is closed with respect to the terms in $q'$, and by the definition of closedness, so is $S_q \cup T_0 \cup \{t\} = S_q \cup T$.

2. If $t = c(s_1, \ldots, s_m)$, $c \in \mathcal{C}$, $m > 0$, then by definition of $abstract_\unlhd$

$$\begin{array}{rcl} q' & = & abstract_\unlhd(q, T) \\ & = & abstract_\unlhd(q, T_0 \cup \{t\}) \\ & = & abstract_\unlhd(abstract_\unlhd(q, T_0), \{c(s_1, \ldots, s_m)\}) \\ & = & abstract_\unlhd(abstract_\unlhd(q, T_0), \{s_1, \ldots, s_m\}) \\ & = & abstract_\unlhd(\ldots abstract_\unlhd(abstract_\unlhd(q, T_0), \{s_1\}), \ldots, \{s_m\}) \\ & = & abstract_\unlhd(q, T_0 \cup \{s_1, \ldots, s_m\}). \end{array}$$

   Since $\mathcal{M}_{S_q \cup T_0 \cup \{s_1, \ldots, s_m\}} <_{mul} \mathcal{M}_{S_q \cup T_0 \cup \{c(s_1, \ldots, s_m)\}} = \mathcal{M}_{S_q \cup T}$ then, by the inductive hypothesis, $S_q \cup T_0 \cup \{s_1, \ldots, s_m\}$ is closed with respect to the terms in $q'$, and by the definition of closedness, so is $S_q \cup T_0 \cup \{c(s_1, \ldots, s_m)\} = S_q \cup T$.

3. If $t = f(s_1, \ldots, s_m)$, $f \in \mathcal{F}$, $m \geq 0$, then by definition of $abstract_\unlhd$

$$\begin{array}{rcl} q' & = & abstract_\unlhd(q, T) \\ & = & abstract_\unlhd(q, T_0 \cup \{t\}) \\ & = & abstract_\unlhd(abstract_\unlhd(q, T_0), t) \\ & = & abs\_def(abstract_\unlhd(q, T_0), E, t) = \\ & = & abs\_def((q_1, \ldots, q_n), E, t) = \\ & = & abs\_def(S'', E, t) \end{array}$$

   where $E = \{q_i \mid root(q_i) = root(t) \wedge q_i \unlhd t,\ i = 1, \ldots, n\}$.

   Here we distinguish two cases, which correspond to the two case values of the function $abs\_def$ in definition of $abstract_\unlhd$, given by $abs\_def(S'', E, t) =$

(a) $(abstract_{\unlhd}(q, T_0), t)$ when $E = \emptyset$. Since $\mathcal{M}_{S_q \cup T_0} <_{mul} \mathcal{M}_{S_q \cup T}$ then, by the inductive hypothesis, $S_q \cup T_0$ is closed with respect to the terms in $S''$, and therefore it is closed with respect to the elements in $(abstract_{\unlhd}(q, T_0), t)$.

Then, the claim follows trivially from the fact that $t$ is closed with respect to the terms in $q' = (abstract_{\unlhd}(q, T_0), t)$.

(b) $abstract_{\unlhd}((q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n), S^*) = abstract_{\unlhd}(S''', S^*)$, where there exists $i \in \{1, \ldots, n\}$ such that $msg(\{q_i, t\}) = \langle w, \{\theta_1, \theta_2\}\rangle$, and $S^* = \{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$ where $E \neq \emptyset$.

First, we have $\mathcal{M}_{S_q \cup T_0} <_{mul} \mathcal{M}_{S_q \cup T_0 \cup \{t\}}$ and then, by the inductive hypothesis, $S_q \cup T_0$ is $\{s_1, \ldots, s_p\}$-closed and therefore $S_q \cup T_0$ is $S''' \cup S^*$-closed.

Then, we prove that $\mathcal{M}_{S''' \cup S^*} <_{mul} \mathcal{M}_{S''' \cup \{s_i\} \cup \{t\}}$ by using the definition of multiset ordering. Here, we distinguish two cases:

i. $depth(w) = depth(s_i)$. Let $M$, $M'$ be the multiset complexities of $S''' \cup S^*$ and $S''' \cup \{s_i\} \cup \{t\}$, respectively. Let $X$, $X'$ be the multiset complexities of $\mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$ and $\{t\}$, respectively. Let $X_1, X_2$ be the multiset complexities of $\mathcal{R}an(\theta_1)$ and $\mathcal{R}an(\theta_2)$, respectively.

Since $t$ embeds $s_i$, then it is immediate to see that $depth(s_i) \leq depth(t)$. Since $s_i$ and $w$ are comparable and $s_i = \theta_2(w)$, then $depth(d) < depth(s_i)$ for all $d \in \mathcal{R}an(\theta_2)$. Therefore, $depth(d) < depth(t)$ for all $d \in \mathcal{R}an(\theta_2)$, and hence $\forall n \in X_1,\ \exists n' \in X'.\ n < n'$.

Since $t$ and $w$ are comparable terms and $t = \theta_2(w)$, then it holds that $\forall n \in X_2,\ \exists n' \in X'.\ n < n'$, which ends the proof.

ii. $depth(w) < depth(s_i)$ This case can be proved in a similar way by considering that $X$ and $X'$ denote the multiset complexities of $\{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$ and $\{s_i\} \cup \{t\}$, respectively.

Since we have proved $\mathcal{M}_{S''' \cup S^*} <_{mul} \mathcal{M}_{S''' \cup \{s_i\} \cup \{t\}}$, then by the inductive hypothesis, $S''' \cup S^*$ is closed w.r.t. the terms in $q'$ and hence $S_q \cup T_0$ is also closed w.r.t. the elements in $q'$, by Lemma 69. Finally, as $t$ is closed w.r.t. $\{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$ by definition of $msg$, the claim follows by using Lemma 69 again.

4. $root(t) \in \mathcal{P}$. Similarly to the previous case:

$$q' = abs\_prim(abstract_{\unlhd}(q, T_0), E, t)$$

where $E = \{q_i \mid root(q_i) = root(t) \wedge q_i \unlhd t,\ i = 1, \ldots, n\}$.

Since the result of $abs\_prim$ is equivalent to the application of $abs\_def$ or the elimination of the outermost symbol of $t$, then the proof is perfectly analogous to cases 2 and 3.

□

The following lemmata are auxiliary.

**Lemma 71** *The computation of the abstraction operator of definition 65 terminates.*

*Proof.* The termination of the computation $abstract_{\trianglelefteq}(q, T)$ can be proven by well-founded induction on $S_q \cup T$. The proof is similar to the proof of Proposition 70. □

Now, we define the notion of nonembedding sequence, which allows us to identify sequences of terms which fulfill a given ordering.

**Definition 72 (nonembedding property)** *Let $q'$ be a finite sequence of terms and $T$ be a finite set of terms. Let $q = (q_1, \ldots, q_n)$ be the sequence of terms computed by $q \longmapsto_{\mathcal{P}} abstract(q', T)$. Then, we say that $S$ satisfies the nonembedding property if it holds that:*

$$\forall l, k \text{ in } \{1, \ldots, n\}, \text{ with } l < k \ . \ (root(q_l) = root(q_k) \ \Rightarrow \ q_l \ntrianglelefteq q_k)$$

**Lemma 73** *Let $S$ be a sequence of terms satisfying the nonembedding property, and let $T$ be an arbitrary set of terms. Then, $q' = abstract_{\trianglelefteq}(q, T)$ satisfies the nonembedding property.*

*Proof.* It can be easily proved that the computation of $abstract_{\trianglelefteq}(q, T)$ satisfies the nonembedding property by well-founded induction on $S_q \cup T$.

□

**Lemma 74 (nonembedding property)** *Let $T_0, \ldots, T_n$ be the sequence of sequences computed by $T_{i+1} \longmapsto_{\mathcal{P}} abstract_{\trianglelefteq}(T_i, \mathcal{R}^i_{calls})$, where $\mathcal{R}^i = U_{dynamic}(T_i, \mathcal{R})$ for some program $\mathcal{R}$. Then, each $T_i$, $i = 0, \ldots, n$ satisfies the nonembedding property.*

*Proof.* We prove this proposition by induction on the number $n$ of computation steps. If $n = 0$, then $T_0 = \emptyset$ and the proof is done. Let us consider the inductive case $n > 0$. By the inductive hypothesis, the property holds for all $T_j$ such that $j < n$. It is immediate to see that the nonembedding property holds for

$$T_n = abstract_{\trianglelefteq}(T_{n-1}, T)$$

by the application of Lemma 73 since $T_{n-1}$ satisfies the nonembedding property (by the inductive hypothesis). □

As a consequence of Lemmata 63, 71, and 74, the following theorem states the termination of the algorithm.

**Theorem 75** *Let $\mathcal{R}$ be a program and $S$ a finite set of terms. The computation of $\mathcal{PE}(\mathcal{R}, S)$ for the unfolding rule $U_{dynamic}$ and the abstraction operator $abstract_{\trianglelefteq}$ terminates.*

Our final example witnesses that $abstract_{\trianglelefteq}$ behaves well w.r.t. Example 24.

**Example 26** Consider again the tree depicted in Fig. 6.3. By applying our method, the following call to $abstract_{\trianglelefteq}$ is undertaken:

$$\begin{array}{ll} abstract_{\trianglelefteq}( & ((\mathtt{app\ x\ y} \approx \mathtt{w}) \ \wedge \ (\mathtt{app\ w\ z} \approx \mathtt{r})), \\ & \{\mathtt{x}' \approx \mathtt{w}' \ \wedge \ (\mathtt{app\ xs}'\ \mathtt{y} \approx \mathtt{ws}') \ \wedge \ (\mathtt{w}' \approx \mathtt{r}') \ \wedge \ (\mathtt{app\ ws}'\ \mathtt{z} \approx \mathtt{rs}')\}). \end{array}$$

Following Definition 65, by two recursive calls to *abs_prim*, we get:

$$((\mathtt{app\ x\ y} \approx \mathtt{w}) \ \wedge \ (\mathtt{app\ w\ z} \approx \mathtt{r}), \ \mathtt{x}' \approx \mathtt{w}').$$

By considering the independent renaming `dapp x y w z r` for the specialized call $(\mathtt{app\ x\ y} \approx \mathtt{w}) \wedge (\mathtt{app\ w\ z} \approx \mathtt{r})$, the method derives a (recursive) rule of the form:

$$\mathtt{dapp}\ (\mathtt{x\!:\!xs})\ \mathtt{y}\ (\mathtt{w\!:\!ws})\ \mathtt{z}\ (\mathtt{r\!:\!rs}) \ = \ (\mathtt{x} \approx \mathtt{w}) \ \wedge \ (\mathtt{w} \approx \mathtt{r}) \ \wedge \ \mathtt{dapp\ xs\ y\ ws\ z\ rs}$$

which embodies the intended optimal specialization for this example.

## 6.4 Experiments

The refinements presented so far have been implemented into the Indy prototype implementation of the NPE framework (see Appendix A for a detailed description of the system).

In order to assess the practicality of our approach, we have benchmarked the speed and specialization achieved by the Indy system. The benchmarks used for the analysis are: `applast`, which appends an element at the end of a given list and returns the last element of the resulting list; `double_app`, the program in Example 22; `double_flip`, which flips a tree structure twice, then returning the original tree back; `fibonacci`, fibonacci's function; `heads&legs`, which computes the number of heads and legs of a given number of birds and cats; `match-app`, the extremely naïve string pattern matcher based on the use of `append`; `match-kmp`, a semi-naïve string pattern matcher; `maxlength`, which returns the maximum and the length of a list; `palindrome`, a program to check whether a given list is a palindrome; and `sorted_bits`, the program in Example 21. Some of the examples are typical partial deduction benchmarks (see [Lam and Kusalik, 1991; Leuschel, 1998]) adapted to a functional logic syntax, while others come from the literature of functional program transformations, such as positive supercompilation [Sørensen *et al.*, 1996], fold/unfold transformations [Burstall and Darlington, 1977; Darlington, 1982], and deforestation [Wadler, 1990]. All benchmarks were executed by needed narrowing. We have considered the following three different alternatives for local control:

Table 6.1: Benchmark results

| | Original | | emb_goal | | emb_redex | | comp_redex | |
|---|---|---|---|---|---|---|---|---|
| Benchmarks | Rw | RT | Rw | Speedup | Rw | Speedup | Rw | Speedup |
| applast | 10 | 90 | 13 | 1.32 | 28 | 2.20 | 13 | 1.10 |
| double_app | 8 | 106 | 39 | 1.63 | 61 | 1.28 | 15 | 3.12 |
| double_flip | 8 | 62 | 26 | 1.51 | 17 | 1.55 | 17 | 1.55 |
| fibonacci | 5 | 119 | 11 | 1.19 | 7 | 1.08 | 7 | 1.08 |
| heads&legs | 8 | 176 | 24 | 4.63 | 22 | 2.41 | 21 | 2.48 |
| match-app | 8 | 201 | 12 | 1.25 | 20 | 2.75 | 23 | 2.79 |
| match-kmp | 12 | 120 | 14 | 3.43 | 14 | 3.64 | 13 | 3.43 |
| maxlength | 14 | 94 | 51 | 1.17 | 20 | 1.27 | 18 | 1.25 |
| palindrome | 10 | 119 | 19 | 1.25 | 10 | 1.35 | 10 | 1.35 |
| sorted_bits | 8 | 110 | 16 | 1.15 | 31 | 2.89 | 10 | 2.68 |
| Average | 9.1 | 119.7 | 22.5 | 1.85 | 23 | 2.04 | 14.7 | 2.08 |
| TMix average | | | 1881 | | 7441 | | 5788 | |

1. `emb_goal`: it expands derivations while new goals do not embed a previous comparable goal in the same branch;

2. `emb_redex`: the concrete unfolding rule of Section 6.3.1 which implements the *dependency_clash* test using homeomorphic embedding on comparable ancestors of selected redexes to ensure finiteness (note that it differs from `emb_goal` in that `emb_redex` implements dynamic scheduling on conjunctions and that homeomorphic embedding is checked on simple redexes rather than on whole goals);

3. `comp_redex`: the unfolding rule of Section 6.3.1 which uses the simpler definition of *dependency_clash* based on comparable ancestors of selected redexes as a whistle.

Abstraction is always done as explained in Definition 65. Table 6.1 summarizes our benchmark results. The first two columns measure the number of rewrite rules (Rw) and the absolute runtimes (RT) for each original program. The other columns show the number of rewrite rules and the speedups achieved for the specialized programs obtained by using the three considered unfolding rules: `emb_goal`, `emb_redex`, and `comp_redex`. The row at the bottom of the table (TMix) indicates the average specialization time for each considered unfolding rule. Runtimes were measured on a HP 712/60 workstation, running under HP Unix v10.01. Times are expressed in milliseconds and are the average of 10 executions. Speedups were computed by running the original and specialized programs under the publicly available functional logic language Curry [Hanus, 2000a]. Runtime input goals were chosen to give a reasonably long overall time.

The figures in Table 6.1 demonstrate that the control refinements that we have incorporated into the INDY system provide satisfactory speedups on all benchmarks (which is very encouraging, given the fact that no partial input data were provided in the examples, except for `match-app`, `match-kmp`, and `sorted_bits`). On the other hand, our extensions are *conservative* in the sense that there is no penalty w.r.t. the specialization achieved by the original system on non-conjunctive goals —compare, e.g., with the benchmarks in [Alpuente *et al.*, 1998b] (although some specialization times are slightly higher due to the more complex processing being done). Let us note that, from the speedup results in Table 6.1, it can appear that there is no significant difference between the strategies `emb_redex` and `comp_redex`. However, when we also consider the specialization times (TMix) and the size of the specialized programs (Rw), we find out that `comp_redex` has a better overall behaviour. Hence, although the speedups achieved by these strategies are somewhat similar, `emb_redex` is inherently more complex and it very often expands local narrowing trees beyond the "optimal" point.

## 6.5 Conclusions

In functional logic languages, expressions can be written by exploiting the *nesting* capability of the functional syntax, as in $\texttt{app (app x y) z} \approx \texttt{r}$, but in some cases it might be appropriate (or necessary) to decompose nested expressions as in logic programming, and write $(\texttt{app x y} \approx \texttt{w}) \wedge (\texttt{app w z} \approx \texttt{r})$ (for instance, if some test such as `sorted_bits w` on the intermediate list `w` were necessary). The original NPE framework behaves well on programs written with the "pure" functional syntax [Alpuente *et al.*, 1998a]. However, it is not able to produce good specialization when programs are written as conjunctions of subgoals (and a slowdown is commonly produced). For this we could not achieve some of the standard, difficult transformations such as tupling [Chin, 1993] within the classical NPE framework. As opposed to the classical partial deduction framework (in which only folding on single atoms can be done), the NPE algorithm is able to perform folding on complex expressions (containing an arbitrary number of function calls). However, this does not suffice to achieve tupling in practice, since complex expressions are often generalized and specialization is lost.

The results in this chapter demonstrate that it is possible to supply the general NPE framework with appropriate control options to specialize complex expressions containing primitive functions (such as conjunctions), thus providing a powerful polygenetic specialization framework with no ad-hoc setting. Nevertheless, if one intends to develop a partial evaluation scheme for a realistic multi-paradigm declarative language, additional high-level constructs have to be considered (e.g., higher-order functions, constraints, program annotations, calls to external functions, etc.) apart from the primitive symbols which we considered in this chapter. In the third part of this thesis, we will focus on this last challenge in deep extent.

# Part III

# Partial Evaluation in Practice

# Chapter 7

# Using an Abstract Representation

This chapter investigates a novel via for the specialization of functional logic languages. In Section 7.2, we consider a maximally simplified abstract representation of programs (which still contains all the necessary information) and define a non-standard (operational) semantics for these programs which is specially well-suited for being used at partial evaluation time. These two key ingredients allow us to design a simple and concise partial evaluation method for modern functional logic languages which we present in Section 7.3. We will show how the new scheme avoids several limitations of previous approaches for specializing functional logic programs. Part of this chapter has been published in [Albert *et al.*, 2000e,f].

## 7.1 Introduction

As explained in Chapter 3, narrowing-driven partial evaluation has the same potential for specialization as *positive supercompilation* [Sørensen *et al.*, 1996] and *conjunctive partial deduction* [De Schreye *et al.*, 1999]. Despite its power, the narrowing-driven approach to partial evaluation suffers from several limitations:

1. Firstly, in the context of *lazy* functional logic languages, expressions in head normal form cannot be evaluated at partial evaluation time. This restriction is imposed to avoid the *backpropagation* of bindings to the left-hand side of residual rules, which may incorrectly restrict the domain of functions (see Example 3).

2. Secondly, if one intends to develop a partial evaluation scheme for a realistic multi-paradigm declarative language, several high-level constructs have to be considered: higher-order functions, constraints, program annotations, calls to

external functions, etc. A complex operational calculus is required to properly deal with these additional features of modern languages. Commonly, a partial evaluator relies on an interpreter of the language. Therefore, as the operational semantics becomes more elaborated, the associated partial evaluation techniques become (more powerful but) also increasingly more complex.

3. Finally, an interesting application of partial evaluation is the generation of compilers and compiler generators [Jones *et al.*, 1993]. For this purpose, the partial evaluator must be self-applicable, i.e., able to partially evaluate itself. This becomes difficult in the presence of high-level constructs such as those mentioned in point 2 above. As advised in [Jones *et al.*, 1993], *it is essential to cut the language down to the bare bones* in order to achieve self-application.

In order to overcome the aforementioned problems, a promising approach successfully tested in the context of other programming paradigms (e.g., [Bondorf, 1989; Nemytykh *et al.*, 1996]) is to consider programs written in a maximally simplified programming language, into which programs written in a higher-level language can be automatically translated. Recently, Hanus and Prehofer [1999] introduced an explicit representation of the structure of definitional trees (used to guide the needed narrowing strategy) within the rewrite rules. This provides more explicit control and leads to a calculus simpler than standard needed narrowing. Moreover, source programs can be automatically translated to the new representation. In this chapter, we consider a very simple abstract representation of functional logic programs which extends the one introduced in [Hanus and Prehofer, 1999]. Our abstract representation includes also information about the evaluation type of functions: *flex* —which enables narrowing steps— or *rigid* —which forces delayed evaluation by rewriting.[1] Then, we define a *non-standard* (operational) semantics which is specially well-suited to perform computations at partial evaluation time. This is a crucial difference w.r.t. previous approaches, where the same mechanism is used both for program execution and for partial evaluation. The use of an abstract representation, together with the new calculus, allows us to design a simple and concise partial evaluation method for modern functional logic languages, beating the limitations of previous approaches.

Finally, since truly lazy functional logic languages can be automatically translated into the abstract representation (which still contains all the necessary information about programs), our technique is widely applicable. Following this scheme, partially evaluated programs will be also written in the abstract representation. Since existing compilers use a similar representation for intermediate code, this is not a restriction.

[1]Indeed, a similar representation constitutes the basis of a recent proposal for a standard intermediate language, FlatCurry, for the compilation of Curry programs [Hanus, 2000a] in PAKCS —the Portland-Aachen-Kiel Curry System— [Hanus *et al.*, 2000].

Rather, our specialization process can be seen as an optimization phase (transparent to the user) performed during the compilation of the program.

## 7.2 The Basic Framework

In this section, we present an appropriate abstract representation for functional logic languages. We also provide a new operational semantics, the RLNT calculus, which is specially well-suited to perform computations during partial evaluation. Finally, we introduce a novel partial evaluation framework based on the RLNT calculus for programs written in the abstract representation.

### 7.2.1 The Abstract Representation

Similarly to Hanus and Prehofer [1999], we present an abstract representation for programs in which the definitional trees are made explicit by means of case constructs. Moreover, here we distinguish two kinds of case expressions in order to make also explicit the flexible/rigid evaluation annotations. In particular, we assume that all functions are defined by one rule whose left-hand side contains only variables as parameters and the right-hand side contains case expressions for pattern-matching. The syntax for programs in the abstract representation is summarized by

$$
\begin{array}{lcll}
\mathcal{R} & ::= & D_1 \ldots D_m & \text{(flat program)} \\
D & ::= & f(x_1, \ldots, x_n) \to t & \text{(definition)} \\
p & ::= & c(x_1, \ldots, x_n) & \text{(constructor pattern)} \\
t & ::= & x & \text{(variable)} \\
 & \mid & c(t_1, \ldots, t_n) & \text{(constructor call)} \\
 & \mid & f(t_1, \ldots, t_n) & \text{(function call)} \\
 & \mid & case\ t_0\ of\ \{p_1 \to t_1; \ldots; p_n \to t_n\} & \text{(rigid case)} \\
 & \mid & fcase\ t_0\ of\ \{p_1 \to t_1; \ldots; p_n \to t_n\} & \text{(flexible case)}
\end{array}
$$

where $\mathcal{R}$ denotes a *flat* program, $D$ a function definition, $p$ a constructor pattern and $t$ an arbitrary expression. A flat program $\mathcal{R}$ consists of a sequence of function definitions $D$ such that the left-hand side is linear and has only variable arguments, i.e., pattern matching is compiled into case expressions. The right-hand side of each function definition is a term $t$ composed by variables, constructor calls, function calls, and case expressions. The form of a case expression is

$$(f)case\ t\ of\ \{c_1(\overline{x_{n_1}}) \to t_1; \ldots; c_k(\overline{x_{n_k}}) \to t_k\}$$

where $t$ is a term, $c_1, \ldots, c_k$ are different constructors of the type of $t$, and $t_1, \ldots, t_k$ are terms (possibly containing case expressions). The variables $\overline{x_{n_i}}$ are called *pattern variables* and are local variables which occur only in the corresponding subexpression

$t_i$. The difference between *case* and *fcase* only shows up when the argument $t$ is a free variable: *case* suspends (which corresponds to residuation) whereas *fcase* nondeterministically binds this variable to the constructor pattern in each branch of the case expression (which corresponds to narrowing). Functions defined only by *fcase* (resp. *case*) expressions are called *flexible* (resp. *rigid*). Thus, flexible functions act as generators (like predicates in logic programming) and rigid functions act as consumers. As mentioned in previous chapters, concurrency is expressed by a built-in function "&" which evaluates its two arguments concurrently. This function can be defined by the rule:

$$\texttt{True}\ \&\ \texttt{True} = \texttt{True}$$

and, hence, in the following we simply consider it as an ordinary operation symbol.

**Example 27** Consider the rules defining the (rigid) function "$\leqslant$":

$$\begin{array}{rcll}
\texttt{0} & \leqslant & \texttt{n} & = \texttt{True} \\
(\texttt{Succ m}) & \leqslant & \texttt{0} & = \texttt{False} \\
(\texttt{Succ m}) & \leqslant & (\texttt{Succ n}) & = \texttt{m} \leqslant \texttt{n}
\end{array}$$

By using case expressions, it can be translated to the following rule:

$$\begin{array}{lllll}
\texttt{x} \leqslant \texttt{y} = \texttt{case x of} & \{\texttt{0} & \rightarrow & \texttt{True}; & \\
 & (\texttt{Succ x}_1) & \rightarrow & \texttt{case y of} & \{\texttt{0} \rightarrow \texttt{False}; \\
 & & & & (\texttt{Succ y}_1) \rightarrow \texttt{x}_1 \leqslant \texttt{y}_1\}\ \}
\end{array}$$

Due to the presence of fresh pattern variables in the right-hand side of the rule, this is not a standard rewrite rule. Nevertheless, Hanus and Prehofer [1999] provide a unique reading by specifying a particular semantics to case expressions. The main idea consists of considering a case expression as a function whose semantics is defined by a set of rewrite rules. For example, they relate the above case expression with the following rewrite rules:[2]

$$\begin{array}{ll}
\texttt{case 0} & \texttt{of}\ \{\texttt{0} \rightarrow \texttt{t}; _ \rightarrow _\} = \texttt{t} \\
\texttt{case (Succ x)} & \texttt{of}\ \{_ \rightarrow _; \texttt{Succ x} \rightarrow \texttt{t}\} = \texttt{t}
\end{array}$$

In principle, the last rule could be considered illegal since its left-hand side is not a linear pattern. However, the repeated occurrences of variable x will be only used to pass subterms from one place to another and not to compare terms. Thus, we can deal with them as a standard program. In general, a case expression

$$case\ x\ of\ \{\overline{p_n \rightarrow t_n}\}$$

[2] "_" denotes an anonymous variable, as in Prolog.

can be considered as a function with arity $2n+1$ defined by the following $n$ rewrite rules:

$$\begin{array}{ll} case\ p_1 & of\ \{p_1 \to x; \ldots; \_ \to \_\} \to x \\ \vdots & \\ case\ p_n & of\ \{\_ \to \_; \ldots; p_n \to x\} \to x \end{array}$$

Rigorously, a different *case* function (and the corresponding meaning) is required for each case expression with a different pattern. For instance, the case expressions could be indexed by the case patterns. Then, in order to apply a narrowing step to a case expression of the form:

$$case\ t\ of\ \{p_1 \to t_1; \ldots; p_n \to t_n\}$$

there are three possibilities:

1. If $t$ is a variable, we can apply any of the $n$ rules for *case*:

   $$case\ x\ of\ \{p_1 \to t_1; \ldots; p_n \to t_n\} \leadsto_\sigma \sigma(t_i)$$

   with $\sigma = \{x \mapsto p_i\},\ i \in \{1, \ldots, n\}$. This corresponds to an instantiation step in the needed narrowing calculus.

2. If $t$ is a constructor-rooted term $c(\overline{s_k})$ and $p_i = c(\overline{x_k})$ for some $i \in \{1, \ldots, n\}$:

   $$case\ x\ of\ \{p_1 \to t_1; \ldots; p_n \to t_n\} \leadsto_{id} \sigma(t_i)$$

   This corresponds to a selection step in the needed narrowing calculus.

3. Otherwise, further evaluation of $t$ is required.

A definitional tree $\mathcal{T}$ can be mapped into a term with case expressions by using the translation function $\mathcal{C}ase(\mathcal{T})$ [Hanus and Prehofer, 1999]:

$$\begin{array}{lcll} \mathcal{C}ase(rule(l \to r)) & = & r & \\ \mathcal{C}ase(branch(\pi, o, \overline{\mathcal{T}_k})) & = & case\ \pi|_o\ of & \{\ pat(\mathcal{T}_1)|_o \to \mathcal{C}ase(\mathcal{T}_1); \\ & & & \quad \vdots \\ & & & \quad pat(\mathcal{T}_k)|_o \to \mathcal{C}ase(\mathcal{T}_k)\} \end{array}$$

where $pat(\mathcal{T})$ denotes the pattern of the definitional tree $\mathcal{T}$.[3] For instance, If $\mathcal{T}$ is a definitional tree with pattern $f(\overline{x_n})$ of the $n$-ary function $f$, then:

$$f(\overline{x_n}) \to \mathcal{C}ase(\mathcal{T})$$

[3]The extension of function $\mathcal{C}ase$ to consider definitional trees with evaluation annotations is straightforward.

is the new rewrite rule expressed in terms of case expressions for $f$. Moreover, we introduce a function $flat$ which, given an inductively sequential program, allows us to represent this program by means of case expressions; namely, if $\mathcal{R}$ is a program defined by a set of functions $f_1, \ldots, f_n$ whose associated definitional trees are $\mathcal{T}_1, \ldots, \mathcal{T}_n$, respectively, we define $flat(\mathcal{R})$ as follows:

$$flat(\mathcal{R}) = \bigcup_{i=1,\ldots,n} \{f_i(\overline{x_{n_i}}) \to \mathcal{C}ase(\mathcal{T}_i)\}$$

The following section makes use of function $flat$.

### 7.2.2 The Residualizing Semantics

In this section, we provide an adequate calculus to perform partial evaluation within this particular kind of programs. Firstly, we introduce a different formulation of needed narrowing. It will become useful in order to establish the equivalence between needed narrowing and the operational semantics for programs in our abstract representation.

**Inference rules for needed narrowing.**

Throughout this chapter, we use a needed narrowing calculus defined in terms of an inference system (Figure 7.1 provides a formal description). An $\mathcal{E}val$-goal is any sequence obtained from the initial goal by applying inference steps of this calculus. Let us briefly describe the inference rules of the calculus:

Initial. The rule Initial attaches the appropriate definitional tree to the initial term.

Apply. If this tree is a rule, then it applies an instance of this rule to the current term.

Select. The Select rule selects the appropriate subtree of the current definitional tree.

Instantiate. This rule non-deterministically selects a subtree of the current definitional tree and instantiates the variable at the current position to the appropriate pattern.

Eval Subterm. The Eval Subterm rule initiates the evaluation of the subterm at the current position by creating a new $\mathcal{E}val$-goal for this subterm.

Replace Subterm. If a rewrite rule has been applied to this subterm (by the inference rule Apply), the rewritten subterm is inserted at the current position by the inference rule Replace Subterm.

In contrast to traditional narrowing steps, where a subterm is directly unified with the left-hand side of some rewrite rule, in this calculus, the derivation steps explicitly

Initial

$$t \Rightarrow^{id}_{\mathsf{NN}} \mathcal{E}val(t, \mathcal{T}) \quad \text{if } t = f(\overline{t_n}) \text{ and } \mathcal{T} \text{ is the definitional tree of } f$$

Apply

$$\mathcal{E}val(t, rule(l \to r)), G \Rightarrow^{id}_{\mathsf{NN}} \sigma(r), G \quad \text{if } \sigma(l) = t$$

Select

$$\mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \Rightarrow^{id}_{\mathsf{NN}} \mathcal{E}val(t, \mathcal{T}_i), G \quad \text{if } t|_o = c(\overline{t_n}) \text{ and } pat(\mathcal{T}_i)|_o = c(\overline{x_n})$$

Instantiate

$$\mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \Rightarrow^{\sigma}_{\mathsf{NN}} \sigma(\mathcal{E}val(t, \mathcal{T}_i), G) \quad \text{if } t|_o = x \text{ and } \sigma = \{x \mapsto pat(\mathcal{T}_i)|_o\}$$

Eval Subterm

$$\mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \Rightarrow^{id}_{\mathsf{NN}} \mathcal{E}val(t|_o, \mathcal{T}), \mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \quad \text{if } t|_o = f(\overline{t_n}) \text{ and } \mathcal{T} \text{ is the definitional tree of } f$$

Replace Subterm

$$\mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k})), G \Rightarrow^{id}_{\mathsf{NN}} \mathcal{E}val(t[t']_o, branch(\pi, o, \overline{\mathcal{T}_k})), G \quad \text{if } t \neq \mathcal{E}val(..., ...)$$

Figure 7.1: Inference rules for the needed narrowing calculus

show the selection of the subterm and the instantiation of the goal variables. Clearly, this calculus is more detailed but equivalent to needed narrowing as presented in Chapter 2.

**The LNT calculus.**

Hanus and Prehofer [1999] provide an appropriate operational semantics for programs which use case expressions: the LNT calculus (Lazy Narrowing with definitional Trees). In Figure 7.2, we give a formalization which is equivalent to the original LNT strategy. The main difference is that we allow nested expressions (while Hanus and Prehofer [1999] use a goal system in which a goal is decomposed into a sequence of subgoals of the form $t \to^? x$ so that narrowing rules are only applied to the outermost function symbol of the leftmost goal and, finally, a rule called Bind is applied in order to propagate terms to the subsequent expressions). On the other hand, we use the symbols "$[\![$" and "$]\!]$" in an expression like $[\![t]\!]$ to *guide* the inference rules and, most importantly, to mark which part of an expression can be still evaluated (within the square brackets) and which part must be definitively residualized (outside the square brackets).[4] That means that square brackets are purely syntactical (i.e., they do not denote "the semantic value of $t$" as is usual in denotational semantics).

[4]This will become useful later in order to define the Residualizing LNT calculus.

HNF

$$\begin{array}{rll}
[\![t]\!] & \Rightarrow^{id}_{\mathsf{LNT}} & t \quad \text{if } t \in \mathcal{X} \text{ or } t = c() \text{ with } c/0 \in \mathcal{C} \\
[\![c(t_1,\ldots,t_i,\ldots,t_n)]\!] & \Rightarrow^{\sigma}_{\mathsf{LNT}} & [\![\sigma(c(t_1,\ldots,t'_i,\ldots,t_n))]\!] \quad \text{if } c \in \mathcal{C} \text{ and} \\
& & \quad \exists i \in \{1,\ldots,n\} \text{ such that } [\![t_i]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} [\![t'_i]\!]
\end{array}$$

Case Eval

$$\begin{array}{rll}
[\![case\ t\ of\ \{\overline{p_k \to t_k}\}]\!] & \Rightarrow^{\sigma}_{\mathsf{LNT}} & [\![\sigma(case\ t'\ of\ \{\overline{p_k \to t_k}\})]\!] \quad \text{if } [\![t]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} [\![t']\!] \\
& & \text{where } t \notin \mathcal{X} \text{ and } root(t) \notin \mathcal{C}
\end{array}$$

Case Select

$$[\![case\ c(\overline{t_n})\ of\ \{\overline{p_k \to t_k}\}]\!] \Rightarrow^{id}_{\mathsf{LNT}} [\![\sigma(t_i)]\!] \quad \text{if } p_i = c(\overline{x_n}),\ c \in \mathcal{C},\ \sigma = \{\overline{x_n \mapsto t_n}\}$$

Case Guess

$$[\![case\ x\ of\ \{\overline{p_k \to t_k}\}]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} [\![\sigma(t_i)]\!] \quad \text{if } \sigma = \{x \mapsto p_i\},\ i = 1,\ldots,k$$

Function Eval

$$\begin{array}{rll}
[\![f(\overline{t_n})]\!] & \Rightarrow^{id}_{\mathsf{LNT}} & [\![\sigma(r)]\!] \quad \text{if } f(\overline{x_n}) \to r \in \mathcal{R} \text{ is a rule with fresh} \\
& & \text{variables and } \sigma = \{\overline{x_n \mapsto t_n}\}
\end{array}$$

Figure 7.2: LNT calculus

The calculus is formed by the following rules:

HNF. The HNF (Head Normal Form) rules can be applied when the considered term is a variable or a constructor-rooted term, and simply return the same term (if it is a value) or continue with the evaluation of some argument in a "don't care" nondeterministic manner.

Case Eval. This rule initiates the evaluation of the case argument by creating a call for this subterm (unless it is a variable or a constructor-rooted term since they are dealt with in other rules). It inserts the evaluated subterm at the current position and continues with the evaluation of the case expression.

Case Select. This rule selects the appropriate branch of the current case expression.

Case Guess. This rule non-deterministically selects a branch of the current case expression and instantiates the variable at the case argument to the appropriate constructor pattern.

Function Eval. This rule performs the unfolding of a function call.

The equivalence of needed narrowing and the original LNT calculus is established by Hanus and Prehofer [1999] in terms of the equivalences between needed narrowing and the LNT calculus w.r.t. leftmost-outermost narrowing[5] with case expressions. The remaining of this section just recalls the proof scheme used to prove this equivalence. This will allow us to justify the equivalence between needed narrowing and the LNT calculus as defined in Figure 7.2.

First, we state the equivalence between needed narrowing and leftmost-outermost narrowing. Let us introduce some preliminary notions of [Hanus and Prehofer, 1999]:

- We denote by $\mathcal{R}$ an inductively sequential program, by $\mathcal{R}'$ its translated version using function $flat$ (i.e., $\mathcal{R}' = flat(\mathcal{R})$) and by $\mathcal{R}_{case}$ the additional case rules as defined in the previous section.
- The auxiliary function $\mathcal{EC}$ is used to relate terms of the original program $\mathcal{R}$ with case expressions of its translated version $\mathcal{R}'$. In particular, function $\mathcal{EC}$ translates $\mathcal{E}val$-goals into terms with case expressions as follows:

$$\begin{array}{l}
\mathcal{EC}(t) = t \\
\mathcal{EC}(\mathcal{E}val(t, \mathcal{T})) = \sigma(\mathcal{C}ase(\mathcal{T})) \quad \text{where } \sigma(pat(\mathcal{T})) = t \\
\mathcal{EC}(G, \mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k}))) = case\ \mathcal{EC}(G)\ of\ \{\overline{p_k \to t_k}\} \\
\quad \text{where } \mathcal{EC}(\mathcal{E}val(t, branch(\pi, o, \overline{\mathcal{T}_k}))) = case\ \ldots\ of\ \{\overline{p_k \to t_k}\}
\end{array}$$

The following lemma is auxiliary to demonstrate the equivalence between needed narrowing and leftmost-outermost narrowing.

**Lemma 76** *[Hanus and Prehofer, 1999, Lemma A.1]*
*Let $G$ be an $\mathcal{E}val$-goal, $t = \mathcal{EC}(G)$, and $G \Rightarrow^{\sigma}_{\mathsf{NN}} G'$ be an inference step in the needed narrowing calculus. Then, either $\mathcal{EC}(G') = t$ and $\sigma = id$, or there exists a unique leftmost-outermost narrowing step $t \leadsto_{\sigma} t'$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$ with $\mathcal{EC}(G') = t'$.*

The equivalence of needed narrowing and leftmost-outermost narrowing is based on the above lemma:

**Theorem 77** *[Hanus and Prehofer, 1999, Theorem 3.3]*
*Let $t$ be a term and $c$ a 0-ary constructor. For each needed narrowing derivation $t \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{NN}} c$ w.r.t. $\mathcal{R}$, there exists a leftmost-outermost narrowing derivation $t \leadsto^{*}_{\sigma} c$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$, and vice versa.*

Then, the equivalence between leftmost-outermost narrowing and the LNT calculus can be proved. The following is an easy consequence of Lemma A.2 in [Hanus and Prehofer, 1999]:

**Lemma 78** *Let $t \leadsto_{\sigma} t'$ be a leftmost-outermost narrowing step w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$. Then, there exists a LNT step $t \Rightarrow^{\sigma}_{\mathsf{LNT}} t'$ w.r.t. $\mathcal{R}'$.*

[5]Leftmost-outermost narrowing selects the position of the leftmost-outermost term among all possible narrowing positions.

*Proof.* Let us note that the original LNT calculus of [Hanus and Prehofer, 1999] considers goal expressions of the form $t_1 \to^? x_1, \ldots, t_n \to^? x_n$ where case expressions are flattened. Therefore, they establish the equivalence between leftmost-outermost narrowing and LNT computations by using two auxiliary functions $\mathcal{F}lat$ and $\mathcal{F}old$. Intuitively, function $\mathcal{F}lat$ takes an expression (possibly) with nested case constructs and returns the sequence of expressions obtained by unnesting the arguments of case constructs. Analogously, function $\mathcal{F}old$ performs the inverse operation. In contrast to [Hanus and Prehofer, 1999], in our calculus, we can simplify this correspondence since our LNT calculus deals with nested case expressions and, therefore, we do not need to use functions $\mathcal{F}lat$ and $\mathcal{F}old$. Therefore, our statement can be proved straightforwardly from Lemma A.2 in [Hanus and Prehofer, 1999] by just disregarding the use of functions $\mathcal{F}lat$ and $\mathcal{F}old$. □

An analogous lemma in the other direction —which demonstrates that each inference step in the LNT calculus w.r.t. $\mathcal{R}'$ corresponds to zero or one leftmost-outermost steps w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$— can be similarly established. Both lemmata are used to prove the equivalence between the leftmost-outermost narrowing and the LNT calculus:

**Theorem 79** *Let $t$ be a term and $c$ a 0-ary constructor. For each leftmost-outermost narrowing derivation $t \leadsto^*_\sigma c$ w.r.t. $\mathcal{R}' \cup \mathcal{R}_{case}$, there exists a LNT derivation $t \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{LNT}} c$ w.r.t. $\mathcal{R}'$, and vice versa.*

*Proof.* Similarly to Theorem 3.5 in [Hanus and Prehofer, 1999], the proof follows from Lemma 78 with a simple induction on the lengths of the derivations. □

Finally, the following theorem establishes the equivalence between needed narrowing and the LNT calculus. This is an easy consequence of Theorems 77 and 79 above.

**Theorem 80** *Let $t$ be a term and $c$ a 0-ary constructor. For each needed narrowing derivation $t \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{NN}} c$ w.r.t. $\mathcal{R}$, there exists a LNT derivation $t \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{LNT}} c$ w.r.t. $\mathcal{R}'$, and vice versa.*

For the sake of simplicity, we have omitted in the above theorem the case in which only the rule Initial is applied in the needed narrowing derivation, which corresponds to zero LNT steps.

**The extended LNT calculus.**

Here, we consider functional logic languages with a more general operational principle, namely a combination of needed narrowing and residuation. Fortunately, the translation method of Hanus and Prehofer [1999] can be easily extended to cover programs containing evaluation annotations; in particular, flexible (resp. rigid) functions are

HNF

$$[\![t]\!] \Rightarrow^{id}_{\mathsf{LNT}} t \quad \text{if } t \in \mathcal{X} \text{ or } t = c() \text{ with } c/0 \in \mathcal{C}$$

$$[\![c(t_1, \ldots, t_i, \ldots, t_n)]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} [\![\sigma(c(t_1, \ldots, t'_i, \ldots, t_n))]\!] \quad \text{if } c \in \mathcal{C} \text{ and } \exists i \in \{1, \ldots, n\} \text{ such that } [\![t_i]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} [\![t'_i]\!]$$

Case Eval

$$[\![(f)case\ t\ of\ \{\overline{p_k \to t_k}\}]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} \begin{cases} [\![\sigma((f)case\ t'\ of\ \{\overline{p_k \to t_k}\})]\!] & \text{if } [\![t]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} [\![t']\!] \\ (f)case\ t\ of\ \{\overline{p_k \to t_k}\} & \text{otherwise} \end{cases}$$
$$\text{where } t \notin \mathcal{X} \text{ and } root(t) \notin \mathcal{C}$$

Case Select

$$[\![(f)case\ c(\overline{t_n})\ of\ \{\overline{p_k \to t_k}\}]\!] \Rightarrow^{id}_{\mathsf{LNT}} [\![\sigma(t_i)]\!] \quad \text{if } p_i = c(\overline{x_n}),\ c \in \mathcal{C},\ \sigma = \{\overline{x_n \mapsto t_n}\}$$

Case Guess

$$[\![fcase\ x\ of\ \{\overline{p_k \to t_k}\}]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} [\![\sigma(t_i)]\!] \quad \text{if } \sigma = \{x \mapsto p_i\},\ i = 1, \ldots, k$$

Case Suspension

$$[\![case\ x\ of\ \{\overline{p_k \to t_k}\}]\!] \Rightarrow^{id}_{\mathsf{LNT}} case\ x\ of\ \{\overline{p_k \to t_k}\}$$

Function Eval

$$[\![f(\overline{t_n})]\!] \Rightarrow^{id}_{\mathsf{LNT}} [\![\sigma(r)]\!] \quad \text{if } f(\overline{x_n}) = r \in \mathcal{R} \text{ is a rule with fresh variables and } \sigma = \{\overline{x_n \mapsto t_n}\}$$

Figure 7.3: Extended LNT Calculus

translated by using only $fcase$ (resp. $case$) expressions. Moreover, the LNT calculus of Hanus and Prehofer [1999] can be also extended to correctly evaluate $case/fcase$ expressions. In Figure 7.3, we formalize such an extension. In the extended calculus, there is a new rule Case Suspension which returns the original case expression when a rigid case structure has a variable argument. On the other hand, we distinguish two cases in the Case Eval rule:

- if there is some progress towards the evaluation of the case argument, i.e., the expression is not suspended, then we proceed to evaluate the resulting case expression;

- otherwise, the case expression is suspended and we return the original case expression (without square brackets).

Therefore, the extended calculus is trivially equivalent to needed narrowing with residuation. In order to formally prove it, one just needs to consider the Case Suspension rule in Lemma 78. In the following, we refer to the LNT calculus to mean the LNT calculus of Figure 7.3.

HNF

$$[\![t]\!] \;\Rightarrow\; t \quad \text{if } t \in \mathcal{X} \text{ or } t = c() \text{ with } c/0 \in \mathcal{C}$$

$$[\![c(t_1, \ldots, t_n)]\!] \;\Rightarrow\; c([\![t_1]\!], \ldots, [\![t_n]\!])$$

Case Eval

$$[\![(f)case\ t\ of\ \{\overline{p_k \to t_k}\}]\!] \;\Rightarrow\; \begin{cases} [\![\sigma((f)case\ t'\ of\ \{\overline{p_k \to t_k}\})]\!] & \text{if } [\![t]\!] \Rightarrow [\![t']\!] \\ (f)case\ t\ of\ \{\overline{p_k \to t_k}\} & \text{otherwise} \end{cases}$$
$$\text{if } t \neq case\ x\ of \ldots, t \notin \mathcal{X} \text{ and } root(t) \notin \mathcal{C}$$

Case Select

$$[\![(f)case\ c(\overline{e_n})\ of\ \{\overline{p_k \to t_k}\}]\!] \;\Rightarrow\; [\![\sigma(t_i)]\!] \quad \text{if } p_i = c(\overline{x_n}),\ c \in \mathcal{C},\ \sigma = \{\overline{x_n \mapsto e_n}\}$$

Case Guess

$$[\![(f)case\ x\ of\ \{\overline{p_k \to t_k}\}]\!] \;\Rightarrow\; (f)case\ x\ of\ \{\overline{p_k \to [\![\sigma_k(t_k)]\!]}\}$$
$$\text{if } \sigma_i = \{x \mapsto p_i\},\ i = 1, \ldots, k$$

Function Eval

$$[\![g(\overline{t_n})]\!] \;\Rightarrow\; [\![\sigma(r)]\!] \quad \text{if } g(\overline{x_n}) = r \in \mathcal{R} \text{ is a rule with fresh variables and } \sigma = \{\overline{x_n \mapsto t_n}\}$$

Case-of-Case

$$[\![(f)case\ ((f)case\ x\ of\ \{\overline{p_k \to t_k}\})\ of\ \{\overline{p'_j \to t'_j}\}]\!]$$
$$\Rightarrow\; [\![(f)case\ x\ of\ \{\overline{p_k \to (f)case\ t_k\ of\ \{\overline{p'_j \to t'_j}\}}\}]\!]$$

Figure 7.4: RLNT Calculus

**The RLNT calculus.**

Unfortunately, by simply using the (extended) LNT calculus during partial evaluation, we do not avoid the problems of previous approaches. In particular, one of the main problems comes from the *backpropagation* of variable bindings to the left-hand sides of residual rules. In the context of lazy functional logic languages, this can cause an incorrect restriction on the domain of functions (regarding the ability to compute head normal forms) and, thus, the loss of correctness for the transformation when a term is evaluated during partial evaluation beyond its head normal form (as witnessed by Example 3). The restriction of forbidding the evaluation of head normal forms can drastically reduce the optimization power of the transformation in some cases. Therefore, we propose a *residualizing* version of the LNT calculus which allows us to avoid this restriction. In the new calculus, variable bindings are encoded by case expressions and are considered "residual" code. The inference rules of the new calculus, RLNT (Residualizing LNT), can be seen in Figure 7.4. Let us explain the inference rules defining the one-step relation $\Rightarrow$.

HNF. The HNF rules are used to evaluate terms in head normal form. If the expression is a variable or a 0-ary constructor, the square brackets are removed and the evaluation process stops. Otherwise, the evaluation proceeds with the arguments. This evaluation can be carried in a “don't care” nondeterministic manner.

Case Guess. It represents the main difference w.r.t. the LNT calculus. Here, the rule has been modified in order not to backpropagate the bindings of variables. In particular, we “residualize” the case structure and continue with the evaluation of the different branches (by applying the corresponding substitution in order to propagate bindings forward into the computation). Let us note that, due to the above modification, we do not need to distinguish between *case* and *fcase*.

Case-of-Case. An undesirable effect of the Case Guess rule is that nested case expressions may suspend unnecessarily. Take, for instance, an expression of the form:

$$[\![\texttt{case (case x of \{} \quad \texttt{0} \to \texttt{True} \\ \qquad\qquad\qquad\qquad (\texttt{Succ y}) \to \texttt{False\})} \ \texttt{of \{True} \to \texttt{C x\}}]\!]$$

The evaluation of this expression suspends since the outer case can be only evaluated if the argument is a variable (Case Guess), a function call (Case Eval) or a constructor-rooted term (Case Select). Therefore, to avoid premature suspensions, the Case-of-Case rule is introduced:

$$\begin{array}{l} [\![(f)case\ ((f)case\ e\ of\ \{\overline{p_k \to e_k}\})\ of\ \{\overline{p'_j \to e'_j}\}]\!] \\ \qquad \Rightarrow\ [\![(f)case\ e\ of\ \{\overline{p_k \to (f)case\ e_k\ of\ \{\overline{p'_j \to e'_j}\}}\}]\!] \end{array}$$

This rule moves the outer case inside the branches of the inner one and, thus, the evaluation of some branches can now proceed (similar rules can be found in the Glasgow Haskell Compiler as well as in Wadler's deforestation [Wadler, 1990]). By using the Case-of-Case rule, the above expression can be reduced to:

$$\begin{array}{ll} [\![\texttt{case x of} & \{\texttt{0} \to \texttt{case True of \{True} \to \texttt{C x\}} \\ & (\texttt{Succ y}) \to \texttt{case False of \{True} \to \texttt{C x\}\}}]\!] \end{array}$$

(which can be further simplified with the Case Guess and Case Select rules).

Rigorously speaking, this rule can be expanded into four rules (with the different combinations for *case* and *fcase* expressions), but we keep the above (less formal) presentation for simplicity. Observe that the outer case expression may be duplicated several times, but each copy is now (possibly) scrutinizing a known value, and so the Case Select rule can be applied to eliminate some case constructs.

Case Eval. This rule differs from the corresponding rule in the LNT calculus in that it does not consider case expressions whose argument is a case expression with a variable argument. Rather, they are dealt with the Case-of-Case rule as explained above.

Function Eval, Case Select. These rules coincide with the corresponding rules in the LNT calculus.

In contrast to the LNT calculus, the inference system of Figure 7.4 does not compute answers. Thus, it is completely deterministic, i.e., there is no "don't know" nondeterminism involved in the computations. This means that only one derivation can be issued from a given term (thus, there is no need to introduce a notion of RLNT "tree").

**Example 28** Consider function app to concatenate two lists:

$$\begin{array}{llll}\texttt{app x y} = \texttt{case x of} & \{ & [] & \to \texttt{y} \; ; \\ & & (\mathsf{a} : \mathsf{b}) & \to \mathsf{a} : (\texttt{app b y}) \;\; \} \end{array}$$

Given the call "app (app x y) z", we can prove the following (partial) derivation using the rules of the RLNT calculus:

$$\begin{array}{l}
[\![\texttt{app (app x y) z}]\!] \\
\Rightarrow [\![\texttt{case (app x y) of } \{[] \to \texttt{z};\ (\mathsf{a} : \mathsf{b}) \to (\mathsf{a} : \texttt{app b z})\}]\!] \\
\Rightarrow [\![\texttt{case (case x of } \{[] \to \texttt{y};\ (\mathsf{a}' : \mathsf{b}') \to (\mathsf{a}' : \texttt{app b}'\texttt{ y})\}\texttt{) of } \{[] \to \texttt{z}; \\
\hfill (\mathsf{a} : \mathsf{b}) \to (\mathsf{a} : \texttt{app b z})\}]\!]
\end{array}$$

$$\begin{array}{llll}
\Rightarrow [\![\texttt{case x of } \{ & [] & \to \texttt{case y of } \{[] \to \texttt{z};\ (\mathsf{a} : \mathsf{b}) \to (\mathsf{a} : \texttt{app b z})\}; \\
 & (\mathsf{a}' : \mathsf{b}') & \to \texttt{case } (\mathsf{a}' : \texttt{app b}'\texttt{ y}) \texttt{ of } \{[] : \texttt{z}; \\
 & & \qquad\qquad (\mathsf{a} : \mathsf{b}) \to (\mathsf{a} : \texttt{app b z})\}]\!] \\
\Rightarrow \texttt{case x of } \{ & [] & \to [\![\texttt{case y of } \{[] \to \texttt{z};\ (\mathsf{a} : \mathsf{b}) \to (\mathsf{a} : \texttt{app b z})\}]\!]; \\
 & (\mathsf{a}' : \mathsf{b}') & \to [\![\texttt{case } (\mathsf{a}' : \texttt{app b}'\texttt{ y}) \texttt{ of } \{[] \to \texttt{z}; \\
 & & \qquad\qquad (\mathsf{a} : \mathsf{b}) \to (\mathsf{a} : \texttt{app b z})\}]\!] \\
\stackrel{*}{\Rightarrow} \texttt{case x of } \{ & [] & \to \texttt{case y of } \{[] \to \texttt{z};\ (\mathsf{a} : \mathsf{b}) \to (\mathsf{a} : [\![\texttt{app b z})]\!]\}; \\
 & (\mathsf{a}' : \mathsf{b}') & \to [\![\texttt{case } (\mathsf{a}' : \texttt{app b}'\texttt{ y}) \texttt{ of } \{[] \to \texttt{z}; \\
 & & \qquad\qquad (\mathsf{a} : \mathsf{b}) \to (\mathsf{a} : \texttt{app b z})\}]\!] \\
\stackrel{*}{\Rightarrow} \texttt{case x of } \{ & [] & \to \texttt{case y of } \{[] \to \texttt{z};\ (\mathsf{a} : \mathsf{b}) \to (\mathsf{a} : [\![\texttt{app b z}]\!])\}; \\
 & (\mathsf{a}' : \mathsf{b}') & \to (\mathsf{a}' : [\![\texttt{app (app b}'\texttt{ y) z}]\!])\}
\end{array}$$

It is worthwhile to note that the resulting RLNT calculus shares many similarities with the driving mechanism of Sørensen *et al.* [1996] and Wadler's deforestation [Wadler, 1990] (although we obtained it independently as a refinement of the original LNT calculus to avoid the backpropagation of bindings). The main differences w.r.t. the driving mechanism are that we include the Case-of-Case rule and that driving is defined for `if_then_else` constructs too (which, at the same time, can be expressed in our representation by means of case expressions). The main difference w.r.t. deforestation lies in the Case Guess rule, where the constructor patterns $p_i$ are substituted in the different branches, like in the driving transformation. Although it may seem only a slight difference, situations may arise during transformation in which our calculus takes advantage of the sharing between different arguments by virtue of instantiation while deforestation may not (e.g., the former passes the so-called KMP test [Consel and Danvy, 1989] while the latter does not).

In the following, we state the relation between LNT and RLNT steps. Note that both the LNT and RLNT calculi have rules which recursively require the application of further inference rules. For instance, this is the case of the Case Eval rule. For these rules, by abuse, we will often say sentences like "the application of rule Case Eval (firing Case Select)" to mean that, after some recursive calls to Case Eval, we finally apply an inference step with the Case Select rule. In order to relate both calculi, first we introduce some preliminary notions:

- To simplify the proof of equivalence between LNT and RLNT calculi, we consider that the HNF rules of the LNT calculus are defined as follows:

  $$\begin{array}{rll} [\![t]\!] & \Rightarrow^{id}_{\mathsf{LNT}} & t \quad \text{if } t \in \mathcal{X} \text{ or } t = c() \text{ with } c/0 \in \mathcal{C} \\ [\![c(t_1, \ldots, t_i, \ldots, t_n)]\!] & \Rightarrow_{\mathsf{LNT}} & c([\![t_1]\!]^{id}, \ldots, [\![t_n]\!]^{id}) \quad \text{if } c \in \mathcal{C} \end{array}$$

  In the second rule, we use superindexes to label each argument with the substitution computed so far. This allows us to state precisely the relation with the LNT calculus including the original HNF rules (see below). Note that, thanks to this modification, the resulting calculus does not propagate bindings between the arguments of a constructor-rooted term (as the RLNT calculus does). Nevertheless, the modified calculus is still equivalent to the original one in the sense that whenever the original calculus computes

  $$[\![c(t_1, \ldots, t_n)]\!] \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{LNT}}\ d$$

  the modified calculus computes

  $$[\![c(t_1, \ldots, t_n)]\!] \overset{*}{\Rightarrow}_{\mathsf{LNT}}\ c(d_1^{\theta_1}, \ldots, d_n^{\theta_n})$$

  with $\sigma = \theta_1 \Uparrow \ldots \Uparrow \theta_n$ and $d = \sigma(c(d_1, \ldots, d_n))$. The operator "$\Uparrow$" is called *parallel composition* [Palamidessi, 1990] and can be defined as follows:

  $$\theta_1 \Uparrow \ldots \Uparrow \theta_n = mgu(\widehat{\theta_1} \cup \ldots \cup \widehat{\theta_n})$$

  where $\widehat{\theta} = \{x_1 = t_n, \ldots, x_k = t_k\}$ if $\theta = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ and the $mgu$ operator is defined on sets of equations. The parallel composition of substitutions can be understood as the $mgu$ over substitutions instead of terms.

  In the following, we will establish the relation between LNT and RLNT calculi by considering the above modification of the HNF rules in the LNT calculus.

- An auxiliary function $\mathcal{CT}$ is defined to translate expressions where bindings are encoded by case expressions (with a variable argument) into ordinary terms:

  $$(\sigma, t) \in \mathcal{CT}(s) \ \text{ if } \ s \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{CRG}}\, t$$

  where $\Rightarrow_{\mathsf{CRG}}$ (Case Residual Guess) is defined by

  $$\begin{array}{lll} fcase\ x\ of\ \{\overline{p_k \to t_k}\} & \Rightarrow^{\sigma}_{\mathsf{CRG}} & \sigma(t_i) \quad \text{if } \sigma = \{x \mapsto p_i\},\ i = 1, \ldots, k \\ c(t_1, \ldots, t_n) & \Rightarrow^{\sigma}_{\mathsf{CRG}} & c(t_1, \ldots, t'_i, \ldots, t_n) \quad \text{if } t_i \ \Rightarrow^{\sigma}_{\mathsf{CRG}}\ t'_i \end{array}$$

  Note that only "residual" (flexible) case structures (i.e., outside square brackets) can be reduced by $\Rightarrow_{\mathsf{CRG}}$. Here, the aim is to remove those case structures already evaluated by the Case Guess rule of the RLNT calculus. Function $\mathcal{CT}$ will be helpful to relate the LNT and RLNT calculi. For instance, if we apply function $\mathcal{CT}$ to the expression:

  $$\begin{array}{ll} \texttt{fcase x of} & \{\texttt{A} \to \texttt{fcase y of } \{\texttt{0} \to [\![\texttt{g 0}]\!]\}; \\ & \ \texttt{B} \to [\![\texttt{fcase y of } \{\texttt{Succ 0} \to \texttt{g (Succ 0)}\}]\!]\} \end{array}$$

  we obtain the set:

  $$\{(\{\texttt{x} \mapsto \texttt{A}, \texttt{y} \mapsto \texttt{0}\}, [\![\texttt{g 0}]\!]),\ (\{\texttt{x} \mapsto \texttt{B}\}, [\![\texttt{fcase y of } \{\texttt{Succ 0} \to \texttt{g (Succ 0)}\}]\!])\}$$

  which denotes the associated computed bindings and terms in the LNT calculus.

- On the other hand, each LNT step using the Case Eval rule (firing Case Guess) may correspond to a number of RLNT steps using the rule Case Eval (firing Case-of-Case to move forward some inner case expression with a variable argument) followed by an application of the rule Case-of-Case and, then, Case Guess. For instance, given an expression $t$ of the form

  $$fcase\ (fcase\ (fcase\ x\ of\ \{\overline{p_k \to t_k}\})\ of\ \ldots)\ of\ \ldots$$

  we can perform the LNT step using Case Eval (firing Case-Guess):

  $$[\![t]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} [\![fcase\ (fcase\ (\sigma(t_i))\ of\ \ldots)\ of\ \ldots]\!]$$

  where $\sigma = \{x \mapsto p_i\}$ for some $i \in \{1, \ldots, k\}$. However, in order to mimic this LNT step, we need to perform a RLNT sequence of the form

  $$\begin{array}{lll} [\![t]\!] & \Rightarrow_{\mathsf{Case\ Eval}} & [\![fcase\ (fcase\ x\ of\ \{\overline{p_k \to fcase\ t_k\ of\ \ldots}\})\ of\ \ldots]\!] \\ & \Rightarrow_{\mathsf{Case-of-Case}} & [\![fcase\ x\ of\ \{\overline{p_k \to fcase\ (fcase\ t_k\ of\ \ldots)\ of\ \ldots}\}]\!] \\ & \Rightarrow_{\mathsf{Case\ Guess}} & fcase\ x\ of\ \{\overline{p_k \to [\![\sigma_k(fcase\ (fcase\ t_k\ of\ \ldots)\ of\ \ldots)]\!]}\} \end{array}$$

  where $\sigma_i = \{x \mapsto p_i\}$ for all $i = 1, \ldots, k$. Note that in the above sequence the first step applies rule Case Eval (firing Case-of-Case). In the following, we are interested in proving that, for each RLNT derivation of the above form, there exists a corresponding LNT step, and vice versa. For this purpose, we introduce the notion of *finished* RLNT "step" (by abuse, here we use the word "step" even if it consists of the application of several ordinary RLNT rules).

**Definition 81 (finished RLNT step)** *Given a RLNT derivation, we say that it corresponds to a "finished" RLNT step if it applies:*

1. *one rule* Case Select, Case Guess *or* Function Eval,
2. *one rule* Case Eval *(firing either* Case Select, Case Guess *or* Function Eval*), or*
3. *a finite number of times rule* Case Eval *(firing* Case-of-Case*) followed by rule* Case-of-Case *and, then,* Case Guess.

Now, we prove the precise relation between a LNT step and a (finished) RLNT step.

**Lemma 82** *Let $\mathcal{R}$ be a flat program and $s, s'$ be terms (possibly containing occurrences of "$[\![$" and "$]\!]$") such that $(\theta, s) \in \mathcal{CT}(s')$. For each LNT step $s \Rightarrow^{\sigma}_{\mathsf{LNT}} t$, there exists a finished RLNT step $s' \stackrel{+}{\Rightarrow} t'$ such that $(\sigma \circ \theta, t) \in \mathcal{CT}(t')$.*

*Proof.* We prove the claim by considering two cases separately depending on the *depth* of the inference step, i.e., the number of inference rules which are fired in order to perform the LNT step $s \Rightarrow^{\sigma}_{\mathsf{LNT}} t$. In particular, we consider the case when the LNT step performs an inference of depth one and when it performs and inference of depth two or higher.

**(depth one).** Here, we consider the following cases depending on the rule applied.

HNF, Case Select and Function Eval. Since they are defined identically in both calculi (note that we consider the modified version of the HNF rules in the LNT calculus), the proof is similar (but simpler) to the proof for Case Guess below.

Case Guess. Here, we assume that $s$ is of the form $[\![ fcase\ x\ of\ \{\overline{p_k \to t_k}\} ]\!]$ [6] and the LNT step:

$$[\![ fcase\ x\ of\ \{\overline{p_k \to t_k}\} ]\!] \Rightarrow^{\sigma}_{\mathsf{LNT}} [\![ \sigma(t_i) ]\!] = t$$

has been performed, with $\sigma = \{x \mapsto p_i\}$ and $i \in \{1, \ldots, k\}$.

[6] Let us note that, in general, $s$ can be of the form $C[\ldots [\![ fcase\ x\ of\ \{\overline{p_k \to t_k}\} ]\!] \ldots]$, where $C$ is an arbitrary context which only contains constructor symbols above the subterm $[\![ fcase\ x\ of\ \{\overline{p_k \to t_k}\} ]\!]$. The proof for this general case would proceed in a similar way by taking into account the appropriate context.

Now, let us consider a term $s'$ with $(\theta, s) \in \mathcal{CT}(s')$. By definition of $\mathcal{CT}$, term $s'$ can contain a number of $fcase$ structures with variable arguments outside square brackets (which encode substitution $\theta$) and, finally, some branch $s$. Here, we denote by $C[\ldots s \ldots]$ such a term $s'$, where $C$ denotes an arbitrary context of $fcase$ constructs with variable arguments (without occurrences of "$\llbracket$" and "$\rrbracket$" above the subterm $s$). Therefore, Case Guess rule (from the RLNT calculus) is also applicable to the term $s'$:

$$C[\ldots \llbracket fcase\ x\ of\ \{\overline{p_k \to t_k}\}\rrbracket \ldots] \Rightarrow C[\ldots fcase\ x\ of\ \{\overline{p_k \to \llbracket \sigma_k(t_k)\rrbracket}\} \ldots]$$

with $\sigma_i = \{x \mapsto p_i\}$ for all $i = 1, \ldots, k$.

Finally, since $(\theta, s) \in \mathcal{CT}(C[\ldots \llbracket fcase\ x\ of\ \{\overline{p_k \to t_k}\}\rrbracket \ldots])$, it is immediate that $(\sigma \circ \theta, t) \in \mathcal{CT}(t')$ with $\sigma = \sigma_i$ (for some $i \in \{1, \ldots, n\}$) and $t' = C[\ldots fcase\ x\ of\ \{\overline{p_k \to \llbracket \sigma(t_k)\rrbracket}\} \ldots]$.

**(depth two or higher).** Here, we have to consider only the application of rule Case Eval of the LNT calculus.

Case Eval. Note that the definition of Case Eval is the same in both calculi (LNT and RLNT) except when the argument of the case expression is a (flexible) case expression with a variable argument, i.e., when we apply Case Eval (firing Case Guess). Therefore, we only prove the latter case.

Then, we have that $s$ is a term which contains $n$ nested case expressions (labeled with a superindex for preciseness):[7]

$$\llbracket (f)case^1(\ldots (f)case^{n-1}((f)case^n\ x\ of\ \{\overline{p_k \to t_k}\})\ of\ \{\overline{p'_k \to t'_k}\} \ldots)\rrbracket$$

and the LNT step:

$$s \Rightarrow^{\sigma_i}_{\mathsf{LNT}} \llbracket \sigma_i((f)case^1((f)case^2(\ldots (f)case^{n-1}\ t_i\ of\ \{\overline{p'_i \to t'_i}\} \ldots)))\rrbracket$$

has been performed using rule Case Eval (firing Case Guess), with $\sigma = \{x \mapsto p_i\}$ and $i \in \{1, \ldots, k\}$.

Analogously to the base case, let us consider a term $s'$ with $(\theta, s) \in \mathcal{CT}(s')$. By definition of $\mathcal{CT}$, term $s'$ can contain a number of $fcase$ structures with variable arguments outside square brackets (which encode substitution $\theta$) and, finally, some branch $s$:

$$\begin{aligned} s' = C[\ldots \llbracket (f)case^1(\ldots (f)case^{n-1}((f)case^n\ x\ of\ \{\overline{p_k \to t_k}\}) \\ of\ \{\overline{p'_k \to t'_k}\} \ldots)\rrbracket \ldots] \end{aligned}$$

[7]Again, we ignore the possible occurrences of outermost constructor symbols for simplicity. The same considerations as in the previous footnote apply.

Here, we denote by $C[\ldots s \ldots]$ such a term $s'$, where $C$ denotes an arbitrary context of $fcase$ constructs with variable arguments (without occurrences of "$\llbracket$" and "$\rrbracket$" above the subterm $s$). Then, there exists a RLNT sequence of length $n-1$ where we apply $n-2$ times rule Case Eval (firing Case-of-Case) and, finally, a RLNT step with rule Case-of-Case:

$$
\begin{array}{lll}
s' & \Rightarrow & C[\ldots \llbracket (f)case^1(\ldots (f)case^n\ x\ of \\
& & \quad \overline{\{p_k \to (f)case^{n-1}\ t_k\ of\ \{\overline{p'_k \to t'_k}\}\}} \ldots) \rrbracket \ldots] \\
& & \vdots \\
& \Rightarrow & C[\ldots \llbracket (f)case^1((f)case^n\ x\ of \\
& & \quad \overline{\{p_k \to (f)case^2((f)case^3(\ldots (f)case^{n-1}\ t_k\ of\ \{\overline{p'_k \to t'_k}\} \ldots)))} \rrbracket \ldots] \\
& \Rightarrow & C[\ldots \llbracket (f)case^n\ x\ of \\
& & \quad \overline{\{p_k \to (f)case^1((f)case^2(\ldots (f)case^{n-1}\ t_k\ of\ \{\overline{p'_k \to t'_k}\} \ldots))} \rrbracket \ldots]
\end{array}
$$

where each step moves the inner case $(f)case^n$ to an outer position. Then, we can apply a RLNT step with rule Case Guess to the last expression, thus obtaining:

$$
\begin{array}{l}
C[\ldots (f)case^n\ x\ of \\
\quad \overline{\{p_k \to \llbracket \sigma_k((f)case^1((f)case^2(\ldots (f)case^{n-1}\ t_k\ of\ \{\overline{p'_k \to t'_k}\rrbracket\} \ldots)))} \ldots]
\end{array}
$$

with $\sigma_i = \{x \mapsto p_i\}$ for all $i = 1, \ldots, k$.

Finally, since $(\theta, s) \in \mathcal{CT}(s')$, it is immediate that $(\sigma \circ \theta, t) \in \mathcal{CT}(t')$ with $\sigma = \sigma_i$ (for some $i \in \{1, \ldots, n\}$).

□

Now, we prove that for each finished RLNT step, there exists an associated LNT step.

**Lemma 83** *Let $\mathcal{R}$ be a flat program and $s'$ a term (possibly containing occurrences of "$\llbracket$" and "$\rrbracket$"). If $s' \stackrel{+}{\Rightarrow} t'$ is a finished RLNT step, then there exists a term $s$ with $(\theta, s) \in \mathcal{CT}(s')$ such that either $s$ suspends or there exist LNT steps $s \Rightarrow^{\sigma_i}_{\mathsf{LNT}} t_i$, where $(\sigma_i \circ \theta, t_i) \in \mathcal{CT}(t')$, for $i = 1, \ldots, k$, $k > 0$.*

*Proof.* We prove the claim by considering two cases separately: when the finished RLNT sequence $s' \stackrel{+}{\Rightarrow} t'$ performs one step and when it performs two or more steps.

**(one step).** Then, we have the following cases depending on the applied rule:

HNF, Case Select and Function Eval. Since they are defined identically in both calculi (note that we consider the modified version of the HNF rules in the LNT calculus), the proof is similar (but simpler) to the proof for Case Guess below.

Case Guess. Then $s'$ is of the form

$$C[\ldots [\![(f)case\ x\ of\ \{\overline{p_k \to t_k}\}]\!] \ldots]$$

where $s'$ can contain a number $fcase$ structures with variable arguments outside square brackets (which encode some substitution $\theta$) and, finally, at least one branch $[\![(f)case\ x\ of\ \{\overline{p_k \to t_k}\}]\!]$.[8] Then, we consider the RLNT step

$$s' \Rightarrow C[\ldots (f)case\ x\ of\ \{\overline{p_k \to [\![\sigma_k(t_k)]\!]}\} \ldots]$$

where $\sigma_i = \{x \mapsto p_i\}$ for all $i = 1, \ldots, k$. By definition of $\mathcal{CT}$, there exists a term $s = [\![(f)case\ x\ of\ \{\overline{p_k \to t_k}\}]\!]$ such that $(\theta, s) \in \mathcal{CT}(s')$ for some substitution $\theta$. Now, if the case expression is rigid, then the claim follows trivially since $s$ suspends in the LNT calculus. Therefore, now we consider that the case expression is flexible. Then, we can perform the steps

$$s \Rightarrow^{\sigma_i}_{\mathsf{LNT}} [\![\sigma_i(t_i)]\!]$$

for all $i = 1, \ldots, k$. Trivially,

$$(\sigma_i \circ \theta, [\![\sigma_i(t_i)]\!]) \in \mathcal{CT}(C[\ldots fcase\ x\ of\ \{\overline{p_k \to [\![\sigma_k(t_k)]\!]}\} \ldots])$$

for all $i = 1, \ldots, k$, which concludes the proof.

Case Eval (firing Case Select, Case Guess or Function Eval). The proof is perfectly analogous to the previous cases.

Case Eval (firing a different rule) and Case-of-Case. The RLNT step would not be finished.

**(two or more steps).** Here we consider finished RLNT steps which apply two or more rules. Thus, we only consider the application of rule Case Eval of the LNT calculus.

Case Eval. By definition of finished RLNT step, the only possibility is that it performs a (finite) number of steps with rule Case Eval (firing Case-of-Case) followed by a step with rule Case-of-Case and, then, a step with rule Case Guess. In order to apply a sequence of rules Case Eval (firing Case-of-Case), $s'$ must be of the form:

$$C[\ldots [\![(f)case^1(\ldots (f)case^{n-1}((f)case^n\ x\ of\ \{\overline{p_k \to t_k}\})\ of\ \{\overline{p'_k \to t'_k}\} \ldots)]\!] \ldots]$$

[8] Let us note that here we ignore the occurrences of outermost constructor symbols. As mentioned before, the extension of the proof to cover this case is simple.

where $s'$ can contain a number of $fcase$ structures with variable arguments outside square brackets (which encode some substitution $\theta$) and, finally, at least one branch of the form

$$[\![(f)case^1(\ldots(f)case^{n-1}((f)case^n\ x\ of\ \{\overline{p_k \to t_k}\})\ of\ \{\overline{p'_k \to t'_k}\}\ldots)]\!]$$

which is composed by $n$ nested case expressions labeled with a superindex (for the sake of preciseness). Then, there exists a RLNT sequence of length $n-1$ where we apply $n-2$ times rule Case Eval (firing Case-of-Case) and, finally, a RLNT step with Case-of-Case:

$$\begin{array}{ll}
s' & \Rightarrow C[\ldots[\![(f)case^1(\ldots(f)case^n\ x\ of \\
& \qquad \overline{\{p_k \to (f)case^{n-1}\ t_k\ of\ \{\overline{p'_k \to t'_k}\}}\}\ldots)]\!]\ldots] \\
& \vdots \\
& \Rightarrow C[\ldots[\![(f)case^1((f)case^n\ x\ of \\
& \qquad \overline{\{p_k \to (f)case^2((f)case^3(\ldots(f)case^{n-1}\ t_k\ of\ \{\overline{p'_k \to t'_k}\}\ldots)))}]\!]\ldots] \\
& \Rightarrow C[\ldots[\![(f)case^n\ x\ of \\
& \qquad \overline{\{p_k \to (f)case^1((f)case^2(\ldots(f)case^{n-1}\ t_k\ of\ \{\overline{p'_k \to t'_k}\}\ldots))}]\!]\ldots]
\end{array}$$

where each step moves the inner case $(f)case^n$ to an outer position. Then, we can apply a RLNT step with rule Case Guess to the last expression, thus obtaining:

$$\begin{array}{l}
C[\ldots(f)case^n\ x\ of \\
\qquad \overline{\{p_k \to [\![\sigma_k((f)case^1((f)case^2(\ldots(f)case^{n-1}\ t_k\ of\ \{\overline{p'_k \to t'_k}]\!]\}\ldots)))}\ldots]
\end{array}$$

with $\sigma_i = \{x \mapsto p_i\}$ for all $i = 1,\ldots,k$.

By definition of $\mathcal{CT}$, there exists a term

$$s = [\![(f)case^1(\ldots(f)case^{n-1}((f)case^n\ x\ of\ \{\overline{p_k \to t_k}\})\ of\ \{\overline{p'_k \to t'_k}\}\ldots)]\!]$$

such that $(\theta, s) \in \mathcal{CT}(s')$ for some substitution $\theta$. Now, if the inner case expression is rigid, then the claim follows trivially since $s$ suspends in the LNT calculus. Therefore, now we consider that this case expression is flexible. Then, we can perform the steps

$$s \Rightarrow^{\sigma_i}_{\mathsf{LNT}} [\![\sigma_i((f)case^1((f)case^2(\ldots(f)case^{n-1}\ t_i\ of\ \{\overline{p'_i \to t'_i}\}\ldots)))]\!]$$

for all $i = 1,\ldots,k$. Trivially,

$$(\sigma_i \circ \theta, [\![\sigma_i((f)case^1((f)case^2(\ldots(f)case^{n-1}\ t_i\ of\ \{\overline{p'_i \to t'_i}\}\ldots))))]\!]) \in \mathcal{CT}(s')$$

for all $i = 1,\ldots,k$, which concludes the proof.

□

As a consequence of the previous lemmata, we can establish the following relation between the LNT and RLNT derivations. The result holds for non-floundering derivations (i.e., derivations which do not end up in an outermost rigid case expression). Note that we make it explicit for LNT derivations. However, this is not necessary for RLNT derivations since the definition of $\mathcal{CT}$ only considers flexible case's and, therefore, it implicitly excludes floundering derivations.

**Theorem 84** *Let $\mathcal{R}$ be a flat program.*

*i) Given a term $s$ and a (non-floundering) LNT derivation $s \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{LNT}} t$ then, for all $s'$ with $(\theta, s) \in \mathcal{CT}(s')$, there exists a RLNT derivation $s' \overset{+}{\Rightarrow} t'$ whose steps are finished (according to Definition 81) and such that $(\sigma \circ \theta, t) \in \mathcal{CT}(t')$.*

*ii) Given a term $s'$ and a RLNT derivation $s' \overset{+}{\Rightarrow} t'$ whose steps are finished (according to Definition 81), there exists a term $s$ with $(\theta, s) \in \mathcal{CT}(s')$ such that either $s$ suspends or there exist LNT derivations $s \overset{*}{\Rightarrow}{}^{\sigma_i}_{\mathsf{LNT}} t_i$, where $(\sigma_i \circ \theta, t) \in \mathcal{CT}(t')$ for $i = 1, \ldots, k,\ k > 0$.*

*Proof.* By induction on the length of the derivations and the application of Lemmata 82 and 83. □

The following result is an immediate consequence of Theorem 84. Informally, it states that, for each (successful) LNT derivation from $t$ to a constructor term $d$ computing $\sigma$, there exists a corresponding RLNT derivation from $t$ to $t'$ in which the computed value $d$ and answer $\sigma$ are encoded in $t'$ by case expressions.

**Corollary 85** *Let $t$ be a term, $d$ a constructor term, and $\mathcal{R}$ a flat program. For each (non-floundering) LNT derivation $t \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{LNT}} d$, there exists a RLNT derivation $t \overset{*}{\Rightarrow} t'$ such that $(\sigma, d) \in \mathcal{CT}(t')$, and vice versa.*

*Proof.* It follows from Theorem 84, considering that the RLNT derivation only performs finished RNLT steps according to Definition 81. □

Let us conclude by summarizing the main ideas presented so far. First, we have presented an appropriate formulation of needed narrowing which has been proven equivalent to the LNT calculus in [Hanus and Prehofer, 1999]. However, for our purposes, it has been more convenient to provide a slightly modified version of the LNT calculus, which essentially differs from the original formulation in that we allow nested case expressions. This LNT calculus has been proven equivalent to needed narrowing relying on the proofs of [Hanus and Prehofer, 1999]. Then, we have provided an extended LNT calculus which considers evaluation annotations and which is equivalent to needed narrowing combined with residuation. Finally, we have stated the relation between our RLNT calculus and the extended LNT calculus (by considering a modified version of the HNF rules to evaluate terms in head normal form). Thus, Theorem 84 and Corollary 85 are also valid w.r.t. the LNT calculus of Figure 7.3 when terms in head normal form are not evaluated. Moreover, this relation can be easily generalized to arbitrary derivations by slightly modifying function $\mathcal{CT}$:

$$(\sigma, t) \in \mathcal{CT}(s) \;\text{ if }\; s \stackrel{*}{\Rightarrow}{}^{\sigma}_{\mathsf{CRG}}\, t$$

where $\Rightarrow_{\mathsf{CRG}}$ (Case Residual Guess) is defined by

$$\begin{array}{lll}
fcase\ x\ of\ \{\overline{p_k \to t_k}\} & \Rightarrow^{\sigma}_{\mathsf{CRG}} & \sigma(t_i) \quad \text{if } \sigma = \{x \mapsto p_i\},\ i = 1, \ldots, k \\
fcase\ c(\overline{e_n})\ of\ \{\overline{p_k \to t_k}\} & \Rightarrow^{\sigma}_{\mathsf{CRG}} & \sigma(t_i) \quad \text{if } p_i = c(\overline{x_n}),\ c \in \mathcal{C},\ \sigma = \{\overline{x_n \mapsto e_n}\} \\
c(t_1, \ldots, t_n) & \Rightarrow^{\sigma}_{\mathsf{CRG}} & \sigma(c(t_1, \ldots, t'_i, \ldots, t_n)) \quad \text{if } t_i \Rightarrow^{\sigma}_{\mathsf{CRG}} t'_i
\end{array}$$

Note that the main difference is that bindings which come from the evaluation of some argument of a constructor-rooted term are propagated to the remaining arguments (as the original HNF rules of Figure 7.3 do). This motivates the inclusion of the second rule above in order to reduce (residual) case expressions in which the variable argument has been instantiated by the application of the third rule. In the following, we consider the above definition of function $\mathcal{CT}$ in order to allow the evaluation of terms in head normal form. As a summary, we have stated a precise relation between needed narrowing (where the previous framework for partial evaluation is defined) and the new RLNT calculus. This will be helpful to prove the correctness of partial evaluation based on the RLNT calculus.

### 7.2.3 Correctness of RLNT-based Partial Evaluation

In this section, we demonstrate the correctness of partial evaluation based on the RLNT calculus. The following definition recalls how to construct the resultants associated to a set of RLNT derivations, which will produce the partially evaluated program.

**Definition 86 (partial evaluation)** *Let $\mathcal{R}$ be a program, $S = \{s_1, \ldots, s_n\}$ a finite set of terms, and $\mathcal{A}_1, \ldots, \mathcal{A}_n$ finite (possibly partial) RLNT computations whose steps are finished (according to Definition 81) for $s_1, \ldots, s_n$:*

$$\mathcal{A}_k = (s_k \stackrel{+}{\Rightarrow} t_k), \quad k = 1, \ldots, n$$

*Let $\rho$ be an independent renaming of $S$. Then, we define a partial evaluation $\mathcal{R}'$ of $S$ in $\mathcal{R}$ (under $\rho$) as the set of rewrite rules:*

$$\bigcup_{k=1,\ldots,n} \{\rho(s_k) \to ren_\rho(t_k)\}$$

A common restriction in nearby related program transformations is to forbid unfolding using program rules whose right-hand side is not linear. This avoids the duplication of calls under an eager (call-by-value) semantics as well as for lazy (call-by-name) semantics implementing the *sharing* of common variables. Since our computation model is based on a lazy calculus which does not embody the sharing of variables, we cannot incur into the risk of duplicated computations. Nevertheless, if sharing is considered (as in the language Curry), this restriction can be implemented by requiring the use of right-linear program rules within the Function Eval rule.

Regarding partial evaluation of programs with *flex*/*rigid* evaluation annotations, in Chapter 4, we introduced a special processing which correctly infers the evaluation annotations for residual function definitions. This method splits resultants by introducing several intermediate functions in order not to mix bindings which come from the evaluation of flexible and rigid functions. Moreover, in order to avoid the creation of a large number of intermediate functions, only the computation of a single needed narrowing step for suspended expressions was allowed. Now, thanks to the use of case expressions (instead of functions defined by patterns as in Chapter 4), we are able to resume the specialization of suspended expressions beyond the threshold of the first narrowing step without being forced to split the associated resultant into different rules (and hence without increasing the size of the residual program). This is justified by the fact that case constructs preserve the *rigid*/*flexible* nature of the functions.[9] The following example illustrates that the use of case constructs may lead to simpler residual programs.

**Example 29** Consider again the program of Example 11:

$$\begin{array}{lcll} \mathtt{f\ 0\ (Succ\ 0)} & = & \mathtt{0} & \%\ \mathtt{flex} \\ \mathtt{g\ 0\ 0} & = & \mathtt{(Succ\ 0)} & \%\ \mathtt{rigid} \\ \mathtt{h\ 0} & = & \mathtt{0} & \%\ \mathtt{flex} \end{array}$$

and the following partial evaluation for the term `f x (g y (h z))`, which can be obtained by using the technique introduced in Chapter 4:

$$\begin{array}{lll} \mathtt{f'\ 0\ y\ z} & \mathtt{f'_1\ y\ z} & \%\ \mathtt{flex} \\ \mathtt{f'_1\ (Succ\ 0)\ z} & \mathtt{f'_2\ z} & \%\ \mathtt{rigid} \\ \mathtt{f'_2\ 0} & \mathtt{f'_3} & \%\ \mathtt{flex} \\ \mathtt{f'_3} & \mathtt{0} & \%\ \mathtt{flex} \end{array}$$

where `f x (g y (h z))` is renamed as $\mathtt{f'\ x\ y\ z}$. The original program can be translated to our abstract representation as follows:

$$\begin{array}{rcl} \mathtt{f\ x\ y} & = & \mathtt{fcase\ x\ of}\ \{\mathtt{0} \to \mathtt{fcase\ y\ of}\ \{\mathtt{(Succ\ 0)} \to \mathtt{0}\}\} \\ \mathtt{g\ x\ y} & = & \mathtt{case\ x\ of}\ \{\mathtt{0} \to \mathtt{case\ y\ of}\ \{\mathtt{0} \to \mathtt{(Succ\ 0)}\}\} \\ \mathtt{h\ x} & = & \mathtt{fcase\ x\ of}\ \{\mathtt{0} \to \mathtt{0}\} \end{array}$$

[9]Indeed, the treatment for case/fcase expressions is the same in the RLNT calculus.

The following partial evaluation for $\texttt{f x (g y (h z))}$, constructed according to Definition 86, avoids the introduction of three intermediate rules and, thus, is noticeably simplified:

$$\texttt{f}'\ \texttt{x y z} \quad = \quad \texttt{fcase x of } \{\texttt{0} \to \texttt{case y of } \{(\texttt{Succ 0}) \to \texttt{fcase z of } \{\texttt{0} \to \texttt{0}\}\}\}$$

In order to demonstrate the correctness of our partial evaluation scheme, we first adapt the notion of closedness to our abstract representation.

**Definition 87** *Let $S$ be a set of terms and $t$ be a term. We say that $t$ is $S$-closed if $closed(S,t)$ holds, where the relation "closed" is defined inductively as follows:* $closed(S,t) =$

$$\begin{cases} true & \textit{if } t \in \mathcal{X} \\ closed(S,t_1) \wedge \ldots \wedge closed(S,t_n) & \textit{if } t = c(t_1,\ldots,t_n),\ c \in \mathcal{C} \\ closed(S,e)\ \wedge\ \bigwedge_{i\in\{1,\ldots,k\}} closed(S,t_i) & \textit{if } t = (f)case\ e\ of\ \{\overline{p_k \to t_k}\} \\ \bigwedge_{t'\in\mathcal{R}an(\theta)} closed(S,t') & \textit{if } \exists\theta, \exists s \in S \textit{ such that } t = \theta(s) \end{cases}$$

*A set of terms $T$ is $S$-closed, written $closed(S,T)$, if $closed(S,t)$ holds for all $t \in T$, and we say that a TRS $\mathcal{R}$ is $S$-closed if $closed(S,\mathcal{R}_{calls})$ holds. Here we denote by $\mathcal{R}_{calls}$ the set of the right-hand sides of the rules in $\mathcal{R}$.*

According to this definition, variables are always closed, whereas an operation-rooted term is $S$-closed if it is an instance of some term in $S$ and the terms in the matching substitution are recursively $S$-closed. On the other hand, for case expressions, we have two nondeterministic ways to proceed: either by checking the closedness of their arguments or by proceeding as in the case of an operation-rooted term. For instance, a case expression such as:

$$case\ e\ of\ \{p_1 \to t_1; \ldots; p_k \to t_k\}$$

can be proved closed w.r.t. $S$ either by checking that the set $\{e, t_1, \ldots, t_k\}$ is $S$-closed[10] or by testing whether the whole case expression is an instance of some call in $S$.

**Example 30** Let us consider the following set of terms:

$$S = \{\texttt{app a b}, \texttt{case (app a b) of } \{[] \to \texttt{z}; (\texttt{x} : \texttt{y}) \to (\texttt{app y z})\}\ \}$$

The following expression

$$\texttt{case (app a}'\ \texttt{b}') \texttt{ of } \{[] \to \texttt{z}'; (\texttt{x}' : \texttt{y}') \to (\texttt{app y}'\ \texttt{z}')\}$$

can be proved $S$-closed using the first element of the set (by checking that the subterms $\texttt{app a}'\ \texttt{b}'$ and $\texttt{app y}'\ \texttt{z}'$ are instances of $\texttt{app a b}$) or by testing that the whole expression is an instance of the second element of the set.

[10] Constructor patterns are not considered here since they are constructor terms and hence closed by definition.

The following theorem is essential to prove the correctness of partial evaluation, since it allows us to apply the correctness results of the NN-PE framework. Informally speaking, our result states the following diagram:

$$\begin{tikzcd}
\mathcal{R} \arrow[d, "\mathrm{PE}"'] & \mathcal{R}'' \arrow[l, "flat"'] \arrow[d, "\mathrm{PE}"] \\
\mathcal{R}' & \mathcal{R}''' \arrow[l, "flat"]
\end{tikzcd}$$

Essentially, we state that, given a flat program $\mathcal{R}$ and a partial evaluation $\mathcal{R}'$ of $\mathcal{R}$, if we consider an inductively sequential program $\mathcal{R}''$ which is equivalent to the flat version of $\mathcal{R}$, i.e., $\mathcal{R} = flat(\mathcal{R}'')$, then there exists an inductively sequential program $\mathcal{R}'''$ which is a NN-PE of $\mathcal{R}''$ such that $\mathcal{R}' = flat(\mathcal{R}''')$. In a precise sense, the proof could be based on the equivalence between the RLNT calculus and needed narrowing (Theorem 80 and Corollary 85) and the correctness of the NN-PE transformation (Theorem 23). However, for the sake of simplicity, we apply the results achieved by the PE framework based on needed narrowing derivations (to construct residual programs) over LNT derivations. This is justified by the fact that both calculi have been proven equivalent (Theorem 80) in the sense that there exists a LNT derivation in a flat program iff the same derivation can be proven using needed narrowing w.r.t. the corresponding inductively sequential program. This amounts to say that our proof relies on the relation between the LNT and RLNT calculi, and the correctness of NN-PE (by assuming implicitly the equivalence between LNT and needed narrowing).

**Theorem 88** *Let $\mathcal{R}$ be a flat program, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{R}'$ be a partial evaluation of $\mathcal{R}$ w.r.t. $S$ (under $\rho$). Let $\mathcal{R}''$ be an inductively sequential program such that $\mathcal{R} = flat(\mathcal{R}'')$. Then, there exists a program $\mathcal{R}'''$ which is a NN-PE of $\mathcal{R}''$ w.r.t. $S$ (under $\rho$), such that $\mathcal{R}' = flat(\mathcal{R}''')$.*

*Proof.* The proof relies on the construction of a single resultant of $\mathcal{R}'$. The extension to consider the complete set of resultants is straightforward.

Let us consider a resultant of $\mathcal{R}'$ of the form: $l = r$, which has been obtained from a RNLT derivation $[\![s]\!] \stackrel{+}{\Rightarrow} t$ (where all steps are finished) in $\mathcal{R}$, where $l = \rho(s)$ and $r = ren_\rho(t)$. By the equivalence between the RLNT and the LNT calculi (Theorem 84), we know that there exists a term $s'$ with $(\theta, s') \in \mathcal{CT}([\![s]\!])$ and the following LNT derivations:

$$\begin{array}{lll} [\![s']\!] & \overset{\sigma_1}{\overset{+}{\Rightarrow}}_{\mathsf{LNT}} & [\![t_1]\!] \\ & \vdots & \\ [\![s']\!] & \overset{\sigma_n}{\overset{+}{\Rightarrow}}_{\mathsf{LNT}} & [\![t_n]\!] \end{array}$$

where $(\sigma_i, [\![t_i]\!]) \in \mathcal{CT}(t)$ for all $i = 1, \ldots, n$. By definition of $\mathcal{CT}$, since $s$ is within square brackets, the only possibility is that $s = s'$ and $\theta = id$. Therefore, we can compute a NN-PE $\mathcal{R}^*$ of $\mathcal{R}''$ w.r.t. $s$ from the above derivations as follows:

$$\begin{array}{lll} \sigma_1(\rho(s)) & \rightarrow & ren_\rho(t_1) \\ & \vdots & \\ \sigma_n(\rho(s)) & \rightarrow & ren_\rho(t_n) \end{array}$$

The proof consists in demonstrating that $flat(\mathcal{R}^*) = \{l = r\}$. This amounts to show that there exists a definitional tree $\mathcal{T}$ for $\rho(s)$ such that $\mathcal{C}ase(\mathcal{T}) = ren_\rho(t) = r$. The proof is by induction on the number $k$ of Case Guess rules applied in the original RLNT derivation $[\![s]\!] \overset{+}{\Rightarrow} t$.

**Base case** $(k = 0)$**.** Since we assume that Case Guess rule is not applied, by definition of the RLNT calculus, the derivation for $[\![s]\!]$ has the form $[\![s]\!] \overset{+}{\Rightarrow} [\![t']\!] = t$. Due to the fact that there has been no branching and that $(\sigma_i, [\![t_i]\!]) \in \mathcal{CT}(t)$ for all $i = 1, \ldots, n$, the only possibility is that $n = 1$, $\sigma_1 = id$, and $t_1 = t'$. Therefore, $\mathcal{R}^*$ is formed by a single resultant of the form $\rho(s) \rightarrow ren_\rho(t)$ whose unique definitional tree is $rule(\rho(s) \rightarrow ren_\rho(t))$. Trivially, it holds that

$$\mathcal{C}ase(rule(\rho(s) \rightarrow ren_\rho(t))) = ren_\rho(t) = r$$

**Inductive case** $(k > 0)$**.** Let us consider that the RLNT derivation has the form $[\![s]\!] \overset{+}{\Rightarrow} t' \Rightarrow t'' \overset{*}{\Rightarrow} t$ where the subderivation $[\![s]\!] \overset{+}{\Rightarrow} t'$ fires $k-1$ times Case Guess rule, then, Case Guess rule is applied to $t'$ to derive $t''$ and, finally, the remaining subderivation, $t'' \overset{*}{\Rightarrow} t$, does not apply Case Guess rule anymore.

If we consider the subderivation $[\![s]\!] \overset{+}{\Rightarrow} t'$, we know that there exist the following LNT derivations:

$$\begin{array}{lll} [\![s]\!] & \overset{\theta_1}{\overset{+}{\Rightarrow}}_{\mathsf{LNT}} & [\![t'_1]\!] \\ & \vdots & \\ [\![s]\!] & \overset{\theta_m}{\overset{+}{\Rightarrow}}_{\mathsf{LNT}} & [\![t'_m]\!] \end{array}$$

where $(\theta_i, [\![t'_i]\!]) \in \mathcal{CT}(t')$ for all $i = 1, \ldots, m$. Therefore, we can compute a

NN-PE from the above derivations as follows:

$$\begin{array}{rcl}\theta_1(\rho(s)) & \to & ren_\rho(t'_1) \\ & \vdots & \\ \theta_m(\rho(s)) & \to & ren_\rho(t'_m)\end{array}$$

By applying the inductive hypothesis to the subderivation $[\![s]\!] \stackrel{+}{\Rightarrow} t'$, we know that there exists a definitional tree $\mathcal{T}'$ for $\rho(s)$ such that $\mathcal{C}ase(\mathcal{T}') = ren_\rho(t')$.

Now, it is sufficient to show that, by extending $\mathcal{T}'$ as dictated by the remaining steps of the original derivation $[\![s]\!] \stackrel{+}{\Rightarrow} t$, we obtain a definitional tree $\mathcal{T}$ which fulfills our claim.

First, we have to apply $\mathsf{Case\ Guess}$ rule to $t'$. Therefore, $t'$ is required to be of the form: $t' = C[\ldots [\![fcase\ x\ of\{\overline{p_i \to s_i}\}]\!] \ldots]$ where $C$ denotes an arbitrary context of $fcase$ constructs with variable arguments and $t'|_o = [\![fcase\ x\ of\{\overline{p_i \to s_i}\}]\!]$ for some position $o$. Now, after a RLNT step, we obtain a term $t''$ of the form:

$$t'' = C[\ldots fcase\ x\ of\{\overline{p_i \to [\![\varphi_i(s_i)]\!]}\} \ldots]$$

where $\varphi_j = \{x \mapsto p_j\},\ j = 1, \ldots, i$, i.e., $t'' = t'[fcase\ x\ of\{\overline{p_i \to [\![\varphi_i(s_i)]\!]}\}]_o$.

Note that since the last subderivation does not apply $\mathsf{Case\ Guess}$ steps, term $t$ must satisfy that: $t|_o = fcase\ x\ of\{\overline{p_i \to [\![s'_i]\!]}\}$, i.e., the position of the square brackets cannot be changed. Clearly, the subterm $[\![fcase\ x\ of\{\overline{p_i \to s_i}\}]\!]$ of $t'$ corresponds to some term $t'_k$, $k \in \{1, \ldots, m\}$, of the last LNT derivations. For simplicity, we assume that it corresponds to the last derived term $[\![t'_m]\!]$. Therefore, we can replace the last LNT derivation by a new sequence of LNT derivations where the $\mathsf{Case\ Guess}$ rule has been applied to $[\![t'_m]\!]$. After that, the new situation is:

$$\begin{array}{l}[\![s]\!] \stackrel{+}{\Rightarrow}{}^{\theta_1}_{\mathsf{LNT}} [\![t'_1]\!] \\ \vdots \\ [\![s]\!] \stackrel{+}{\Rightarrow}{}^{\theta_{m-1}}_{\mathsf{LNT}} [\![t'_{m-1}]\!] \\ [\![s]\!] \stackrel{+}{\Rightarrow}{}^{\varphi_1 \circ \theta_m}_{\mathsf{LNT}} [\![s'_1]\!] \\ \vdots \\ [\![s]\!] \stackrel{+}{\Rightarrow}{}^{\varphi_i \circ \theta_m}_{\mathsf{LNT}} [\![s'_i]\!]\end{array}$$

Let us notice that this sequence of LNT derivations can only correspond to the $n$ initial LNT derivations, i.e., $[\![t'_1]\!] = [\![t_1]\!], \ldots, [\![s'_i]\!] = [\![t_n]\!]$.

By definition of NN-PE, the resultants associated to the above derivations have the form:

$$\begin{array}{rcl}
\theta_1(\rho(s)) & \to & ren_\rho(t'_1) \\
\vdots & & \\
\theta_{m-1}(\rho(s)) & \to & ren_\rho(t'_{m-1}) \\
\varphi_1 \circ \theta_m(\rho(s)) & \to & ren_\rho(s'_1) \\
\vdots & & \\
\varphi_i \circ \theta_m(\rho(s)) & \to & ren_\rho(s'_i)
\end{array}$$

By definition of $\mathcal{T}'$, the definitional tree contains a leaf of the form:

$$rule(\theta_m(\rho(s)) \to ren_\rho(t'_m))$$

Now, we can construct a definitional tree $\mathcal{T}$ from $\mathcal{T}'$ by replacing the above $rule$ node by a $branch$ node; namely, we replace $rule(\theta_m(\rho(s)) \to ren_\rho(t'_m))$ by

$$\begin{array}{ll}
branch(\theta_m(\rho(s)), r, & rule(\varphi_1 \circ \theta_m(\rho(s)) = ren_\rho(s'_1)) \\
& \ldots \\
& rule(\varphi_i \circ \theta_m(\rho(s)) = ren_\rho(s'_i)))
\end{array}$$

where $r$ is a position such that $\theta_m(\rho(s))|_r = x$. Trivially, $\mathcal{T}$ is a definitional tree for the above set of rules.

Finally, we proceed to compose the results to deduce that $\mathcal{C}ase(\mathcal{T}) = r$. Let us consider the sequence of calls produced during the execution of $\mathcal{C}ase(\mathcal{T}')$:

$$\mathcal{C}ase(\mathcal{T}') = \ldots = C_1 = C_2 = ren_\rho(t')$$

As discussed above, there exists a position $o$ in $C_1$ which corresponds to the depth of the leaf $rule(\theta_m(\rho(s)) \to ren_\rho(t'_m))$ in the definitional tree $\mathcal{T}'$ such that:

$$\begin{array}{l}
C_1|_o = \mathcal{C}ase(rule(\theta_m(\rho(s)) \to ren_\rho(t'_m))) \\
C_2|_o = ren_\rho(t'_m)
\end{array}$$

Therefore, $ren_\rho(t')|_o = ren_\rho(t'_m)$. Similarly, we consider the sequence of calls for the computation of $\mathcal{C}ase(\mathcal{T})$:

$$\mathcal{C}ase(\mathcal{T}) = \ldots = P_1 = P_2 = P_3$$

Since we constructed $\mathcal{T}$ from $\mathcal{T}'$ by replacing the above *rule* node with a *branch* node, we know that:

$$\begin{array}{l} P_1|_o = Case(branch(\theta_m(\rho(s)), o, rule(\varphi_1 \circ \theta_m(\rho(s)) = ren_\rho(s'_1)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad rule(\varphi_i \circ \theta_m(\rho(s)) = ren_\rho(s'_i)) \\ P_2|_o = fcase\ x\ of\ \{\overline{p_i \to Case(rule(\varphi_i \circ \theta_m(\rho(s)) = ren_\rho(s'_i))}\} \\ P_3|_o = fcase\ x\ of\ \{\overline{p_i \to Case(ren_\rho(s'_i))}\} \end{array}$$

Finally, since $t = t'[fcase\ x\ of\{\overline{p_i \to [\![s'_i]\!]}\}]_o$ and $P_3 = ren_\rho(t')[P_3|_o]_o$, we have that $P_3 = ren_\rho(t) = r$, which concludes the proof.

□

Since we have demonstrated that, for a given partial evaluation $\mathcal{R}'$ obtained according to Definition 86, there exists an inductively sequential program (equivalent to $\mathcal{R}'$ modulo function *flat*) which is a NN-PE, we can use the correctness results of the NN-PE framework (Theorem 23). However, there are still two points which require some further attention:

- In Theorem 23, expressions cannot be evaluated beyond their head normal forms at partial evaluation time. In fact, it is not safe due to the backpropagation of bindings and the form in which residual rules are constructed (see Chapter 3). From a theoretical viewpoint, the problem can be seen as follows. Given a LNT derivation for $s_0$ in $\mathcal{R}$:

  $$s_0 \Rightarrow^{\sigma_1}_{\mathsf{LNT}} s_1 \Rightarrow^{\sigma_2}_{\mathsf{LNT}} \ldots \Rightarrow^{\sigma_n}_{\mathsf{LNT}} s_n \qquad (*)$$

  If we produce the associated residual rule (before renaming) as follows:

  $$\sigma_n \circ \ldots \circ \sigma_1(s_0) \to s_n$$

  Then, it could happen that some instance $\theta(s_0)$ can be reduced to head normal form in $\mathcal{R}$ but the above residual rule cannot be applied due to the incompatibility of $\theta$ and $\sigma_n \circ \ldots \circ \sigma_1$ (which destroys the completeness of the transformation if $\theta(s_0)$ is a subterm of some expression $e[\theta(s_0)]_p$ whose computation only requires the evaluation of $\theta(s_0)$ to head normal form). If the derivation $(*)$ has not surpassed the head normal form of $s_0$, then this situation cannot occur, as demonstrated in Theorem 23. In our new scheme, this restriction is no longer necessary, since the RLNT calculus computes no substitution, but encodes it as case expressions. Therefore, the previous incompatibility between $\theta$ and $\sigma_n \circ \ldots \circ \sigma_1$ cannot occur (i.e., $\sigma_n \circ \ldots \circ \sigma_1 = id$) and, hence, it is perfectly safe to surpass head normal forms at specialization time. In fact, whenever the residual rule is applied to some term $\theta(s_0)$, the evaluation of the corresponding right-hand side proceeds by applying the Case Guess rule as much as needed, so that only the "compatible" part of $\sigma_n \circ \ldots \circ \sigma_1$ is computed.

- On the other hand, in order to establish the *total* correctness of the partial evaluation transformation, the preservation of floundering freeness is needed. Informally, one needs to prove that $\mathcal{R} \cup \{e\}$ does not flounder iff $\mathcal{R}' \cup \{e'\}$ does not flounder either, where $e' = ren_\rho(e)$ and $\mathcal{R}'$ is a partial evaluation of $\mathcal{R}$ (see, e.g., Theorem 42). This result is implicitly given by our framework, since the floundering behaviour of derivations is determined by the evaluation annotations of program functions, and our syntax explicitly incorporates evaluation annotations by means of $case/fcase$ constructs. Thus, we explicitly preserve the floundering behaviour throughout the partial evaluation process.

**Theorem 89 (correctness)**
*Let $\mathcal{R}$ be a flat program, $e$ a term, $c$ a 0-ary constructor, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming of $S$. Let $\mathcal{R}'$ be a partial evaluation of $\mathcal{R}$ w.r.t. $S$ (under $\rho$) such that $\mathcal{R}' \cup \{e'\}$ is $S'$-closed, where $e' = ren_\rho(e)$ and $S' = \rho(S)$. Then, $e \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{LNT}} c$ is a LNT derivation for $e$ in $\mathcal{R}$ iff there exists a LNT derivation $e' \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{LNT}} c$ in $\mathcal{R}'$.*

*Proof.* Let $e \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{LNT}} c$ be a LNT derivation for $e$ in $\mathcal{R}$. Let us consider an inductively sequential program $\mathcal{R}''$ such that $\mathcal{R} = flat(\mathcal{R}'')$. By Theorem 80, there exists a needed narrowing derivation $e \leadsto^*_\sigma c$ in $\mathcal{R}''$ for $e$. Then, by Theorem 88, we know that there exists an inductively sequential program $\mathcal{R}'''$ which is a NN-PE of $\mathcal{R}''$ w.r.t. $S$ such that $\mathcal{R}' = flat(\mathcal{R}''')$. By the correctness of the NN-PE framework (Theorem 23), there exists a needed narrowing derivation $e' \leadsto^*_\sigma c$ in $\mathcal{R}'''$, where $e' = ren_\rho(e)$. Now, by the equivalence between needed narrowing and the LNT calculus (Theorem 80), there exists a LNT derivation $e' \overset{*}{\Rightarrow}{}^{\sigma}_{\mathsf{LNT}} c$ in $\mathcal{R}'$, which concludes the proof. □

## 7.3 Control Issues

In this section, we present a program specialization algorithm which uses the abstract representation and the calculus introduced so far.

Following the ideas introduced in Chapter 5, a simple *on-line* partial evaluation algorithm can be defined as follows. Given a program $\mathcal{R}$ and an input term $t$, we compute a finite (possibly incomplete) RLNT derivation $t \overset{+}{\Rightarrow} s$ for $t$ in $\mathcal{R}$.[11] Then,

[11] Note that, since the RLNT calculus is deterministic, there is no branching in the search space. Thus, only a single derivation can be computed from a given term.

this process is iteratively repeated for any subterm which occurs in the expression $s$ and which is not closed w.r.t. the set of terms already evaluated.

As framed in Chapter 5, a partial evaluation algorithm involves two control issues: the so-called *local* control, which concerns the computation of partial evaluations for single terms, and the *global* control, which ensures the termination of the iterative process but still guaranteeing that the closedness condition is eventually reached.

Concerning the global control, a special treatment of case expressions is required, which was not considered in previous abstraction operators. Of course, a straightforward solution would be to consider the case symbol as an ordinary constructor symbol. Unfortunately, this would often cause a drastical loss of specialization, as witnessed by the following example.

**Example 31** Consider again the program `app`, the initial expression `app (app x y) z`, and the RLNT derivation of Example 28:

$$
\begin{array}{l}
[\![\texttt{app (app x y) z}]\!] \\
\stackrel{*}{\Rightarrow} [\![\texttt{case (case x of \{}\ \ \texttt{[]} \to \texttt{y;} \\
\qquad\qquad\qquad\qquad\quad (\texttt{a}' : \texttt{b}') \to (\texttt{a}' : \texttt{app b}'\ \texttt{y})\})\\
\qquad\qquad\qquad\qquad\qquad \texttt{of \{[]} \to \texttt{z;} \\
\qquad\qquad\qquad\qquad\qquad\quad (\texttt{a : b}) \to (\texttt{a : app b z})\}]\!] \\
\stackrel{*}{\Rightarrow} \texttt{case x of \{}\ \ \texttt{[]} \qquad\quad \to \texttt{case y of \{[]} \to \texttt{z; (a : b)} \to (\texttt{a} : [\![\texttt{app b z}]\!])\}; \\
\qquad\qquad\qquad (\texttt{a}' : \texttt{b}') \quad \to (\texttt{a}' : [\![\texttt{app (app b}'\ \texttt{y) z}]\!])\}
\end{array}
$$

If we consider an unfolding rule which stops the derivation at the intermediate case expression, then the abstraction operator of Definition 65 will attempt to add only the operation-rooted subterms "app b′ y" and "app b y" to the set of calls to be subsequently specialized. This will prevent us from obtaining an efficient (recursive) residual function for the original call, since we never reach again an expression containing "app (app x y) z" (see Example 32).

On the other hand, by treating case expressions as operation-rooted terms, the problem is not solved either. For instance, if we consider an unfolding rule which stops at the last term in the above derivation, then adding the whole term to the current set is not convenient. In this case, the best choice would be to treat the *case* symbol as a constructor symbol. Moreover, a similar situation occurs when considering constructor-rooted terms, since the RLNT calculus does not have restrictions to evaluate terms in head normal form.

Luckily, the RLNT calculus gives us some leeway. The key idea is to take into special consideration the position of the square brackets of the calculus: an expression within square brackets should be added to the set of partially evaluated calls (if possible), whereas expressions which are not within square brackets should be definitively residualized (i.e., ignored by the abstraction operator, except for operation-rooted terms).

**Definition 90** *Given a sequence of terms $q$ and a finite set of terms $T$, we define:*

$$abstract^{\#}(q,T) \;=\; \begin{cases} q & \textit{if } T = \emptyset \\ abs(\ldots abs(q,t_1),\ldots,t_n) & \textit{if } T = \{t_1,\ldots,t_n\}, n \geq 1 \end{cases}$$

*The function $abs(q,t)$ distinguishes the following cases:*

$$abs(q,t) \;=\; \begin{cases} q & \textit{if } t \in \mathcal{X} \\ abstract^{\#}(q,\{t_1,\ldots,t_n\}) & \textit{if } t = c(t_1,\ldots,t_n),\ c \in \mathcal{C} \\ abstract^{\#}(q,\{e,t_1,\ldots,t_n\}) & \textit{if } t = (f)case\ e\ of\ \{\overline{p_n \to t_n}\} \\ try\_add(q,t) & \textit{if } t = f(t_1,\ldots,t_n),\ f \in \mathcal{F} \\ try\_add(q,e) & \textit{if } t = [\![e]\!] \end{cases}$$

*Finally, the function $try\_add((q_1,\ldots,q_n),t)$ is defined as follows:*
$try\_add((q_1,\ldots,q_n),t) \;=$

$$\begin{cases} abstract^{\#}((q_1,\ldots,q_{i-1},q_{i+1},\ldots,q_n),\{s'\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)) \\ \qquad\qquad \textit{if } \exists i \in \{1,\ldots,n\}.\ root(q_i) = root(t) \textit{ and } q_i \trianglelefteq t, \\ \qquad\qquad \textit{where } \langle s', \{\theta_1,\theta_2\}\rangle = msg(\{q_i,t\}) \\ ((q_1,\ldots,q_n),t) \qquad\qquad \textit{otherwise} \end{cases}$$

Let us informally explain this definition. Given a set of terms $q$, in order to add a new term $t$, the abstraction operator distinguishes the following cases:

- variables are disregarded;
- if $t$ is rooted by a constructor symbol or by a case symbol, then it recursively inspects the arguments;
- if $t$ is rooted by a defined function symbol or it is enclosed within square brackets, then the abstraction operator tries to add it to $q$ with *try_add* (even if it is constructor-rooted or a case expression). Now, we proceed as follows:
  - if $t$ does not embed any *comparable* (i.e., with the same root symbol) term in $q$, then $t$ is simply added to $q$;
  - if $t$ embeds some comparable term of $q$, say $q_i$, then the $msg$ of $q_i$ and $t$ is computed, say $\langle s', \{\theta_1,\theta_2\}\rangle$, and it finally attempts to add $s'$ as well as the terms in $\theta_1$ and $\theta_2$ to the set resulting from removing $q_i$ from $q$.

The left-hand sides of the residual flat program can be renamed by using the independent renaming transformation of Definition 14, by simply disregarding square brackets from function calls and treating case expressions as ordinary functions. The right-hand sides are renamed by means of the auxiliary function $ren_\rho$, which gives a special treatment to case expressions and also considers square brackets.

**Definition 91** *Let $S$ be a set of terms, $e$ a term, and $\rho$ an independent renaming of $S$. The function $ren_\rho(e)$ is defined inductively as follows:* $ren_\rho(e) ::=$

$$\begin{cases} e & \textit{if } e \in \mathcal{X} \\ c(ren_\rho(t_1), \ldots, ren_\rho(t_n)) & \textit{if } e = c(t_1, \ldots, t_n),\ c \in \mathcal{C} \\ (f)case\ c(\overline{t_n})\ of\ \{\overline{p_k \to ren_\rho(t_k)}\} & \textit{if } e = (f)case\ c(\overline{t_n})\ of\ \{\overline{p_k \to t_k}\} \\ \theta'(\rho(s)) & \textit{if } (e = f(\ldots),\ f \in \mathcal{F} \textit{ and } e = \theta(s)), \\ & \quad or\ (e = [\![e']\!] \textit{ and } e' = \theta(s)),\ where \\ & \qquad \theta' = \{(x \mapsto ren(\theta(x))) \mid x \in Dom(\theta)\} \end{cases}$$

In the above definition, case expressions can be (non-deterministically) renamed either by independently renaming the case subterms or by replacing the considered case expression by a call to the corresponding new, renamed function (when the expression is an instance of some specialized call in $S$).

Let us consider an example to illustrate the whole partial evaluation process.

**Example 32** Consider the program $\mathcal{R}_{\texttt{app}}$ which contains the rule defining the function `app`. In order to compute $\mathcal{PE}(\mathcal{R}_{\texttt{app}}, \{\texttt{app (app x y) z}\})$, we start with:

$$\mathtt{q_0} = abstract^{\#}(nil, \{\texttt{app (app x y) z}\}) = (\texttt{app (app x y) z})$$

For the first iteration, we assume that: $\mathcal{U}(\texttt{app (app x y) z}) =$

$$\begin{array}{lll} \texttt{case x of } \{ & [] & \to \texttt{case y of } \{[] \to \texttt{z};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : [\![\texttt{app b z}]\!])\}; \\ & (\texttt{a}' : \texttt{b}') & \to (\texttt{a}' : [\![\texttt{app (app b}' \texttt{ y) z}]\!])\} \end{array}$$

(see derivation in Example 28). Then, we compute:

$$\mathtt{q_1} = abstract^{\#}(\mathtt{q_0}, \{\mathcal{U}(\texttt{app (app x y) z})\}) = (\texttt{app (app x y) z}, \texttt{app b z})$$

For the next iteration, we assume that:

$$\mathcal{U}(\texttt{app b z}) = \texttt{case x of } \{[] \to \texttt{y};\ (\texttt{a} : \texttt{b}) \to \texttt{a} : [\![\texttt{app b y}]\!]\ \}$$

Therefore, $abstract^{\#}(\mathtt{q_1}, \{\mathcal{U}(\texttt{app b z})\}) = \mathtt{q_1}$ and the process finishes. The associated residual rules are (after renaming the original expression as `dapp x y z`):

$$\begin{array}{lll} \texttt{dapp x y z} & = & \texttt{case x of } \{\ [] \to \texttt{case y of } \{[] \to \texttt{z}; \\ & & \qquad\qquad\qquad\qquad (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b z})\}; \\ & & \qquad (\texttt{a}' : \texttt{b}') \to (\texttt{a}' : \texttt{dapp b}' \texttt{ y z})\} \\ \texttt{app x y} & = & \texttt{case x of } \{\ [] \to \texttt{y};\ (\texttt{a} : \texttt{b}) \to (\texttt{a} : \texttt{app b y})\} \end{array}$$

As expected, the optimized function `dapp` is able to concatenate three lists by traversing the first list only once, which is not possible in the original program.

Now, we proceed to demonstrate the termination of the partial evaluation algorithm presented so far.

The following proposition states that the operator $abstract^{\#}$ of Definition 90 is a correct abstraction operator according to Definition 46. The first property of Definition 46 amounts to say that the abstraction operator does not "create" new function symbols (thus, it is trivial to prove it). Now, we proceed to demonstrate the second property which ensures the correct propagation of closedness information.

**Proposition 92** *Given a sequence of terms $q$ and a finite set of terms $T$. Then, for all $t \in (S_q \cup T)$, $t$ is closed with respect to the elements in $q' = abstract^{\#}(q, T)$.*

*Proof.* We proceed by well-founded induction on $S_q \cup T$. If $S_q \cup T = \{\,\}$, then $abstract^{\#}(nil, \{\,\}) = nil$, and the proof is done. Let us consider the inductive case $S_q \cup T \neq \{\,\}$, and assume that $T$ is not empty (otherwise the proof is trivial). Then, $T = T_0 \cup \{t\}$, with $T_0 = \{t_1, \ldots, t_{n-1}\}$. By the inductive hypothesis, the property holds for all $q'$ and for all $T'$ such that $\mathcal{M}_{S_{q'} \cup T'} <_{mul} \mathcal{M}_{S_q \cup T}$. Following the definition of $abstract^{\#}$, we now consider four cases:

1. If $t$ is a variable, then $q' = abstract^{\#}(q, T_0 \cup \{t\}) = abstract^{\#}(q, T_0)$. Since $\mathcal{M}_{S_q \cup T_0} <_{mul} \mathcal{M}_{S_q \cup T_0 \cup \{t\}} = \mathcal{M}_{S_q \cup T}$ then, by the inductive hypothesis, $S_q \cup T_0$ is closed with respect to the terms in $q'$ and, by the definition of closedness, so is $S_q \cup T_0 \cup \{t\} = S_q \cup T$.

2. If $t = c(s_1, \ldots, s_m)$, $c \in \mathcal{C}$, then $q' = abstract^{\#}(q, T_0 \cup \{s_1, \ldots, s_m\})$. Since $\mathcal{M}_{S_q \cup T_0 \cup \{s_1, \ldots, s_m\}} <_{mul} \mathcal{M}_{S_q \cup T_0 \cup \{c(s_1, \ldots, s_m)\}} = \mathcal{M}_{S_q \cup T}$ then, by the inductive hypothesis, $S_q \cup T_0 \cup \{s_1, \ldots, s_m\}$ is closed with respect to the terms in $q'$ and, by the definition of closedness, so is $S_q \cup T_0 \cup \{c(s_1, \ldots, s_m)\} = S_q \cup T$.

3. If $t = (f)case\ e\ of\ \{\overline{p_k \rightarrow s_k}\}$, then $q' = abstract^{\#}(q, T_0 \cup \{e, s_1, \ldots, s_k\})$ and the proof is perfectly analogous to the previous case.

4. If $t = f(s_1, \ldots, s_m)$, $f \in \mathcal{F}$, $m \geq 0$, or $t$ is within square brackets, then

   $$q' = try\_add(abstract^{\#}(q, T_0), t)$$

   Assume that $abstract^{\#}(q, T_0)$ returns $q'' = (q_1, \ldots, q_p)$, $p \geq 1$ (since the case when $abstract^{\#}(q, T_0)$ is $nil$ is straightforward). Here we distinguish two cases, which correspond to the two case values of the function $try\_add$ in the definition of $abstract^{\#}$, given by $try\_add(q'', t) = try\_add((q_1, \ldots, q_n), t) =$

   (a) $abstract^{\#}((q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n), S^*) = abstract^{\#}(q''', S^*)$, if there exists $i \in \{1, \ldots, n\}$ such that $root(t) = root(q_i)$ and $q_i \trianglelefteq t$, where

   $$msg(\{q_i, t\}) = \langle w, \{\theta_1, \theta_2\}\rangle$$

and $S^* = \{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$.
First, we have $\mathcal{M}_{S_q \cup T_0} <_{mul} \mathcal{M}_{S_q \cup T_0 \cup \{t\}}$ and then, by the inductive hypothesis, $S_q \cup T_0$ is $S_{q''}$-closed and therefore $S_q \cup T_0$ is $S_{q'''} \cup S^*$-closed. Then, we prove that $\mathcal{M}_{S_{q'''} \cup S^*} <_{mul} \mathcal{M}_{S_{q'''} \cup \{q_i\} \cup \{t\}}$ by using the definition of multiset ordering. Here, we distinguish two cases:

i. $depth(w) = depth(q_i)$. Let $M$, $M'$ be the multiset complexities of $S_{q'''} \cup S^*$ and $S_{q'''} \cup \{q_i\} \cup \{t\}$, respectively. Let $X$, $X'$ be the multiset complexities of $\mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$ and $\{t\}$, respectively. Let $X_1, X_2$ be the multiset complexities of $\mathcal{R}an(\theta_1)$ and $\mathcal{R}an(\theta_2)$, respectively. Since $t$ embeds $q_i$, then it is immediate to see that $depth(q_i) \leq depth(t)$. Since $q_i$ and $w$ are comparable and $q_i = \theta_2(w)$, then $depth(d) < depth(q_i)$ for all $d \in \mathcal{R}an(\theta_2)$. Therefore, $depth(d) < depth(t)$ for all $d \in \mathcal{R}an(\theta_2)$, and hence $\forall n \in X_1,\ \exists n' \in X'.\ n < n'$.
Since $t$ and $w$ are comparable terms and $t = \theta_2(w)$, then it holds that $\forall n \in X_2,\ \exists n' \in X'.\ n < n'$, which proves the claim.

ii. $depth(w) < depth(q_i)$. It can be proved in a similar way by considering that $X$ and $X'$ denote the multiset complexities of $\{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$ and $\{q_i\} \cup \{t\}$, respectively.

Since we have proved $\mathcal{M}_{S_{q'''} \cup S^*} <_{mul} \mathcal{M}_{S_{q'''} \cup \{q_i\} \cup \{t\}}$, then by the inductive hypothesis, $S_{q'''} \cup S^*$ is closed w.r.t. the terms in $q'$ and hence $S_q \cup T_0$ is also closed w.r.t. $q'$ by Lemma 69. Finally, as $t$ is closed w.r.t. $\{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$ by definition of $msg$, the claim follows by using Lemma 69 again. Then, the result follows trivially from the fact that $t$ is closed with respect to the terms in $q' = (abstract^{\#}(q, T_0), t)$.

(b) $(abstract^{\#}(q, T_0), t)$. Since $\mathcal{M}_{S_q \cup T_0} <_{mul} \mathcal{M}_{S_q \cup T}$ then, by the inductive hypothesis, $S_q \cup T_0$ is closed w.r.t. the terms in $S_{q''}$, and therefore it is closed w.r.t. the elements in $(abstract^{\#}(q, T_0), t)$.

□

Finally, we establish the termination of the partial evaluation process.

**Lemma 93**
*The computation of the operator $abstract^{\#}$ of Definition 90 terminates.*

*Proof.* It can be easily proved by well-founded induction on the union of the input arguments of $abstract^{\#}$. □

**Lemma 94**

*Let $S$ be a sequence of terms satisfying the nonembedding property, and $T$ an arbitrary set of terms. Then, $q' = abstract^{\#}(q, T)$ satisfies the nonembedding property (Definition 72).*

*Proof.* We prove that the computation of $abstract^{\#}(q, T)$ satisfies the nonembedding property by well-founded induction on $S_q \cup T$.

If $S_q \cup T = \{\ \}$, then $abstract^{\#}(nil, \{\ \}) = nil$, and the proof is done. Let us consider the inductive case $S_q \cup T \neq \{\ \}$, and assume that $T$ is not empty (otherwise the proof is trivial). Then, $T = T_0 \cup \{t\}$, with $T_0 = \{t_1, \ldots, t_{n-1}\}$. By the inductive hypothesis, the property holds for all $q'$ and for all $T'$ such that $\mathcal{M}_{S_{q'} \cup T'} <_{mul} \mathcal{M}_{S_q \cup T}$. Following the definition of $abstract^{\#}$, we now consider four cases:

1. If $t \in \mathcal{X}$, then $q' = abstract^{\#}(q, T) = abstract^{\#}(q, T_0)$. Since $\mathcal{M}_{S_q \cup T_0} <_{mul} \mathcal{M}_{S_q \cup T_0 \cup \{t\}} = \mathcal{M}_{S_q \cup T}$ then, by the inductive hypothesis, $q'$ satisfies the nonembedding property.

2. If $t = c(s_1, \ldots, s_m)$, $c \in \mathcal{C}$, $m \geq 0$, then by definition of $abstract^{\#}$, $q' = abstract^{\#}(q, T_0 \cup \{s_1, \ldots, s_m\})$.
   Since $\mathcal{M}_{S_q \cup T_0 \cup \{s_1, \ldots, s_m\}} <_{mul} \mathcal{M}_{S_q \cup T_0 \cup \{c(s_1, \ldots, s_m)\}} = \mathcal{M}_{S_q \cup T}$ then, by the inductive hypothesis, the nonembedding property holds for $q'$.

3. If $t = (f)case\ e\ of\ \{\overline{p_k \to s_k}\}$, then $q' = abstract^{\#}(q, T_0 \cup \{e, s_1, \ldots, s_k\})$ and the proof is perfectly analogous to the previous case.

4. If $t = f(s_1, \ldots, s_m)$, $f \in \mathcal{F}$, $m \geq 0$ or if $t$ is within square brackets, then by definition of $abstract^{\#}$:

   $$q' = try\_add(abstract^{\#}(q, T_0), t)$$

   Assume that $abstract^{\#}(q, T_0)$ returns $q'' = (q_1, \ldots, q_n)$, $n \geq 1$ (since the case when $abstract^{\#}(q, T_0)$ is $nil$ is straightforward). Here we distinguish two cases, which correspond to the two case values of the function $try\_add$ in the definition of $abstract^{\#}$, given by $try\_add(q'', t) = try\_add((q_1, \ldots, q_n), t) =$

   (a) $abstract^{\#}((q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n), T')$, if there exists $i \in \{1, \ldots, p\}$ such that $root(t) = root(q_i)$ and $q_i \trianglelefteq t$, where $msg(\{q_i, t\}) = \langle w, \{\theta_1, \theta_2\}\rangle$, $T' = \{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$. Since $\mathcal{M}_{S_{q''} \cup T'} <_{mul} \mathcal{M}_{S_{q''} \cup \{q_i\} \cup \{t\}}$, by the inductive hypothesis, the set of comparable terms in $q'$ satisfies the nonembedding property.

   (b) $(S_{q''}, t)$, otherwise. Since $\mathcal{M}_{S_q \cup T_0} <_{mul} \mathcal{M}_{S_q \cup T}$ then, by the inductive hypothesis, the computation of $abstract^{\#}(q, T_0)$ satisfies the nonembedding property and, by the premise of this case, the considered term $t$ does not infringe the nonembedding property for any of the comparable terms in $abstract^{\#}(q, T_0)$.

$\square$

The above property can be extended to a sequence of $\longmapsto$ steps (by an easy induction on the number of steps in the derivation).

**Lemma 95** *Let $T_0, \ldots, T_n$ be the sequences computed by:*

$$T_i \longmapsto_{\mathcal{P}} abstract^{\#}(T_i, S')$$

*where $S' = \{s' \mid s \in T_i \ \wedge\ \mathcal{U}(s) = s'\}$ for some flat program $\mathcal{R}$. Then, for all $i = 0, \ldots, n$, $T_i$ satisfies the nonembedding property.*

From the previous results, we can conclude that the partial evaluation process terminates by using the abstraction operator of Definition 90 and a finite unfolding rule.

**Theorem 96** *Let $\mathcal{R}$ be a flat program and $S$ a finite set of terms. The computation of the partial evaluation algorithm $\mathcal{PE}(\mathcal{R}, \mathcal{S})$, as defined in Definition 48, terminates using a finite unfolding rule and the abstraction operator of Definition 90.*

*Proof.* The following facts suffice to prove the proposition: i) the number of sets of incomparable terms which can be formed using a finite number of defined function symbols is finite; ii) by Kruskal's Tree Theorem and the nonembedding property stated above (Lemma 95), the subsets of comparable terms of any set computed from $\mathcal{PE}(\mathcal{R}, \mathcal{S})$ are finite; iii) the function $abstract^{\#}$ is well defined, in the sense that the computation of the new set always terminates (Lemma 93). $\square$

## 7.4 Conclusions

In this chapter, we introduced a novel approach for the partial evaluation of truly lazy functional logic languages. The new scheme is carefully designed for an abstract representation in which high-level programs can be automatically translated. We have shown how the appropriate level of abstraction of the intermediate language allows us to design a simple and concise partial evaluation framework. Moreover, by virtue of a non-standard (residualizing) semantics, we can avoid several limitations of previous frameworks for the partial evaluation of integrated languages. The next chapter illustrates how our new framework can be applied to define a practical partial evaluator which covers all the features of modern declarative languages.

# Chapter 8

# A Practical Partial Evaluator for Curry

This chapter summarizes our findings in the development of partial evaluation tools for the language Curry. From a practical point of view, the most promising approach appears to be the partial evaluation scheme introduced in Chapter 7. We support this statement by showing how the underlying method can be extended in order to develop a practical partial evaluation tool for the language Curry. The process is fully automatic and can be incorporated into a Curry compiler as a source-to-source transformation on intermediate programs. An implementation of the partial evaluator has been undertaken. Experimental results confirm that our partial evaluator pays off in practice. Part of the developments presented in this chapter are published in [Albert *et al.*, 2001b].

## 8.1 Introduction

Curry [Hanus, 1997b, 2000a] is a modern multi-paradigm declarative language which integrates features from functional, logic and concurrent programming. The most important features of the language include lazy evaluation, higher-order functions, nondeterministic computations, concurrent evaluation of constraints with synchronization on logical variables, and a unified computation model which integrates narrowing and residuation. Furthermore, Curry is a complete programming language which has been used to implement distributed applications (e.g., Internet servers [Hanus, 1999], dynamic web pages [Hanus, 2001]) and graphical user interfaces [Hanus, 2000b]. Several (efficient) implementations of the language already exist (see, e.g., [Alpuente *et al.*, 1999a; Antoy and Hanus, 2000; Hanus *et al.*, 2000; Lux and Kuchen, 1999]), although there is still room for further optimizations. Existing compilers for pure functional languages have been successfully improved by semantics-based program transformation techniques. For instance, the Glasgow Haskell Compiler includes a number of source-to-source program transformations which are able to optimize the quality of code in many different aspects [Peyton-Jones, 1996]. Encouraged by these successful experiences, we develop an automatic program transformation technique to improve the efficiency of Curry functional logic programs.

In the context of functional logic languages, it is very common to define functions which are correct from a declarative point of view, but which are also computationally expensive. This is the case, for instance, of functions defined by means of higher-order constructs such as `map`, `foldr`, etc. Although these functions can be defined in a concise and simple way, overhead is introduced at runtime. Program transformation techniques can play an important role in these situations. For example, consider the following function `foo`:

$$\texttt{foo}\;\texttt{xs} \;=\; \texttt{foldr}\;(+)\;0\;(\texttt{map}\;(+1)\;\texttt{xs})$$

to add `1` to the elements of a given list `xs` and then compute their total sum. From the programmer point of view, this definition is perfectly right, but there exist more efficient definitions for `foo`, like the following one:

$$\begin{array}{lcl} \texttt{foo}\;[] & = & 0 \\ \texttt{foo}\;(\texttt{x}:\texttt{xs}) & = & (\texttt{x}+1)+(\texttt{foo}\;\texttt{xs}) \end{array}$$

In contrast to the original definition, it is a first-order function and it can be executed more efficiently over existing functional logic compilers. In principle, the INDY system cannot be used to optimize the above function `foo` due to occurrences of the built-in function + and the higher-order functions `map` and `foldr`. Furthermore, the NPE framework of Alpuente *et al.* [1998b] (recalled in Chapter 3) suffers from the limitation that, within a lazy (call-by-name) semantics, terms in head normal form (i.e., rooted by a constructor symbol) cannot be evaluated during the partial evaluation process. This can drastically reduce the optimization power of the method in many cases (see Chapter 3). To overcome this problem, in Chapter 7, we introduced a novel approach for the partial evaluation of functional logic languages. The new scheme considers a maximally simplified representation into which programs written in a higher-level language (i.e., inductively sequential programs [Antoy, 1992] with evaluation annotations) are automatically translated. The restriction to not evaluate terms in head normal form has been avoided by defining a non-standard semantics which is well-suited to perform computations at partial evaluation time.

Now, the objective is to show how the improved framework of Chapter 7 can be successfully applied in practice. To this end, we first enrich the intermediate representation considered in Section 7.2.1 in order to cover all the facilities of the language

Curry. The resulting representation is essentially equivalent to the standard intermediate language FlatCurry [Hanus *et al.*, 2000], which has been proposed to provide a common interface for connecting different tools working on Curry programs (e.g., back ends for various compilers [Antoy and Hanus, 2000; Hanus and Sadre, 1999]). Then, the non-standard semantics of Section 7.2.2 is carefully extended in order to cover the additional language features. This extension is far from trivial, since the underlying calculus does not compute bindings but represents them by "residual" case expressions. However, there are a number of functions, such as equalities, (concurrent) conjunctions, some arithmetic functions, etc., in which the propagation of bindings among their arguments is crucial to achieve a good level of specialization. Therefore, we are committed to define a specific treatment for these important features of the language. Finally, in order to make the resulting framework practically applicable, we define appropriate control strategies which take into account the particularities of the considered language and the (non-standard) semantics. The resulting method is able to transform realistic Curry programs, what is not possible in partial evaluators in existence up to now. For instance, the "declarative" definition of `foo` can be automatically transformed into the more efficient version (see Section 8.4).

The implementation of a partial evaluator which follows the above ideas has been undertaken and experimentally tested. Essentially, the partial evaluator performs two phases. Firstly, it reads a program and translates it to FlatCurry. Secondly, the FlatCurry program is partially evaluated. The developed system is ready to be incorporated into the PAKCS compiler [Hanus *et al.*, 2000] for Curry as a source-to-source transformation on FlatCurry programs. For an effective deployment of the system, the programmer should simply include some annotations in the original program (indicating the function calls to be specialized). The fact that the partial evaluator is written in Curry also confirms that it is a complete language which is useful to implement realistic applications in a purely declarative manner.

## 8.2 Extending the Basic Framework

The aim of this section is to extend the basic approach of Chapter 7 in order to cover the facilities of a realistic multi-paradigm language: Curry [Hanus, 2000a]. To this end, we first empower the abstract representation of Section 7.2.1 with some additional features which constitute the most useful and sophisticated facilities of the language. Then, we correspondingly extend the rules of the RLNT calculus to properly deal with these new features.

### 8.2.1 An Intermediate Representation for Curry Programs

Our enriched abstract representation essentially coincides with the standard intermediate representation, FlatCurry [Hanus *et al.*, 2000], used during the compilation of Curry programs. It contains all the necessary information about a Curry program with all "syntactic sugar" compiled out and type-checking and lambda-lifting performed. In the extended representation, we allow the following expressions:

$$
\begin{array}{lcll}
e & ::= & x & \text{(variable)} \\
 & | & c(e_1,\ldots,e_n) & \text{(constructor)} \\
 & | & f(e_1,\ldots,e_n) & \text{(function call)} \\
 & | & (f)case\ e_0\ of\ \{p_1 \to e_1;\ldots;p_n \to e_n\} & \text{(case expression)} \\
 & | & external(e) & \text{(external function call)} \\
 & | & partcall(f,e_1,\ldots,e_k) & \text{(partial application)} \\
 & | & apply(e_1,e_2) & \text{(application)} \\
 & | & constr(\{x_1,\ldots,x_n\},e) & \text{(constraint)} \\
 & | & or(e_1,e_2) & \text{(disjunction)} \\
 & | & guarded(\{x_1,\ldots,x_n\},e_1,e_2) & \text{(guarded expression)}
\end{array}
$$

The right-hand side of each function definition is now an expression $e$ composed by variables, constructors, function calls, case expressions, and additional features like: non user-defined ("external") functions, higher-order features such as partial applications and applications of a functional expression to an argument, constraints (like equational constraints $e_1 =:= e_2$, possibly containing existentially quantified variables), disjunctions (to represent functions with overlapping left-hand sides), and guarded expressions (to represent conditional rules, i.e., the guard is always a constraint and the list of variables are the local variables which are visible in the constraint and in the right-hand side). A detailed description of the above features and their intended semantics can be found in [Hanus, 2000a]. We omit the committed choice construct of FlatCurry (as described in [Hanus *et al.*, 2000]) because, unfortunately, completeness is not generally preserved and more sophisticated analysis tools are necessary. This decision has almost no negative influence on the applicability of the partial evaluator since the committed choice is rarely used nor supported by the current version of the Curry→Prolog compiler [Antoy and Hanus, 2000], and in most reactive applications it can be replaced by the higher concept of "ports" for distributed programming [Hanus, 1999].

### 8.2.2 Extending the RLNT Calculus

In principle, one could extend the RLNT calculus in order to deal with all the facilities of FlatCurry in a simple way. The naïve idea is to treat all the additional features of the language as constructor symbols at partial evaluation time. This means that they are never partially evaluated but their original definitions are returned as they are by the partial evaluation process. However, in realistic Curry programs, the use of these additional features is perfectly common, hence it is not acceptable to residualize them at partial evaluation time. Our experimental tests have shown that no specialization is obtained in most cases if we follow this simple approach. For example, consider the expression "`foldr` (+) `0` [1, 2, 3]" which is represented in FlatCurry as "$foldr(partcall(external(+)), 0, [1, 2, 3])$" (where numbers are represented as constructor constants). By unfolding the call to $foldr$ (see its definition in Section 8.4), we get:

$$\begin{array}{lll} case\ [1,2,3]\ of\ \{ & [\,] \to 0; & \\ & (a:b) \to apply( & apply(partcall(external(+)), a), \\ & & foldr(partcall(external(+)), 0, b))\} \end{array}$$

which can be further reduced by using the Case Select rule to:

$$apply(apply(partcall(external(+)), 1), foldr(partcall(external(+)), 0, [2, 3]))$$

However, if we treat $apply$, $partcall$, $external$ and $+$ as constructor symbols, the RLNT calculus will not be able to evaluate this expression to `6`, i.e., no specialization is obtained.

On the other hand, extending the RLNT calculus of Section 7.2.2 with the standard semantics for the additional features of FlatCurry is not a good solution either. The problem stems from the fact that the RLNT calculus only propagates bindings forward into the branches of a case expression. However, there are a number of functions, such as equalities, (concurrent) conjunctions, some arithmetic functions, etc., in which the propagation of bindings among their arguments is crucial to achieve a good level of specialization. In order to propagate bindings[1] among different arguments, we lift some case expressions from inner positions to the top-level position, and propagate the corresponding bindings to the remaining arguments. For example, the expression[2]

$$[\![(\texttt{x} =:= \texttt{1})\ \&\ (\texttt{fcase x of}\ \{\texttt{1} \to \texttt{success}\})]\!]$$

can be transformed into

$$[\![\texttt{fcase x of}\ \{\texttt{1} \to (\texttt{x} =:= \texttt{1}\ \&\ \texttt{success})\}]\!]$$

The transformed expression can be now evaluated by the Case Guess rule, thus propagating the binding $\{\texttt{x} \mapsto \texttt{1}\}$ to the first conjunct:

$$\texttt{fcase x of}\ \{\texttt{1} \to [\![\texttt{1} =:= \texttt{1}\ \&\ \texttt{success}]\!]\}$$

[1]Recall that bindings are represented in our calculus by case expressions with a variable argument.
[2]Following [Hanus, 2000a], "`success`" denotes a constraint which is always solvable.

We notice that this transformation cannot be applied over arbitrary expressions since the intended (lazy) semantics is only preserved when the given function is *strict* in the position of the case expression. Nevertheless, typical FlatCurry programs contain many elements where the evaluation order is fixed. For instance, the guard of a conditional expression is strict, since it must be reduced to `True` (or "`success`") before applying a conditional rule, the arguments of most external functions are also strict, because they must be reduced to ground constructor terms before executing the external call, etc.

Furthermore, there are a number of situations in which an expression cannot be evaluated until all (or some) of its arguments have some particular form. For example, a call of the form $apply(e_1, e_2)$ can be only reduced if the first argument $e_1$ is of the form $partcall(\ldots)$. In these cases, we will try to evaluate the arguments of the function in hopes of achieving the required form. For the sake of a simpler presentation, we introduce the auxiliary function $try\_eval$. Given a function call $f(\overline{e_n})$ and a set of natural numbers $I$ which represents the set of strict arguments of function $f$, we define $try\_eval$ as follows:

$$try\_eval(f(\overline{e_n}), I) \;=\; \begin{cases} [\![(f)case\ x\ of\ \{\overline{p_k \to f(e_1, \ldots, e_{i-1}, e'_k, e_{i+1}, \ldots, e_n)}\}]\!] \\ \qquad\qquad \text{if } e_i = (f)case\ x\ of\ \{\overline{p_k \to e'_k}\} \\ \qquad\qquad\quad \text{for some } i \in I \\ [\![f(e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_n)]\!] \\ \qquad\qquad \text{if } \exists i \in \{1, \ldots, n\}.\ [\![e_i]\!] \Rightarrow e''_i, \\ \qquad\qquad\quad e'_i = del_{sq}(e''_i),\ e_i \neq e'_i \\ f(\overline{e_n}) \qquad \text{otherwise} \end{cases}$$

Here, we denote by $del_{sq}(e)$ the expression which results from deleting all occurrences of "$[\![$" and "$]\!]$" in $e$. We use it to test syntactic equality between expressions without taking into account the relative positions of "$[\![$" and "$]\!]$". Let us informally explain the function above. First, $try\_eval$ tries to float a case expression which occurs in the $i$-th argument (with $i \in I$) outside this argument. If this is not possible, it tries to evaluate some argument and, if this does not lead to a progress, the expression is just residualized. Since this definition of $try\_eval$ is non deterministic, we additionally require that the different cases are tried in their textual order and the arguments are evaluated from left to right (as standard in FlatCurry).

The remaining of this section makes extensive use of the above function to describe the concrete way in which our partial evaluator treats those features which are not covered by the RLNT calculus of Section 7.2.2.

**Non User-Defined Functions.**

FlatCurry programs often contain functions which have not been defined in the user's program code. Examples of these functions are those which are defined in the prelude[3] and those which are implemented in another language (*external* functions).

Regarding the prelude functions, we find some utility functions like `length`, `head`, and `tail`, the Boolean operators `&&` (sequential conjunction) and `||` (disjunction), or the "`if then else`" construct. An explicit treatment of these functions is not necessary since their definitions are available in the prelude. Hence, we simply consider them as any common user-defined function.

Regarding external functions, there are a number of arithmetic operators (e.g., $+$, $-$, $*$), as well as the basic input/output facilities (e.g., `putChar`, `readChar`), etc. Such functions are executed in Curry only if all arguments are evaluated to ground constructor terms.[4] When computing the partial evaluation of an external function, it seems reasonable to impose the same restriction. This implies that all arguments of external functions are assumed to be strict and, thus, the call to *try_eval* is performed with the complete set of argument positions:

$$[\![ external(f(\overline{e_n})) ]\!] \;\Rightarrow\; \begin{cases} external\_call(f(\overline{e_n})) \\ \qquad \text{if } \forall i \in \{1,\ldots,n\},\ e_i \text{ is ground constructor} \\ [\![ external(e') ]\!] \\ \qquad \text{if } try\_eval(f(\overline{e_n}), \{1,\ldots,n\}) = [\![ e' ]\!] \\ external(f(\overline{e_n})) \\ \qquad \text{otherwise} \end{cases}$$

where $external\_call(e)$ evaluates $e$ using its predefined definition. Basically, the partial evaluator first tries to execute the external function and, if this is not possible because all arguments are not ground constructor terms, then it tries to evaluate its arguments. Furthermore, we need to add the rule

$$[\![ external((f)case\ x\ of\ \{\overline{p_k \to e_k}\}) ]\!] \;\Rightarrow\; [\![ (f)case\ x\ of\ \{\overline{p_k \to external(e_k)}\} ]\!]$$

to move a case expression obtained by *try_eval* outside the external call (in order to allow further evaluation of the branches).

The only exception to the above rule are I/O actions, for which Curry follows the monadic approach to I/O. These functions act on the current "state of the outside world", and hence they are residualized since this state is not known at partial evaluation time, i.e., the result of partially evaluating them should not depend on the particular time when the specialization is performed.

[3]The prelude of Curry contains a set of standard datatype and function definitions which are added to each Curry program.

[4]There are few exceptions to this general rule but typical external functions (like arithmetic operators) comply with this condition. Therefore, we assume this request for the sake of simplicity.

**Constraints.**

The treatment for constraints heavily depends on the associated constraint solver. In the following, we only consider *equational* constraints. An elementary constraint is an equation $e_1 =:= e_2$ between two expressions which is solvable if both sides are reducible to unifiable constructor terms. This notion of equality, the so-called *strict* equality, is considered within our calculus by the following rule:

$$[\![e_1 \ =:= \ e_2]\!] \ \Rightarrow \ \begin{cases} case_\sigma(success) & \text{if } \sigma = mgu(e_1, e_2) \text{ and } e_1, e_2 \\ & \text{are constructor terms} \\ try\_eval(e_1 \ =:= \ e_2, \{1,2\}) & \text{otherwise} \end{cases}$$

Note that we call to *try_eval* with the set of positions $\{1, 2\}$ since function "=:=" is strict in its two arguments. Here, we use $case_\sigma(success)$ as a shorthand for denoting the encoding of $\sigma$ by nested (flexible) case expressions with *success* at the leaf of the last branch. For example, the expression $[\![\texttt{C x 2 =:= C 1 y}]\!]$, whose *mgu* is $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}$ is evaluated to

$$\texttt{fcase x of } \{1 \to \texttt{fcase y of } \{2 \to \texttt{success}\}\}$$

This simple treatment of constraints is not sufficient in practical programs since they are often used in concurrent conjunctions, which is written as $c_1 \ \& \ldots \& \ c_n$ ("&" is a built-in operator which evaluates its arguments concurrently). In this case, the evaluation of constraints can instantiate variables and the corresponding bindings must be propagated to the remaining conjuncts. The problematic point here is that we cannot move arbitrary case expressions to the top level, but only flexible case expressions (otherwise, we could change the floundering behavior of the program). Consider, for instance, the following simple functions:

$$\begin{array}{lcl} \texttt{f x} & = & \texttt{case x of } \{1 \to \texttt{success}\} \\ \texttt{g x} & = & \texttt{fcase x of } \{1 \to \texttt{success}\} \end{array}$$

where `f` is rigid and `g` is flexible. Given the expression $[\![\texttt{f x \& g x}]\!]$, if we allow to float arbitrary case expressions out, we could perform the following evaluation:

$$\begin{array}{lcl} [\![\texttt{f x \& g x}]\!] & \Rightarrow & [\![\texttt{case x of } \{1 \to \texttt{success}\} \ \& \ \texttt{g x}]\!] \\ & \Rightarrow & [\![\texttt{case x of } \{1 \to \texttt{success \& g x}\}]\!] \\ & \Rightarrow & \texttt{case x of } \{1 \to [\![\texttt{success \& g 1}]\!]\} \end{array}$$

which ends up in $\texttt{case x of } \{1 \to \texttt{success}\}$. Note that this residual expression suspends if variable `x` is not instantiated, whereas the original expression could be reduced by evaluating first function `g` and then function `f`. Therefore, we handle concurrent conjunctions as follows:

$$[\![c_1 \,\&\, \ldots \,\&\, c_n]\!] \;\Rightarrow\; \begin{cases} success \\ \qquad\qquad \text{if } c_i = success \text{ for all } i \in \{1,\ldots,n\} \\ [\![fcase\ x\ of\ \{\overline{p_k \to (c_1 \,\&\ldots\&\, c_{i-1} \,\&\, e'_k \,\&\, c_{i+1} \,\&\, \ldots \,\&\, c_n)}\}]\!] \\ \qquad\qquad \text{if } c_i = fcase\ x\ of\ \{\overline{p_k \to e'_k}\} \text{ for some } i \in \{1,\ldots,n\} \\ [\![c_1 \,\&\ldots\&\, c_{i-1} \,\&\, c'_i \,\&\, c_{i+1} \,\&\, \ldots \,\&\, c_n]\!] \\ \qquad\qquad \text{if } \exists i \in \{1,\ldots,n\}.\ [\![c_i]\!] \Rightarrow c''_i,\ c'_i = del_{sq}(c''_i), c_i \neq c'_i \\ c_1 \,\&\, \ldots \,\&\, c_n \\ \qquad\qquad \text{otherwise} \end{cases}$$

In contrast to external functions, note that only flexible case expressions are moved to the top level. Equational constraints can also contain local existentially quantified variables. In this case they take the form $constr(vars, c)$, where $vars$ are the existentially quantified variables in the constraint $c$. We treat these constraints as follows:

$$[\![constr(vars, c)]\!] \;\Rightarrow\; \begin{cases} success & \text{if } c = success \\ try\_eval(constr(vars, c), \{2\}) & \text{otherwise} \end{cases}$$

Note that the above rule moves all bindings to the top level, even those for the local variables in $vars$. In practice, case expressions denoting bindings for the variables in $vars$ are removed since they are local, but we keep the above formulation for simplicity.

**Guarded Expressions.**

In Curry, functions can be defined by conditional rules of the form

$$f\ e_1 \ldots e_n \mid c \;=\; e$$

where $c$ is a constraint (rules with multiple guards are also allowed but considered as syntactic sugar for denoting a sequence of rules). Conditional rules are represented in FlatCurry by the *guarded* construct. At partial evaluation time, we are interested in inspecting not only the guard but also the right-hand side of the equation whose guard is evaluated to $success$. However, only bindings produced from the evaluation of the guard can be floated out (since this is the unique strict argument). In particular, if the guard becomes trivial, i.e., $success$, we can delete it and proceed with the evaluation of the right-hand side:

$$[\![guarded(vars, gc, e)]\!] \;\Rightarrow\; \begin{cases} [\![e]\!] & \text{if } gc = success \\ try\_eval(guarded(vars, gc, e), \{2\}) & \text{otherwise} \end{cases}$$

As in the case of constraints, the application of $try\_eval$ can unnecessarily propagate some bindings (i.e., those for the variables in $vars$) outside the guarded expression. A precise treatment can be easily defined, but we preserve the above presentation for the sake of readability.

**Higher-Order Functions.**

The higher-order features of functional programming are implemented in Curry in Warren's style [Warren, 1982], i.e., by providing a (first-order) definition of the application function (*apply*). Since Curry excludes higher-order unification, the operational semantics of Curry covers the usual higher-order features of functional languages by means of the following axiom [Hanus, 2000a]:

$$[\![ \mathit{apply}(f(e_1, \ldots, e_m), e) ]\!] \Rightarrow f(e_1, \ldots, e_m, e)$$

if $f$ has arity $n > m$. Thus, an application is evaluated by simply adding the argument to a partial call. In FlatCurry, we distinguish partial applications from total functions; namely, partial applications are distinguished by means of the *partcall* symbol. The next rule illustrates the way in which our partial evaluator treats these higher-order features:

$$[\![ \mathit{apply}(e_1, e_2) ]\!] \;\Rightarrow\; \begin{cases} [\![ f(\overline{c_k}, e_2) ]\!] & \text{if } e_1 = \mathit{partcall}(f, \overline{c_k}),\; k+1 = \mathit{ar}(f) \\ \mathit{partcall}(f, \overline{c_k}, e_2) & \text{if } e_1 = \mathit{partcall}(f, \overline{c_k}),\; k+1 < \mathit{ar}(f) \\ \mathit{try\_eval}(\mathit{apply}(e_1, e_2), \{1\}) & \text{otherwise} \end{cases}$$

where $ar(f)$ denotes the arity of the function $f$. Roughly speaking, we allow a partial function to become a total function by adding the missing argument, if possible. In general, after adding the extra argument, the function does not have the right number of arguments yet and, thus, it is still a partial function. Otherwise, we evaluate the arguments of *apply* in prospect of achieving a partial call after evaluation. Note that *try_eval* is called with the set $\{1\}$ in order to avoid the propagation of bindings from the evaluation of non-strict arguments (i.e., from the second argument of *apply*).

**Overlapping Left-Hand Sides.**

Overlapping left-hand sides in Curry programs produce a disjunction whenever the different alternatives have to be considered. Similarly, we treat *or* expressions in FlatCurry as follows:

$$[\![ \mathit{or}(e_1, e_2) ]\!] \;\Rightarrow\; \mathit{or}([\![ e_1 ]\!], [\![ e_2 ]\!])$$

Roughly speaking, we residualize *or* expressions and proceed to evaluate the different alternatives.

## 8.3 The Partial Evaluator in Practice

Following the structure of many on-line partial evaluators (see, e.g., [Gallagher, 1993]), we sketch the implemented partial evaluation algorithm which is parametric w.r.t. an unfolding rule $\mathcal{U}$ and an abstraction operator *abstract*:

> **Input**: a program $\mathcal{R}$ and a set of terms $T$
> **Output**: a set of terms $S$
> **Initialization**: $i := 0;\ T_0 := T$
> **Repeat**
> 1. $S := \mathcal{U}(T_i, \mathcal{R})$;
> 2. $T_{i+1} := abstract(T_i, S)$;
> 3. $i := i + 1$;
>
> **Until** $T_i = T_{i-1}$ (modulo renaming)
> **Return** $S := T_i$

The above partial evaluation algorithm involves the two classical control issues: as for local control, we are concerned with the definition of an unfolding rule $\mathcal{U}$ to compute finite partial evaluations; as for global control, we define a safe abstraction operator *abstract* to ensure the termination of the iterative process.

**Local Control.**

Regarding the local control, the main novelty w.r.t. previous partial evaluators for functional logic programs is the use of a non-standard semantics, the RNLT calculus, to perform computations at partial evaluation time. Since RLNT computations are deterministic and do not produce bindings (they are represented by case structures), the restriction to not evaluate terms in head normal form of previous partial evaluators is avoided.

In order to ensure the finiteness of RLNT derivations, the INDY partial evaluator uses an unfolding rule based on the homeomorphic embedding ordering. However, in the presence of an infinite signature (e.g., natural numbers in Curry), this unfolding rule can lead to non-terminating computations. For example, consider the following Curry program which generates a list of natural numbers within two given limits:

```
enum a b  =  if a > b then [] else (a : enum (a + 1) b)
```

During its specialization w.r.t. the call `enum 1 n`, the following calls are produced:

`enum 1 n`, `enum 2 n`, `enum 3 n`, ...

where no call embeds any previous call.

Therefore, we have chosen a safe (and "cheap") unfolding rule:[5] only the unfolding of one function call is allowed. The main advantage of this approach is that expressions can be "folded back" (i.e., can be proved closed) w.r.t. any partially evaluated call. In practice, this generates optimal recursive functions in many cases. As a

[5]The positive supercompiler of Jones *et al.* [1993] employs a similar strategy.

counterpart, many (unnecessary) intermediate functions may appear in the residual program. This does not mean that we incur in a "code explosion" problem since this kind of redundant rules can be easily removed by a postunfolding phase (similarly to [Jones *et al.*, 1993]). In the implemented partial evaluator, we consider that a rule defining function $f$ is redundant provided that: (a) $f$ does not belong to the set of original function calls to be partially evaluated, (b) $f$ is called only once in the residual program, and (c) $f$ is not recursive (this avoids infinite unfolding). In the case when a rule is redundant, the call to function $f$ is unfolded and the definition of $f$ is removed from the program. Our experiments with the one-step unfolding rule and the postunfolding phase indicate that this leads to optimal (and concise) residual functions in many cases.

**Global Control.**

As discussed above, a well-quasi ordering like the homeomorphic embedding ordering cannot be used in practice since we consider an infinite signature. Therefore, we implement an abstraction operator which uses a *well-founded* order to ensure termination and generalizes those calls which do not satisfy this ordering by using the $msg$ (*most specific generalization*). Abstraction operators based on a well-founded order are computationally less expensive than operators based on a well-quasi ordering. The reason is that, for well-quasi orders, one is forced to compare the expression to be abstracted with all terms already partially evaluated. In contrast, by using a well-founded order, it suffices to check whether the expression is strictly greater than the last partially evaluated call. Abstraction operators based on this relation are defined in, e.g., [Martens and Gallagher, 1995].

The main novelty of our abstraction operator w.r.t. previous operators implemented in existing partial evaluators is that it is *guided* by the RLNT calculus. The key idea is to take into account the position of the square brackets of the calculus in expressions (see Definition 90); namely, subterms within square brackets should be added to the set of partially evaluated terms (if possible, otherwise generalized) since further evaluation is still required, while subterms which are not within square brackets should be definitively residualized (i.e., ignored by the abstraction operator, except for operation-rooted terms). The combination of this strategy with the above unfolding rule gives rise to efficient residual programs in many cases, while still guaranteeing termination.

## 8.4 Experimental Results

This section describes some experiments with an implementation of a partial evaluator for Curry programs which follows the guidelines presented in this chapter. Our

| Benchmark | mix | original | specialized | speedup |
|---|---|---|---|---|
| `allones` | 450 | 320 | 250 | 1.28 |
| `double_app` | 1550 | 370 | 270 | 1.37 |
| `double_flip` | 580 | 530 | 400 | 1.32 |
| `kmp` | 16250 | 300 | 30 | 10 |
| `length_app` | 650 | 330 | 260 | 1.27 |

Table 8.1: Benchmark results

partial evaluation tool is implemented in Curry itself and it is publicly available at

`http://www.dsic.upv.es/users/elp/soft.html`

The implemented partial evaluator first translates the subject program into a FlatCurry program. To do this, we use the module `Flat.curry` for meta-programming in Curry (included in the current distribution of PAKCS [Hanus *et al.*, 2000]). This module contains datatype definitions to treat FlatCurry programs as data objects (using a kind of *ground representation*). In particular, the I/O action `readFlatCurry` reads a Curry program, translates it to the intermediate language FlatCurry, and finally returns a data structure representing the input program.

We used the implementation to conduct a number of experiments starting with pure functional logic programs (without the additional features of Curry). Table 8.1 shows the results obtained from some benchmarks already used in Chapter 6: `allones`, `double_app`, `double_flip`, `kmp` and `length_app`. For each benchmark, we show the specialization time (column mix), the timings for executing the original and the specialized programs (columns original and specialized), and the speedups achieved (column speedup). Times are expressed in milliseconds and are the average of 10 executions. Runtime input goals were chosen to give a reasonably long overall time. The results obtained are comparable to those obtained with the INDY system, which demonstrates that no appreciable slowdown is produced for pure programs when extra features are not used. All benchmarks have been specialized w.r.t. function calls containing no static data, except for the `kmp` example (what explains the larger speedup attained).

Unfortunately, these ad-hoc programs do not happen very frequently in real Curry applications. This prompted us to benchmark more realistic examples where the most important features of Curry programs may appear. One of the most useful features of functional languages are higher-order functions since they improve code reuse and modularity in programming. Thus, such features are often used in practical Curry programs (much more than in logic programs, which are usually based on first-order logic and offer only weak features for higher-order programming, e.g., Prolog programs).

Furthermore, almost every practical program uses built-in arithmetic functions which are available in Curry as external functions (whereas they are not common in purely narrowing-based functional logic languages). These practical features were not treated in previous approaches to partial evaluation of functional logic languages.

The following functions `map` (for applying a function to each element of a list) and `foldr` (for accumulating all list elements) are often used in functional (logic) programs:

```
map   _ []       = []
map   f (x : xs) = f x : map f xs

foldr _ z []      = z
foldr f z (h : t) = f h (foldr f z t)
```

For instance, the expression `foldr (+) 0 [1, 2, 3]` is the sum of the elements of the list `[1, 2, 3]`. Due to the special handling of higher-order features (*apply* and *partcall*) and built-in functions (*external*), our partial evaluator is able to reduce occurrences of this expression to `6`. However, realistic programs contain (partially instantiated) calls to higher-order functions instead of such constant expressions. For instance, the expression `foldr (+) 0 xs` is specialized by our partial evaluator to `f xs` where `f` is a first-order function defined by:

```
f []       = 0
f (x : xs) = x + f xs
```

Calls to this residual function run `3.5` times faster (in the Curry→Prolog compiler [Antoy and Hanus, 2000] of PAKCS) than calls to the original definitions; also, heap usage has been reduced significatively (see Table 8.2). Similarly, the expression `foldr (+) 0 (map (+1) xs)` has been successfully specialized to the efficient version of Section 8.1. It is interesting to note that our partial evaluator neither requires function definitions in a specific format (like "foldr/build" in short cut deforestation [Gill *et al.*, 1993]) nor it is restricted to "higher-order macros" (as in [Wadler, 1990]), but can handle arbitrary higher-order functions. For instance, the higher-order function

```
iterate f n = if n == 0 then f else iterate (f.f) (n - 1)
```

which modifies its higher-order argument in each recursive call (`f.f` denotes function composition) can be successfully handled by our partial evaluator. Table 8.2 shows the results of specializing some calls to higher-order and built-in functions. For each benchmark, we show the execution time and heap usage for the original and specialized calls and the speedups achieved. The input list `xs` contains 20,000 elements in each call. Times are expressed in milliseconds and measured on a 700 MHz Linux-PC. The programs were executed with the Curry→Prolog compiler of PAKCS.

| Benchmark | original | | specialized | | speedup |
|---|---|---|---|---|---|
| | time | heap | time | heap | |
| `foldr (+) 0 xs` | 210 | 2219168 | 60 | 619180 | 3.5 |
| `foldr (+) 0 (map (+1) xs)` | 400 | 4059180 | 100 | 859180 | 4.0 |
| `foldr (+) 0 (map square xs)` | 480 | 4970716 | 170 | 1770704 | 2.8 |
| `foldr (++) [] xs` (*concat*) | 290 | 2560228 | 110 | 560228 | 2.6 |
| `filter (>100) (map (*3) xs)` | 750 | 6639908 | 430 | 3120172 | 1.7 |
| `map (iterate (+1) 2) xs` | 1730 | 17120280 | 600 | 6720228 | 2.9 |

Table 8.2: Benchmark results

Another important feature of Curry is the use of (concurrent) constraints. Consider, for instance, the following function `arith`:

```
digit 0    =  success
...
digit 9    =  success
arith x y  =  x + x =:= y & x * x =:= y & digit x
```

Calls to "`arith`" might be completely evaluated at partial evaluation time. Actually, our partial evaluator returns the residual function `arith'` :

```
arith' 0 0  =  success
arith' 2 4  =  success
```

for the partial evaluation of `arith x y`.

In this section, we have shown the specialization of calls to some small functions. However, this does not mean that only small programs can be specialized with our partial evaluation tool. Actually, for larger programs, the programmer would simply include some annotations indicating the function calls to be specialized (what usually involve only calls to small functions). In this case, the same program would be returned by the system except for the annotated calls, which are replaced by new calls to the specialized functions, together with the definitions of such specialized functions.

## 8.5 Conclusions

This chapter describes a successful experience in the development of a program transformation engine for a realistic multi-paradigm declarative language. Our method builds on the theoretical basis of Chapter 7 for the partial evaluation of functional logic languages. We have shown how this framework can be successfully applied in practice by extending the underlying method in order to cover the facilities of the language Curry. The resulting method is able to transform realistic Curry programs, which is not possible in partial evaluators in existence up to now. For instance, almost every practical Curry program uses built-in arithmetic functions, constraints, and higher-order functions. These practical features were not treated in previous approaches to partial evaluation of functional logic languages. A partial evaluator has been implemented which is able to generate efficient (and reasonably small) specialized programs. The system is ready to be incorporated into the PAKCS compiler [Hanus *et al.*, 2000] for Curry as a source-to-source transformation on FlatCurry programs. For an effective deployment of the system, the programmer may include some annotations in the original program (indicating the function calls to be specialized). We fairly think that the fact that the whole system is written in Curry also confirms that Curry is a complete, general-purpose language, useful to implement realistic applications in a purely declarative manner.

Regarding efficiency, we have not considered yet a formal technique to measuring the improvement in efficiency achieved by partial evaluation. In the last part of the thesis, we provide several abstract criteria to formally measure the potential benefit of our partial evaluation algorithm as well as a technique to derive recurrence equations which relate the cost of executing the original and the residual program.

# Part IV

# Effectiveness

# Chapter 9

# Effectiveness of Partial Evaluation

We introduce a framework for assessing the effectiveness of partial evaluators in functional logic languages. Our framework is based on properties of the rewrite system which underlies a functional logic program. Consequently, our assessment is independent of any specific language implementation or computing environment. We define several criteria for measuring the cost of a computation: number of steps, number of function applications, and pattern matching effort. Most importantly, we express the cost of each criterion by means of recurrence equations over algebraic data types, which can be automatically inferred from the partial evaluation process itself. In some cases, the equations can be solved by transforming their arguments from arbitrary data types to natural numbers. In other cases, it is possible to estimate the improvement of a partial evaluation by analyzing the associated cost recurrence equations. A short version of this chapter appeared in [Albert *et al.*, 2001a, 2000a,b].

## 9.1 Introduction

A common motivation of partial evaluation techniques is to improve the efficiency of a program while preserving its meaning. Rather surprisingly, relatively little attention has been paid to the development of formal methods for reasoning about the effectiveness of the partial evaluation transformation; usually, only experimental tests on particular languages and compilers are undertaken. Clearly, a machine-independent way of measuring the effectiveness of partial evaluation would be useful to both users and developers of partial evaluators.

Predicting the speedup achieved by partial evaluators is generally undecidable.

We begin this chapter by reviewing some approaches to this problem. Andersen and Gomard's *speedup analysis* [Andersen and Gomard, 1992] predicts a relative interval of the speedup achieved by a program specialization. Nielson's type system [Nielson, 1988] formally expresses when a partial evaluator is better than another. Other interesting efforts investigate *cost analyses* for logic and functional programs which may be useful for determining the effectiveness of program transformations; for instance, Debray and Lin's method [Debray and Lin, 1993] for the (semiautomatic) analysis of the worst-case cost of a large class of logic programs and Sands's theory of cost equivalence [Sands, 1995] for reasoning about the computational cost of *lazy* functional programs. Laziness introduces a considerable difficulty since the cost of lazy (call-by-name) computations is not *compositional* [Sands, 1995]. Although the two cost analyses above can be used to study the effectiveness of partial evaluation, their authors did not address this issue.

All these efforts mainly base the cost of executing a program on the number of steps performed within a computation. However, simple experiments show that the number of steps and the associated runtime are not easily correlated. Consider, for instance, the positive supercompiler described in [Sørensen *et al.*, 1996]. As noted by [Sørensen, 1994, Chapter 11], the residual program obtained by positive supercompilation — without the *postunfolding* phase—performs *exactly* the same number of steps as the original one. This does not mean that the process is useless. Rather, all intermediate data structures are gone, and such structures take up space and garbage collection time in actual implementations. Furthermore, the reduction in number of steps of a computation does not imply a proportional reduction in its execution time. The following example illustrates this point.

**Example 33** Reconsider operation `app` to concatenate lists:

$$\begin{array}{lcl} \texttt{app [] y} & = & \texttt{y} \\ \texttt{app } (\texttt{x}_1 : \texttt{x}_\texttt{s}) \texttt{ y} & = & \texttt{x}_1 : \texttt{app x}_\texttt{s} \texttt{ y} \end{array}$$

and the following partial evaluation w.r.t. `app x y` obtained by the partial evaluator Indy (Appendix A):[1]

$$\begin{array}{lcl} \texttt{app2s [] y} & = & \texttt{y} \\ \texttt{app2s } (\texttt{x} : \texttt{[]}) \texttt{ y} & = & \texttt{x} : \texttt{y} \\ \texttt{app2s } (\texttt{x}_1 : \texttt{x}_2 : \texttt{x}_\texttt{s}) \texttt{ y} & = & \texttt{x}_1 : \texttt{x}_2 : (\texttt{app2s x}_\texttt{s} \texttt{ y}) \end{array}$$

This residual program computes the same function as the original one but in approximately half the number of steps. This might suggest that the execution time of `app2s` (for sufficiently large inputs) should be about one half the execution time of

[1]Note that no input data are provided. In spite of this, Indy can still improve programs by shortening computations and removing intermediate data structures.

app. However, executions of function app2s in several environments (e.g., in the lazy functional language Hugs [Jones and Reid, 1998] and the functional logic language Curry [Hanus, 2000a]) show that speedup is only around 10%.

In order to reason about these counterintuitive results, we introduce several formal criteria to measure the efficiency of a functional logic computation. In particular, we define the cost of evaluating an expression in terms of the number of steps, the number of function applications, and the complexity of the pattern-matching or unification involved in the computation. Similar criteria are taken into account (only experimentally) in traditional profiling approaches (e.g., [Sansom and Peyton-Jones, 1997]). Let us remark that our aim is not to define a complex cost analysis for functional logic programs, but to introduce some representative cost criteria and then investigate their variations by the application of a particular partial evaluation method. The above criteria seem specially well-suited to estimate the speedup achieved by the general partial evaluation scheme based on narrowing.[2] In particular, we use *recurrence equations* to compare the cost of executing the original and residual programs. These equations can be automatically derived from the partial evaluation process itself and are parametric w.r.t. the considered cost criteria. Unlike traditional recurrence equations used to reason about the complexity of programs, the derived equations are defined on data structures rather than on natural numbers. This complicates the computation of their solutions, although in some cases useful statements about the improvements achieved by partial evaluation can be made by a simple inspection of the sets of equations. In other cases, these equations can be transformed into traditional recurrence equations and then solved by well-known mathematical methods. In this case, the solution yields an accurate value for the speedup achieved by the partial evaluation process.

## 9.2 Formal Cost Criteria

In many situations, the execution time of a program is the single most important sign of its efficiency. However, correlating execution times to program transformations is more of an art than a science. The reason is that execution times are affected by the hardware's instructions, the implementation of the compiler or interpreter, the effects of optimizations and caching, the facilities of the operating system to measure time intervals, the bias due to program instrumentation, etc.

The cost criteria that we introduce in this section are independent of the particular implementation of the language. Rather, they are formulated for a rewrite system,

[2]Nevertheless, our technique can be easily adapted to other related partial evaluation methods, such as partial deduction [Lloyd and Shepherdson, 1991] and positive supercompilation [Sørensen *et al.*, 1996].

which can be thought of as an abstract model of a program, and rely on operations which are, in one form or another, performed by likely implementations of rewriting and narrowing. Our cost criteria may be more difficult to compute than execution times, but provide detailed and accurate information that can shed some light on the results of traditional profiling approaches.

In the following, we introduce three criteria to measure the cost of the evaluation of a term. Informally, they count the number of steps of the evaluation, the number of symbol applications to construct intermediate and final results and the number of symbol comparisons for rule selection.

The first cost criterion that we consider has been widely used in the literature. This is the *number of steps*, or *length*, of the evaluation of a term. The following trivial definition is presented only for uniformity with the remaining costs.

**Definition 97 (number of steps)** *We denote by $\mathcal{S}$ a function on rewrite rules, called the number of steps, as follows. If $R$ is a rewrite rule, then $\mathcal{S}(R) = 1$.*

The second cost criterion that we consider is the number of symbol applications that occur within a computation. By symbol, we mean either a constructor or a defined operation. Although constructor applications differ from defined operation applications, we do not need to make a distinction for most of our discussion. Counting applications is sensible because, in most implementations of a functional logic language, an evaluation will execute some machine instructions that directly correspond to each symbol application.

For example, an eager language might implement a defined operation application with a call to a subroutine. The cost abstracted by our criterion is then the creation and destruction of the call allocation record and the instructions executed to transfer control both in and out of the subroutine. Similarly, the cost abstracted in a lazy language would be the creation of a thunk.

Commonly, both eager and lazy languages store the result of a computation and intermediate results in linked data structures that closely represent terms. These structures are created by the allocation of a portion of dynamic memory and its initialization by means of tags and pointers or references to other similar portions. A constructor application abstracts the cost to create these data structures.

The following definition bundles together all applications. It can be easily specialized to consider constructor or defined symbol applications separately, denoted by $\mathcal{A}_c$ and $\mathcal{A}_d$, respectively.

**Definition 98 (number of applications)** *We denote by $\mathcal{A}$ a function on rewrite rules, called the number of applications. If $R = l \to r$ is a rewrite rule, then $\mathcal{A}(R)$ is the number of occurrences of non-variable symbols in $r$.*

The above definition is appropriate for a first-order language in which function applications are not curried. In a fully curried language, $\mathcal{A}(l \to r)$ would return one less the number of symbols in $r$ (including variables).

**Example 34** Consider again the operations `app` and `app2s` discussed in earlier examples. From the above definition and the operations' rewrite rules, we derive:

$$\mathcal{A}(\texttt{app [] y} = \texttt{y}) = \texttt{0}$$
$$\mathcal{A}(\texttt{app (x : x}_\texttt{s}\texttt{) y} = \texttt{x : app x}_\texttt{s}\texttt{ y}) = \texttt{2}$$

$$\mathcal{A}(\texttt{app2s [] y} = \texttt{y}) = \texttt{0}$$
$$\mathcal{A}(\texttt{app2s (x : []) y} = \texttt{x : y}) = \texttt{1}$$
$$\mathcal{A}(\texttt{app2s (x}_\texttt{1}\texttt{ : x}_\texttt{2}\texttt{ : x}_\texttt{s}\texttt{) y} = \texttt{x}_\texttt{1}\texttt{ : x}_\texttt{2}\texttt{ : app2s x}_\texttt{s}\texttt{ y}) = \texttt{3}$$

The third cost criterion that we consider abstracts the effort necessary to select which program rule must be fired in a single computation step. In traditional functional languages, this corresponds to pattern matching. In functional logic languages, this is typically generalized to unification.

Our definition is influenced by the computation of a needed step using a definitional tree. We make the assumption that the number of rewrite rules in a program does not affect the efficiency of a computation. The reason is that a reference to the symbol being applied can be resolved at compile-time, i.e., when a source program is translated into code for a concrete or abstract machine. However, when a defined operation $f$ is applied to its arguments, in non-trivial cases, one needs to inspect certain occurrences of several arguments of the application of $f$ to determine which rule defining $f$ should be fired. This determination must be performed at run-time.

Continuing our examples, the application of defined operation `app` to arguments `x` and `y` requires the inspection of `x`. Argument `x` must be evaluated to a head normal form before a rewrite rule can be applied. If the root symbol of (the evaluation to head normal form of) `x` is constructor "[]", then, the first rewrite rule defining `app` must be fired. By contrast, if the root symbol is constructor ":", then, the second rewrite rule defining `app` applies.

**Definition 99 (pattern matching effort)** *Let $\mathcal{R}$ be a program. We denote by $\mathcal{P}$ a function on rewrite rules, called pattern matching effort, as follows. If $R = l \to r$ is a rewrite rule of $\mathcal{R}$, then $\mathcal{P}(R)$ is the number of constructor symbols in $l$.*

We note that $\mathcal{P}$ gives a worst-case measure of the pattern matching effort. In particular, if a ground expression $e$ is evaluated using rule $R$, then $\mathcal{P}(R)$ returns exactly the number of constructor symbols of $e$ whose inspection is required. However, whenever $e$ contains free variables, the value returned by $\mathcal{P}(R)$ represents an upper bound.

**Example 35** Consider again the operations app and app2s discussed in earlier examples. From the above definition and the operations' rewrite rules, we derive:

$$\mathcal{P}(\mathtt{app}\ []\ \mathtt{y} \to \mathtt{y}) = 1$$
$$\mathcal{P}(\mathtt{app}\ (\mathtt{x} : \mathtt{x_s})\ \mathtt{y} \to \mathtt{x} : \mathtt{app}\ \mathtt{x_s}\ \mathtt{y}) = 1$$
$$\mathcal{P}(\mathtt{app2s}\ []\ \mathtt{y} \to \mathtt{y}) = 1$$
$$\mathcal{P}(\mathtt{app2s}\ (\mathtt{x} : [])\ \mathtt{y} \to \mathtt{x} : \mathtt{y}) = 2$$
$$\mathcal{P}(\mathtt{app2s}\ (\mathtt{x_1} : \mathtt{x_2} : \mathtt{x_s})\ \mathtt{y} \to \mathtt{x_1} : \mathtt{x_2} : \mathtt{app2s}\ \mathtt{x_s}\ \mathtt{y}) = 2$$

For simplicity, we do not consider non-deterministic computations explicitly, although our cost measures could be extended following the ideas of Debray and Lin [1993]. On the other hand, many partial evaluation methods do not change the non-determinism of computations, i.e., the *search spaces* of a given goal in the original and residual programs have essentially the same structure. In particular, this is the case of the NPE method. Therefore, a cost measure which quantifies the amount of non-determinism would not be affected by our partial evaluation method.

In the remainder of this chapter, we denote by $C$ a generic cost criterion which stands for $\mathcal{S}$, $\mathcal{A}$ or $\mathcal{P}$. Furthermore, most of the developments in the following sections are independent of the considered criteria; thus, $C$ could also denote more elaborated criteria, like the number of variable bindings, the "size" of the reduced expression, etc.

The previous definitions allow us to define the cost of a derivation as the total cost of its constituent steps:

**Definition 100 (cost of a derivation)** *Let $\mathcal{R}$ be a program and $t$ a term. Let $C$ denote a generic cost criterion. We overload function $C$ by partially defining it on derivations as follows. If $D : t_0 \leadsto_{(p_1,R_1,\sigma_1)} t_1 \cdots \leadsto_{(p_n,R_n,\sigma_n)} t_n$ is a derivation, then $C(D) = \sum_{i=1}^{n} C(R_i)$.*

For the developments in the next section, it is more convenient to reason about the efficiency of programs when a cost measure is defined over terms rather than entire *computations*. We use "computation" as a generic name for the *evaluation* of a term, i.e., a narrowing derivation ending in a constructor term. In general, different strategies applied to a same term may produce evaluations of different lengths and/or fail to terminate. For instance, if term $t$ contains uninstantiated variables, then there may exist distinct evaluations of $t$ obtained by distinct instantiations of $t$'s variables. Luckily, needed narrowing gives us some chance. We allow uninstantiated variables in a term $t$ as long as these variables are not instantiated during its evaluation, i.e., we have a derivation of the form $t \leadsto^*_{id} d$. In this case, there is a concrete implementation of needed narrowing, namely the mapping denoted by $\lambda$ in Definition 5, in which $t \leadsto^*_{id} d$ is the only possible derivation stemming from $t$. We call *$\lambda$-derivation* a derivation computed by $\lambda$. Therefore, to be formal, in the technical results of this chapter we consider only $\lambda$-derivations.[3] The next property is an immediate consequence of Theorem 9 (disjointness of computed solutions). It allows us to define the cost of a term as the cost of its $\lambda$-derivation when the computed substitution is empty (since it is unique).

**Lemma 101 (uniqueness of identity derivations)** *Let $\mathcal{R}$ be an inductively sequential program and $t$ a term built over the signature of $\mathcal{R}$. If in $\mathcal{R}$ needed narrowing computes a $\lambda$-derivation, $t \leadsto^*_{id} d$, where $d$ is a constructor term, then this is the only $\lambda$-derivation for $t$ in $\mathcal{R}$.*

*Proof.* The proof is by contradiction. If there exists another $\lambda$-derivation $t \leadsto^*_{\sigma} d'$, then the set of computed answers for $t$ would not be disjoint (since $id$ is more general than any substitution), contrary to Theorem 9. $\square$

Now, we overload function $C$ by partially defining it on terms as follows.

**Definition 102 (cost of a term)** *Let $\mathcal{R}$ be a program and $t$ a term. Let $C$ denote a cost criterion. If $t \leadsto_{(p_1,R_1,\sigma_1)} \cdots \leadsto_{(p_k,R_k,\sigma_k)} d$ is a $\lambda$-derivation, where $d$ is a constructor term and $\sigma_1 = \cdots = \sigma_k = id$, we define $C(t) = \sum_{i=1}^{k} C(R_i)$.*

We apply the previous definitions to Example 33. The next table summarizes (with minor approximations to ease understanding) the cost of computations with both functions when the arguments are constructor terms:

| | $\mathcal{S}$ | $\mathcal{A}_c$ | $\mathcal{A}_d$ | $\mathcal{P}$ |
|---|---|---|---|---|
| `app` | $n$ | $n$ | $n$ | $n$ |
| `app2s` | $0.5\,n$ | $n$ | $0.5\,n$ | $n$ |

Here $n$ represents the *size* of the inputs to the functions, i.e., the number of elements in the first argument of `app` and `app2s` (the second argument does not affect significatively the cost of computations). The first observation is that not all cost criteria have been improved. In fact, the number of constructor applications and the pattern matching effort remain unchanged. To obtain a "global" value of the improvement achieved by a partial evaluation, one might assume an equal unit cost for all criteria. In this way, the total cost of a computation using `app` is $4\,n$. Likewise, the cost of a computation using `app2s` is $3\,n$. The speedup is only 25%. In general, one should determine the appropriate weight of each cost criterion for a specific language environment. For instance, if we increase the unit cost of $\mathcal{A}_c$ and decrease that of $\mathcal{S}$—a more realistic choice in our environment—the improvement of `app2s` over `app` estimated by our criteria closely explains the lower speedup measured experimentally (10%).

[3]Note that this is not a practical restriction in our context, since $\lambda$-derivations are efficient, easy to compute, and used in the implementations of modern functional logic languages such as Curry [Hanus, 2000a] and Toy [López-Fraguas and Sánchez-Hernández, 1999].

## 9.3 Effectiveness of Partial Evaluation

In this section, we are concerned with the problem of determining the improvement achieved by a partial evaluation in the context of a functional logic language.

### 9.3.1 Automatic Generation of Recurrence Equations

We propose the use of recurrence equations to analyze how a cost criterion is affected by the partial evaluation process. Although we will illustrate our proposal over the cost criteria introduced in the previous section, our developments are parametric w.r.t. the considered cost criteria. Our approach is inspired by the standard use of recurrence equations to analyze the complexity of algorithms in terms of their inputs (see, e.g., [Aho *et al.*, 1974] for imperative, [Sands, 1995] for functional, and [Debray and Lin, 1993] for logic programs).

**Definition 103 (recurrence equation)** *Let $\mathcal{R}$ be a program and $s$ be a term. Given a needed narrowing derivation $s \leadsto_\sigma^+ t$, its associated recurrence equation is: $C(\sigma(s)) = C(t) + k$, where $k = C(s \leadsto_\sigma^+ t)$.*

However, we are not interested in arbitrary recurrence equations, since in general they would not be useful. Rather, we present a technique for deriving sets of recurrence equations tightly associated to the partial evaluation process. These equations, specifically the form in which we present them, are informative about the effectiveness of partial evaluation even in the absence of an explicit solution of the equations.

**Definition 104** *Let $\mathcal{R}$ be a program, $S$ a finite set of operation-rooted terms, and $\rho$ an independent renaming for $S$. Let $\mathcal{R}'$ be a partial evaluation of $S$ in $\mathcal{R}$ (under $\rho$) computed from the finite (possibly incomplete) needed narrowing trees $\mathcal{N}_1, \ldots, \mathcal{N}_n$, according to Definition 16. We produce a pair of equations*

$$C(\sigma(s)) \;=\; C(t) + k \quad / \quad C(\sigma(\rho(s))) \;=\; C(ren_\rho(t)) + k'$$

*for each needed narrowing derivation $s \leadsto_\sigma^+ t$ in $\mathcal{N}_1, \ldots, \mathcal{N}_n$.*[4] *Constants $k$ and $k'$ denote the observable cost of the considered derivation in the original and residual programs, respectively, i.e., $k = C(s \leadsto_\sigma^+ t)$ and $k' = C(\rho(s) \leadsto_\sigma ren_\rho(t))$.*

We observe that if $C = \mathcal{S}$, then $k \geq 1$ and $k' = 1$, i.e., the application of each resultant corresponds to a sequence of one or more rules in the original program.

**Example 36** Consider again operation app and the narrowing derivation:

$$\begin{array}{rll} s = \texttt{app (app x y) z} & \leadsto_{\{\texttt{x} \mapsto \texttt{x}' : \texttt{x}_\texttt{s}\}} & \texttt{app (x}' : \texttt{app x}_\texttt{s} \texttt{ y) z} \\ & \leadsto_{id} & \texttt{x}' : \texttt{app (app x}_\texttt{s} \texttt{ y) z} = t \end{array}$$

[4] Note that there is no risk of ambiguity in using the same symbol, $C$, for both equations, since the signatures of $\mathcal{R}$ and $\mathcal{R}'$ are disjoint by definition of PE.

Then, we produce the associated resultant rule:

$$\underbrace{\texttt{dapp (x}' : \texttt{x}_\texttt{s}\texttt{) y z}}_{\sigma(\rho(s))} \quad = \quad \underbrace{\texttt{x}' : \texttt{dapp x}_\texttt{s}\texttt{ y z}}_{ren_\rho(t)}$$

with $\sigma = \{\texttt{x} \mapsto \texttt{x}' : \texttt{x}_\texttt{s}\}$ and $\rho = \{\texttt{app (app x y) z} \mapsto \texttt{dapp x y z}\}$. According to Definition 104, we produce the following equations for the cost criteria of Section 9.2:

$$\begin{array}{rclcrcl}
\mathcal{S}(\sigma(\texttt{s})) & = & \mathcal{S}(\texttt{t}) + 2 & / & \mathcal{S}(\sigma(\rho(\texttt{s}))) & = & \mathcal{S}(\texttt{ren}_\rho(\texttt{t})) + 1 \\
\mathcal{A}(\sigma(\texttt{s})) & = & \mathcal{A}(\texttt{t}) + 4 & / & \mathcal{A}(\sigma(\rho(\texttt{s}))) & = & \mathcal{A}(\texttt{ren}_\rho(\texttt{t})) + 2 \\
\mathcal{P}(\sigma(\texttt{s})) & = & \mathcal{P}(\texttt{t}) + 2 & / & \mathcal{P}(\sigma(\rho(\texttt{s}))) & = & \mathcal{P}(\texttt{ren}_\rho(\texttt{t})) + 1
\end{array}$$

which represent the cost of performing the above narrowing derivation in the original / residual program. Note that some useful conclusions about the improvement achieved by this residual rule can be easily inferred from the equations. For instance, we can see that all cost criteria have been halved.

The following two lemmata are auxiliary to demonstrate the *local* correctness of the derived recurrence equations. They hold for arbitrary needed narrowing derivations (and, in particular, for $\lambda$-derivations). The next result establishes a key property of needed narrowing derivations. This will become useful to compare the costs of two derivations.

**Lemma 105** *Let $\mathcal{R}$ be an inductively sequential program, and $t$ and $u$ be terms built over the signature of $\mathcal{R}$. If needed narrowing computes in $\mathcal{R}$ a derivation:*

$$t = t_0 \rightsquigarrow_{(p_1,R_1,\sigma_1)} t_1 \rightsquigarrow_{(p_2,R_2,\sigma_2)} \cdots \rightsquigarrow_{(p_n,R_n,\sigma_n)} t_n = u \tag{9.1}$$

*and $\sigma = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_n$, then needed narrowing also computes in $\mathcal{R}$ the derivation:*

$$\sigma(t) = t'_0 \rightsquigarrow_{(p_1,R_1,id)} t'_1 \rightsquigarrow_{(p_2,R_2,id)} \cdots \rightsquigarrow_{(p_n,R_n,id)} t'_n = u \tag{9.2}$$

*where, $t'_i = \sigma(t_i)$, for all $i \in \{0, \ldots, n\}$.*

*Proof.* The proof is by induction on $n$. The base case, i.e., $n = 0$, is trivial.[5] For the inductive step, we preliminary observe that if for some term, $v$, needed narrowing computes $v \rightsquigarrow_{(p,R,\eta)} v'$, then, for any substitution $\eta' \geqslant \eta$, needed narrowing computes $\eta'(v) \rightsquigarrow_{(p,R,id)} \eta'(v')$. The proof is by induction on the Noetherian ordering, $\prec$, on which the recursive definition of needed narrowing (Definition 5) is based. The inductive step is then a consequence of this result. $\square$

The following result establishes the stability of needed narrowing derivations computing the identity substitution.

[5]By convention, if $n = 0$, the expression $\sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_n$ is the identity substitution.

**Lemma 106 (stability of deterministic needed narrowing derivations)** *Let $\mathcal{R}$ be an inductively sequential program, $t$ and $s$ arbitrary terms, and $\sigma$ a constructor substitution. If needed narrowing computes in $\mathcal{R}$ a derivation:*

$$t \leadsto_{id}^{*} s$$

*then needed narrowing also computes in $\mathcal{R}$ the derivation:*

$$\sigma(t) \leadsto_{id}^{*} \sigma(s)$$

*where both derivations have the same number of steps and fire exactly the same rules at the same positions in corresponding steps.*

*Proof.* It is immediate from the stability of rewriting and the fact that needed narrowing and outermost-needed rewriting (as defined in [Antoy, 1992]) coincide when needed narrowing derivations compute the identity substitution. □

Now, we are ready to prove the *local* correctness of recurrence equations:

**Theorem 107 (local correctness)** *Let $\mathcal{R}$ be an inductively sequential program and $u$ a term such that $C(u) = n$ in $\mathcal{R}$. If there exists an equation $C(s) = C(t) + k$ (associated to a $\lambda$-derivation in $\mathcal{R}$) with $u = \theta(s)$, then $C(\theta(t)) = n - k$.*

*Proof.* Since $C(u) = n$ is defined, there exists a $\lambda$-derivation from $u$ to a constructor term $d$ computing the empty substitution:

$$D : u \leadsto_{id}^{*} d$$

whose associated cost is: $C(u) = C(D) = n$. By hypothesis, there exists a recurrence equation:

$$C(s) = C(t) + k$$

such that $u = \theta(s)$. By Definition 103, this equation has been obtained from a $\lambda$-derivation of the form:

$$s' \leadsto_{\sigma}^{+} t$$

such that $s = \sigma(s')$, and whose associated cost is $C(s' \leadsto_{\sigma}^{+} t) = k$. By Lemma 105, the following sequence of steps

$$\sigma(s') = s \leadsto_{id}^{+} t$$

is a $\lambda$-derivation, too. Since $u = \theta(s)$, by Lemma 106, we have the $\lambda$-derivation:

$$\theta(s) = u \leadsto_{id}^{+} \theta(t)$$

which has the same number of steps and fires exactly the same rules at the same positions in corresponding steps. Trivially, $C(u \leadsto_{id}^{+} \theta(t)) = C(s' \leadsto_{\sigma}^{+} t) = k$.

Finally, by Lemma 101, the $\lambda$-derivation $u \leadsto_{id}^{+} d$ is unique and, thus, derivation $u \leadsto_{id}^{+} \theta(t)$ must be a prefix of it. Therefore, it follows that $C(u \leadsto_{id}^{+} d) = C(\theta(t)) - k$. $\square$

In particular, the above result applies to the sets of equations constructed according to Definition 104. However, reasoning about recurrence equations considered above is not easy. The problem comes from the laziness of the computation model, since interesting cost criteria are not compositional for non-strict semantics [Sands, 1995]. In particular, the cost of evaluating an expression of the form $f(e)$ will depend on how much function $f$ needs argument $e$. For instance, given the definition of `app` and the following functions:

$$
\begin{array}{lcl}
\mathtt{h\ x\ y} & = & \mathtt{head\ (app\ x\ y)} \\
\mathtt{head\ (x : xs)} & = & \mathtt{x}
\end{array}
$$

we could generate a set of equations of the form:

$$
\begin{array}{lcl}
\mathcal{S}(\mathtt{h\ x\ y}) & = & \mathcal{S}(\mathtt{head\ (app\ x\ y)}) + 1 \\
\mathcal{S}(\mathtt{head\ (x : y)}) & = & 1 \\
\mathcal{S}(\mathtt{app\ []\ y}) & = & 1 \\
\mathcal{S}(\mathtt{app\ (x : xs)\ y}) & = & \mathcal{S}(\mathtt{app\ xs\ y}) + 1
\end{array}
$$

Now, in order to compute the number of steps for an arbitrary expression $\mathtt{h\ l_1\ l_2}$ (with $\mathtt{l_1}$ and $\mathtt{l_2}$ non-empty lists), we still need the knowledge about the considered narrowing strategy. By considering needed narrowing, $\mathcal{S}(\mathtt{h\,l_1\,l_2})$ is a constant number (i.e., 3). However, by considering an eager narrowing strategy, $\mathcal{S}(\mathtt{h\ l_1\ l_2})$ depends on the number of elements in $\mathtt{l_1}$. In eager (call-by-value) languages, the cost of $f(e)$ can be obtained by first computing the cost of evaluating $e$ to some normal form $d$, and then adding the cost of evaluating $f(d)$. Trivially, this procedure can be used to compute an *upper-bound* for $f(e)$ under a lazy semantics. Nevertheless, we have identified a class of recurrence equations which allows us to state a stronger result: the condition of *closed* recurrence equations.

Recall from previous chapters that the partial evaluation of a program with respect to $S$ is required to be $S$-closed, and only $S$-closed goals are considered. Roughly speaking, a term $t$ was considered $S$-closed iff each outer subterm $t'$ of $t$ headed by a defined function symbol is an instance of some term in $S$ and the terms in the matching substitution are recursively $S$-closed. For recurrence equations, we have required a stronger notion of closedness since only terms whose maximal operation-rooted subterms are constructor instances of some terms in $S$ are considered closed (this corresponds to the restricted notion of closedness $closed^-$ of Chapter 4).

**Definition 108 (closed set of recurrence equations)** *Let $\mathcal{R}$ be a program and $S = \{s_1, \ldots, s_n\}$ be a finite set of operation-rooted terms. Let $\mathcal{N}_1, \ldots, \mathcal{N}_n$ be finite (possibly incomplete) needed narrowing trees for $s_i$ in $\mathcal{R}$, $i = 1, \ldots, n$. Let $E$ be the set of recurrence equations associated to the narrowing derivations in $\mathcal{N}_1, \ldots, \mathcal{N}_n$. We say that $E$ is $S$-closed iff for each equation in $E$ of the form:*

$$C(s) = C(t) + k$$

*$t$ is either a constructor term or a constructor instance of some term in $S$.*

Note that producing closed partial evaluations (i.e., residual programs whose right-hand sides are closed w.r.t. the set of calls used to compute the partial evaluation, according to Definition 13) is not sufficient to guarantee that the corresponding recurrence equations are closed. Indeed, both requirements coincide only when a "plain" notion of closedness is considered for partial evaluations, where a term $t$ is $S$-closed iff $t$ is an instance of some term in $S$ by a *constructor* substitution (this was called "basic closedness" in [Alpuente *et al.*, 1998b]). This is the case, for instance, of partial deduction [Lloyd and Shepherdson, 1991] or positive supercompilation (when the *perfect* "$\alpha$-identical" closedness test of Sørensen *et al.* [1996] is used during PE).

The relevance of closed recurrence equations stems from their use to *compute* the cost of a term:

**Theorem 109 (cost computation)** *Let $\mathcal{R}$ be an inductively sequential program and $S = \{s_1, \ldots, s_k\}$ a finite set of operation-rooted terms. Let $\mathcal{N}_1, \ldots, \mathcal{N}_k$ be finite (possibly incomplete) needed narrowing trees (using $\lambda$-derivations) for $s_i$ in $\mathcal{R}$, $i = 1, \ldots, k$. Let $E$ be a set of $S$-closed recurrence equations associated to the $\lambda$-derivations in $\mathcal{N}_1, \ldots, \mathcal{N}_k$. If $t$ is a constructor instance of some term in $S$ and $C(t) = n$, then there is a rewrite sequence $C(t) \to^* n$ using the (oriented) equations in $E$ and the definition of "+".*

*Proof.* Since $C(t) = n$ and $t$ is operation-rooted, we know that there exists a non-null $\lambda$-derivation

$$A : t \leadsto^+_{id} d$$

in $\mathcal{R}$ such that $d$ is a constructor term and $C(t \leadsto^+_{id} d) = n$. Since $t$ is a constructor instance of some term, say $s_i$ in $S$ ($1 \leq i \leq k$), i.e., $t = \theta(s_i)$ for some constructor substitution $\theta$, we know (following a similar reasoning as in the proof of Theorem 107) that there exists a recurrence equation in $E$ of the form

$$C(\sigma(s_i)) = C(t') + k$$

inferred from a $\lambda$-derivation $s_i \leadsto^+_\sigma t'$ in $\mathcal{N}_i$, such that

$$B : \theta(s_i) \leadsto^+_{id} \theta(t')$$

is a prefix of $A$. By Theorem 107, $C(\theta(t')) = n - k$.

We prove the claim by induction on the number of steps, $m$, of this derivation. The case $m = 0$ is impossible since $s_i$, and hence $t$, are operation-rooted and $t$ is narrowable to a constructor term.

**Base case:** $m = 1$. If $m = 1$, then $t \leadsto_{id} d$ in $\mathcal{R}$ and consequently $t' = \theta(d)$, hence $t'$ is a constructor term, $C(t') = 0$, and $n = k$. Hence, $C(\sigma(s_i)) = n$ and the claim holds.

**Ind. step:** $m > 1$. In this case, derivation $A$ has the form $t \leadsto^+_{id} \theta(t') \leadsto^+_{id} d$. Since $t'$ is $S$-closed, by the induction hypothesis, $C(\theta(t')) \to^+ n - k$. By Theorem 107, $C(t) = C(\theta(t')) + k$. Hence, $C(t) \to^+ n$ and the claim holds.

□

Roughly speaking, this result states that, by considering the set of recurrence equations as a rewrite system (implicitly oriented from left to right) and performing additions as usual, we have all the necessary information for computing the associated cost of a term (whenever the cost of this term is defined, i.e., it is narrowable to a constructor term with empty substitution). Therefore, if the cost of term $t$ is $C(t) = n$ in program $\mathcal{R}$, then there exists a rewrite sequence which *computes* this cost: $C(t) \to^* n$ in $E$.

In other words, this result shows that a finite set $E$ of recurrence equations captures the cost of an infinite set of derivations or terms, under appropriate conditions. For example, $E$ could be used to mechanically compute the cost of a term according to the considered cost criteria. Observe that the rewrite strategy for $E$ is irrelevant, since the sole defined functions in $E$ are $C$ and $+$ (note that any function defined in the original and residual programs plays now the role of a constructor), and $+$ is strict in both arguments.

Theorem 109 becomes useful when the sets of recurrence equations associated to a concrete partial evaluation are closed. In particular, given a partial evaluation $\mathcal{R}'$ of $S$ in $\mathcal{R}$ (under $\rho$), we compute a set of recurrence equations $E = E_{\mathcal{R}} \cup E_{\mathcal{R}'}$ (according to Definition 104), where $E_{\mathcal{R}}$ (resp. $E_{\mathcal{R}'}$) is the subset of recurrence equations associated to program $\mathcal{R}$ (resp. $\mathcal{R}'$). Then, Theorem 109 can be applied whenever $E_{\mathcal{R}}$ is $S$-closed (and, thus, $E_{\mathcal{R}'}$ is $\rho(S)$-closed). This is always ensured if one considers the perfect closedness test of Sørensen *et al.* [1996] or the plain notion of closedness of Alpuente *et al.* [1998b] (denoted by *closed*$^-$ throughout) during partial evaluation. However, this property is not guaranteed when stronger notions of closedness are considered during partial evaluation (e.g., the recursive closedness of Definition 13) or when some partitioning techniques are used (as in Chapter 6).

According to Definition 108, the equations of Example 36 are not closed due to calls like $\mathcal{S}(\texttt{x} : \texttt{app (app x}_\texttt{s}\ \texttt{y) z})$. However, this is not a drawback. We can simplify constructor-rooted calls using the following (straightforward) properties:

$$C(x) = 0, \text{ for all } x \in \mathcal{V} \tag{1}$$

$$C(t) = C(t_1) + \ldots + C(t_n), \text{ if } t = c(t_1, \ldots, t_n),\ c \in \mathcal{C},\ n \geq 0 \tag{2}$$

This amounts to say that the cost of reducing constructor constants and variables is zero. Furthermore, the cost of evaluating a constructor-rooted term is the sum of the costs of evaluating its arguments. In the following, we assume that recurrence equations are possibly simplified using the above properties.

### 9.3.2 Usefulness of Recurrence Equations

The preceding section presents a formal approach for assessing the effectiveness of partial evaluation. Here, we illustrate its usefulness by showing several possible applications of our recurrence equations.

#### Recurrence Equations over Natural Numbers.

Our recurrence equations provide a formal characterization of the computational cost of executing a program w.r.t. a class of goals (those which are constructor instances of the left-hand sides). Therefore, as a first approach to analyze the improvement achieved by a concrete partial evaluation of a set of calls, one can *solve* the recurrence equations associated to both the original and partially evaluated programs and determine the speedup. In general, it is hard to find explicit solutions of these equations. Nevertheless, we can use a *size* function that maps the arguments of an expression to natural numbers. In some cases, using this function one can transform a cost $C$ over terms into a cost $T$ on natural numbers. Basically, for each recurrence equation $C(\sigma(s)) = C(t) + k$, we define an equation $T_s(n_1, \ldots, n_i) = T_t(m_1, \ldots, m_j) + k$, where $n_1, \ldots, n_i$ is a sequence of natural numbers representing the sizes of arguments $\sigma(x_1), \ldots, \sigma(x_i)$, with $x_1, \ldots, x_i$ the distinct variables of $s$. Similarly, $m_1, \ldots, m_j$ denote the sizes of the different variables in $t$. Note that the subscript $s$ of $T_s$ is only a device to uniquely identify the recurrence equations associated to term $s$.

**Example 37** Consider the following recurrence equations defining $\mathcal{S}$:

$$\begin{array}{lcll}
\mathcal{S}(\sigma(\texttt{app (app x y) z})) & = & \mathcal{S}(\texttt{app y z}) + 1 & \text{with } \sigma = \{\texttt{x} \mapsto \texttt{[]}\} \\
\mathcal{S}(\sigma(\texttt{app (app x y) z})) & = & \mathcal{S}(\texttt{app (app x}_\texttt{s}\texttt{ y) z}) + 2 & \text{with } \sigma = \{\texttt{x} \mapsto \texttt{x}' : \texttt{x}_\texttt{s}\} \\
\mathcal{S}(\sigma(\texttt{app x y})) & = & 1 & \text{with } \sigma = \{\texttt{x} \mapsto \texttt{[]}\} \\
\mathcal{S}(\sigma(\texttt{app x y})) & = & \mathcal{S}(\texttt{app x}_\texttt{s}\texttt{ y}) + 1 & \text{with } \sigma = \{\texttt{x} \mapsto \texttt{x}' : \texttt{x}_\texttt{s}\}
\end{array}$$

We can transform them into the following standard recurrence equations over natural numbers:

$$
\begin{array}{lcll}
\mathtt{T}_1(0,\mathtt{n}_2,\mathtt{n}_3) & = & \mathtt{T}_2(\mathtt{n}_2,\mathtt{n}_3)+1 & \\
\mathtt{T}_1(\mathtt{n}_1,\mathtt{n}_2,\mathtt{n}_3) & = & \mathtt{T}_1(\mathtt{n}_1-1,\mathtt{n}_2,\mathtt{n}_3)+2 & \mathtt{n}_1>0 \\
\mathtt{T}_2(0,\mathtt{n}_3) & = & 1 & \\
\mathtt{T}_2(\mathtt{n}_2,\mathtt{n}_3) & = & \mathtt{T}_2(\mathtt{n}_2-1,\mathtt{n}_3)+1 & \mathtt{n}_2>0
\end{array}
$$

where $\mathtt{n}_1$, $\mathtt{n}_2$ and $\mathtt{n}_3$ denote the length of the corresponding lists $\sigma(\mathtt{x})$, $\sigma(\mathtt{y})$ and $\sigma(\mathtt{z})$, respectively. Here, $\mathtt{T}_1$ stands for $\mathtt{T}_{\mathtt{app\ (app\ (x\ y)\ z)}}$ and $\mathtt{T}_2$ for $\mathtt{T}_{\mathtt{app\ x\ y}}$. The *explicit solutions* of these equations are the functions:

$$
\begin{array}{ll}
\mathtt{T}_1(\mathtt{n}_1,\mathtt{n}_2,\mathtt{n}_3) & = 2\,\mathtt{n}_1+\mathtt{n}_2+2 \\
\mathtt{T}_2(\mathtt{n}_2,\mathtt{n}_3) & = \mathtt{n}_2+1
\end{array}
$$

Generalizing and formalizing a useful notion of size does not seem to ease the understanding or manipulation of recurrence equations, because one must carefully distinguish different occurrences of a same constructor. For example, if `x` is a list of lists, only the "top-level" occurrences of the constructors of list `x` affect to the length of the evaluation of `app x y`. The number of occurrences of constructors in an element of `x` is irrelevant. However, these additional occurrences should be counted in sizing an argument of operation `flatten` defined below:

$$
\begin{array}{lcl}
\mathtt{flatten\ []} & = & \mathtt{[]} \\
\mathtt{flatten\ ([]:y)} & = & \mathtt{flatten\ y} \\
\mathtt{flatten\ ((x_1:x_s):y)} & = & \mathtt{x_1:flatten\ (x_s:y)}
\end{array}
$$

Furthermore, a solution of arbitrary recurrence equations does not always exist. In particular, *non-linear* recurrence equations, which might arise in some cases, do not have a mathematical explicit solution (i.e., a solution in terms of some size-measure of the arguments).

Nevertheless, in the following we present some alternative approaches. Basically, we are interested in computing the improvement achieved by partial evaluation. Therefore, it may suffice to compute a speedup interval from the corresponding sets of equations, rather than their exact solutions.

**Bounds for the Effectiveness of Partial Evaluation.**

It is well-known that, in general, partial evaluation cannot accomplish *superlinear* speedup (see, e.g., [Amtoft, 1991; Andersen and Gomard, 1992] for traditional partial evaluation and [Sørensen, 1994] for positive supercompilation). This is also true in partial deduction if the same execution model is also used for performing computations during partial evaluation.[6] In our framework, where we use the same mechanism both for execution and for partial evaluation, it is obvious from Definition 104 that the recurrence equations associated to the original and residual programs have exactly the same structure, i.e., they are identical except for the renaming of terms and the associated costs. Hence, it is straightforward to conclude that narrowing-driven partial evaluation cannot achieve superlinear speedups, either.

An important observation is that the overall speedup of a program is determined by the speedups of loops, since sufficiently long runs will consume most of the execution time inside loops. In our scheme, loops are represented by recurrence equations. Since the recurrence equations associated to the original and the residual programs have the same structure, as justified in the above paragraph, they constitute by themselves a useful aid to the user for determining the speedup (or slowdown) associated to each loop. Moreover, this information is inexpensive to obtain since the recurrence equations can be generated from the same partial derivations used to produce residual rules. In principle, we can easily modify existing partial evaluators for functional logic programs (e.g., the Indy system of Appendix A or the partial evaluator described in Chapter 8) to provide rules decorated with the associated cost improvement. Each resultant rule can be augmented with a pair of integers, $(k, k')$, for each cost criterion. This pair describes a cost variation (according to Definition 104) of the resultant rule in the original and in the residual program. For instance, given the partial derivation of Example 36, we produce the (decorated) residual rule:

$$\mathtt{dapp}\ (\mathtt{x'} : \mathtt{x_s})\ \mathtt{y}\ \mathtt{z}\ \rightarrow\ \mathtt{x'} : \mathtt{dapp}\ \mathtt{x_s}\ \mathtt{y}\ \mathtt{z} \quad /*\ \{(2,1),(4,2),(2,1)\}\ */$$

From this information, we immediately see that all cost criteria have been improved and, moreover, we can quantify the improvement achieved.

However, as residual programs grow larger, it becomes more difficult to estimate the effectiveness of a partial evaluation from the decorated rules. In this case, it would be valuable to design an automatic speedup analysis tool to determine the improvement achieved by the whole program w.r.t. each cost criterion. For instance, we could define a simple speedup analysis along the lines of Andersen and Gomard [1992]. For this purpose, it suffices to consider a speedup interval $\langle l, u\rangle$ where $l$ and $u$ are the smallest and largest ratios $k'/k$ among all the computed recurrence equations, respectively.

[6] Of course, if one uses a different computational mechanism at partial evaluation time, superlinear speedup becomes possible. For instance, one can use call-by-name evaluation at partial evaluation time when specializing call-by-value functional languages, or a refined selection rule when specializing Prolog programs (see, e.g., the discussion in [Amtoft, 1991]).

## 9.4 Related Work

A considerable effort has been devoted to reason about the complexity of imperative programs (see, e.g., [Aho *et al.*, 1974]). However, relatively little attention has been paid to the development of methods for reasoning about the computational cost of declarative programs. For logic programs, Debray and Lin [1993] introduce a method for the (semi-)automatic analysis of the worst-case cost of a large class of logic programs, including nondeterminism and the generation of multiple solutions via backtracking. Regarding eager (call-by-value) functional programs, [Liu and Gomez, 1998] describes a general approach for time-bound analysis of programs. Essentially, it is based on the construction of time-bound functions which mimic the original functions to compute the associated cost of evaluations. In contrast to [Liu and Gomez, 1998], we note that our (cost) recurrence equations are tightly associated to the partial evaluation process and, thus, they allow us to assess the effectiveness achieved by the partial evaluation transformation. The techniques of Liu and Gomez [1998] cannot be easily adapted to lazy (call-by-name) functional languages since cost criteria are not usually compositional. On the other hand, Sands [1995] develops a theory of cost equivalence to reason about the cost of lazy functional programs. Essentially, Sands [1995] introduces a set of *time rules* extracted from a suitable operational semantics, together with some equivalence laws. The aim of this calculus is to reveal enough of the "algorithmic structure" of operationally opaque lazy functional programs to permit the use of more traditional techniques developed in the context of imperative programs.

None of the above references apply the introduced analyses to predict the improvement achieved by partial evaluation techniques. Indeed, we found very little work directed to the formal study of the cost variation due to partial evaluation techniques. For instance, [Nielson, 1988] introduces a type system to formally express when a partial evaluator is better than another, i.e., when a residual program is more *efficient* than another. Amtoft [1991] is aimed to the definition of a general framework to study the effects of several unfolding-based transformations over logic programs. The framework is applied to partial evaluation, but the considered measures are very simple (e.g., unification is not taken into account). A well-known technique appears in [Andersen and Gomard, 1992; Jones *et al.*, 1993] which introduces a simple *speedup analysis* to predict a relative interval of the speedup achieved by a partial evaluation. Finally, [Sørensen, 1994, Chapter 11] presents a theoretical study of the efficiency of residual programs by positive supercompilation. It proves that the number of steps is not improved by positive supercompilation alone (a post-unfolding phase is necessary). This would be also true in our context if resultants were constructed only from one-step derivations. However, as discussed in [Sørensen, 1994], this does not mean that the process is useless, since intermediate data structures are frequently eliminated.

All the cited references study the effectiveness of partial evaluation by taking into account only the number of steps in evaluations. Although this is an important measure, we think that other criteria should also be considered (as in traditional experimental profiling approaches [Sansom and Peyton-Jones, 1997]). The discussion in [Sørensen, 1994] (see above) as well as situations like that in Example 33 justify this claim.

## 9.5 Conclusions

To the best of our knowledge, this is the first attempt to formally measure the effectiveness of partial evaluation with cost criteria different from the number of evaluation steps. Our characterization of cost enables us to estimate the effectiveness of a partial evaluation in a precise framework. We also provide an automatic method to infer some recurrence equations which help us to reason about the improvement achieved. The combination of these contributions helps us to reconcile theoretical results and experimental measures. Although the introduced notions and techniques are specialized to NPE, they could be adapted to other related partial evaluation methods (e.g., the partial evaluation method presented in Chapter 7, positive supercompilation [Sørensen *et al.*, 1996] and partial deduction [Lloyd and Shepherdson, 1991]).

# Chapter 10

# Conclusions and Future Work

We conclude by summarizing the main contributions of this thesis and pointing out several directions for further research.

## 10.1 Conclusions

Narrowing-driven partial evaluation [Alpuente *et al.*, 1998b] is the first fully-automatic and correct approach to the partial evaluation of functional logic programs. The framework is parametric w.r.t. the narrowing strategy used to guide the specialization process. Several instances of the framework have been presented: [Alpuente *et al.*, 1997] (based on lazy narrowing), [Alpuente *et al.*, 1998b] (based on innermost narrowing) and [Alpuente *et al.*, 1999c] (based on needed narrowing). However, these instances of the narrowing-driven partial evaluation framework are not endowed to support the development of a useful partial evaluation tool for a modern multi-paradigm language such as Curry [Hanus, 2000b], Escher [Lloyd, 1995] or Toy [López-Fraguas and Sánchez-Hernández, 1999]. The problem is that there are still many features of these languages which are not considered in the general framework. For instance, it is essential to design special techniques to deal with residuation, conjunctive expressions, external functions, higher-order functions, constraints, local declarations, etc.

The most important contribution of this thesis is the development of methods and techniques which make feasible the definition of effective partial evaluators for realistic functional logic languages. In short, the main contributions of the thesis are:

1. A partial evaluation framework for residuating functional logic programs.

   Modern multi-paradigm declarative languages combine functional, logic and concurrent programming styles by unifying (needed) narrowing and residuation into a single model. The model usually provides evaluation annotations to specify the precise evaluation mechanism. Hence, we generalized the original narrowing-driven partial evaluation framework in order to deal with (inductively sequential) programs containing evaluation annotations for program functions. Correctness results were provided, including the equivalence between the original and specialized programs w.r.t. the floundering behaviour.

2. Improved control issues for partial evaluation.

   The partial evaluation method is based on an automatic unfolding algorithm which builds narrowing trees. The process usually involves two termination problems. The first problem —local termination— aims to keep the expansion of the narrowing trees finite. The global control concerns the termination of the recursive unfolding. Our contribution was the formulation of concrete control issues that effectively handle primitive function symbols in functional logic languages based on needed narrowing; namely, a dynamic unfolding rule and a novel abstraction operator which highly improve the specialization power of the method. These refinements have been experimentally evaluated using a needed narrowing strategy in the INDY system, a prototype implementation of a partial evaluator for functional logic languages.

3. Using an abstract representation for partial evaluation.

   We considered a maximally simplified abstract representation of programs and defined a non-standard semantics for these programs which is specially tailored for being used at partial evaluation time. The intermediate representation is essential to represent the different features of the language at the appropriate level of abstraction. This allowed us to define a simple and concise partial evaluation method for modern functional logic languages. Thanks to the simplicity of the intermediate language, self-application becomes feasible.

4. A realistic partial evaluator for the Curry language.

   Following the partial evaluation approach based on the use of an abstract representation, we defined a partial evaluator to perform automatic program transformation of Curry programs. First, we considered the intermediate language, FlatCurry, to which Curry programs can be automatically translated. Then, we extended the control issues to deal with all the features of FlatCurry programs and developed the first purely declarative partial evaluator for a realistic multi-paradigm language.

5. A formal approach to measuring the effectiveness of partial evaluation.

   The effectiveness of partial evaluation tools is also crucial for our aims. Therefore, we have developed a formal approach to measuring the effectiveness of the partial evaluation process. To this purpose, we have defined some formal criteria to measure the efficiency of functional logic computations, which are independent of the concrete implementation of the language and compiler. Moreover, we expressed the cost of each criterion by means of recurrence equations over algebraic data types, which can be automatically inferred from the partial evaluation process itself.

## 10.2 Future Work

There are several interesting directions for continuing the research presented in this thesis. Let us briefly enumerate some of them:

1. Mixed On-line/Off-line Partial Evaluation

   In the literature on program specialization, control problems have been tackled from two different approaches. *On-line* partial evaluators perform all the control decisions *during* the actual specialization phase, whereas *off-line* specializers perform a pre-processing phase based on a *binding-time analysis* that generates annotations on the program *before* the actual specialization begins [Jones *et al.*, 1993]. Off-line methods can be faster than on-line methods, and they are easier to stop [Consel and Danvy, 1993]. For these reasons, in order to achieve *effective* self-application, partial evaluation of functional programs has mainly stressed the off-line approach.

   We have introduced a novel approach which uses an abstract representation to perform partial evaluation of modern multi-paradigm declarative languages. Within the new framework, self-application is already theoretically possible. Nevertheless, in order to achieve *effective* self-application, a binding-time analysis able to guide the specialization process seems mandatory. During the pre-processing phase, a binding time analysis provides information which leads to a more efficient specialization, since the inferred information reduces the amount of decisions that the partial evaluator has to take on-line.

   Therefore, we believe that a challenging approach within our integrated paradigm is an hybrid partial evaluation method, i.e., a binding-time analysis embedded into our online partial evaluator such that we can make some reduce/residualize decisions off-line while leaving others to the specializer. Such analyses are usually able to determine that substantial parts of the specialization phase are indeed static, which makes self-application possible. This would unify advantages of both approaches.

2. Formal Properties of Partial Evaluation

   In this line of research, we are interested in establishing several formal properties about the improvements achieved by the narrowing-driven partial evaluation method. These properties will be given w.r.t. the simple measures defined in this thesis: number of evaluation steps, number of function applications, and pattern matching effort. In particular, we want to ascertain the conditions under which these measures cannot be degraded by partial evaluation.

   For instance, supercompilation does not generally reduce the number of evaluation steps. Therefore, in order to ensure improvement in efficiency, some other measures must be reduced. Finally, we are also interested in the usefulness of our results in the context of neighbouring partial evaluation methods.

3. Formal Profiling of Multi-Paradigm Programs

   The importance of profiling tools is associated to its ability to improve program's performance. The information gathered by the profiler may help to detect parts or features of a program that consume most machine resources.

   We intend to develop a source-level profiler for a lazy multi-paradigm language capable of measuring both execution time and space usage. We hope that the experience gained from using the profiler to monitoring real applications provides feedback valuable for improving our transformation techniques.

   An interesting starting point is to develop a formal model for cost attribution in the style of [Sansom and Peyton-Jones, 1997]. This will provide a formal framework in which to discuss the improvement achieved by our transformation techniques and prove certain properties without getting enmeshed in the details of particular implementations.

This thesis has been written in English, which is not the author's native language. The author apologizes for any mistakes which may be due to this circumstance.

# Bibliography


Aho A., Hopcroft J., and Ullman J., 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

Aït-Kaci H., 1990. An Overview of LIFE. In J. Schmidt and A. Stogny, editors, *Proc. Workshop on Next Generation Information Systems Technology*, pages 42–58. Springer LNCS 504.

Aït-Kaci H., Lincoln P., and Nasr R., 1987. Le Fun: Logic, equations, and Functions. In *Proc. of Fourth IEEE Int'l Symp. on Logic Programming*, pages 17–23. IEEE, New York.

Albert E., Alpuente M., Falaschi M., Julián P., and Vidal G., 1998a. Improving Control in Functional Logic Program Specialization. In *Proc. of the Int'l Static Analysis Symposium, SAS'98*, pages 262–277. Springer LNCS 1503.

Albert E., Alpuente M., Falaschi M., Julián P., and Vidal G., 1998b. Polygenetic Partial Evaluation of Lazy Functional Logic Programs. In *Proc. of Appia-Gulp-ProDe, AGP'98*, pages 151–164.

Albert E., Alpuente M., Falaschi M., and Vidal G., 1998c. INDY User's Manual. Technical Report DSIC-II/12/98, UPV. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

Albert E., Alpuente M., Hanus M., and Vidal G., 1999a. A Partial Evaluation Framework for Curry Programs. In *Proc. of the 6th Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, pages 376–395. Springer LNAI 1705.

Albert E., Alpuente M., Hanus M., and Vidal G., 1999b. Partial Evaluation of Residuating Functional Logic Programs. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 376–395.

Albert E., Antoy S., and Vidal G., 2000a. A Formal Approach to Reasoning about the Effectiveness of Partial Evaluation. In *Proc. of 9th Int'l Workshop on Functional and Logic Programming, WFLP'2000.*

Albert E., Antoy S., and Vidal G., 2000b. Measuring the Effectiveness of Partial Evaluation. In *Proc. of 10th Int'l Workshop on Logic-based Program Synthesis and Transformation, LOPSTR'00*, pages 72–79.

Albert E., Antoy S., and Vidal G., 2001a. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Logic-based Program Synthesis and Transformation*. Springer LNCS.

Albert E., Ferri C., Steiner F., and Vidal G., 2000c. Improving Functional Logic Programs by Difference-Lists. In *Proc. of 6th Asian Computing Science Conference, ASIAN'00*, pages 238–255. Springer LNCS 1961.

Albert E., Ferri C., Steiner F., and Vidal G., 2000d. List-Processing Optimizations in a Multi-Paradigm Declarative Language. In *Proc. of 9th Int'l Workshop on Functional and Logic Programming, WFLP'2000*.

Albert E., Hanus M., and Vidal G., 2000e. Realistic Program Specialization in a Multi-Paradigm Language. In *Proc. of 9th Int'l Workshop on Functional and Logic Programming, WFLP'2000*.

Albert E., Hanus M., and Vidal G., 2000f. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of the 7th Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR'00)*, pages 381–398. Springer LNAI 1955.

Albert E., Hanus M., and Vidal G., 2001b. A Practical Partial Evaluator for a Multi-Paradigm Declarative Language. In *Proc. of Fifth International Symposium on Functional and Logic Programming, FLOPS'01*. Springer LNCS.

Alpuente M., Escobar S., and Lucas S., 1999a. UPV-Curry: an Incremental Curry Interpreter. In J. Pavelka, G. Tel, and M. Bartosek, editors, *Proc. of 26th Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM'99*, volume 1725 of *Lecture Notes in Computer Science*, pages 327–335. Springer-Verlag, Berlin.

Alpuente M., Falaschi M., Julián P., and Vidal G., 1997. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, volume 32, 12 of *Sigplan Notices*, pages 151–162. ACM Press, New York.

Alpuente M., Falaschi M., Moreno G., and Vidal G., 1999b. A Transformation System for Lazy Functional Logic Programs. In A. Middeldorp and T. Sato, editors, *Proc. of the 4th Fuji Int'l Symp. on Functional and Logic Programming, FLOPS'99*, pages 147–162. Springer LNCS 1722.

Alpuente M., Falaschi M., Moreno G., and Vidal G., 2000. An Automatic Composition Algorithm for Functional Logic Programs. In V. Hlaváč, K.G. Jeffery, and J. Wiedermann, editors, *Sofsem 2000—Theory and Practice of Informatics*, volume 1963 of *LNCS*, pages 289–297. Springer-Verlag.

Alpuente M., Falaschi M., and Vidal G., 1996. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H.R. Nielson, editor, *Proc. of the 6th European Symp. on Programming, ESOP'96*, pages 45–61. Springer LNCS 1058.

Alpuente M., Falaschi M., and Vidal G., 1998a. Experiments with the Call-by-Value Partial Evaluator. Technical Report DSIC-II/13/98, UPV. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

Alpuente M., Falaschi M., and Vidal G., 1998b. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844. Short version appeared in Alpuente *et al.* [1996].

Alpuente M., Hanus M., Lucas S., and Vidal G., 1999c. Specialization of Functional Logic Programs Based on Needed Narrowing. In P. Lee, editor, *Proc. of ICFP'99*, pages 273–283. ACM, New York.

Amtoft T., 1991. Properties of Unfolding-based Meta-level Systems. In *Proc. ACM Symp. on Partial Evaluation and Semantics-based Program Transformation, PEPM'91.*

Andersen L. and Gomard C., 1992. Speedup Analysis in Partial Evaluation: Preliminary Results. In *Proc. of PEPM'92*, pages 1–7. Yale University, New Haven, CT.

Antoy S., 1992. Definitional Trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632.

Antoy S., 1997. Optimal Non-Deterministic Functional Logic Computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, pages 16–30. Springer LNCS 1298.

Antoy S., 2000. Personal Communication. Portland, USA.

Antoy S., Echahed R., and Hanus M., 1997. Parallel Evaluation Strategies for Functional Logic Languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 138–152. MIT Press.

Antoy S., Echahed R., and Hanus M., 2000. A Needed Narrowing Strategy. In *Journal of the ACM*, volume 47(4), pages 776–822.

Antoy S. and Hanus M., 2000. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the 3rd International Workshop on Frontiers of Combining Systems (FroCoS 2000)*, pages 171–185. Springer LNCS 1794.

Baader F. and Nipkow T., 1998. *Term Rewriting and All That*. Cambridge University Press.

Bellegarde F., 1995. ASTRE: Towards a Fully Automated Program Transformation System. In J. Hsiang, editor, *Proc. of RTA'95*, pages 403–407. Springer LNCS 914.

Benkerimi K. and Hill P., 1993. Supporting Transformations for the Partial Evaluation of Logic Programs. *Journal of Logic and Computation*, 3(5):469–486.

Benkerimi K. and Lloyd J., 1990. A Partial Evaluation Procedure for Logic Programs. In S. Debray and M. Hermenegildo, editors, *Proc. of the 1990 North American Conf. on Logic Programming*, pages 343–358. The MIT Press, Cambridge, MA.

Bockmayr A., Krischer S., and Werner A., 1993. Narrowing Strategies for Arbitrary Canonical Rewrite Systems. Technical Report MPI-I-93-233, Max-Planck-Institut fuer Informatik, Saarbruecken.

Bol R., 1993. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46.

Bonacina M.P., 1988. Partial evaluation by completion. In G.I. et al., editor, *Conferenza dell'Associazione italiana per il calcolo automatico*.

Bondorf A., 1988. Towards a Self-Applicable Partial Evaluator for Term Rewriting Systems. In D. Bjørner, A. Ershov, and N. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 27–50. North-Holland, Amsterdam.

Bondorf A., 1989. A Self-Applicable Partial Evaluator for Term Rewriting Systems. In *Proc. of Int'l Conf. on Theory and Practice of Software Development, Barcelona, Spain*, pages 81–95. Springer LNCS 352.

Bossi A. and Cocco N., 1993. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87.

Boye J., 1993. Avoiding Dynamic Delays in Functional Logic Languages. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93*, pages 12–27. Springer LNCS 714.

Bruynooghe M., De Schreye D., and Martens B., 1992. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79.

Burstall R. and Darlington J., 1977. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67.

Chin W., 1993. Towards an Automated Tupling Strategy. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 119–132. ACM, New York.

Chin W., Goh A., and Khoo S., 1999. Effective Optimisation of Multiple Traversals in Lazy Languages. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA (Technical Report BRICS-NS-99-1)*, pages 119–130. University of Aarhus, DK.

Consel C. and Danvy O., 1989. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, 30:79–86.

Consel C. and Danvy O., 1993. Tutorial notes on Partial Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York.

Danvy O., Glück R., and Thiemann P., editors, 1996. *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany.* Springer LNCS 1110.

Darlington J., 1982. Program Transformation. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press.

Darlington J. and Guo Y., 1989. Narrowing and Unification in Functional Programming: an evaluation mechanism for absolute set abstraction. In *Proc. of the Conf. on Rewrite Techniques and Applications, RTA'89*, pages 92–108. Springer LNCS 355.

Darlington J. and Pull H., 1988. A Program Development Methodology Based on a Unified Approach to Execution and Transformation. In D. Bjørner, A. Ershov, and N. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 117–131. North-Holland, Amsterdam.

De Schreye D., Glück R., Jørgensen J., Leuschel M., Martens B., and Sørensen M., 1999. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277.

Debray S. and Lin N., 1993. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–975.

Dershowitz N. and Jouannaud J.P., 1990. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam.

Dershowitz N. and Jouannaud J.P., 1991. Notations for Rewriting. *Bulletin of the European Association of Theoretical Computer Science*, 43:162–172.

Dershowitz N. and Reddy U., 1993. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*, 15:467–494.

Echahed R., 1990. On completeness of narrowing strategies. *Theoretical Computer Science*, 72(2-3):133–146.

Etalle S., Gabbrielli M., and Marchiori E., 1997. A Transformation System for CLP with Dynamic Scheduling and CCP. In *Proc. of the ACM Sigplan PEPM'97*, pages 137–150. ACM Press, New York.

Fribourg L., 1985. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 172–185. IEEE, New York.

Futamura Y. and Nogi K., 1988. Generalized Partial Computation. In D. Bjørner, A. Ershov, and N. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, Amsterdam.

Gallagher J., 1993. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. ACM, New York.

Gallagher J. and Bruynooghe M., 1990. Some Low-Level Source Transformations for Logic Programs. In M. Bruynooghe, editor, *Proc. of 2nd Workshop on Meta-Programming in Logic*, pages 229–246. Department of Computer Science, KU Leuven, Belgium.

Gill A., Launchbury J., and Peyton Jones S., 1993. A Short Cut to Deforestation. In *Proc. of the Conf. on Functional Programming Languages and Computer Architecture*, pages 223–232. ACM Press, New York, NY, USA.

Giovannetti E., Levi G., Moiso C., and Palamidessi C., 1991. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377.

Glück R., Jørgensen J., Martens B., and Sørensen M., 1996. Controlling Conjunctive Partial Deduction of Definite Logic Programs. In *Proc. Int'l Symp. on Programming Languages: Implementations, Logics and Programs, PLILP'96*, pages 152–166. Springer LNCS 1140.

Glück R. and Sørensen M., 1994. Partial Deduction and Driving are Equivalent. In *Proc. of PLILP'94*, pages 165–181. Springer LNCS 844.

Glück R. and Sørensen M., 1996. A Roadmap to Metacomputation by Supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 137–160. Springer LNCS 1110.

Gurr C., 1994. *A Self-Applicable Partial Evaluator for the Logic Programming Language Goedel*. Ph.D. thesis, Department of Computer Science, University of Bristol.

Guttag J. and Horning J., 1978. The Algebraic Specification of Abstract Data Types. Technical report, Acta Informatica.

Hans W., Loogen R., and Winkler S., 1992. On the Interaction of Lazy Evaluation and Backtracking. In M. Bruynooghe and M. Wirsing, editors, *Proc. Int'l Symp. on Programming Languages: Implementations, Logics and Programs, PLILP'92*, pages 355–369. Springer-Verlag.

Hanus M., 1994. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628.

Hanus M., 1995a. Analysis of Residuating Logic Programs. *Journal of Logic Programming*, 24(3):161–199.

Hanus M., 1995b. On Extra Variables in (Equational) Logic Programming. In *Proc. of 20th Int'l Conf. on Logic Programming*, pages 665–678. The MIT Press, Cambridge, MA.

Hanus M., 1997a. Lazy Narrowing with Simplification. *Computer Languages*, 23(2–4):61–85.

Hanus M., 1997b. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93. ACM, New York.

Hanus M., 1999. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702.

Hanus M., 2000a. Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~curry`.

Hanus M., 2000b. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753.

Hanus M., 2001. Server Side Web Scripting in Curry. In *Proc. of the Third International Workshop on Practical Aspects of Declarative Languages (PADL'01)*. Springer LNCS.

Hanus M., Antoy S., Koj J., Sadre R., and Steiner F., 2000. PAKCS 1.2: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany.

Hanus M., Kuchen H., and Moreno-Navarro J., 1995. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107.

Hanus M., Lucas S., and Middeldorp A., 1998. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8.

Hanus M. and Prehofer C., 1999. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75.

Hanus M. and Sadre R., 1999. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, 1999(6).

Huet G. and Hullot J., 1982. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266.

Huet G. and Lévy J., 1992. Computations in Orthogonal Rewriting Systems, Part I + II. In J. Lassez and G. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443.

Hullot J., 1980. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87.

Ida T. and Nakahara K., 1997. Leftmost outside-in narrowing calculi. *Journal of Functional Programming*, 7(2):129–161.

Jones M. and Reid A., 1998. The Hugs 98 User Manual. Available at `http://haskell.cs.yale.edu/hugs/`.

Jones N., 1994. The Essence of Program Transformation by Partial Evaluation and Driving. In N. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language and Computation*, pages 206–224. Springer LNCS 792.

Jones N., Gomard C., and Sestoft P., 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ.

Julián P., 2000. *Especialización de Programas Lógico-Funcionales Perezosos*. Ph.D. thesis, DSIC-UPV. In spanish.

Klop J., 1992. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press.

Klop J. and Middeldorp A., 1991. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, pages 161–195.

Knuth D., Morris J., and Pratt V., 1977. Fast Pattern Matching in Strings. *SIAM Journal of Computation*, 6(2):323–350.

Komorowski H., 1982. 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 9th ACM Symp. on Principles of Programming Languages*, pages 255–267.

Lafave L. and Gallagher J., 1997. Partial Evaluation of Functional Logic Programs in Rewriting-based Languages. Technical Report CSTR-97-001, Department of Computer Science, University of Bristol, Bristol, England.

Lam J. and Kusalik A., 1991. A Comparative Analysis of Partial Deductors for Pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada. Revised April 1991.

Lassez J.L., Maher M.J., and Marriott K., 1988. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca.

Leuschel M., 1998. On the Power of Homeomorphic Embedding for Online Termination. In G. Levi, editor, *Proc. of the Int'l Static Analysis Symposium, SAS'98*, pages 230–245. Springer LNCS 1503.

Leuschel M., De Schreye D., and de Waal A., 1996. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In M. Maher, editor, *Proc. of JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA.

Leuschel M. and Martens B., 1996. Global Control for Partial Deduction through Characteristic Atoms and Global Trees. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 263–283. Springer LNCS 1110.

Leuschel M., Martens B., and De Schreye D., 1998. Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258.

Levi G. and Sirovich F., 1975. Proving Program Properties, Symbolic Evaluation and Logical Procedural Semantics. In *Proc. of MFCS'75*, pages 294–301. Springer LNCS 32.

Liu Y.A. and Gomez G., 1998. Automatic Accurate Time-Bound Analysis for High-Level Languages. In *Proc. of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 31–40. Springer LNCS 1474.

Lloyd J., 1994. Combining Functional and Logic Programming Languages. In *Proc. of the International Logic Programming Symposium*, pages 43–57. The MIT Press, Cambridge, MA.

Lloyd J., 1995. Declarative Programming in Escher. Technical Report CSTR-95-013, Computer Science Department, University of Bristol.

Lloyd J. and Shepherdson J., 1991. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242.

Loogen R., López-Fraguas F., and Rodríguez-Artalejo M., 1993. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714.

López-Fraguas F. and Sánchez-Hernández J., 1999. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631.

Lux W. and Kuchen H., 1999. An Efficient Abstract Machine for Curry. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 171–181.

Martens B. and Gallagher J., 1995. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. of ICLP'95*, pages 597–611. MIT Press.

Middeldorp A. and Okui S., 1998. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757.

Miniussi A. and Sherman D.J., 1996. Squeezing Intermediate Construction in Equational Programs. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 284–302. Springer LNCS 1110.

Moreno G., 2000. *Rules and Strategies for Transforming Functional Logic Programs.* Ph.D. thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia. In spanish.

Moreno-Navarro J. and Rodríguez-Artalejo M., 1992. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224.

Moreno-Navarro J.J., Kuchen H., Loogen R., and Rodriguez-Artalejo M., 1990. Lazy Narrowing in a Graph Machine. In *Proceedings of the 2nd International Conference on Algebraic and Logic Programming*, pages 298–317. Springer LNCS 463.

Naish L., 1991. Adding equations to NU-Prolog. In J. Maluszyński and M. Wirsing, editors, *Proc. of the 3rd Int'l Symp. on Programming Languages Implementation and Logic Programming*, pages 15–26. Springer LNCS 528.

Nemytykh A., Pinchuk V., and Turchin V., 1996. A Self-Applicable Supercompiler. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 322–337. Springer LNCS 1110.

Nielson F., 1988. A Formal Type System for Comparing Partial Evaluators. In D. Bjørner, A. Ershov, and N. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 349–384. North-Holland, Amsterdam.

O'Donnell M., 1977. *Computing in Systems Described by Equations.* Springer LNCS 58.

Padawitz P., 1988. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin.

Palamidessi C., 1990. Algebraic Properties of Idempotent Substitutions. In M. Paterson, editor, *Proc. of 17th Int'l Colloquium on Automata, Languages and Programming*, pages 386–399. Springer LNCS 443.

Pettorossi A., 1977. Transformation of Programs and Use of "Tupling Strategy". In *Informatica 77 Conference*, pages 1–6.

Pettorossi A. and Proietti M., 1994. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320.

Pettorossi A. and Proietti M., 1996a. A Comparative Revisitation of Some Program Transformation Techniques. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 355–385. Springer LNCS 1110.

Pettorossi A. and Proietti M., 1996b. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414.

Peyton-Jones S., 1996. Compiling Haskell by Program Transformation: a Report from the Trenches. In H.R. Nielson, editor, *Proc. of the 6th European Symp. on Programming, ESOP'96*, pages 18–44. Springer LNCS 1058.

Plasmeijer R. and van Eekelen M., 1993. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley.

Plotkin G., 1972. Building-in equational theories. *Machine Intelligence*, 7:73–90.

Reddy U., 1985. Narrowing as the Operational Semantics of Functional Languages. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, New York.

Robinson J., 2000. Computational Logic: Memories of the Past and Challenges for the Future. In J. Lloyd, editor, *Proc. of First Int'l Conference on Computational Logic, CL'2000*, pages 1–23. Springer LNCS.

Romanenko A., 1991. Inversion and Metacomputation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–22. Sigplan Notices, 26(9), ACM, New York.

Sands D., 1995. A Naive Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4):495–541.

Sansom P. and Peyton-Jones S., 1997. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385.

Scherlis W., 1981. Program Improvement by Internal Specialization. In *Proc. of 8th Annual ACM Symp. on Principles of Programming Languages*, pages 41–49. ACM Press.

Smolka G., 1995. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000.

Sørensen M., 1994. Turchin's Supercompiler Revisited: An Operational Theory of Positive Information Propagation. Technical Report 94/7, Master's Thesis, DIKU, University of Copenhagen, Denmark.

Sørensen M. and Glück R., 1995. An Algorithm of Generalization in Positive Supercompilation. In J. Lloyd, editor, *Proc. of ILPS'95*, pages 465–479. The MIT Press, Cambridge, MA.

Sørensen M., Glück R., and Jones N., 1994. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In D. Sannella, editor, *Proc. of the 5th European Symp. on Programming, ESOP'94*, pages 485–500. Springer LNCS 788.

Sørensen M., Glück R., and Jones N., 1996. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838.

Tamaki H. and Sato T., 1984. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of Second Int'l Conf. on Logic Programming*, pages 127–139.

Thiel J., 1984. Stop Losing Sleep over Incomplete Data Type Specifications. In *In 11th ACM Symposium on Principles of Programming Languages*, pages 76–82.

Turchin V., 1986a. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325.

Turchin V., 1986b. Program Transformation by Supercompilation. In H. Ganzinger and N. Jones, editors, *Programs as Data Objects, 1985*, pages 257–281. Springer LNCS 217.

Turchin V., 1988. The Algorithm of Generalization in the Supercompiler. In D. Bjørner, A. Ershov, and N. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, Amsterdam.

Turchin V., 1996. Metacomputation: Metasystem Transitions plus Supercompilation. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 481–509. Springer LNCS 1110.

Vidal G., 1996. *Semantics-Based Analysis and Transformation of Functional Logic Programs*. Ph.D. thesis, DSIC-UPV. In spanish.

Wadler P., 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248.

Warren D.H.D., 1982. Higher-Order Extensions to Prolog – Are they needed? In M. Hayes-Roth and Pao, editors, *Machine Intelligence*, volume 10. Ellis Horwood.

You J.H., 1991. Unification Modulo an Equality Theory for Equational Logic Programming. *Journal of Computer and System Sciences*, 42:54–75.

# Appendix A

# The Indy System

This appendix introduces the Indy system: an automatic partial evaluator for functional logic programs based on the NPE methodology. Indy (Integrated Narrowing-Driven specialization sYstem) is a rather concise implementation of the generic partial evaluation algorithm originally introduced in [Alpuente *et al.*, 1998b]; namely, it implements the different instances of the narrowing-driven partial evaluation algorithm which have been presented in [Alpuente *et al.*, 1998b] (based on innermost narrowing), [Alpuente *et al.*, 1997] (based on lazy narrowing), and [Alpuente *et al.*, 1999c] (based on needed narrowing), as well as the extension of [Albert *et al.*, 1999a] to deal with residuation (see Chapter 4). Additionally, the control improvements presented in Chapter 6 have been also incorporated in the system. See the Indy user's manual [Albert *et al.*, 1998c] for details.

## A.1 Introduction

The Indy system is structured following the general scheme presented in Chapter 5, where a clear distinction between local and global control is set. The algorithm starts by partially evaluating the initial set of terms, and then it recursively specializes the terms which are introduced dynamically during this process. As remarked in previous chapters, appropriate unfolding and abstraction (generalization) operators are needed to ensure that infinite unfolding is not attempted (local termination) and to ensure that no infinite attempts for specializing calls are carried out (global termination). Moreover, the inclusion of primitive function symbols (and, in particular, the conjunctive operator) in input calls poses specific control problems that require an special treatment in order to achieve a good specialization (see Chapter 6). In particular, the dynamic unfolding rule of Definition 62 and the refined abstraction operator of Definition 65 are implemented in the Indy system in order to care for the appropriate handling of primitive function symbols. Using the terminology of Glück and Sørensen [1996], INDY inherits from the NPE framework the ability to produce both polyvariant and polygenetic specializations, i.e., it can produce different specializations for the same function definition, and can also combine distinct original function definitions into a comprehensive specialized function. The INDY system is written in SICStus Prolog and is publicly available from:

`http://www.dsic.upv.es/users/elp/soft.html`

The implementation consists of about 800 clauses (4000 lines of code). The partial evaluator is expressed by 95 clauses, the metainterpreter by 415 clauses (including the code needed to handle the *ground representation*), the parser and other utilities by 90 clauses, the user interface by 100 clauses, and the postprocessing renaming by 95 clauses. The implementation only considers unconditional programs. Conditional programs are treated by using the predefined functions `and`, `if_then_else`, and `case_of`, which are reduced by standard defining rules (see, e.g., [Moreno-Navarro and Rodríguez-Artalejo, 1992]). The current implementation allows the user to select either the RN calculus (cf. Section 4.2), or simpler *call-by-name* (lazy or needed narrowing) and *call-by-value* (innermost narrowing) evaluation strategies. Normalization of goals between narrowing steps is also provided for innermost and lazy narrowing[1] (using a terminating subset of program rules).

The specializer also performs the post-processing renaming transformation of Definition 14, which is useful for automatically fulfilling the independence condition as well as for ensuring that the left-hand sides of the specialized rules are linear patterns. This ensures that different specializations for the same function definition are correctly distinguished, which is crucial for polyvariant specialization.

## A.2 Control Options

The INDY system needs no previous installation. It runs under SICStus Prolog v3.x and is entered by simply typing:

```
?- consult('indyv1.8').
```

Once the system is loaded, you can type `help` to display the complete range of commands of the system:

[1] Note that normalization steps do not optimize needed narrowing derivations since they have been proved optimal in [Antoy *et al.*, 2000].

```
>> help.

------------------------------------------------------------
load   -> Read in a file containing the source program
read   -> Read a source program from the standard input
list   -> Display the source program
nwing  -> Set the narrowing strategy
normal -> Switch normalization on/off
unfold -> Set the unfolding rule
eval   -> Change the current evaluation settings
split  -> Set the splitting option
mix    -> Start the specialization process
flags  -> Consult the current settings
help   -> See this info
cd     -> Change the working directory
sh     -> Start a new interactive default shell process
quit   -> Leave the system
------------------------------------------------------------
```

Commands must be followed by a period. These commands are described below.

### A.2.1 Loading Programs

Language syntax is very simple. Programs consist of a sequence of rules:

```
Term -> Term # Annotation
```

where annotation can be `rigid` or `flex`. The evaluation annotation is optional and indicates the precise evaluation mechanism for each function (`flex` functions are evaluated by narrowing while `rigid` functions are executed by residuation). The command `eval` allows us to change the evaluation annotation of functions. Also, the user can specify a sequence of rules of the form `Term --> Term` used for normalization. This distinction is useful when non-terminating programs are considered, since the (possibly empty) terminating subset of the program rules can be safely used for simplification [Fribourg, 1985; Hanus, 1997a,a].

The following grammar formalizes our language.[2] We use the following notations in grammar rules. Terminals are character sequences enclosed within apostrophes, `{...}` denotes a (possibly empty) repetition, `[...]` denotes an optional part, and `...|...` denotes a choice.

[2] To facilitate the integration of INDY in modern functional logic compilers, our environment incorporates a Curry to Indy translator. The development of a Toy to Indy translator is a subject of on-going work.

```
Program     := Rule ';' { Rule ';' } { RwRule ';' }
Rule        := Pattern '->' Term '#' Annotation
RwRule      := Pattern '-->' Term
TermList    := Term { ',' Term }
ConstrList  := ConstrTerm { ',' ConstrTerm }
Term        := Variable | FuncTerm
FuncTerm    := FuncName [ '(' TermList ')' ]
Pattern     := Defined [ '(' ConstrList ')' ]
ConstrTerm  := Constructor [ '(' ConstrList ')' ]
Annotation  := 'Flex' | 'Rigid'
FuncName    := Defined | Constructor
Defined     := Identifier
Constructor := ':' Identifier
Identifier  := Lowercase { Letter | Digit }
Variable    := Uppercase { Letter | Digit }
Letter      := Uppercase | Lowercase
Uppercase   := 'A' | 'B' | ... | 'Z'
Lowercase   := 'a' | 'b' | ... | 'z'
Digit       := '0' | '1' | ... | '9'
```

We note that a Prolog-like notation for lists is also allowed in programs. In order to load a source program, you can use one of these two commands:

- `load`: it is used to read a file which contains the source program and to put the program into the internal database. Filenames comply with the Prolog notation for terms, i.e., simple quotes are required when special characters appear in the filename (e.g., you should type `'append.fl'` rather than `append.fl`). The (terminating) subset of program rules to be used for normalization must be duplicated, with the symbol `->` replaced by `-->`.

- `read`: this option reads a source program from the standard input on-line. Let us illustrate the use of `read` on the well-known program `append`:

```
>> read.

Introduce program rules (quit to finish):

"lhs -> rhs;" : "append([],Ys) -> Ys;".
"lhs -> rhs;" : "append([X|Xs],Ys) -> [X|append(Xs,Ys)];".
"lhs -> rhs;" : quit.
```

```
Same rules for normalization? (yes/no): no.

Introduce rules for normalization (quit to finish):

"lhs --> rhs;" : "append([],Ys) --> Ys;".
"lhs --> rhs;" : "append([X|Xs],Ys) --> [X|append(Xs,Ys)];".
"lhs --> rhs;" : quit.

Program loaded!
```

Of course, the user can avoid re-typing the rules in this example by answering `yes` to the question asked on-line.

The program which is currently loaded into the system can be displayed by using the INDY command `list`.

### A.2.2 Operational Mechanism

The operational mechanism can be fixed by typing the command `nwing`. Currently, the user can select one of these options:

- `innermost narrowing`: it implements the innermost narrowing strategy of Fribourg [1985]; namely, only the (leftmost) innermost position of the goal is considered for the next narrowing step. This strategy requires programs to be constructor-based, completely-defined for completeness.
- `lazy narrowing`: it implements the lazy narrowing strategy of Moreno-Navarro and Rodríguez-Artalejo [1992]. At each step, only the outermost position is narrowed, unless further evaluation of inner subterms are *demanded* by some program rule. The completeness of this strategy is ensured, e.g., for constructor-based (weakly) orthogonal programs. Note that, since programs are not required to terminate, a terminating subset of the program rules has to be selected for normalization in order to guarantee the finiteness of the partial evaluation process.
- `needed narrowing`: it implements the needed narrowing strategy as defined in Section 2.4.2. This strategy only performs narrowing steps which are necessary for computing solutions. The completeness and optimality of this strategy is ensured for inductively sequential programs.
- `RN calculus`: it implements the combination of two different operational principles: (needed) narrowing and residuation, as described in Section 4.2. This strategy requires evaluation annotations on functions.

Innermost as well as lazy narrowing strategies can be speeded up by the use of normalization between narrowing steps. This setting can be fixed by using the command `normal`. The default setting for the narrowing strategy is: `needed narrowing`.

### A.2.3 Unfolding Rule

The unfolding rule determines how to build local narrowing trees and, what is of even greater significance, when to stop constructing them. The unfolding rule can be set up by typing the command `unfold`. Currently, there are three options:

- `emb_goal`: the nonembedding unfolding rule, which uses the homeomorphic embedding ordering (cf. Chapter 6) to stop the construction of the narrowing trees;
- `emb_redex`: this unfolding rule uses a dynamic selection strategy to determine the set of positions to be unfolded by a "don't care" selection within a conjunction. It implements the *dependency_clash* test of Definition 62 using homeomorphic embedding on comparable ancestors of selected redexes to ensure finiteness (note that it differs from `emb_goal` in that `emb_redex` implements dynamic scheduling on conjunctions and that homeomorphic embedding is checked on simple redexes rather than on whole goals);
- `comp_redex`: it differs from `emb_redex` in that `comp_redex` uses a simpler definition of *dependency_clash* based on comparable ancestors of selected redexes as a whistle;
- `depthk`: a depth-$k$ bound strategy, which expands derivations down to the "depth-$k$ frontier" of the tree.

The default unfolding rule is `emb_goal`. For instance, in order to select an `emb_redex` strategy, you should proceed as follows:

```
>> unfold.

Unfolding rule? (emb_goal/emb_redex/comp_redex/depthk): emb_redex.
```

### A.2.4 Splitting strategy

The abstraction operator corresponds to the operator $abstract_{\trianglelefteq}$ of Definition 65, which carefully considers primitive functions to provide for accurate polygenetic specialization. By setting the splitting strategy on, the partition of expressions containing selected primitive function symbols is enabled (in order to avoid unnecessary generalization).

The splitting strategy is not fixed by default. This option may be fixed by using the command `split`. For example, in order to permit the splitting of terms containing the primitive functions `and` (conjunction) and `eq` (equality), you can proceed as follows:

```
>> split.

List of primitive symbols? ["and","eq"].
```

### A.2.5 Specialization

The partial evaluation process is started by the command `mix`. The input data is a list containing the initial calls (terms) to specialize. For example, if we have loaded the program `append`, it can be specialized (using needed narrowing and the `emb_redex` unfolding rule) w.r.t. the term `append(append(X,Y),Z)` as follows:

```
>> mix.

List of expressions? ["append(append(X,Y),Z)"].

Final Set of Calls:
  append(RLN,YLN)
  append(append(X,Y),Z)

Independent Renaming:
  append(RLN,YLN) => append2_1(RLN,YLN)
  append(append(X,Y),Z) => append3_1(X,Y,Z)

Specialized program:

append2_1([],A) -> A;
append2_1([A|B],C) -> [A|append2_1(B,C)];
append3_1([],[],A) -> A;
append3_1([],[A|B],C) -> [A|append2_1(B,C)];
append3_1([A|B],C,D) -> [A|append3_1(B,C,D)];

Mix Time: 70 ms.

Save to a file? (yes/no) no.
```

The partial evaluator returns the final set of specialized calls[3], the renamed functions

[3]Note that the final set of calls can be different from the original one, since the abstraction operator can add or delete terms from the current set at each iteration of the algorithm.

for these calls, and the partially evaluated program (which can be saved into a file).

This specialized program is able to concatenate three lists by traversing the first input list only once, whereas the source program passes twice over this list.

### A.2.6 Utilities

The INDY system also has some utility commands:

- `cd`: it is used to change the working directory;
- `flags`: it displays the current settings (narrowing strategy and unfolding rule) of the system;
- `sh`: it starts an external shell.

The command `quit` is used to leave the system.

## A.3 An Example of Specialization

The following example is a kind of standard test in partial evaluation and similar transformation techniques: the specialization of a semi-naïve pattern matcher for a fixed pattern into an efficient algorithm (sometimes called "the KMP-test" [Sørensen *et al.*, 1996]). This example is particularly interesting because it provides a kind of optimization that neither (conventional) partial evaluation nor deforestation can achieve. We consider the specialization of the program `kmp` to the pattern `[a,a,b]` using needed narrowing and the `comp_redex` unfolding rule:

```
>> load.

Filename? 'Examples/kmp'.

File loaded!

>> list.

--Program Rules--

  match(P,S) -> loop(P,S,P,S);
  loop([],SS,OP,OS) -> :true;
  loop([P|PP],[],OP,OS) -> :false;
  loop([P|PP],[S|SS],OP,OS) -> if(eq(P,S),loop(PP,SS,OP,OS),next(OP,OS));
  next(OP,[]) -> :false;
```

```
  next(OP,[S|SS]) -> loop(OP,SS,OP,SS);
  if(:true,A,B) -> A;
  if(:false,A,B) -> B;
  eq(:a,:a) -> :true;
  eq(:b,:b) -> :true;
  eq(:a,:b) -> :false;
  eq(:b,:a) -> :false;


>> nwing.

Narrowing strategy? (lazy/needed/innermost): needed.


>> normal.

Normalization? (yes/no): yes.


>> unfold.

Unfolding rule? (emb_goal/emb_redex/comp_redex/depthk): comp_redex.

>> mix.

List of expressions? ["match([:a,:a,:b],S)"].


Final Set of Calls:
  loop([:b],SSZT,[:a,:a,:b],[:a,:a|SSZT])
  loop([:a,:a,:b],SSF,[:a,:a,:b],SSF)
  loop([:a,:b],SSF,[:a,:a,:b],[:a|SSF])
  match([:a,:a,:b],S)

Independent Renaming:
  loop([:b],SSZT,[:a,:a,:b],[:a,:a|SSZT]) => loop1_1(SSZT)
  loop([:a,:a,:b],SSF,[:a,:a,:b],SSF) => loop1_2(SSF)
  loop([:a,:b],SSF,[:a,:a,:b],[:a|SSF]) => loop1_3(SSF)
  match([:a,:a,:b],S) => match1_1(S)
```

```
Specialized program:

loop1_1([]) -> :false;
loop1_1([:b|A]) -> :true;
loop1_1([:a|A]) -> loop1_1(A);
loop1_2([]) -> :false;
loop1_2([:a|A]) -> loop1_3(A);
loop1_2([:b|A]) -> loop1_2(A);
loop1_3([]) -> :false;
loop1_3([:a|A]) -> loop1_1(A);
loop1_3([:b|A]) -> loop1_2(A);
match1_1([]) -> :false;
match1_1([:a|A]) -> loop1_3(A);
match1_1([:b|A]) -> loop1_2(A);

Mix Time: 1100 ms.
```

This program never backs up on the subject string when a mismatch is detected: whenever three consecutive `a`'s are found, the algorithm looks for one `b`, instead of attempting to match the whole pattern `[a,a,b]` from scratch. This is essentially a KMP-style pattern matcher [Knuth *et al.*, 1977].