Strategies and analysis techniques for functional program optimization

Santiago Escobar Román
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

UNIVERSIDAD
POLITECNICA
DE VALENCIA

A thesis submitted for the degree of:

Doctor in Computer Science

Advisors:

María Alpuente Frasnedo
Salvador Lucas Alba

Committee:

Isidro Ramos Salavert	**U.P. Valencia, Spain**
José Meseguer	**Urbana-Champaign, USA**
Hélène Kirchner	**LORIA INRIA-Lorraine, France**
Francisco López Fraguas	**U.C. Madrid, Spain**
Albert Rubio	**U.P. Cataluña, Spain**

October 31th, 2003

Estrategias y técnicas de análisis para la optimización de programas funcionales

Santiago Escobar Román
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

UNIVERSIDAD POLITECNICA DE VALENCIA

Memoria presentada para optar al título de:

Doctor en Informática

Directores:

María Alpuente Frasnedo
Salvador Lucas Alba

Tribunal de lectura:

Presidente:	**Isidro Ramos Salavert**	**U.P. Valencia, Spain**
Vocales:	**José Meseguer**	**Urbana-Champaign, USA**
	Hélène Kirchner	**LORIA INRIA-Lorraine, France**
	Francisco López Fraguas	**U.C. Madrid, Spain**
Secretario:	**Albert Rubio**	**U.P. Cataluña, Spain**

31 de Octubre de 2003

Acknowledgements

I would like to express my gratitude to the people who helped in the achievement of this PhD thesis.

First of all I would like to thank my advisors María Alpuente and Salvador Lucas for their dedication to the development of this work. I am grateful to María for her unconditional trust in me since I joined the "Extensions of Logic Programming Group" (ELP) that she leads. I admire her enthusiastic and excellent way of doing research as well as her ability to have told me so many times what I really needed to listen. This thesis would have never been possible without her collaboration and experience. I am also grateful to Salva for his rigorous way of looking at scientific problems and for his patience to show me how to do research. He helped me to acquire self-confidence, to add rigor to my thoughts, and to do not take assumptions for granted. They have been a reference point since I started doing research.

Thanks to my colleagues in the ELP group. Cèsar Ferri for his patience and help during this joint time that we have dedicated to research and teaching. Germán Vidal for his friendly interest in my research work and for many discussions during the coffee time. I am also grateful to María José Ramírez, José Hernàndez, and Alicia Villanueva for many interesting discussions and for encouraging me during the difficult moments. I would also like to thank all the people who have been part of the ELP research group during these approximately four years for their unconditional support and for providing a friendly atmosphere.

Part of the research work included in this PhD thesis has been done during several research stays in different countries. I thank Michael Hanus and Frank Steiner for their hospitality during my stays in Aachen and Kiel, as well as the members of the "Lehrstuhl für Informatik II" at "RWTH Aachen Universität" and the "Institut für Informatik" of the "Christian-Albrechts-Universität zu Kiel". Thanks also to Frank Huch and Bernd Brassel for their friendship during my last stay in Kiel. I am grateful to Rachid Echahed, Jérémie Blanc, Wendelin Serwe, and Frederic Prost for their valuable advice and amiability during my stay in Grenoble, as well as the members of the "Laboratoire Leibniz" at the "Institut IMAG" in Grenoble. I am also grateful to Bernhard Gramlich for his collaboration and amiability during my stay in Wien, as well as the members of the "Institut für Computersprachen" of the "Technische Universität Wien". And I am also grateful to Moreno Falaschi and Marco Comini for their friendly support and cooperation during my stay in Udine, as well as the members of the "Dipartimento di Matematica e Informatica" of the "Università di Udine".

I would like to thank the members of the "Grupo de Programación Lógica e Ingeniería del Software", founded by Isidro Ramos, and to those colleagues, technicians, and administrative staff of the "Departamento de Sistemas Informáticos y Computación". Especially to Antonio Molina, Patricio Letelier, and Emilio Insfrán who introduced me in teaching Software Engineering.

Moreover, I received some economical support from different institutions: "Departamento de Sistemas Informáticos y Computación", "Universidad Politécnica de Valencia", "Generalitat Valenciana", "Ministerio de Ciencia y Tecnología", "Universidad Politécnica de Madrid", and "ACM Sigplan".

Finally, thanks to my parents and my brother, who always helped me in so many different ways to achieve this PhD thesis. And thanks also to my friends, who encouraged me and showed me the strength to continue doing research.

Santiago Escobar
Valencia, October 2003

Abstract

Computer systems play an important role in the modern information society. However, the low quality of software and its low level of abstraction, inhibit the necessary confidence of final users and system developers in software engineering. Correctness of computer programs by a mathematical theory of computation is the fundamental concern of the theory of programming and of its application in large-scale software engineering. Formal methods provide software engineering with the suitable scientific and technological framework to become a real engineering, as predictable as civil or electrical engineering are. Indeed, the use of declarative rule-based programming languages during all program development stages ensures that correct and certified formal methodologies are followed during the whole software production process.

Programs in declarative rule-based programming languages are usually described as *term rewriting systems*. Program execution consists of reducing (or *rewriting*) *input terms* to *output terms* by applying a sequence of rules. *Narrowing* is an extension of rewriting for term rewriting systems which permits the instantiation of variables in input terms in order to enable rewriting steps on the instantiated expression. Rewriting (as well as narrowing) is generally undecidable, i.e. it is not possible to determine if a term rewrites (narrows) to another one. The reduction space associated to a given input term is huge due to the different possibilities for selecting subterms to reduce and the rules applicable to those subterms. This situation is even worse in the case of narrowing due to the instantiation process. In fact, the reduction (or narrowing) space usually contains useless (no interesting output expression is achievable), dangerous (no interesting output expression is guaranteed), and inefficient reduction sequences (an alternative more efficient sequence exists which delivers an equivalent outcome).

This thesis faces the problem of how to define efficient methods to improve the computational behavior of term rewriting systems, i.e. to shrink the reduction space associated to an input term by selecting which subterms and/or rules should be used for rewriting (or narrowing). We consider the following different approaches to optimize programs:

1. By dynamically selecting the allowed reductions in execution time due to the

definition of appropriate reduction or narrowing strategies. We refine this case into two parts:

(a) Whether the allowed reductions are automatically calculated from the program, i.e. the appropriate reduction strategy is inferred from the program. Here, we study the definition of optimal rewriting and narrowing strategies. Roughly speaking, we provide an incremental definition of the *outermost-needed narrowing* strategy (the best narrowing strategy for term rewriting systems) and we improve it, obtaining the *natural narrowing* strategy.

(b) Whether the allowed reductions are provided by assertions of the programmer included into the program, i.e. the appropriate reduction strategy is induced by the programmer. Here, we study different aspects of the inclusion of syntactic strategy annotations into term rewriting systems. First, we reformulate (and improve) the computational model associated in the literature to *on-demand strategy annotations* by handling demandedness in a different way. Also, we provide two program transformations: one for solving the incompleteness problem w.r.t. the absence of some annotations; and the other to encode on-demand strategy annotations into standard annotations in order to introduce a flavour of laziness into languages which only consider standard strategy annotations.

2. By statically analyzing the program in compilation time in order to discard (and remove) some parts. We use program analysis and transformation techniques. Here, we study a different semantics-based problem which is orthogonal to the definition of reduction or narrowing strategies. Roughly speaking, we analyze and remove the irrelevant data appearing in the program in the form of redundant arguments of functions, which produce inefficient reduction steps which cannot be avoided by a reduction strategy.

On the other hand, the proposed techniques are semantically correct, i.e. they are strong enough to avoid undesired sequences but not too much restrictive in order to preserve the sequences of interest. All these research lines lie on the common intuitive idea of optimizing term rewriting systems at the most simple but flexible and powerful level: symbol arguments.

Parts of this work have appeared in [Alpuente *et al.*, 1999a,b,c,d, 2000, 2002a,b,c,d,e,f, 2003a,b; Escobar, 2003a,b].

Resumen

Los sistemas informáticos desempeñan un papel importante en la moderna sociedad de la información. Sin embargo, la baja calidad del software y su bajo nivel de abstracción inhiben la necesaria confianza de los usuarios finales y desarrolladores de sistemas en la ingeniería de la programación. La corrección de programas informáticos a través de una teoría matemática de la computación es el primordial interés de la teoría de la programación y de su aplicación a la ingeniería de la programación a gran escala. Los métodos formales proveen a la ingeniería de la programación del adecuado marco científico y tecnológico para convertirse en una ingeniería real, tan predecible como son la ingeniería de caminos, canales y puertos o la ingeniería industrial. De hecho, el uso durante todas las etapas del desarrollo de programas, de lenguajes de programación declarativos basados en reglas, asegura la utilización de metodologías formales correctas y certificadas a lo largo del proceso de producción de programas.

Los programas se describen normalmente como *sistemas de reescritura de términos* en los lenguajes de programación declarativos basados en reglas. La ejecución de un programa consiste en reducir (o *reescribir*) *términos de entrada* en *términos de salida* con la aplicación de una secuencia de reglas. *El estrechamiento* es una extensión de la reescritura para sistemas de reescritura de términos que permite la instanciación de variables en los términos de entrada con el objetivo de activar pasos de reescritura en la expresión instanciada. La reescritura (así como el estrechamiento) son en general indecidibles, es decir no se puede determinar si un término se reescribe (o estrecha) a otro. El espacio de reducción asociado a un término de entrada es enorme debido a las diferentes posibilidades de selección de los subtérminos a reducir y las reglas aplicables a dichos subtérminos. Esta situación se agrava en el caso del estrechamiento debido al proceso de instanciación. De hecho, el espacio de reducción (o estrechamiento) normalmente contiene secuencias de reducción inútiles (no se puede alcanzar ninguna expresión de salida interesante), peligrosas (no se asegura ninguna expresión de salida interesante), e ineficientes (existe una secuencia alternativa más eficiente que proporciona una salida equivalente).

Esta tesis afronta el problema de cómo definir métodos eficientes para mejorar el comportamiento computacional de los sistemas de reescritura de términos, es decir,

restringir el espacio de reducción asociado a un término de entrada seleccionando qué términos y/o reglas serán usados por la reescritura (o el estrechamiento). Consideramos las siguientes distintas aproximaciones para optimizar los programas:

1. Seleccionando dinámicamente las reduciones permitidas en tiempo de ejecución a través de la definición de estrategias de reducción o estrechamiento apropiadas. Este caso lo refinamos en las siguientes dos partes:

   (a) Cuando las reducciones permitidas son calculadas automáticamente a partir del programa, es decir, la estrategia de reducción apropiada se infiere del programa. Aquí estudiamos la definición de estrategias óptimas de reescritura y estrechamiento. En concreto, proporcionamos una definición incremental del *estrechamiento necesario más externo* (la mejor estrategia de estrechamiento para sistemas de reescritura de términos) y lo mejoramos, obteniendo la estrategia de *estrechamiento natural*.

   (b) Cuando las reducciones permitidas son proporcionadas por afirmaciones del programador incluidas en el programa, es decir, la estrategia de reducción apropiada se induce por el programador. Aquí estudiamos los diferentes aspectos de la inclusión de anotaciones sintácticas de estrategias en sistemas de reescritura de términos. Primero, reformulamos (y mejoramos) el modelo computacional asociado en la literatura a las *anotaciones de estrategia bajo demanda* manejando la noción de demanda de una forma distinta. Además, proporcionamos dos transformaciones de programas: una para solventar los problemas de incompletitud asociados a la ausencia de anotaciones; y otra para codificar las anotaciones de estrategia bajo demanda como anotaciones estándar con el objetivo de incluir cierta pereza en los lenguajes que sólo consideran anotaciones de estrategia estándar.

2. Analizando estáticamente el programa en tiempo de compilación para descartar (y eliminar) ciertas partes. Utilizamos técnicas de análisis y transformación de programas. Aquí estudiamos un problema diferente basado en la semántica que es ortogonal a la definición de estrategias de reducción o de estrechamiento. En concreto, analizamos y eliminamos la información irrelevante incluida en el programa en forma de argumentos de función redundantes, que puede producir pasos de reducción ineficientes imposibles de evitar por una estrategia de reducción.

Por otra parte, las técnicas propuestas son semánticamente correctas, es decir, son lo suficientemente potentes como para evitar las secuencias no deseadas pero no demasiado restrictivas con el objetivo de preservar aquellas secuencias de interés. Todas estas líneas de investigación yacen sobre la idea común e intuitiva de optimizar los sistemas de reescritura de términos al nivel más simple pero flexible y potente: los argumentos de los símbolos.

Partes de esta tesis han aparecido en [Alpuente *et al.*, 1999a,b,c,d, 2000, 2002a,b,c,d,e,f, 2003a,b; Escobar, 2003a,b].

Resum

Els sistemes informàtics exerceixen un paper important en la moderna societat de la informació. Però, la baixa qualitat del software i el seu baix nivell d'abstracció inhibeixen la necessària confiança dels usuaris finals i els desenvolupadors de sistemes en la ingenieria del programari. La correcció de programes informàtics a través d'una teoria matemàtica de la computació és el primordial interés de la teoria de la programació i de la seua aplicació a la ingenieria del programari a gran escala. Els mètodes formals proveeixen la ingenieria del programari de l'adequat marc científic i tecnològic per convertir-se en una ingenieria real, tan predicible com són la ingenieria de camins, canals i ports o la ingenieria industrial. De fet, l'ús durant totes les etapes del desenvolupament de programes, de llenguatges de programació declaratius basats en regles, assegura que durant tot el procés de producció de programes s'utilitzen metodologies formals correctes i certificades.

El programes es describeixen normalment com *sistemes de reescriptura de termes* als llenguatges de programació declaratius basats en regles. L'execució d'un programa consisteix en reduir (o *reescriure*) *termes d'entrada* a *termes d'eixida* amb l'aplicació d'una seqüència de regles. *L'estrenyiment* de termes és una extensió de la reescriptura per a sistemes de reescriptura de termes que permet la instanciació de variables als termes d'entrada amb l'objectiu d'activar passes de reescriptura a l'expressió instanciada. La reescriptura (aixi com l'estrenyiment de termes) són en general indecidibles, és a dir, no es pot determinar si un terme es pot reescriure (o estrenyir) a un altre. L'espai de reducció associat a un terme d'entrada és enorme a causa de les diferents possibilitats de selecció dels subtermes a reduir i les regles aplicables a eixos subtermes. Aquesta situació s'agreuja en el cas de l'estrenyiment de termes a causa del procés d'instanciació. De fet, l'espai de reducció (o estrenyiment) normalment conté seqüències de reducció inútils (no es pot aconseguir cap expressió d'eixida interessant), perilloses (no s'assegura cap expressió d'eixida interessant), e ineficients (existeix una altra seqüència més eficient que produeix una eixida equivalent).

Aquesta tesi afronta el problema de cóm definir metodes eficients per a millorar el comportament computacional dels sistemes de reescriptura de termes, és a dir, restringir l'espai de reducció asociat a un terme d'entrada seleccionant quins termes

i/o regles es poden usar a la reescriptura (o l'estrenyiment). Considerem les següents distintes aproximacions per a optimitzar els programes:

1. Seleccionant dinàmicament les reduccions permeses en temps d'execució mitjançant la definició d'estratègies de reducció o estrenyiment apropiades. Aquest cas el refinem en les següents dues parts:

   (a) Quan les reduccions permeses són calculades automàticament a partir del programa, és a dir, l'estrategia de reducció apropiada s'infereix del programa. Ací estudiem la definició d'estratègies òptimes de reescriptura i estrenyiment. Informalment, proporcionem una definició incremental de l'*estrenyiment necessari més extern* (la millor estrategia d'estrenyiment per a sistemes de reescriptura de termes) i el millorem, obtenint l'estratègia de *estrenyiment natural.*

   (b) Quan les reduccions permeses són proporcionades per afirmacions del programador incloses al programa, és a dir, l'estratègia de reducció apropiada és induïda pel programador. Ací estudiem els diferents aspectes de la inclusió d'anotacions sintàctiques d'estratègies a sistemes de reescriptura de termes. Primerament, reformulem (i millorem) el model computacional associat en la literatura a les *anotacions d'estratègia sota demanda* manejant la noció de demanda d'una forma distinta. A més, proporcionem dues transformacions de programes: una per a resoldre els problemes d'incompletitud associats a l'absència d'anotacions; i una altra per a codificar les anotacions d'estratègia sota demanda com anotacions estàndard amb l'objectiu d'incloure certa peresa als llenguatges que sols consideren anotacions d'estratègia estàndard.

2. Analitzant estàticament el programa en temps de compilació per a descartar (i eliminar) certes parts. Utilitzem tècniques d'anàlisi i transformació de programes. Ací estudiem un problema diferent basat en la semàntica que és ortogonal a la definició d'estratègies de reducció o d'estrenyiment. Informalment, analitzem i eliminem la informació irrellevant inclosa al programa en forma d'arguments de funció redundants, que poden produir passes de reducció ineficients impossibles d'evitar per una estratègia de reducció.

D'altra banda, les tècniques proposades són semànticament correctes, és a dir, són el suficientment potents com per a evitar les seqüències no desitjades però no massa restrictives amb l'objectiu de preservar aquelles seqüències d'interés. Totes aquestes línies d'investigació jauen sobre la idea comuna e intuitiva d'optimitzar els sistemes de reescriptura de termes al nivell més simple però flexible i potent: els arguments dels símbols.

Parts d'aquest treball han aparegut a [Alpuente *et al.*, 1999a,b,c,d, 2000, 2002a,b,c,d,e,f, 2003a,b; Escobar, 2003a,b].

Contents

Chapter 1

Introduction

1.1 Information Society and Software Technology in the 2000's

Computer systems play an important role in the modern information society. In this novel scenario, the growing avalanche of new possibilities fetched by hardware and communication improvements (e.g. Internet) for the day to day development of companies, institutions and families, dodges (and sometimes conceals) the popular fact that the software on which we rely almost all our necessities is highly unsafe and inefficient. This situation is clearly unacceptable in the area of safety-critical and mission-critical software, where fail is inadmissible and can provoke irremediable human or material loses, e.g. nuclear powerplant control, space rocket launching, air traffic control, medical instrumentation, phone nets, or electronic commerce. Beyond these obvious shocking examples, all computer users are commonly annoyed by bugs in mass market software, by recent well-known viruses which have been widely reported by the media (and have been estimated to cost billions), and by fear of cyber-terrorism which is nowadays widespread. The public will welcome their elimination or reduction.

The low quality of software and its low level of abstraction, inhibit the necessary confidence of final users and system developers in software production. For instance, viruses obtain entry by exploiting errors like buffer overflow, which could be caught quite easily by a simple automatic analysis tool. Trustworthy software is now recognized by major vendors as a primary long-term goal. However, most of the general public, and even many programmers, are unaware of the possibility that computers might check the correctness of their own programs. Indeed, correctness of computer programs by a mathematical theory of computation is the fundamental concern of the theory of programming and of its application in large-scale software engineering (see

[Brooks, 2003; Hoare, 2003; McCarthy, 2003] for recent references that support this idea).

Modern large-scale software systems involve such level of complexity that the development, exploitation, and management of these systems require the use of automated aid techniques and tools during the different software production stages to ensure appropriate levels of confidence. At present, the most widely accepted means of bearing trust software is by massive and expensive testing. Test oracles detect errors as close as possible to their place of occurrence. Availability of actually reliable techniques will encourage programmers to formulate assertions as specifications in advance of code, and many of them would be verifiable by automated or semi-automated semantics-based tools.

1.1.1 Formal Methods

Several experiences have demonstrated that specification or design errors are the most frequent in software production and the most expensive to fix and repair [Boehm, 1981; Fairley, 1985]. Moreover, manufactured software does not usually meet their initial specifications [McCarthy, 2003]. Software engineering is devoted to solve these problems by stressing the fact that the errors embedded in an application design or requirement specification should be detected at early stages of its development at the same time as such development should follow strengthener methodologies.

Formal methods decrease the number of errors in specifications thanks to the absence of imprecision and ambiguity and by avoiding incompleteness and inconsistency. These properties entitle the application of formal analysis techniques which are not available to more traditional methodologies, e.g. typical methods based on simulation or testing can only detect the existence of an error but not the absence of errors (which is provided only by formal methods). Nevertheless, program certification by mathematical proofs is an absolute scientific ideal, like purity of materials or accuracy of measurement, pursued for its own sake in the controlled environment of the research laboratory. Indeed, the practicing software engineer has to resign himself to work around the impurities and inaccuracies of the real world. However, the value of purity and accuracy (and even correctness) are often not appreciated until the scientist has shown that they are achievable.

Despite the continuous and permanent interest of the theoretical community during the last twenty-five years [Broy, 1999] in specification, analysis, verification, diagnosis, optimization and certification of software systems, formal methods have not been widely accepted at a practical level by the industry [Bicarregui, 1999; Bowen and Hinchey, 1995; Broy, 1999; Ehrig, 2001; Ehrig and Mahr, 2001a,b; Hall, 1990; Wordsworth, 1999]. This unpopularity was mainly due to the generalized belief [Hall, 1990; Bowen and Hinchey, 1995] among software designers, analysts, and programmers that formal methods were so unnecessarily complicated and burdened with mathematical logic formalisms that the time and effort investment does not pay off in practice. But the fact is that a large fraction of computer science research in industry and academia over the last 20 years has been wasted, because the projects have had too short a time scale and focussed in short term goals [McCarthy, 2003]. As a consequence, there is a lack of practical tools and techniques to support formal methods in specific domains [Bicarregui, 1999; Hoare, 2003]. The methods of research application into formal methods are well established, though they need to be scaled up to meet the needs of modern software construction. On the one hand, the results of pure research will have to be adapted, extended, combined and tested by application on a broad scale to legacy code expressed in legacy languages. On the other hand, commercial programming tool-sets are driven mainly by fashionable slogans and by the politics of standardisation. Their elegant pictorial representations often have no semantics attributed to them. Their designers are constrained by compatibility with legacy practices and code, and by lack of scientific understanding on the part of their customers (and themselves).

During last decade, a *lightweight* approach pursuing a formal methods application view has been adopted [Maiden and Sutcliffe, 1992]. This approximation considers less ambitious solutions which combine different methods and programming languages, oriented to specific domains and activities, and which do not require a mathematical background or experience in formal methods from the final users. Moreover, it seizes some good ideas of *eXtreme programming* [Beck, 1999] to adapt theories to software development team's requirements.

On the other hand, formal methods provide software engineering with the suitable scientific and technological framework to become a real engineering, creator and sustainer of the software technology that the recent information society is asking for. The challenge is to make software engineering as predictable a discipline as civil or electrical engineering are [Brooks, 1986, 1995, 2003].

Verifying Compiler

To make software engineering a predictable real engineering discipline, several long-term projects should reinforce the use of formal methods. A challenging long-term project has been recently retaken: "*The Verifying Compiler*" [Hoare, 2003]. A "verifying compiler" uses mathematical and logical reasoning to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations associated with the code of the program. The compiler works in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components.

The "verifying compiler" has a strong historical background. The idea of using assertions to check a large routine goes back to [Turing, 1949]. The idea of the computer checking the correctness of its own programs was put forward by [McCarthy, 1963]. The two ideas were brought together in the "verifying compiler" by [Floyd, 1967]. Early attempts to implement the idea were severely inhibited by the difficulty of proof support with the machines of that day. At that time, the ephemeral nature and limited market for software written by hardware manufacturers reduced the motivation for any major verification effort. Furthermore, the source code was usually written in assembler, and kept secret. The project was abandoned in the 1970s and since those days, further difficulties have arisen from the complexities of modern software practice, for example, concurrent programming, object orientation, inheritance, components, interoperability, etc. However, experience has been gained in specification and verification of moderately scaled systems. Advances in unifying theories of programming suggest that many aspects of correctness of concurrent and object-oriented programs can be expressed by assertions or annotations supplemented by automatic or machine-assisted tools. Many of the global program analyses that are needed to underpin correctness proofs for systems involving concurrency and pointer manipulation have now been developed for being used in optimizing compilers. And, even more important, market pressure for trustworthy software is now greater than ever before. These reasons justify the maturity and feasibility of the project nowadays; this is in concordance with the recent objective of software engineering being a real engineering discipline by the use and adaptation of formal methods to specific domains and activities in a transparent and seamless way to the final users.

1.1.2 Declarative rule-based Programming Languages

Formal methods can be applied at any software production stage (as certification or optimization tools). Even when the considered software system does not meet the initial requirement specification, or it is produced without any correctness or completeness guarantee with respect to the initial requirement specification, or the initial requirement specification does not provide for the expected characteristics of a good program, formal specification and verification techniques of properties can be used in a post-processing phase; e.g. techniques to extract correct abstract information about general properties of interest. Alike the "verifying compiler", all these collected informations can be simply deferred to the user (e.g. proof of program termination, error absence, etc) or they can be used to automatically optimize (reprogram) the system w.r.t. some criterion or property (e.g. computation efficiency, space consumption, etc).

On the other hand, formal methods can be applied to each software production stage (and not only as certification or optimization tools). That is, formal methods can be seamed into software production processes from the beginning, where directly executable specification languages are clearly interesting, since they provide a concise and clear expressive vehicle which avoids the ambiguity inherent to natural or pictorial specification languages. Directly executable specification languages permit specifications to be considered as programs which can be immediately tested. This mechanism provides a simple prototyping vehicle which allows the software engineer to obtain a first approximation of the results bore by its own specification (in the case it would be correctly implemented). Furthermore, the use of executable specification languages during all program development stages ensures that strengthener, correct, and certified formal methodologies are followed during the whole software production process.

For instance, declarative languages (such as OBJ [Goguen *et al.*, 2000] and Maude [Clavel *et al.*, 1996] –algebraic rule-based languages–, Haskell [Hudak *et al.*, 1992] and ML [Milner *et al.*, 1990, 1997] –functional languages–, Prolog [Nadathur and Miller, 1988] and Gödel [Hill and Lloyd, 1994] –logic languages–, or Curry [Hanus *et al.*, 1995], $\mathcal{TOY}$ [López-Fraguas and Sánchez-Hernández, 1999], and Escher [Lloyd, 1999] –multiparadigm languages–) are adequate to be used during the software production processes, especially in the definition of executable prototypying specifications [Plasmeijer and Eekelen, 1993; Hudak and Jones, 1993] where they compete with advantage against more popular languages such as C or Ada [Hudak and Jones, 1993] or the wandering semiformal notations which cannot be completely automated. The diversity of resources for the definition and structuration of data (polimorfism, algebraic data types, high order, infinite data structures, abstract data types) and the easiness to manipulate them through lazy or eager evaluation techniques constitute a valuable asset of functional languages as specification languages [Goguen *et al.*, 2000; Hudak, 2000] and also as programming languages [Bird, 1998; Rabhi and Lapalme, 1999; Thompson, 1999]. On the other hand, the ability to search different solutions, the possibility to play with animated partial data structures and the use of logic variables through the unification mechanism, which are typical of logic languages, offer a range of new expressive possibilities and computational features that complement the functional dimension [Apt, 1997; Doets, 1994; O'Donnell, 1995b]. The *integration* of these two classes of programming languages is, then, desirable since all these features would be available in a seamless and unique environment; this is the bet of multiparadigm languages, in particular functional logic languages [Hanus, 1995b; Reddy, 1995; Rodriguez-Artalejo, 1995].

These features qualify declarative (or multiparadigm) programming languages as powerful and appropriate supporter candidates of the "verifying compiler" project. For instance, classical imperative languages are not equipped with formal analysis mechanisms which can prove properties such as termination, no buffer overflow, or the no violation of array bounds [Métayer *et al.*, 1998]; properties whose absence in manufactured software systems is too frequently manifested (commonly without the user knowing that the origin comes from concealable application errors).

1.2 Term Rewriting

Term rewriting [Baader and Nipkow, 1998; Dershowitz and Plaisted, 2001; Klop, 1992; Ohlebusch, 2002; TeReSe, 2003] is the standard operational principle for the definition of functional [Field and Harrison, 1988; Hudak *et al.*, 1992; Milner *et al.*, 1990, 1997; Plasmeijer and Eekelen, 1993; Peyton-Jones, 1987; Reade, 1993], algebraic [Bidoit *et al.*, 1993; Clavel *et al.*, 1996; Ehrig and Mahr, 1985; Goguen *et al.*, 1978, 2000], equational [Hoffmann and O'Donnell, 1982; O'Donnell, 1985, 1987, 1995a, 1998], or multiparadigm programming languages [Hanus, 1995b; Reddy, 1995; Rodriguez-Artalejo, 1995]. It is also important as the deductive mechanism used in automated theorem proving within equational systems [Bachmair and Dershowitz, 1994; Bachmair and Ganzinger, 2001; Dershowitz and Plaisted, 2001; Jouannaud, 1994; Plaisted, 1994; Rusinowitch, 1987]. Programs are usually described as Term Rewriting Systems (TRSs) [Ohlebusch, 2002; TeReSe, 2003]. In this thesis, term rewriting systems are used as the suitable basis computational model for encoding programs.

A term rewriting system consists of equations which are regarded as directed rules; namely, the rule is only applied from left to right whereas an equation could be applied in both directions. A computation (reduction or rewrite) step involves the search of a subterm (called *redex*) within the term to be reduced which is exactly an instance of the left hand side of a program rule and results in the replacement of the redex by the corresponding instance of the right hand side of the rule (if such redex and rule exist). Program execution consists of reducing *input terms* to *output terms* by applying a sequence of rules. These *output terms* belong to a concrete *family* of canonical expressions which corresponds to a semantics associated by the user to the program. Different families of canonical expressions (or semantics) are usually used in the literature, e.g. normal forms for general term rewriting systems [TeReSe, 2003], head normal forms for algebraic programming languages [Goguen *et al.*, 2000], and values for functional programming languages [Milner *et al.*, 1997].

Narrowing is an extension of rewriting for term rewriting systems where pattern matching is substituted by unification in the redex search process (as in *logic programming*) [Fribourg, 1985; Hanus, 1997; Hölldobler, 1989; Hullot, 1980; Reddy, 1985]. The narrowing mechanism permits the instantiation of variables in input terms in order to enable reduction steps (by rewriting) on the instantiated expression. Narrowing was originally used as an automatic theorem prover resolution method in equational systems [Fay, 1979; Slagle, 1974] and it is now the basic computational model of (multiparadigm) functional logic languages [Hanus, 1994; Hanus *et al.*, 1995, 2003]. The narrowing mechanism is complete in the sense of logic programming (computation of *answers*) as well as in the sense of functional programming (computation of *values*).

Rewriting (as well as narrowing) is generally undecidable, i.e. it is not possible to determine if a term rewrites (narrows) to another one. In fact, since the reduction space associated to a given input term is huge due to the different possibilities for selecting the redexes and rules, it usually contains useless –no canonical expression is achievable–, dangerous –no canonical expression is guaranteed–, and inefficient reduction sequences –an alternative efficient sequence exists which delivers an equivalent outcome–. This situation is even worse in the case of narrowing due to the instantiation process. Therefore, it seems interesting to formalize techniques for avoiding these kinds of undesired reduction sequences, i.e. to shrink the reduction space associated to an input term by selecting which subterms and/or rules should be used for rewriting. This is achievable in term rewriting thanks to its formal mathematical basis.

Traditionally, the reduction space associated to an input term is shrunk either in execution time, by dynamically selecting the allowed reductions (which is used to break down the non-determinism inherent to program execution), or in compilation time, by statically analyzing the program in order to obtain some abstract information (which is used to discard some program parts). However, the proposed techniques must be semantically correct, i.e. they should be strong enough to avoid sequences considered undesired w.r.t. some criteria but not too much restrictive in order to preserve the sequences of interest w.r.t. the considered semantics (alike trustworthy correct software).

In this thesis, we consider two main approaches to shrink the computation space associated to term rewriting systems. First, we consider the idea of cutting off the reduction space (in execution time) by enforcing some reduction (and narrowing) sequences while avoiding others. This is done by means of reduction and narrowing strategies which hold good computational properties. Second, we consider the idea of transforming the term rewriting system (in compilation time) into a new one with a smaller (or simplified) reduction space. In this case, we use program analysis and transformation techniques to optimize the term rewriting system. In the following, we give some motivation for both techniques.

1.2.1 Reduction Strategies

When implementing term rewriting (as well as term narrowing), a function is used which takes an input term and returns the terms which stem from it, i.e. what is called a *reduction* (or *narrowing*) *strategy*. A reduction strategy takes the control of which subterms and rules are going to be considered for rewriting (a *narrowing* *strategy* also determines which instantiations are going to be considered). In general, rewriting/narrowing strategies are deterministic [Antoy and Middeldorp, 1996; Klop, 1992], i.e. given a term, it reduces to exactly one term. A rewriting/narrowing relation can be transformed in several ways (under some circumstances) to achieve a deterministic strategy. In the following, we concentrate on rewriting (or reduction) strategies rather than narrowing strategies, although similar concepts exists or can be established.

Most programming languages impose a *predefined* reduction strategy which guides program execution and determines the possible semantics associated to the program. There exist two traditional strategies to evaluate a function call: *eager evaluation* (also known as *strict*, *call-by-value*, or *innermost* evaluation) and *lazy evaluation* (also known as *non-strict*, *call-by-need*, or *outermost* evaluation). The former one forces the evaluation of all the arguments before evaluating the outer function call over the evaluated parameters. The latter one only evaluates those arguments whose evaluation is necessary in order to evaluate the outer function call. The selection of one of them determines the behavior and the properties of the reduction sequences. For instance, eager rewriting will not complete the evaluation of a function call if the evaluation of some of the arguments does not terminate, whereas lazy rewriting will always obtain the outcome of the function call if it exists, independently of the complete evaluation of the arguments. Furthermore, the time and space consumption of the two strategies differ considerably; lazy rewriting needs more resources than eager rewriting [Peyton-Jones, 1987]. Though lazy rewriting is generally better (e.g. w.r.t. program termination), a great difficulty is how to define concretely *what a needed argument is* and how to efficiently determine whether a redex is *needed*.

In general, the problem of performing only needed reduction sequences is not trivial nor decidable [Huet and Lévy, 1979, 1992; Middeldorp, 1997; Lucas, 1998c]. However, the definition of reduction strategies by suitably restricting the rewriting relation associated to a TRS is a common solution to avoid undesired reduction sequences and master the hugeness of the reduction space without losing completeness. On the one hand, several reduction strategies have been developed in term rewriting to help perform only needed reduction sequences and achieve a real lazy evaluation [Antoy, 1992; Antoy *et al.*, 2000; Comon, 1995; Durand and Middeldorp, 1997; Huet and Lévy, 1979, 1992; Jacquemard, 1996; Kennaway, 1989; Klop and Middeldorp, 1991; Loogen *et al.*, 1993; Moreno-Navarro and Rodríguez-Artalejo, 1992; O'Donnell, 1977, 1985; Oyamaguchi, 1993; Sekar and Ramakrishnan, 1993; Thatte, 1987; Toyama *et al.*, 1993]. On the other hand, several authors have considered the idea of interleaving the two evaluation strategies in order to achieve better computational properties w.r.t. some criteria, i.e. perform a number of eager evaluation steps on some arguments of the function call before using lazy rewriting to reduce it [Burn, 1991; Mycroft, 1980;

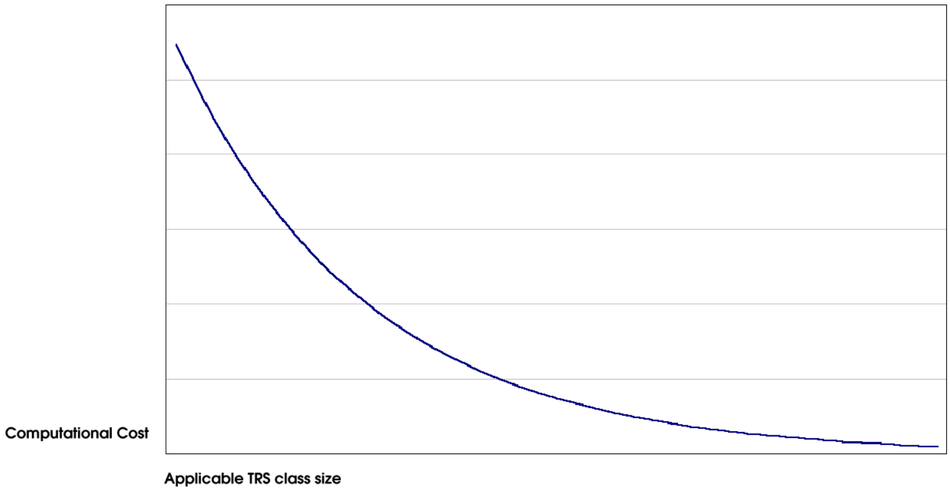

Figure 1.1: Comparison of properties of existing reduction strategies (i)

Mycroft and Norman, 1992; Sekar *et al.*, 1990].

All those possible strategies performing needed reduction sequences or combining lazy and eager rewriting have their merits and demerits. In recent years, strategies whose eager/lazy behavior can be fixed by the user have deserved great interest. A number of programming languages (e.g., ASF+SDF [van Deursen *et al.*, 1996], CafeOBJ [Futatsugi and Nakagawa, 1997], ELAN [Borovanský *et al.*, 1998], Maude [Clavel *et al.*, 1996], OBJ2 [Futatsugi *et al.*, 1985], OBJ3 [Goguen *et al.*, 2000], or Stratego [Visser, 1999]) permit (to some extent) the explicit specification of strategies aimed at controlling the execution of programs (see [Visser, 2001] for a recent survey). Other programming languages such as Clean [van Eekelen *et al.*, 1991], Curry [Hanus *et al.*, 1995], Gofer [Jones, 1992], Standard ML [Milner *et al.*, 1997], or Haskell [Hudak and Fasel, 1992] have a predefined computational strategy whose exact behavior can be modified (to some extent) by means of program annotations. For instance, lazy languages (e.g., Haskell, Clean) permit to introduce syntactic annotations (i.e., associated to symbol arguments) and interpret them as guidelines to become more eager and efficient.

As noticed by [Antoy, 2002], reduction strategies have always evolved to be more efficient (optimal or needed) but also to enlarge the class of TRSs to which they are applicable (though possibly sub-optimal). Antoy also pointed out that the more efficient a reduction strategy is, the less costly its calculus is, which is also an advantage. Figure 1.1 graphically illustrates these facts. However, researchers have not much worried about the cost associated to the applicability test of a reduction strategy when optimality is to be ensured. Indeed, the more “needed” a reduction

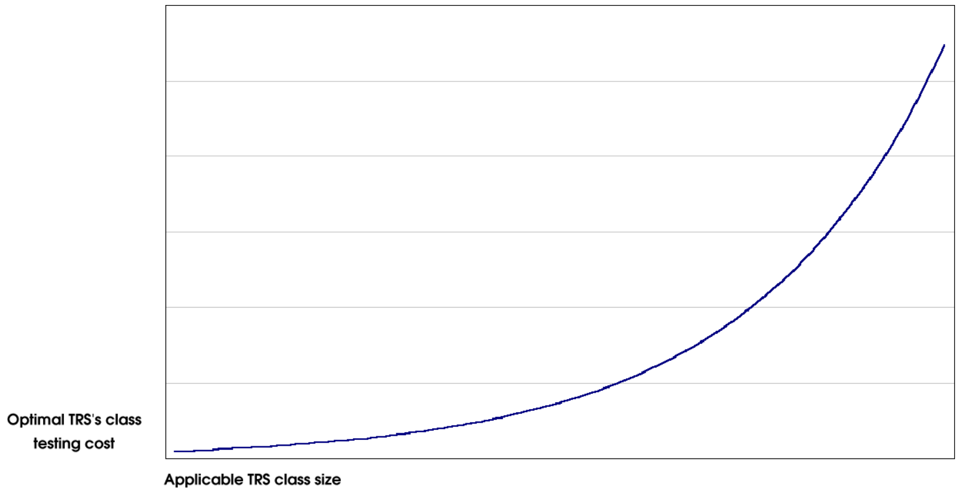

Figure 1.2: Comparison of properties of existing reduction strategies (ii)

strategy is, the more costly the optimal TRS class test is; or in other words, the larger the applicable TRS class is, the shorter the optimal TRS class is. Figure 1.2 graphically illustrates this other point. For instance, *needed rewriting* of [Huet and Lévy, 1992] was applicable to *orthogonal* TRSs and the calculus was very complex and difficult but the conditions to ensure optimality were simple (only orthogonality); *outermost-needed rewriting* (narrowing) of [Antoy *et al.*, 2000] is applicable to left-linear constructor systems, the calculus is simpler, but optimality is only ensured for the class of inductively sequential TRSs, which is a more difficult condition to test; and *context-sensitive rewriting* of [Lucas, 2002a] is also applicable to left-linear constructor systems, the calculus is even simpler, but optimality (and correctness) is only ensured for μ-terminating TRSs, which is a very difficult condition to test.

By using modern reduction strategies, suitable efficient strategies for a given program or specification class can be defined. However, tools and techniques to support such modern strategies are also important. These considerations motivate the first half of this thesis, where improved reduction strategies are considered as well as support tools and techniques. We address two main problems: 1) how to define effectively reduction strategies which consider only needed reduction sequences, and 2) how to deal with reduction strategies whose eager/lazy behavior can be programmed by the user.

1.2.2 Program Analysis

Program analysis techniques have been proved useful to statically approximate dynamic properties of programs in different programming paradigms such as logic, functional, or equational programming [Aho *et al.*, 1986; Cousot and Cousot, 1977, 1979, 1994; Cousot, 2002; Hughes, 1988]. Its applications include certification, optimization and transformation of programs, automated verification of properties, debugging, etc. The development and application of such techniques to term rewriting systems (and particularly to, functional or equational programming) has been fostered during the last decades. The optimization and transformation of programs is a way to improve the efficiency of term rewriting systems while keeping correctness. This also supports the "verifying compiler" [Hoare, 2003] within the "lightweight" application of formal methods to software engineering [Maiden and Sutcliffe, 1992].

Many analysis and program transformation techniques require some previous knowledge about the program. This information can be gained by static analysis of the program, i.e some analysis performed at compilation time. The analysis and transformation of programs consists in the automatic processing of the source text of the program in order to either check some property (analysis) or obtain a new program which behaves better w.r.t. some criteria, generally efficiency (transformation). In the last decades, several methodologies to approximate properties of interest have been developed: *flow-based analysis*, *semantics-based analysis* (or *abstract interpretation*), and *type-based analysis*. These techniques have been applied to different programming paradigms, in particular to rule-based languages and term rewriting systems.

As introduced in Section 1.2.1, a common method to reduce the computation space associated to a term rewriting system is to use reduction strategies which perform eager rewriting steps on some arguments of a function call before enabling lazy rewriting. Some of these reduction strategies usually rely on program analysis techniques to decide whether it is possible and safe to perform such eager rewriting steps (correctness). In the literature, this is called *strictness analysis* [Burn *et al.*, 1986; Burn, 1991; Jensen, 1991; Mycroft, 1980; Mycroft and Norman, 1992; Sekar *et al.*, 1990; Wadler and Hughes, 1987]. Strictness analysis ensures whether an argument evaluation is "strictly" necessary to lazy evaluation and then the argument is evaluated in an eager form. Strictness analysis is closely related to the notion of argument neededness (see Section 1.2.1). Strictness analysis has often been complemented by program analysis for detecting parts of a program which are clearly unneeded w.r.t. lazy evaluation. In the literature, this is called *dead code analysis* (also known as *unneededness*, *absence*, *filtering*, or *useless* analysis) [Aho *et al.*, 1986; Berardi *et al.*, 2000; Cousot and Cousot, 1994; Hughes, 1988; Kobayashi, 2000; Kennaway *et al.*, 1996; Kuper, 1994; Leuschel and Sørensen, 1996; Liu and Stoller, 2002; Pettorossi and Proietti, 1994; Wand and Siveroni, 1999]. In fact, unneeded terms w.r.t. a program are understood as *computationally irrelevant* terms, since their evaluation is not "needed" w.r.t. lazy evaluation and these terms can be safely erased in order to shrink the associated reduction space of the program.

Other similar notions have been widely used during last decades to detect and remove parts of a program which are computationally irrelevant. Examples are program specialization [Alpuente *et al.*, 1997, 1998, 1999e; Leuschel and Martens, 1995; Pettorossi and Proietti, 1994, 1996a], slicing [Gouranton, 1998; Schoenig and Ducasse, 1996; Reps and Turnidge, 1996; Szilagyi *et al.*, 2002; Tip, 1995; Weiser, 1984], and compile-time garbage collection [Jones and Métayer, 1989; Park and Goldberg, 1992; Knoop *et al.*, 1994].

However, traditionally, no method has considered what could be called *semantically irrelevant* terms, in contrast to computationally irrelevant terms. The notion of being semantically irrelevant (or redundant) means that replacing it by whatever expression we like, the final result does not change; independently of actual computations. This concept is clearly independent of the notions of dead code since a subterm can be needed (computationally relevant) though redundant (semantically irrelevant). Indeed, the reader should note that no reduction strategy can dodge the problem since the evaluation of a redundant subterm can be strictly necessary to achieve the canonical form of the input term w.r.t. the associated semantics; though such canonical form does not depend on the redundant subterm. In other words, *computationally irrelevant* terms (dead code) are always *semantically irrelevant* (redundant) whereas the opposite does not hold. This point justifies that there are examples where no possible best reduction strategy can be defined to improve the reduction space associated to a program whereas optimizing assertions are available.

In the second half of the thesis, we consider how to efficiently detect and remove *semantically irrelevant* terms in term rewriting systems, i.e. how to formulate program analysis techniques to detect such kind of *inefficiencies* and then apply program transformation techniques to eliminate them.

1.3 Objective and Contributions of the Thesis

This thesis is framed into the improvement of term rewriting computation as a model for the programs that can be written in more sophisticated programming languages. We face the problem of how to define efficient methods to safely improve the computational behavior of programs either by manipulating the rewriting calculus or by manipulating the term rewriting system itself. We provide four main contributions concerning *reduction strategies* and *program analysis and transformation*:

Incremental Needed Narrowing Ordinary narrowing can have a huge search space and great effort has been made to develop sophisticated narrowing strategies without losing completeness. In concrete, several lazy narrowing strategies have been designed (see e.g., [Antoy *et al.*, 2000; Loogen *et al.*, 1993; Moreno-Navarro and Rodríguez-Artalejo, 1992]). *Outermost-needed narrowing* [Antoy *et al.*, 2000] (also known simply as needed narrowing) is currently the best lazy narrowing strategy for functional logic programs due to the optimality properties w.r.t. the length of the derivations (functional view) and the number of computed solutions (logic view). Needed narrowing is based on the idea of a *definitional tree* which codifies the term rewriting system and the reduction strategy [Antoy, 1992]. A definitional tree can be defined for *inductively sequential* TRSs.

As briefly introduced in Section 1.2.1, term rewriting involving only needed reduction sequences is usually known as *call by need computation* and is related to the notions of *neededness* and *root-neededness* [Huet and Lévy, 1979, 1992; Middeldorp, 1997; Lucas, 1998c]. In [Huet and Lévy, 1979, 1992], Huet and Lévy provided a formal basis for the mechanization of call by need computations for the class of strongly sequential TRSs. They showed that the steady reduction of strongly needed redexes is normalizing, i.e. provides a correct normal form. Strandh provided an *incremental* algorithm which preserves the optimality and normalizing properties of Huet and Lévy's reduction of needed redexes for *forward branching* TRSs [Durand, 1994]. For constructor systems, inductive sequentiality and strong sequentiality coincide [Hanus *et al.*, 1998b]. Since it is known that constructor strongly sequential systems are forward branching [Durand and Salinier, 1993], it seems reasonable to consider the possibility to define an *incremental needed narrowing* which is able to efficiently conduct needed functional logic reductions.

We address this problem in the thesis and show how pre-eminent properties of reduction in (a distinguished subclass of) strongly sequential programs, namely the incrementality of the evaluation, can be inherited by needed narrowing. We give an incremental definition of needed narrowing and demonstrate that the original optimality properties are kept. Moreover, we experimentally demonstrate that the proposed incremental refinement can lead to substantial improvements in the overall evaluation process.

Natural Narrowing *Outermost-needed rewriting* [Antoy, 1992] is a sound and complete rewrite strategy for the class of inductively sequential CSs whereas *outermost-needed narrowing* [Antoy *et al.*, 2000] is the extension to narrowing. Its optimality properties and the fact that inductively sequential CSs are a subclass of strongly sequential programs explains why outermost-needed narrowing has become useful in functional logic programming as the functional logic counterpart of Huet and Lévy's strongly needed reduction. *Weakly outermost-needed rewriting* [Antoy *et al.*, 1997] is defined for non-inductively sequential CSs. The extension to narrowing is called *weakly outermost-needed narrowing* [Antoy *et al.*, 1997] and is considered as the functional logic counterpart of Sekar and Ramakrishnan's parallel needed reduction, which is the extension of Huet and Lévy needed rewriting for non-inductively sequential CSs (namely, almost orthogonal CSs).

However, some improvement is possible for (weakly) outermost-needed rewriting and narrowing since they do not work appropriately both on non-inductively sequential CSs and on failing input terms. In [Sekar and Ramakrishnan, 1993], Sekar and Ramakrishnan argued that strongly sequential systems have been widely used in programming languages and, unfortunately, violations are not frequent. However, this is a cumbersome restriction and its relaxation is quite useful in some contexts. It would be convenient for programmers that such a restriction could be relaxed while optimal evaluation is preserved for strongly sequential parts of a program. The availability of optimal and effective evaluation strategies without such a restriction could encourage programmers to formulate non-inductively sequential programs, which could still be executed in an optimal way.

Hence, we study here the improvement (or refinement) of outermost-needed rewriting and narrowing, and their weakly extensions, to provide better lazy rewriting and narrowing strategies for functional (logic) programs based on (non-inductively) sequential programs. We provide a refinement of the demandedness (understood as neededness) notion associated to outermost-needed rewriting and narrowing. Indeed, for the class of inductively sequential CSs, natural rewriting and narrowing behave even better than outermost-needed rewriting and narrowing in the avoidance of failing computations. And regarding inductively sequential CSs, we introduce a larger class of CSs called *inductively sequential preserving* where natural rewriting and narrowing preserve optimality for sequential parts of the program.

(On-demand) Strategy Annotations Strategies whose eager/lazy behavior can be provided by the user by means of syntactic strategy annotations have been developed recently [Lucas, 1998a,c, 2002a; Visser, 2001; Dolstra, 2001]. *Syntactic annotations* (generally specified as sequences of integers associated to function arguments, called *local strategies*) have been used in OBJ2 [Futatsugi *et al.*, 1985], OBJ3 [Goguen *et al.*, 2000], CafeOBJ [Futatsugi and Nakagawa, 1997], and Maude [Clavel *et al.*, 1996] to improve efficiency and (hopefully) avoid nontermination. Local strategies are used in OBJ programs for guiding the *evaluation strategy*: only those indices appearing in the local strategy are able to be reduced. Unfortunately, using rewriting restrictions may cause *incompleteness*, i.e., normal forms of input expressions could be unreachable by such restricted computation. The obvious problem is that the lack of some indices in the local strategies can have a negative impact in the ability of such strategies to compute normal forms. Some solutions to these problems naturally arise.

On the one hand, the absence of some indices w.r.t. constructor symbols (which form the canonical expressions) can be solved by transformation processes which add those indices necessary for canonicity while we are still able to preserve program termination. We provide a program transformation for solving incompleteness w.r.t. absence of indices while termination is preserved.

On the other hand, the absence of some indices w.r.t. function symbols can be solved by the rather intuitive notion of demanded evaluation of an argument of a function call by the left hand side of a rule [Antoy and Lucas, 2002]. In [Ogata and Futatsugi, 2000; Nakamura and Ogata, 2001], *on-demand* indices are proposed, which indicate those arguments that should be evaluated only 'on-demand'. The 'demand' is identified as an attempt to match an argument term with the left-hand side of a rewrite rule [Eker, 2000; Goguen *et al.*, 2000; Ogata and Futatsugi, 2000]. We recall in the thesis the current proposals for dealing with on-demand strategy annotations and discuss some drawbacks regarding the treatment of demandedness. We (re-)formulate the computational model of *on-demand strategy annotations* by handling demandedness in a different way. We demonstrate that the new on-demand strategy outperforms the original one. We also show (even experimentally) that our definition behaves better than other on-demand evaluation methods regarding the ability to compute (head-)normal forms. Furthermore, we provide a program transformation which can be used to formally prove termination of programs that use our computational model for implementing arbitrary strategy annotations.

Furthermore, we demonstrate how this new on-demand evaluation strategy can be made available to languages that do not provide evaluation with on-demand strategy annotations (e.g., Maude and OBJ3), thereby introducing a flavour of laziness into such languages. Our proposal is based on an automatic, semantics–preserving program transformation for OBJ programs which achieves correctness and completeness by producing a program without negative annotations which can be then correctly executed by typical OBJ interpreters.

Redundant Arguments of Functions The application of automatic transformation processes during the formal development and optimization of programs can introduce encumbrances in the generated code that programmers usually (or presumably) do not write [Aho *et al.*, 1986; Berardi *et al.*, 2000; Cousot and Cousot, 1994; Hughes, 1988; Kobayashi, 2000; Leuschel and Sørensen, 1996; Liu and Stoller, 2002; Pettorossi and Proietti, 1994; Wand and Siveroni, 1999]. Examples are *irrelevant data* in the shape of redundant arguments in the functions defined by the program or useless program rules. It seems interesting to formalize program analysis techniques for detecting these kinds of redundancies as well as some transformations for eliminating dead code which appears in the form of redundant function arguments or useless rules and which, in some cases, can be safely erased without jeopardizing correctness.

We study this problem in the last part of the thesis. We provide a semantic characterization of redundancy which is parametric w.r.t. the observed semantics. We consider different (reduction) semantics, including the normal forms semantics (typical of pure rewriting) and the evaluation semantics (closer to functional programming). We introduce the notion of redundancy of an argument w.r.t. a semantics and derive some decidability results. Inefficiencies caused by the redundancy of arguments cannot be avoided by using standard rewriting strategies. Therefore, we formalize an elimination procedure which gets rid of the useless arguments and provide sufficient conditions for the preservation of the semantics. We present some experiments that demonstrate that our approach is both practical and useful.

All these research lines lie on the common intuitive idea of optimizing term rewriting systems at the most simple but flexible and powerful level: symbol arguments.

Chapter 2

Preliminaries

Term rewriting systems provide an adequate computational model for functional (logic) languages which allow the definition of functions by means of patterns (e.g., Haskell, Hope or Miranda) [Baader and Nipkow, 1998; Dershowitz and Jouannaud, 1990; Klop, 1992; Plasmeijer and Eekelen, 1993; Ohlebusch, 2002; TeReSe, 2003]. In the remainder of this thesis we follow the standard framework of term rewriting for developing our results (see [Baader and Nipkow, 1998; Ohlebusch, 2002; TeReSe, 2003] for missing definitions). Definitions are mainly given in the one-sorted case all through the manuscript. The extension to many-sorted signatures is not difficult [Padawitz, 1988].

2.1 Binary Relations

Given a set A, $\mathcal{P}(A)$ denotes the set of all subsets of A. Let $R \subseteq A \times A$ be a binary relation on a set A. We denote the reflexive closure of R by $R^=$, its transitive closure by R^+, and its reflexive and transitive closure by R^*. An element $a \in A$ is an R-normal form, if there exists no b such that $a\ R\ b$. We say that b is an R-normal form of a (written $a\, R^!\, b$), if b is an R-normal form and $a\ R^* b$. We say that R is *confluent* if, for every $a, b, c \in A$, whenever aR^*b and aR^*c, there exists $d \in A$ such that bR^*d and cR^*d. We say that R is *terminating* (or *well-founded*) iff there is no infinite sequence $a_1\ R\ a_2\ R\ a_3\ \cdots$. In concrete, let $\rightarrow \subseteq A \times A$ be a binary relation on a set A. We denote the inverse of $\rightarrow$ by $\leftarrow$, the symmetric closure by $\leftrightarrow$, the transitive closure by $\rightarrow^+$, the reflexive and transitive closure by $\rightarrow^*$, the reflexive, symmetric and transitive closure by $\leftrightarrow^*$, and the normalizing relation by $\rightarrow^!$.

2.2 Terms, Variables and Substitutions

Throughout this thesis, $\mathcal{X}$ denotes a countable set of variables $\{x, y, \ldots\}$ and $\mathcal{F}$ denotes a finite set of function symbols $\{f, g, \ldots\}$, each one having a fixed arity given by a function $ar : \mathcal{F} \to \mathbb{N}$. If $ar(f) = 0$, we say that f is a constant symbol. By $\mathcal{T}(\mathcal{F}, \mathcal{X})$ we denote the set of terms; $\mathcal{T}(\mathcal{F})$ is the set of ground terms or Herbrand domain, i.e., terms without variable occurrences. A term is said to be linear if it has no multiple occurrences of a single variable. A k-tuple $t_1, \ldots, t_k$ of terms is written $\overline{t}$. The number k of elements of the tuple $\overline{t}$ will be clarified by the context. $\mathcal{V}ar(t)$ is the set of variables in t.

A *substitution* is a mapping $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ which homomorphically extends to a mapping $\sigma : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{T}(\mathcal{F}, \mathcal{X})$. We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ 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 any other variable x. We denote by id the "identity" substitution: $id(x) = x$ for all $x \in \mathcal{X}$. Let $Subst(\mathcal{T}(\mathcal{F}, \mathcal{X}))$ denote the set of substitutions and $Subst(\mathcal{T}(\mathcal{F}))$ be the set of ground substitutions, i.e., substitutions on $\mathcal{T}(\mathcal{F})$. If $\sigma(t)$ is a ground term, we call σ a *grounding substitution* for t. A *unifier* of two terms t, s is a substitution σ with $\sigma(t) = \sigma(s)$. A *most general unifier* (*mgu*) of t, s is a unifier σ such that for each unifier σ' of t, s there exists θ such that $\sigma' = \theta \circ \sigma$. Given a substitution θ and a set of variables $V \subseteq \mathcal{X}$, we denote by $\theta|_V$ (or $\theta_{\restriction V}$) the substitution obtained from θ by restricting its domain to V. We write $\theta = \sigma\ [V]$ if $\theta|_V = \sigma|_V$, and $\theta \leq \sigma\ [V]$ denotes the existence of a substitution γ such that $\gamma \circ \theta = \sigma\ [V]$.

Given terms t, s, we write $t \leq s$ if $\exists \sigma. s = \sigma(t)$ and say that t is a prefix of s (said *strict* if $t < s$).

2.3 Positions and Contexts

Terms are viewed as labelled trees in the usual way. Positions $p, q, \ldots \in \mathbb{N}_+^*$ are represented by chains of positive natural numbers used to address subterms of t. By Λ, we denote the empty chain (referring to the root of the term). Given positions p, q, we denote its concatenation by $p.q$. If p is a position, and Q is a set of positions, $p.Q$ is the set $\{p.q \mid q \in Q\}$. By $\mathcal{P}os(t)$, we denote the set of positions of a term t. Positions are ordered by the *prefix* ordering: $p \leq q$, if there exists p' such that $p.p' = q$. Positions can also be ordered by the lexicographical ordering: $p \leq_{lex} q$ iff $p \leq q$ or $p = w.i.p'$, $q = w.j.q'$, $i, j \in \mathbb{N}$, and $i < j$. Positions p, q are *disjoint*, denoted $p \perp q$, if neither $p \leq q$ nor $q \leq p$. Given a set of positions P, $minimal_{\leq}(P)$ is the set of minimal positions of P w.r.t. order $\leq$. Given a set $S \subseteq \mathcal{F} \cup \mathcal{X}$, $\mathcal{P}os_S(t)$ denotes positions in t where symbols in S occur. When no confusion arises, we denote $\mathcal{P}os_{\{f\}}(t)$ as $\mathcal{P}os_f(t)$ for a symbol $f \in \mathcal{F} \cup \mathcal{X}$. Positions of non-variable symbols in

t are denoted as $\mathcal{P}os_{\mathcal{F}}(t)$ and $\mathcal{P}os_{\mathcal{X}}(t)$ are the variable occurrences. The subterm at position p of t is denoted as $t|_p$ and $t[s]_p$ is the term t with the subterm at position p replaced by s. The symbol labelling the root of t is denoted as $root(t)$ and $root(t,p)$ is $root(t|_p)$. A context $C[\,]$ is a term from $\mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{X})$, where $\Box$ is a fresh constant symbol denoting one 'hole'. The number and location of holes is clarified 'in situ'. If $C[\,]$ is a context and t a term, $C[t]$ denotes the result of replacing the hole in C by t.

2.4 Term Rewriting Systems

A rewrite rule is an ordered pair (l,r), written $l \to r$, with $l,r \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $l \notin \mathcal{X}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The left-hand side (*lhs*) of the rule is l and r is the right-hand side (*rhs*). A *term rewriting system* (TRS) is a pair $\mathcal{R} = (\mathcal{F}, R)$ where R is a set of rewrite rules and $\mathcal{F}$ is called the signature. $L(\mathcal{R})$ denotes the set of *lhs*'s of $\mathcal{R}$. Given $\mathcal{R} = (\mathcal{F}, R)$, we consider $\mathcal{F}$ as the disjoint union $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ of symbols $c \in \mathcal{C}$, called *constructors*, and symbols $f \in \mathcal{D}$, called *defined functions* (or *operators*), where $\mathcal{D} = \{f \mid f(\bar{l}) \to r \in R\}$ and $\mathcal{C} = \mathcal{F} - \mathcal{D}$. Then, $\mathcal{T}(\mathcal{C},\mathcal{X})$ is the set of constructor terms. A pattern is a term $f(l_1,\ldots,l_n)$ such that $f \in \mathcal{D}$ and $l_1,\ldots,l_n \in \mathcal{T}(\mathcal{C},\mathcal{X})$. The set of patterns is $\mathcal{P}att(\mathcal{F},\mathcal{X})$. Let $\mathcal{P}os_{\mathcal{D}}(t)$ (resp. $\mathcal{P}os_{\mathcal{C}}(t)$) be the set of positions of defined (resp. constructor) symbols of term t. In this thesis, we often underline the redex applied in a rewrite step when several possible redexes are available.

A term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ rewrites to s (at position p), written $t \xrightarrow{p}_{\mathcal{R}} s$ (or $t \to_{\mathcal{R}} s$, or $t \to_p s$, or just $t \to s$), if $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$, for some $l \to r \in R$, $p \in \mathcal{P}os(t)$ and substitution σ. The subterm $\sigma(l)$ is called a *redex* and we say t rewrites to s by contracting redex $\sigma(l)$ at position p. We often underline the contracted redex of a rewriting step. The subterm $\sigma(r)$ is called a *contractum* of the redex $\sigma(l)$. We write $t \xrightarrow{>\Lambda} s$ if t rewrites to s by contracting a redex at a non-root position. We denote by $\mathcal{P}os_{\mathcal{R}}(t)$ the set of redexes of term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$. A term is a normal form of $\mathcal{R}$ if it is a $\to$-normal form. Let $\mathsf{NF}_{\mathcal{R}}$ be the set of normal forms of $\mathcal{R}$. A term t is a head-normal form (or root-stable) if it cannot be rewritten to a redex, i.e. there is no left-hand side l and substitution $\sigma \in Subst(\mathcal{T}(\mathcal{F},\mathcal{X}))$ such that $t \to^* \sigma(l)$. Let $\mathsf{HNF}_{\mathcal{R}}$ be the set of head-normal forms of $\mathcal{R}$.

A TRS $\mathcal{R}$ is left-linear if for all $l \in L(\mathcal{R})$, l is a linear term. A constructor system (*CS*) is a TRS whose lhs's are patterns. A TRS $\mathcal{R}$ is *terminating* (resp. *confluent*) if the relation $\to_{\mathcal{R}}$ is terminating (resp. confluent). A TRS $\mathcal{R}$ is *canonical* or *convergent* if the relation $\to_{\mathcal{R}}$ is terminating and confluent. If the TRS $\mathcal{R}$ is canonical, the normal form of a term $t \in \mathcal{T}(\mathcal{F})$ exists, it is unique, and it will be denoted by $t{\downarrow_{\mathcal{R}}} \in \mathsf{NF}_{\mathcal{R}}$. Two terms t, s are *joinable*, denoted by $t \downarrow s$, if there exists a term u such that $t \to^* u$ and $s \to^* u$. A TRS $\mathcal{R}$ is *sufficiently complete* if $\forall t \in \mathcal{T}(\mathcal{F})$, $\exists t' \in \mathcal{T}(\mathcal{C})$ such that $t \leftrightarrow^*_{\mathcal{R}} t'$. A defined function $f \in \mathcal{D}$ is *completely defined* if it does not occur in any ground term in normal form, i.e. functions are reducible on all ground terms (of appropriate sort). A TRS $\mathcal{R}$ is *completely defined* (or CD) if each defined function of the signature is completely defined.

Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap*, if there is a non-variable position $p \in \mathcal{P}os_{\mathcal{D}}(l)$ and a most-general unifier σ 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 $p = \epsilon$. A critical pair $\langle t, s\rangle$ is trivial if $t = s$. A left-linear TRS without critical pairs is called *orthogonal*. A left-linear TRS where its critical pairs are trivial overlays is called *almost orthogonal*. And if it only has trivial critical pairs it is called *weakly orthogonal*. Note that, in CSs, almost orthogonality and weak orthogonality coincide.

2.5 Strategies and Normalization

A (non-deterministic) *rewriting strategy* for a TRS $\mathcal{R}$ is a function $\mathcal{S}$ that assigns a non-empty set of non-empty finite rewrite sequences each beginning with t to every term t which is not a normal form [Barendregt *et al.*, 1987; Middeldorp, 1997]. As a specialization of the previous notion, by a *sequential* or *one-step* (non-deterministic) rewriting strategy for a TRS $\mathcal{R}$, we mean a function $\mathcal{S}$ that assigns a non-empty set $\mathcal{S}(t) \subseteq \mathcal{P}os_{\mathcal{R}}(t)$ of redex positions of t to every reducible term t [Middeldorp, 1997]. For TRSs that are not weakly orthogonal, we also need to supply the rewrite rule according to which the selected redex is to be contracted, since a redex may have more than one contractum (see [Antoy and Middeldorp, 1996] for explanation). We write $t \to_{\mathcal{S}} s$ for a sequential strategy $\mathcal{S}$ if $t \stackrel{p}{\to} s$ and $p \in \mathcal{S}(t)$. An $\mathcal{S}$-rewrite sequence is a reduction sequence of the form $t_1 \to_{\mathcal{S}} t_2 \to_{\mathcal{S}} t_3 \cdots$. If $t \to^*_{\mathcal{S}} s$, we say s is a $\mathcal{S}$-reduct of t (or reduct of t w.r.t. strategy $\mathcal{S}$).

A strategy $\mathcal{S}$ for a TRS $\mathcal{R}$ is called *normalizing* [Middeldorp, 1997] if there are no infinite $\mathcal{S}$-rewrite sequences starting from a term that has a normal form, i.e. it is able to normalize terms when a normal form exists. We say strategy $\mathcal{S}$ is *root-normalizing* [Middeldorp, 1997] if for all terms t that have a root-stable form, every infinite $\mathcal{S}$-rewrite sequence starting from t contains a root-stable term, i.e. it is able to find a root-stable form of a term if it exists, though no terminating sequences exist from a term. Not every normalizing strategy is root-normalizing and viceversa (see [Middeldorp, 1997] for examples). Moreover, some strategies are defined in the literature in terms of root-normalization instead of normalization. In such cases, a root-normalizing strategy can be extended to a normalizing strategy.

Definition 2.1 [Middeldorp, 1997] *Let $\mathcal{S}$ be a sequential root-normalizing strategy. The extension to a normalizing strategy $\mathcal{S}'$ is inductively defined as follows:*

$$\mathcal{S}'(t) = \begin{cases} \mathcal{S}(t) & \textit{if } t \textit{ is not root-stable.} \\ 1.\mathcal{S}(t_1) \cup \cdots \cup k.\mathcal{S}(t_k) & \textit{if } t = f(t_1, \ldots, t_k) \textit{ is root-stable.} \end{cases}$$

In order to formally (and practically) address the definition of strategies in this thesis, we recall the following 'intuitive' principles that a strategy must satisfy [Lucas, 2002a]:

Existence: To guarantee that $\mathcal{S}(t) \neq \varnothing$ for every reducible term t.

Computability: To provide an effective method for computing the strategy.

Good behavior: To provide evidence of some good computational property for the strategy; typically $\mathcal{S}$ is normalizing, root-normalizing, etc.

Efficiency: To ensure that computations achieved by using the strategy satisfy some criterion for efficiency; for instance, minimality of normalizing derivations.

2.6 Neededness

Definition 2.2 (Descendants of a position) [Huet and Lévy, 1992] *Let $A : t \xrightarrow{p}_{l \to r} s$ be a rewriting step and $q \in \mathcal{P}os(t)$. The set $q \backslash A$ (alternatively, $q \backslash \langle p, l \to r \rangle$) of* descendants *(or* residuals*) of q in s is defined as follows:*

$$q\backslash A = \begin{cases} \{q\} & \text{if } q < p \text{ or } q \parallel p, \\ \{p.p_3.p_2 \mid r|_{p_3} = l|_{p_1}\} & \text{if } q = p.p_1.p_2 \text{ with } p_1 \in \mathcal{P}os_{\mathcal{X}}(l), \\ \varnothing & \text{otherwise.} \end{cases}$$

If $Q \subseteq \mathcal{P}os(t)$ then $Q\backslash A$ denotes the set $\bigcup_{q \in Q} q\backslash A$. The notion of descendant extends to rewrite sequences in the obvious way. If Q is a set of pairwise disjoint positions in t and $A : t \to^* s$ then the positions in $Q\backslash A$ are pairwise disjoints. Descendants of redexes are usually called *residuals*. Almost orthogonal TRSs have the nice property that descendants of a redex are still redexes (with the same patterns); i.e. redexes are not destroyed by reduction in almost orthogonal TRSs.

For orthogonal term rewriting systems (TRSs), Huet and Lévy's *needed rewriting* is optimal.

Definition 2.3 (Needed redex) [Huet and Lévy, 1992] *Let $\mathcal{R}$ be a TRS. A redex Δ in a term t is called* needed *if in every rewrite sequence from t to a normal form, a descendant of Δ is contracted.*

Theorem 2.4 [Huet and Lévy, 1992] *Let $\mathcal{R}$ be an orthogonal TRS.*

1. *Every term which is not in normal form has a needed redex.*

2. *Repeated contraction of needed redexes results in a normal form, whenever the term under consideration has a normal form.*

Sekar and Ramakrishnan [Sekar and Ramakrishnan, 1993] noticed that Theorem 2.4 does not extend to almost orthogonal TRSs, where there is no a unique needed redex for all rewriting sequences but a set of redexes such that at least one of them is needed for each rewriting sequence.

Definition 2.5 (Necessary set of redexes) [Sekar and Ramakrishnan, 1993] *Let $\mathcal{R}$ be a TRS. A non-empty set of redexes in a term t is called* necessary *if in every rewrite sequence from t to a normal form, a descendant of at least one of the redexes in the set is contracted.*

Theorem 2.6 [Sekar and Ramakrishnan, 1993] *Let $\mathcal{R}$ be an almost orthogonal TRS.*

1. *Every term which is not in normal form has a necessary set of redexes.*
2. *Repeated contraction of necessary set of redexes results in a normal form, whenever the term under consideration has a normal form.*

However, [Middeldorp, 1997] showed that needed reduction is not useful when programming languages with lazy semantics are considered (e.g. functional programming languages such as Haskell) since a redex is always needed if the term under consideration does not have a normal form but an infinite normal form. In [Middeldorp, 1997], *root-neededness* of a redex is introduced as the basis to develop *normalizing* and *infinitary normalizing* rewrite strategies for lazy (or call-by-need) rewriting. In [Hanus *et al.*, 1998b], it is proved that (almost) orthogonal strongly sequential TRSs amount to reduce root-needed redexes.

Definition 2.7 (Root-neededness) [Middeldorp, 1997] *A position p of a term t is called* root-needed *if in every reduction sequence from t to a head-normal form, either $t|_p$ or a descendant of $t|_p$ is reduced. A rewriting step $t \xrightarrow{p} s$ is root-needed if the reduced position p is.*

2.7 Narrowing

Narrowing is an extension of rewriting for term rewriting systems where pattern matching is substituted by unification (as in *logic programming*) [Hullot, 1980; Slagle, 1974]. Narrowing was originally used as an automatic theorem resolution method in equational systems [Fay, 1979; Slagle, 1974]. Narrowing is the basic computational model of (multiparadigm) functional logic languages [Hanus, 1994; Hanus *et al.*, 2003, 1995; Reddy, 1985]. The narrowing mechanism is complete in the sense of logic programming (computation of *answers*) as well as in the sense of functional programming (computation of *values*). Functional logic programming inherits advantages of the different functional and logic paradigms in a unique and seamless way: functional programming provides nested expressions, efficient evaluation by deterministic (often lazy) evaluation and higher-order functions; whereas logic programming provides existentially quantified variables, partial data structures and built-in search. See [Hanus, 1994] for a survey on narrowing and functional logic programming.

To evaluate terms containing variables, *narrowing* non-deterministically instantiates the variables in order to make a rewrite step 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, $t \leadsto_{(p,\mathcal{R},\sigma)} t'$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \xrightarrow{p}_{\mathcal{R}} t'$. The substitution σ of a narrowing step $t \leadsto_{(p,\mathcal{R},\sigma)} t'$ is usually restricted to the variables in t; this is correct because the definition of narrowing only requires the rewrite step to be possible. We denote by $t_1 \leadsto_{\sigma}^{*} t_n$ a sequence of narrowing steps $t_1 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_{n-1}} t_n$ with $\sigma = \sigma_{n-1} \circ \cdots \circ \sigma_1$. Since in functional logic programming, we are interested in computing *values* (constructor terms) as well as *answers* (substitutions), we say that the narrowing derivation $t \leadsto_{\sigma}^{*} c$ *computes the result* c *with answer* σ if c is a constructor term.

Part I

Strategies: Neededness

Chapter 3

Weakly Outermost-Needed Rewriting and Narrowing

This chapter recalls the *weakly outermost-needed rewriting* strategy of [Antoy, 1992] and the *weakly outermost-needed narrowing* strategy of [Antoy *et al.*, 1994, 2000]. Outermost-needed narrowing (also called need narrowing) is currently the best sound and complete narrowing strategy for functional logic programming. Section 3.1 gives some introduction about the needed narrowing evaluation and its advantages. *Definitional trees* are introduced in Section 3.2, which are a fundamental data structure for the needed narrowing calculus. Section 3.3 recalls the weakly outermost-needed narrowing strategy and its original version for rewriting. Section 3.4 summarizes some useful properties of needed narrowing.

3.1 Introduction

Functional logic languages (see [Hanus, 1994] for a survey) with a complete operational semantics are usually based on *narrowing*, an operational mechanism which combines reduction (from the functional part) and variable instantiation (from the logic part) [Hullot, 1980; Reddy, 1985; Slagle, 1974]. A *narrowing step* instantiates variables of an expression and then applies a reduction step to a redex of the instantiated expression. Unfortunately, simple narrowing (similarly to rewriting) can have a huge search space and great effort has been made to develop sophisticated narrowing strategies without losing completeness. A challenge in the design of functional logic languages is the definition of a "good" narrowing strategy, i.e., a restriction on the narrowing steps issuing from t. Such narrowing strategy should be *sound* (i.e., only correct solutions are computed) and *complete* (i.e., all solutions or more general representations of the solutions are computed). Following the improvements in reduction strategies, several efforts have been developed w.r.t. narrowing in order to cut off the search space: basic narrowing [Hullot, 1980], innermost narrowing [Fribourg, 1985], outermost narrowing [Echahed, 1990], standard narrowing [Darlington and Guo, 1989; Ida and Nakahara, 1997; Middeldorp and Okui, 1998; You, 1991], lazy narrowing [Giovannetti *et al.*, 1991; Hans *et al.*, 1992; Moreno-Navarro *et al.*, 1990; Moreno-Navarro and Rodríguez-Artalejo, 1992; Reddy, 1985] and narrowing with redundancy tests [Bockmayr *et al.*, 1993]. Each strategy requires certain conditions on the term rewrite system in order to ensure completeness.

Needed narrowing [Antoy *et al.*, 1994, 2000] is a narrowing strategy which is complete for *inductively sequential* programs, a class of constructor term rewriting systems (CSs) with discriminating left-hand sides, i.e., typical functional programs. Moreover, it is optimal for such class of TRSs from both the functional programming side (length of successful derivations) as well as the logic programming view (number of computed solutions). The optimality is achieved by performing only "unavoidable" steps for solving equations in a non-deterministic way. The notion of "unavoidable" is related to the well-known notion of "neededness" from rewriting presented in Section 2.6. Needed narrowing rebuilds, in the functional logic setting, the unique (peerless) properties of the strongly needed reduction of (pure) equational and functional programs [O'Donnell, 1998; Plasmeijer and Eekelen, 1993].

Furthermore, there is a second optimality result concerned with the substitutions computed by narrowing derivations: 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 3.1 *[Antoy et al., 2000]*

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

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

In comparison to Definition 2.3 of a rewriting step which is needed, 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 3.2 *[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 $(p, l \rightarrow r, \sigma) \in \mathcal{S}(t)$, then p is a position of t, $l \rightarrow r$ is a rewrite rule, and σ a substitution such that*

$$t \rightsquigarrow_{p,l\rightarrow r,\sigma} \sigma(t[r]_p)$$

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 have some pattern-matching sequential organization representable in a structure called definitional tree [Antoy, 1992, 1997], which is used to implement needed narrowing.

3.2 Definitional Trees

A definitional tree can be seen as set of patterns organized in a hierarchical structure with some additional constraints. Here, we introduce the original definition of [Antoy, 1992] and the more "declarative" definition of [Antoy, 1997]. The symbols *branch* and $leaf$, used in the next definition, are uninterpreted functions used to classify the nodes of the tree.

Definition 3.3 *[Antoy, 1992] $\mathcal{T}$ is a partial definitional tree, or pdt, with pattern π iff one of the following cases hold:*

$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$ *where π is a pattern, o is the occurrence of a variable of π, 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 π is a pattern.*

A definitional tree can also be seen as a partially ordered set of patterns with some additional constraints. This alternative definition is useful to get rid of trees and symbols "branch" or "leaf".

Definition 3.4 *[Antoy, 1997] A definitional tree of a defined symbol f is a finite, non-empty set $\mathcal{T}$ of linear patterns partially ordered by subsumption order $\leq$ and having the following properties (up to renaming of variables) [Antoy, 1992]:*

leaf property *The maximal elements of $\mathcal{T}$, referred to as the* leaves*, are variants of the left-hand sides of the rules defining f. Non-maximal elements are referred to as* branches*.*

root property *The minimum element of $\mathcal{T}$ is referred to as the* root *and it is denoted as pattern($\mathcal{T}$).*

parent property *If π is a pattern of $\mathcal{T}$ different from the root, there exists in $\mathcal{T}$ a unique pattern π' strictly preceding π such that there exists no other pattern strictly between π and π'. π' is referred to as the* parent *of π and π as a* child *of π'.*

induction property *All the children of a same parent differ from each other only at the position of a variable of their parent, referred to as* inductive position*.*

We denote by $\mathcal{P}(\mathcal{F})$ the set of *pdt*s over the signature $\mathcal{F}$. A defined symbol f is called *inductively sequential* if there exists a definitional tree $\mathcal{T}$ with pattern $f(x_1, \ldots, x_k)$ (where $x_1, \ldots, x_k$ are different variables) whose leaves are variants (i.e.,

$$\begin{tikzcd}[column sep=small]
& \boxed{\texttt{x}} \leqslant \texttt{y} \arrow[dl] \arrow[dr] & & \\
\texttt{0} \leqslant \texttt{y = True} & & \texttt{s(x')} \leqslant \boxed{\texttt{y}} \arrow[dl] \arrow[dr] & \\
& \texttt{s(x')} \leqslant \texttt{0 = False} & & \texttt{s(y')} \leqslant \texttt{s(y') = x'} \leqslant \texttt{y'}
\end{tikzcd}$$

Figure 3.6: Definitional tree for the function "$\leqslant$"

equal modulo variable renaming) of all and only the rules defining f. In this case, we say that $\mathcal{T}$ is a definitional tree for f, denoted as $\mathcal{T}_f$. A left-linear CS $\mathcal{R}$ is *inductively sequential* if all its defined function symbols are inductively sequential. An inductively sequential CS can be viewed as a set of definitional trees, each defining a function symbol. It is often convenient and simplifies understanding to provide a graphical representation of definitional trees as a tree of patterns where the inductive position in branch nodes is surrounded by a box [Antoy, 1992].

Let us note that, even if we consider a first-order language, we use a curried notation in the examples (as it is usual in functional languages).

Example 3.5 *Consider the rules defining the "$\leqslant$" function:*

$\texttt{0} \leqslant \texttt{y = True}$
$\texttt{s(x)} \leqslant \texttt{0 = False}$
$\texttt{s(x)} \leqslant \texttt{s(y) = x} \leqslant \texttt{y}$

The definitional tree for the function "$\leqslant$" is the following (graphically illustrated in Figure 3.6):

$$\begin{aligned}
&\mathit{branch}(\texttt{x} \leqslant \texttt{y}, 1, \mathit{rule}(\texttt{0} \leqslant \texttt{y = True}),\\
&\qquad\qquad \mathit{branch}(\texttt{s(x')} \leqslant \texttt{y}, 2, \mathit{rule}(\texttt{s(x')} \leqslant \texttt{0 = False}),\\
&\qquad\qquad\qquad\qquad \mathit{rule}(\texttt{s(x')} \leqslant \texttt{s(y') = x'} \leqslant \texttt{y'})))
\end{aligned}$$

When a left-linear CS is not inductively sequential, a rule partition which obtains inductively sequential subsets is applied. The symbol *or* is used to represent the rule partition.

Example 3.7 *Consider the well-known parallel-or TRS $\mathcal{R}$ of [Middeldorp, 1997]:*

$\texttt{True} \vee \texttt{X = True}$
$\texttt{X} \vee \texttt{True = True}$
$\texttt{False} \vee \texttt{False = False}$

The definitional tree for the function "$\vee$" is the following (graphically illustrated in

$$\boxed{\texttt{x}} \vee \texttt{y} \quad \swarrow \; \searrow \quad \texttt{True} \vee \texttt{y = True} \qquad \texttt{False} \vee \boxed{\texttt{y}} \;\downarrow\; \texttt{False} \vee \texttt{False = False}$$

$$\texttt{x} \vee \boxed{\texttt{y}} \;\downarrow\; \texttt{x} \vee \texttt{True = True}$$

Figure 3.8: Partition of definitional trees for the function "∨"

Figure 3.8):

$$\begin{array}{l} or(branch(\texttt{x} \vee \texttt{y}, 1, rule(\texttt{True} \vee \texttt{y = True}), \\ \qquad\qquad\qquad branch(\texttt{False} \vee \texttt{y}, 2, rule(\texttt{False} \leqslant \texttt{False = False}))), \\ \quad branch(\texttt{x} \vee \texttt{y}, 2, rule(\texttt{x} \vee \texttt{True = True}))) \end{array}$$

For the definition of the needed narrowing strategy, $rule(\pi, \sigma(l) \to \sigma(r))$ is used to abbreviate the fact that $leaf(\pi)$ is a *pdt* representing some rule $l \to r$ of the considered rewrite system, where σ 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 (CD) 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 each *branch* node of the possible definitional trees contains a subtree for each constructor symbol of the appropriate sort.

3.3 The Weakly Outermost-Needed Narrowing 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 π denote such a pattern, τ be the most general unifier of t and π, and $\mathcal{T}$ be the *pdt* in which π 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 a *branch pdt*, then we recur on $\tau(t|_o)$, where o is the occurrence contained in $\mathcal{T}$ and τ 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 τ and o.

The outermost-needed narrowing strategy is a mapping, λ, that implements the above computation. λ 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, σ), where p is a position of t, R is a rule $l \to r$ of $\mathcal{R}$, and σ is a substitution. For $R = l \to r$, the strategy performs the narrowing step

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

In the following, $pattern(\mathcal{T})$ denotes the pattern argument of $\mathcal{T}$, and $\prec$ denotes the Noetherian ordering on $\mathcal{T}(\mathcal{F},\mathcal{X}) \times \mathcal{P}(\mathcal{F})$ 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 3.9 *[Antoy et al., 2000] The function λ takes two arguments: an operation-rooted term, t, and a pdt, $\mathcal{T}$, such that $pattern(\mathcal{T})$ and t unify. The function λ yields a set of triples of the form (p, R, σ), where p is a position of t, R is a rule $l \to r$ of $\mathcal{R}$, and σ is a unifier of $pattern(\mathcal{T})$ and t. Thus, let t be a term and $\mathcal{T}$ a pdt in the domain of λ. The function λ is defined by induction on $\prec$ as follows.*

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

The function λ is trivially well-defined in the second 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, λ is well-defined in the third 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 π 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 λ is well-defined in this case as well.

Example 3.10 *Consider the TRS and the definitional tree of Example 3.5. Consider also the following rules defining the "+" function:*

```
0 + y = y
s(x) + y = s(x+y)
```

The term `x ⩽ (0 + 0)` *has the following narrowing sequences according to Definition 3.9 since* $\lambda(t, \mathcal{T}_{\leqslant}) = \{(\Lambda, \texttt{0} \leqslant \texttt{y = True}, \{\texttt{x} \mapsto \texttt{0}\}), (2, \texttt{0 + y = y}, \{\texttt{x} \mapsto \texttt{s(x)}\})\}$*:*

$\underline{\texttt{x} \leqslant \texttt{(0 + 0)}} \leadsto_{\{\texttt{x}\mapsto\texttt{0}\}} \texttt{True}$

$\texttt{x} \leqslant \texttt{(}\underline{\texttt{0 + 0}}\texttt{)} \leadsto_{\{\texttt{x}\mapsto\texttt{s(x')}\}} \underline{\texttt{s(x')} \leqslant \texttt{0}} \leadsto_{id} \texttt{False}$

As in proof procedures for logic programming, we assume that the definitional trees always contain new, fresh 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 τ, when $t|_o$ is operation-rooted, is the *anticipated* substitution guaranteeing the neededness of the computed position. It is pushed down in the recursive call to λ 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 π. Hence, σ extends the anticipated substitution.

For non-inductively sequential left-linear CSs, the *weakly outermost-needed narrowing* strategy is defined [Antoy *et al.*, 1997]. The rules defining a non-inductively defined symbol f must be partitioned into inductively sequential subsets, i.e. subsets for which there exists a definitional tree (see [Antoy, 1992; Antoy *et al.*, 1997]). The function λ is then applied to all inductively sequential subsets and the union of all computed triples is selected.

Example 3.11 *Consider the TRS and the definitional tree of Example 3.7. The term* `x ∨ y` *has the following narrowing sequences:*

$\underline{\texttt{x} \vee \texttt{y}} \leadsto_{\{\texttt{x}\mapsto\texttt{True}\}} \texttt{True}$

$\underline{\texttt{x} \vee \texttt{y}} \leadsto_{\{\texttt{x}\mapsto\texttt{False},\texttt{y}\mapsto\texttt{False}\}} \texttt{True}$

$\underline{\texttt{x} \vee \texttt{y}} \leadsto_{\{\texttt{y}\mapsto\texttt{True}\}} \texttt{True}$

3.3.1 Weakly Outermost-Needed Rewriting

The outermost-needed strategy was originally defined in [Antoy, 1992] for rewriting instead of narrowing. We recall here such definition. However, its behaviour and properties are subsumed by those of the outermost-needed narrowing.

Definition 3.12 *[Antoy, 1992] The function φ takes two arguments: an operation-rooted term, t, and a pdt, $\mathcal{T}$, such that $pattern(\mathcal{T})$ and t unify. The function λ yields a tuple (p, R), where p is a position of t and R is a rule $l \to r$ of $\mathcal{R}$. Thus, let t be a term and $\mathcal{T}$ a pdt in the domain of φ. The function λ is defined by induction on $\prec$ as follows.*

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

Note that the computation of $\varphi(t, \mathcal{T})$ is always deterministic for a definitional tree in contrast to function λ. For non-inductively sequential left-linear CSs, the *weakly outermost-needed rewriting* strategy is defined similarly to the narrowing case [Antoy *et al.*, 1997]. The rules defining a non-inductively defined symbol f must be partitioned into inductively sequential subsets, i.e. subsets for which there exists a definitional tree (see [Antoy, 1992; Antoy *et al.*, 1997]). The function φ is then applied to all inductively sequential subsets and the union of all computed tuples is selected.

3.3.2 Normalization

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 σ 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 σ is a *computed answer substitution* for an equation e if there is a narrowing derivation $e \leadsto_\sigma^* \mathit{true}$. 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 3.9, 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[1]: *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].*

3.4 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 3.13 (Correctness) [Antoy *et al.*, 2000] *Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules. If $t \approx t' \leadsto_\sigma^* \mathit{true}$ is an outermost-needed narrowing derivation, then σ 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. The following theorem states the completeness of outermost-needed narrowing for inductively sequential rewrite systems:

Theorem 3.14 (Completeness) [Antoy *et al.*, 2000] *Let $\mathcal{R}$ be an inductively sequential rewrite system extended by the equality rules. Let σ be a constructor substitution that is a solution of an equation $t \approx t'$ and V be a finite set of variables*

[1] In a same manner to the extension of root-normalizing strategies to normalizing ones of Definition 2.1.

containing $\mathcal{V}ar(t) \cup \mathcal{V}ar(t')$. Then there exists a derivation $t \approx t' \leadsto^{*}_{\sigma'}$ true computed by λ such that $\sigma' \leq \sigma$ $[V]$.

The following theorem strengthens the characterization of the needed narrowing strategy by showing that no redundant solutions are computed by λ. 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 σ and σ', 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 as $\{x \mapsto 0\}$ and $\{x \mapsto s(x')\}$. Therefore, disjointness is a stronger notion than incomparability. Disjoint substitutions correspond to the notion of "*irreconcilable*" substitution of [Palamidessi, 1990].

Theorem 3.15 [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 λ. Then, σ and σ' 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 3.16 For instance, consider the rules in Example 3.5 and the following rules defining the addition on natural numbers:

```
0    + n = True
s(m) + n = s(m+n)
```

Given the term `0` $\leqslant$ `x+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 `+(x,x,z)` $\wedge$ `⩽(0,z,b)`, which causes a non-deterministic[2] computation due to the predicate call `+(x,x,z)`.

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 3.17 *[Alpuente et al., 1999e] 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.*

3.4.1 Properties of Weakly Outermost-Needed Narrowing

Nevertheless, in [Antoy *et al.*, 1997], weakly outermost-needed narrowing is proved to be a sound and complete procedure for non-inductively sequential rewrite systems. The following theorem states the soundness of weakly outermost-needed narrowing for inductively sequential rewrite systems.

Theorem 3.18 (Correctness) [Antoy *et al.*, 1997] *Let $\mathcal{R}$ be an almost orthogonal constructor rewrite system extended by the equality rules. If $t \approx t' \leadsto^*_\sigma$ true is an outermost-needed narrowing derivation, then σ is a solution for $t \approx t'$.*

Moreover, weakly outermost-needed narrowing is complete for the entire class of orthogonal rewrite systems w.r.t. irreducible substitutions, although neither needed rewriting nor needed narrowing steps are computable for this class of rewrite systems.

Theorem 3.19 (Completeness) [Antoy *et al.*, 1997] *Let $\mathcal{R}$ be an almost orthogonal constructor rewrite system extended by the equality rules. Let σ be a constructor substitution that is a solution of an equation $t \approx t'$ and V be a finite set of variables*

[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.

containing $\mathcal{V}ar(t) \cup \mathcal{V}ar(t')$*. Then there exists a derivation* $t \approx t' \leadsto^*_{\sigma'}$ *true computed by* λ *such that* $\sigma' \leq \sigma$ $[V]$.

Chapter 4

Incremental Needed Narrowing

This chapter addresses the problem of how to conduct *needed narrowing* [Antoy *et al.*, 2000], the best sound and complete narrowing strategy for functional logic programming, in an *incremental* way, i.e. without starting from scratch when a new needed narrowing step is to be applied. We show how a pre-eminent property of reduction in (a distinguished subclass of) strongly sequential programs of [Huet and Lévy, 1992], namely the incrementality of the evaluation, can be inherited by needed narrowing. We give an incremental definition of needed narrowing and show that the original optimality properties for inductively sequential programs are kept. Section 4.1 motivates the search of incrementality of the needed narrowing evaluation. Section 4.2 describes the proposed incremental optimization of definitional trees, namely *incremental definitional trees*. Section 4.3 formalizes the *incremental needed narrowing* calculus. In Section 4.4, we experimentally demonstrate that the incremental refinement can lead to substantial improvements in the overall evaluation process. Section 4.5 gives some related work and Section 4.6 concludes. Benchmark programs are given in Appendix A at the end of the chapter.

A short version of this chapter appeared in [Alpuente et al., 1999a,b,c,d]

4.1 Introduction

For orthogonal term rewriting systems (TRSs), Huet and Lévy's strong sequentiality provides a formal basis for the mechanization of sequential, normalizing rewriting computations [Huet and Lévy, 1979, 1992]. In [Huet and Lévy, 1979, 1992], Huet and Lévy defined the notion of strongly needed redex and showed that, for the class of strongly sequential TRSs (SS), the steady reduction of strongly needed redexes is normalizing (see Section 2.6). Huet and Lévy showed that the strong sequentiality of a TRS is decidable. They also provided an algorithm to identify a strongly needed redex within a reducible term, thus giving an effective procedure for implementing rewriting computations: given an input expression, first use Huet and Lévy's algorithm to find out a redex; if such a redex does not exist, then stop (the term is in normal form); otherwise, reduce the redex using the (unique) rule that can be applied; repeat these steps as long as possible.

Within Huet and Lévy's algorithm, the search for a new redex after each replacement starts from the root of the term. In general, only this way to proceed is safe; nevertheless, Strandh introduced the so-called forward branching TRSs (FB) and provided an algorithm which does not only hit upon a strongly needed redex in the term but also performs the corresponding replacement and resumes the reduction process incrementally, that is, *without* (necessarily) turning back to the root of the term in order to start the search for a new redex [Strandh, 1989]. FB is a subclass of SS, as it was shown in [Durand, 1994]. Strandh proved that his *incremental* algorithm preserves the optimality and normalizing properties of Huet and Lévy's reduction of strongly needed redexes. Toyama et al. defined the class of transitive systems (TR) and the notion of *transitive index* which, for transitive systems, can be advantageously used for incremental evaluation of terms also [Toyama *et al.*, 1993]. Durand also showed that FB = TR [Durand, 1994].

Needed narrowing [Antoy *et al.*, 1994] is currently the best complete narrowing strategy due to its optimality properties regarding the length of the derivations and the independence of computed answers [Antoy *et al.*, 2000]. Based on the notion of constructor system and inductive sequentiality [Antoy, 1992], needed narrowing rebuilds, in the functional logic setting, the properties of the strongly needed reduction of (pure) equational and functional programs (see Chapter 3). In [Antoy *et al.*, 2000], needed narrowing is formalized by means of a function λ which, for a given term, yields the position, rule, and substitution to be applied in order to reduce the considered expression (see Chapter 3). Therefore, the function λ accomplishes the role of Huet and Lévy's procedure to implement needed reduction and, similarly to that one, the corresponding calculus is not incremental.

For constructor systems, inductive sequentiality and strong sequentiality coincide [Hanus *et al.*, 1998b]. Since it is known that constructor strongly sequential systems are forward branching [Durand and Salinier, 1993], it seems reasonable to consider the possibility to define an incremental needed narrower which is able to efficiently conduct needed functional logic reductions. We first give an incremental definition λ^{ι} of needed narrowing and demonstrate that it is strongly equivalent to the original function λ of Definition 3.9 (also concerning its optimality results) and more efficiently implementable thanks to the incremental extent. Our incremental definition covers two different features of the implementation of needed narrowing: first, we provide in Section 4.2 an incremental definition of the data structure which is used to guide needed narrowing computations, namely, the *definitional trees*, and then we formalize in Section 4.3 an incremental definition of λ, i.e., an incremental procedure which actually implements needed narrowing computations.

We have applied our techniques to the optimization of the multi-paradigm language Curry [Hanus *et al.*, 1995, 2003], an extension of Haskell [Hudak *et al.*, 1992] which is supported by an international initiative with the aim of becoming a standard in the area and whose operational model is based on needed narrowing. In Section 4.4, we prove that all the proposed optimizations are actually (and independently) effective by comparing a basic, non-optimized version of UPV-Curry (a public implementation of Curry developed at UPV [Alpuente *et al.*, 1999d]), and three refined implementations. We provide an experimental comparison of our implementation and other Curry interpreters, namely TasteCurry and PAKCS-TasteCurry, the most popular implementations of Curry.

4.2 Incremental Definitional Trees

Definitional trees (see Definition 3.4) are used to guide the search for a redex within the computation of λ (also instrumenting the possible variable instantiations). From Definition 3.9 of λ, it is immediate to see that the pattern π of a branch node $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ is simply *augmented* by a flat term $c_i(x_1, \ldots, x_k)$, which is placed at the inductive position o of π, in order to attain the patterns of the (root nodes of the) sons $\mathcal{T}_i$, for $1 \leq i \leq n$. On the other hand, the decision about the definitional tree to be considered in the next step, only depends on the constructor symbol c_i rather than on the entire pattern of the son $\mathcal{T}_i$.

Thus, we have developed an optimized representation of the definitional trees which moves the pattern matching information to the exact place where it is needed. Given a TRS $\mathcal{R}$, an *incremental definitional tree* $\mathcal{I}$ with pattern π is an expression of the form:

$\mathcal{I} = irule(\pi = r')$ where $\pi = r'$ is a variant of a rule $l = r \in \mathcal{R}$.

$\mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n))$ where o is a variable position of π, $c_1, \ldots, c_n$

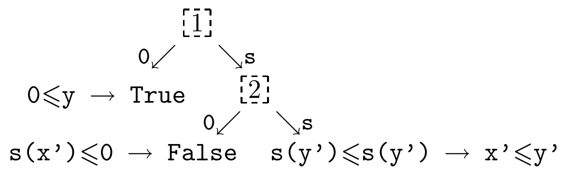

Figure 4.2: Incremental definitional tree for the function `first`

are constructors for $n > 0$, and each $\mathcal{I}_i$ is an incremental definitional tree with pattern $\pi[c_i(x_1, \ldots, x_k)]_o$ where k is the arity of the constructor c_i and $x_1, \ldots, x_k$ are new variables.

The pattern of an incremental definitional tree, which is not explicitly included in its syntactic structure, can be easily obtained as follows:

$$pattern(\mathcal{I}) = \begin{cases} \pi & \text{if } \mathcal{I} = irule(\pi = r); \\ pattern(\mathcal{I}_1)[x]_o & \text{if } \mathcal{I} = ibranch(o, (c_1, \mathcal{I}_1), \ldots, (c_n, \mathcal{I}_n)) \text{ and} \\ & x \notin \mathcal{V}ar(pattern(\mathcal{I}_1)). \end{cases}$$

Standard and incremental definitional trees are related by means of a function ρ which maps standard definitional trees to incremental definitional trees:

$$\rho(\mathcal{T}) = \begin{cases} irule(l = r) & \text{if } \mathcal{T} = rule(l = r); \\ ibranch(o, (c_1, \rho(\mathcal{T}_1)), \ldots, (c_n, \rho(\mathcal{T}_n))) & \text{if } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n), \text{ and} \\ & \quad pattern(\mathcal{T}_i) = \pi[c_i(\overline{x_k})]_o, \text{ for } 1 \leq i \leq n \end{cases}$$

Example 4.1 *The definitional tree of Example 3.5 is represented as follows:*

$$\begin{array}{l} ibranch(1, (\mathtt{0}, irule(\mathtt{0} \leqslant \mathtt{y} \rightarrow \mathtt{True})), \\ \qquad (\mathtt{s}, ibranch(2, (\mathtt{0}, irule(\mathtt{s(x')} \leqslant \mathtt{0} \rightarrow \mathtt{False})), \\ \qquad\qquad (\mathtt{s}, irule(\mathtt{s(x')} \leqslant \mathtt{s(y')} \rightarrow \mathtt{x'} \leqslant \mathtt{y'}))))) \end{array}$$

Figure 4.2 shows the corresponding graphical representation. The main differences w.r.t. Figure 3.6 are: a) patterns have been removed from branch nodes, b) a dashed box encloses the inductive position which labels branch nodes, and c) we label arcs with the corresponding constructor symbol.

Note that it is immediate to redefine the original outermost-needed narrowing strategy λ in terms of incremental definitional trees, although we don't include this definition in the chapter. Moreover, we consider only inductively sequential TRSs, i.e. we consider definitional trees without "or" nodes. In the following section, we provide an incremental definition λ^ι of the original outermost-needed narrowing strategy λ of Definition 3.9.

4.3 Incremental Evaluation

Consider a needed narrowing sequence $e_1 \leadsto e_2 \leadsto \cdots \leadsto e_n$. Since each step $e_i \leadsto_{(p_i,R_i,\sigma_i)} e_{i+1}$ is scheduled by $\lambda(e_i, \mathcal{T}_i)$ (where $\mathcal{T}_i$ is a definitional tree for $root(e_i)$), and it usually changes only small parts of e_i, it is obvious that, if $p_i \in \mathcal{P}os(e_i) - \{\Lambda\}$, then the needed narrowing step for e_{i+1} (i.e., the computation of $\lambda(e_{i+1}, \mathcal{T}_{i+1})$) might *wastefully* examine some of the symbols already considered in the preceding step and take almost the same decisions.

In order to avoid this wasting, we look for an *incremental* definition of λ which is able to locally resume the search from $e_{i+1}|_{q_i}$ for some $q_i \leq p_i$. Intuitively, whenever $(p, l{\to}r, \sigma) \in \lambda(t, \mathcal{T})$ and $t \leadsto_{(p,l\to r,\sigma)} s$, we can be sure that the symbols occurring at the position immediately above p have not been modified. On the other hand, the new term $s|_p$ may or may not be operation-rooted. If $root(s|_p) \in \mathcal{D}$, then $s|_p$ should be further narrowed until a constructor-rooted term or a variable is obtained. If $root(s|_p) \in \mathcal{C} \cup \mathcal{X}$, then the next needed narrowing step is determined by the definitional tree $\mathcal{T}$ of $root(s|_q)$, where $q < p$ is the position of the defined symbol occurring at the position immediately above p. The following result formally defines the former conditions.

Proposition 4.3 *Let t be a term, such that $t \leadsto_{(p,R,\sigma)} s$ using $(p, R, \sigma) \in \lambda(t, \mathcal{T})$, where $\mathcal{T}$ is a definitional tree such that $pattern(\mathcal{T}) \leq t$. Let $\mathcal{T}'$ be a definitional tree such that $pattern(\mathcal{T}') \leq s$ and $q = max(\{q' \in \mathcal{P}os_{\mathcal{D}}(s) \mid q' \leq p\})$. If $(p', R', \sigma') \in \lambda(s, \mathcal{T}')$, then $p' \geq q$.*

Proof. Let us first consider the case when $q \neq p$ (note that, since we impose $pattern(\mathcal{T}') \leq s$, it follows that $\{q' \in \mathcal{P}os_{\mathcal{D}}(s) \mid q' \leq p\}$ is not empty; i.e., q does exist.). Then, the computation of $\lambda(t, \mathcal{T})$ has reached a branch node $branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ and there is an operation-rooted subterm $t|_{q.o}$ at the inductive position o. Thus the computation of $\lambda(s, \mathcal{T}')$ first reaches the same branch node and then proceeds by checking whether $s|_{q.o}$ is a constructor-rooted term or a variable, since it cannot be an operation-rooted term unless $q = p$, which contradicts the original assumption. In both cases, a subtree $\mathcal{T}_i$ for $1 \leq i \leq n$ is selected before resuming the computation. If $\mathcal{T}_i$ is a branch subtree, then the computed position p' will be below q; otherwise (that is, $\mathcal{T}_i$ is a rule subtree), $p' = q$.

Now consider the case when $q = p$. Then $s|_p$ is operation-rooted and this position needs to be further narrowed in order to compute a constructor-rooted term. Then, the computed position is trivially q or below q. □

Similarly, the following result formally defines that after a needed-narrowing step, the subterm at the narrowed position should be further narrowed until a constructor-rooted term or a variable arises at that position, i.e. if it is rooted by a defined symbol, we must continue narrowing from that position, but if it is rooted by a constructor symbol or a variable, we must continue in some position above.

Corollary 4.4 *Let t be a term, such that $t \rightsquigarrow_{(p,R,\sigma)} s$ using $(p, R, \sigma) \in \lambda(t, \mathcal{T})$, where $\mathcal{T}$ is a definitional tree such that $pattern(\mathcal{T}) \leq t$. Let $\mathcal{T}'$ be a definitional tree such that $pattern(\mathcal{T}') \leq s$. If $s|_p$ is operation-rooted and $(p', R', \sigma') \in \lambda(s, \mathcal{T}')$, then $p' \geq p$.*

Proof. Immediate by Proposition 4.3. □

Since each step $e_i \rightsquigarrow_{(p_i,R_i,\sigma_i)} e_{i+1}$ in a needed narrowing sequence $e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots \rightsquigarrow e_n$ is performed using $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$, where $\mathcal{T}_i$ is a definitional tree for $root(e_i)$, Corollary 4.4 ensures that, whenever $e_{i+1}|_{p_i}$ is operation-rooted, then $p_{i+1} \geq p_i$. Thus, we take $(p_{i+1}, R_{i+1}, \sigma_{i+1}) = (p_i.p, R, \sigma)$, where $(p, R, \sigma) \in \lambda(e_{i+1}|_{p_i}, \mathcal{T})$ and $\mathcal{T}$ is a definitional tree for $root(e_{i+1}|_{p_i})$.

On the other hand, if $e_{i+1}|_{p_i}$ is constructor-rooted or a variable, then it is necessary to resume the evaluation for some q somewhere above p_i. Proposition 4.3 allows us to come back to $e_{i+1}|_q$ by taking the ('most defined') definitional subtree $\mathcal{T}_i'$ of the definitional tree of $root(e_i|_q)$ which has been used in the computation of $\lambda(e_i, \mathcal{T}_i)$. Thus, we let $(p_{i+1}, R_{i+1}, \sigma_{i+1}) = (q.p, R, \sigma)$, where $(p, R, \sigma) \in \lambda(e_{i+1}|_q, \mathcal{T}_i')$. Besides, we also know that in the case $q = p_i$ and $q > \Lambda$, the expression e_{i+1} has a defined symbol at q which needs a constructor-rooted term at position p_i to continue processing its definitional tree. Therefore, for a sequence n_i of needed narrowing steps in the computation $e = e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots \rightsquigarrow e_n$ such that the operation-rooted term at q can not be narrowed, $\forall i \leq j \leq i + n_i, (p_j, Rj, \sigma_j) \in \lambda(e_j, \mathcal{T}_j)$, and $p_j > q$. Since in the needed narrowing step $i + n_i + 1$ the position q will be narrowed, we will need the definitional subtree $\mathcal{T}$ for $root(e_i|_q)$ and the position q to compute only a call $\lambda(e_{i+n_i+1}|_q, \mathcal{T}')$ instead of $\lambda(e_{i+n_i+1}, \mathcal{T}_{i+n_i+1})$.

Thus, we define a data structure $\mathcal{L}$ which contains the information needed to perform the needed narrowing steps incrementally. Given a term t, we let $\mathcal{L}$ denote a list of pairs $(p, \mathcal{I})$ where p is a position and $\mathcal{I}$ is an incremental definitional tree. The *incremental needed narrowing strategy* is denoted by $\lambda^\iota(t, \mathcal{L})$. It takes as arguments a term t and an auxiliary list $\mathcal{L}$ which plays the role of the definitional tree $\mathcal{T}$ in the original strategy $\lambda(t, \mathcal{T})$.

Definition 4.5 *The function λ^ι takes an operation-rooted term t and a list $\mathcal{L} = [(q, \mathcal{I}), \ldots]$ with $pattern(\mathcal{I}) \leq t|_q$, and computes quadruples of the form $(p, R, \sigma, \mathcal{L}') \in \lambda^\iota(t, \mathcal{L})$, where p is the position pointing to the subterm of t to be narrowed, R is the rule to be applied, σ is the computed substitution, and $\mathcal{L}'$ is a list containing the information which allows us to proceed with the subsequent evaluation steps incrementally.*

$$\lambda^\iota(t,[(q,\mathcal{I})|\mathcal{L}]) \ni \begin{cases} (q, l{\to}r, id, \mathcal{L}') & \textit{if } \mathcal{I} = \textit{irule}(l=r) \textit{ and} \\ & \mathcal{L}' = \begin{cases} [(q,\mathcal{I}')|\mathcal{L}] & \textit{if } \theta(r) \textit{ is operation-rooted} \\ \mathcal{L} & \textit{otherwise} \end{cases} \\ & \textit{where } t|_q = \theta(l) \textit{ and } \mathcal{I}' \textit{ is an incremental} \\ & \quad \textit{definitional tree for } root(\theta(r)) \\ (p, R, \sigma\circ\tau, \mathcal{L}') & \textit{if } \mathcal{I} = \textit{ibranch}(o,(c_1,\mathcal{I}_1),\ldots,(c_n,\mathcal{I}_n)),\ t|_{q.o} = x \in \mathcal{X}, \\ & \tau = \{x \mapsto c_i(\overline{x_k})\},\ \textit{and } (p,R,\sigma,\mathcal{L}') \in \lambda^\iota(\tau(t),[(q,\mathcal{I}_i)|\mathcal{L}]); \\ (p,R,\sigma,\mathcal{L}') & \textit{if } \mathcal{I} = \textit{ibranch}(o,(c_1,\mathcal{I}_1),\ldots,(c_n,\mathcal{I}_n)), \\ & t|_{q.o} = c_i(\overline{t_k}),\ \textit{and } (p,R,\sigma,\mathcal{L}') \in \lambda^\iota(t,[(q,\mathcal{I}_i)|\mathcal{L}]); \\ (p, l{\to}r, \sigma, \mathcal{L}') & \textit{if } \mathcal{I} = \textit{ibranch}(o,(c_1,\mathcal{I}_1),\ldots,(c_n,\mathcal{I}_n)),\ t|_{q.o} = f(\overline{t_k}) \\ & \textit{for } f \in \mathcal{D},\ \mathcal{I}' \textit{ is an incremental definitional tree for } f, \\ & \textit{and } (p,l{\to}r,\sigma,\mathcal{L}') \in \lambda^\iota(t,[(q.o,\mathcal{I}'),(q,\mathcal{I})|\mathcal{L}]); \end{cases}$$

The following notations are helpful. Given a position p and a list $\mathcal{L}$ of pairs position/incremental definitional tree, we define:

$$p.\mathcal{L} = \begin{cases} [\,] & \text{if} \quad \mathcal{L} = [\,] \\ [(p.q,\mathcal{I}) \mid p.\mathcal{L}'] & \text{if} \quad \mathcal{L} = [(q,\mathcal{I}) \mid \mathcal{L}'] \end{cases}$$

We also use the standard operator '++' for concatenating lists. The following auxiliary result sets that the evaluation can be done incrementally.

Lemma 4.6 *Let t be a term and $\mathcal{I} = \textit{ibranch}(o,(c_1,\mathcal{I}_1),\ldots,(c_n,\mathcal{I}_n))$ be such that $\textit{pattern}(\mathcal{I}) \leq t$ and $t|_o$ is operation-rooted. Let $\mathcal{I}'$ be an incremental definitional tree such that $\textit{pattern}(\mathcal{I}') \leq t|_o$. Then,*

$$\lambda^\iota(t,[(\Lambda,\mathcal{I})]) = \{(o.p, l\to r, \sigma, (o.\mathcal{L}) \mathbin{++} [(\Lambda,\mathcal{I})]) \mid (p, l\to r, \sigma, \mathcal{L}) \in \lambda^\iota(t|_o,[(\Lambda,\mathcal{I}')])\}$$

Proof. Immediate, by definition of λ^ι. □

The following proposition states that λ^ι is able to compute the same needed narrowing step than λ starting from a given term and an initial list of incremental information.

Proposition 4.7 *Let t be a term and $\mathcal{T}$ be a definitional tree such that $\textit{pattern}(\mathcal{T}) \leq t$. Then, $(p, l{\to}r, \sigma) \in \lambda(t,\mathcal{T})$ if and only if $\exists \mathcal{L}.(p,l{\to}r,\sigma,\mathcal{L}) \in \lambda^\iota(t,[(\Lambda,\rho(\mathcal{T}))])$.*

Proof. Immediately from the definition of λ and λ^ι. □

Now, we have to ensure that the defined function symbols occurring at positions which are above (or at) a previously narrowed position are obtained by the strategy λ^ι and thus considered for the subsequent narrowing steps.

Proposition 4.8 (Completeness) *Let t be a term and $\mathcal{T}$ be a definitional tree such that $pattern(\mathcal{T}) \leq t$. Let $t \leadsto_{(p,l\to r,\sigma)} s$ using $(p, l\to r, \sigma) \in \lambda(t, \mathcal{T})$. Let $(p, l\to r, \sigma, \mathcal{L}) \in \lambda^\iota(t, [(\Lambda, \rho(\mathcal{T}))])$, $F_s = \{q \in \mathcal{P}os_\mathcal{D}(s) \mid q \leq p\}$. Then, $q \in F_s$ if and only if $\exists \mathcal{I}.(q, \mathcal{I}) \in \mathcal{L}$.*

Proof. By noetherian induction on pairs $(t, \mathcal{T})$ (strictly) ordered by the ordering introduced in Section 3.3: $(t, \mathcal{T}) \prec (t', \mathcal{T}')$ iff $n_\mathcal{D}(t) < n_\mathcal{D}(t')$ or $n_\mathcal{D}(t) = n_\mathcal{D}(t')$ and $\mathcal{T} < \mathcal{T}'$, i.e., $\mathcal{T}$ is a subtree of $\mathcal{T}'$ [Antoy *et al.*, 2000]. Here, $n_\mathcal{D}(t)$ is the number of defined function symbols in t.

- (Base) Since $pattern(\mathcal{T}) \leq t$, then t is operation-rooted. Hence, the base case consists in considering those pairs $(t, \mathcal{T})$ such that t is operation-rooted and does not contain any other defined function symbol, and $\mathcal{T}$ is a rule node $\mathcal{T} = rule(l = r)$. By hypothesis, $l \leq t$, hence let $t = \theta(l)$. Thus, only the first case for λ applies, i.e., $t \leadsto_{(\Lambda,l\to r,id)} s$ and we have either

  1. $s = \theta(r)$ is not operation-rooted, hence $F_s = \emptyset$. Then, from the definition of λ^ι, we have that $\mathcal{L} = [\ ]$.
  2. $\theta(r)$ is operation-rooted, hence $F_s = \{\Lambda\}$. Then, from the definition of λ^ι, $\mathcal{L} = [(\Lambda, \mathcal{I})]$, where $\mathcal{I}$ is an incremental definitional tree for $root(\theta(r))$.

  On the other hand, if $\mathcal{L} = [\ ]$, then from the definition of λ^ι, we have that $s = \theta(r)$ is not operation-rooted and hence $F_s = \emptyset$. If $\mathcal{L}$ is not empty, then $\mathcal{L} = [(\Lambda, \mathcal{I})]$, and thus $s = \theta(r)$ is operation-rooted, i.e., $F_s = \{\Lambda\}$.

- (Induction) We consider the different cases for λ separately.

  1. If $\mathcal{T}$ is a rule node, we reason as in the base case.
  2. If $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ and $t|_o$ is a variable x, then $\lambda(t, \mathcal{T}) = (p, l\to r, \sigma \circ \tau)$, where $(p, l\to r, \sigma) \in \lambda(\tau(t), \mathcal{T}_i)$ for some $1 \leq i \leq n$ and $\tau = \{x \mapsto c_i(\overline{x_k})\}$. Thus, $\tau(t) \leadsto_{(p,l\to r,\sigma)} s$. Since $n_\mathcal{D}(\tau(t)) = n_\mathcal{D}(t)$ and $\mathcal{T}_i < \mathcal{T}$, by induction hypothesis we have that, for $(p, l\to r, \sigma, \mathcal{L}) \in \lambda^\iota(\tau(t), [(\Lambda, \rho(\mathcal{T}_i))])$, $q' \in F_s$ iff $\exists \mathcal{I}.(q', \mathcal{I}) \in \mathcal{L}$. From the definition of λ^ι, the conclusion follows.
  3. If $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ and $t|_o$ is constructor-rooted, the conclusion follows by a similar reasoning to case 2.
  4. If $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_n)$ and $t|_o$ is operation-rooted, then $(p', l\to r, \sigma) \in \lambda(t|_o, \mathcal{T}')$ (where $\mathcal{T}'$ is a definitional tree for $root(t|_o)$ and $p = o.p'$), i.e., $t|_o \leadsto_{(p',l\to r,\sigma)} s'$. Since $n_\mathcal{D}(t|_o) < n_\mathcal{D}(t)$, by the induction hypothesis $q' \in F_{s'}$ if and only if $\exists \mathcal{I}.(q', \mathcal{I}) \in \mathcal{L}'$, for $(p', l\to r, \sigma, \mathcal{L}') \in \lambda^\iota(t|_o, [(\Lambda, \rho(\mathcal{T}'))])$. Since $F_s = \{\Lambda\} \cup o.F_{s'}$, by Lemma 4.6 the conclusion follows. □

The following theorem shows that the needed narrowing steps performed by the incremental strategy λ^ι and the computation steps carried out by the original needed narrowing strategy λ are equivalent.

Theorem 4.9 (Correctness) *Let $\mathcal{R}$ be an inductively sequential program, $e_1 \rightsquigarrow e_2 \rightsquigarrow \cdots \rightsquigarrow e_n$ be a needed narrowing derivation where $\mathcal{T}_i$ is a definitional tree such that $pattern(\mathcal{T}_i) \leq e_i$ for all $1 \leq i \leq n$, and $\mathcal{L}_0 = [(\Lambda, \rho(\mathcal{T}_1))]$. Then, for all $1 \leq i \leq n$, $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$ if and only if $(p_i, R_i, \sigma_i, \mathcal{L}_i) \in \lambda^\iota(e_i, \mathcal{L}_{i-1})$.*

Proof. We prove it by induction on the length of the derivation.

- $i = 1$. This case is trivial, since by Proposition 4.7 it amounts to a call to $\lambda^\iota(e, \mathcal{L}_0)$.
- $i > 1$. Let $\mathcal{L}_{i-1} = [(q, \rho(\mathcal{T}))|\mathcal{L}']$ for some definitional tree $\mathcal{T}$. By the induction hypothesis, we have that $(p_{i-1}, R_{i-1}, \sigma_{i-1}) \in \lambda(e_{i-1}, \mathcal{T}_{i-1})$ if and only if $(p_{i-1}, R_{i-1}, \sigma_{i-1}, \mathcal{L}_{i-1}) \in \lambda^\iota(e_{i-1}, \mathcal{L}_{i-2})$. Since $e_{i-1} \rightsquigarrow_{(p_{i-1}, R_{i-1}, \sigma_{i-1})} e_i$, by Proposition 4.8, q points to a defined symbol of e_i which is above or at the position p_{i-1}. By the ordering between the positions recorded in $\mathcal{L}_{i-1}$ (see Lemma 4.6), q is the position of the defined symbol immediately above or at the position p_{i-1}. By Proposition 4.3, $(p', R_i, \sigma_i) \in \lambda(e_i|_q, \mathcal{T}) \Leftrightarrow (q.p', R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i)$, where $q.p' = p_i$. Then, by Proposition 4.7, $(p', R_i, \sigma_i) \in \lambda(e_i|_q, \mathcal{T}) \Leftrightarrow (p', R_i, \sigma_i, \mathcal{L}'_i) \in \lambda^\iota(e_i|_q, [(\Lambda, \rho(\mathcal{T}))])$. Now, since all positions in the first component of the position/incremental definitional tree lists correspond to operation-rooted subterms, by using Lemma 4.6 it is immediate that $(q.p', R_i, \sigma_i, \mathcal{L}_i) \in \lambda^\iota(e_i, [(q, \rho(\mathcal{T}))|\mathcal{L}']) \Leftrightarrow (p', R_i, \sigma_i, \mathcal{L}'_i) \in \lambda^\iota(e_i|_q, [(\Lambda, \rho(\mathcal{T}))])$ if $(q', \mathcal{I}') \in \mathcal{L}'_i \Rightarrow (q.q', \mathcal{I}') \in \mathcal{L}_i$.

  Thus, $(p_i, R_i, \sigma_i) \in \lambda(e_i, \mathcal{T}_i) \Leftrightarrow (p_i, R_i, \sigma_i, \mathcal{L}_i) \in \lambda^\iota(e_i, [(q, \rho(\mathcal{T}))|\mathcal{L}'])$. □

Although we only provide in this chapter the definition of λ^ι in terms of incremental definitional trees, the function λ^ι can be easily redefined to work with standard definitional trees, similarly to λ. By abuse, we freely use this in the following section, where different implementations of needed narrowing are compared.

4.4 Experimental Results

Curry [Hanus *et al.*, 2003] is a functional logic programming language based on needed narrowing which combines the best ideas of declarative languages such as Haskell [Hudak *et al.*, 1992] and SML [Milner *et al.*, 1990] (functional languages), Gödel [Hill and Lloyd, 1994] and λProlog [Nadathur and Miller, 1988] (logic languages), and

ALF [Hanus and Schwab, 1991], Babel [Kuchen and Anastasiadis, 1996] and $\mathcal{TOY}$ [López-Fraguas and Sánchez-Hernández, 1999] (functional logic languages).

UPV-Curry [Alpuente *et al.*, 1999d] is an interpreter of Curry developed by the ELP group at the Technical University of Valencia (UPV), which provides an almost complete implementation of the language according to [Hanus *et al.*, 1998a] (it lacks Curry modules and encapsulated search) and which is is publicly available at

`http://www.dsic.upv.es/users/elp/soft.html`

The system is written in SICStus Prolog v3.6 and consists of about 250 clauses (3000 lines of code). The parser is expressed by 95 clauses, the typechecker module by 55 clauses, the algorithm which builds definitional trees by 20 clauses, the execution module by 45 clauses, and the interface and some other utilities by 35 clauses. See [Escobar *et al.*, 1998] for a complete description.

We have performed an empirical comparison of the original needed narrowing strategy and the optimized definition when used in the implementation of Curry (see Table 4.1). We first consider a basic UPV-Curry implementation which mechanizes the standard λ function, and the original definitional trees. A second UPV-Curry implementation includes the refinement due to the incremental definitional trees. The third UPV-Curry implementation includes the incremental strategy λ^ι, but it considers the original definitional trees. We finally consider a completely refined implementation, including all optimizations. Times were measured on a SUN SparcStation, running under UNIX System V Release 4.0. They are expressed in milliseconds and are the average of 10 executions. The benchmarks used for the analysis are given in Appendix A; most of them are standard Curry test programs[1]. The functions `ackermann`, `fibonacci`, `last`, `quicksort` and `mergesort` are the standard ones; `horseman` computes, by means of constraints, the number of men and horses that have a certain number of heads and feet; finally, `iter` produces a sequence of n nested calls to a given function and is used to set the potentiality of our method off.

Runtime input goals were chosen to give a reasonably long overall time, and are shown in Table 4.2. Natural numbers are implemented using Z/S-terms, and lists are shown within goals by using a subindex which represents its size. L is `[10,9,8,7,6,5,4,3,2,1,0]` in the goal for `quicksort`.

The figures in Table 4.1 reveal that the proposed refinements are actually (and independently) effective. The goals which require much computation, such as `ackermann`, `fibonacci`, `quicksort`, and `mergesort` get a sustained speed-up when incrementality is used (e.g. 3.20 for the `ackermann` example). Indeed, the `iter` benchmark highlights the benefits which are brought by the incremental evaluation when the goal

[1] Look at the URL `http://www.informatik.uni-kiel.de/~curry/examples/` for a list of available Curry test programs.

Benchmark	Original	Incremental def. trees	Incremental evaluation	Both optimizations	Global Speed-up
`iter`	3345	1145	395	344	9.72
`ackermann`	2097	1112	834	653	3.20
`mergesort`	556	466	340	323	1.72
`quicksort`	4137	3114	2668	2658	1.55
`fibonacci`	869	717	678	622	1.40
`horseman`	3342	3180	2762	2751	1.21
`last`	1025	1008	983	968	1.05

Table 4.1: Comparison between UPV-Curry optimizations (in ms.)

Benchmark	Goal	UPV-Curry	TasteCurry	PAKCS
`iter`	`iter 100 sub1 100`	344	1174	909
`ackermann`	`ackermann 20`	653	2505	1203
`mergesort`	`sort intMerge [3,1,2] xs`	323	870	442
`quicksort`	`qsort` L	2658	1829	1511
`fibonacci`	`fibonacci 10`	622	2262	1106
`horseman`	`horseman x y 8 20`	2751	3542	1658
`last`	`last [1]`$_{100}$	977	58736	4256

Table 4.2: Comparison with other Curry interpreters (in ms.)

contains some nested calls and the innermost one must be evaluated first. On the other hand, even in the cases when no significant speed-up is obtained (this is the case of `last`), execution is not punished with any appreciable overhead. In other words, the use of incremental definitional trees causes no overload, which substantiate the idea that they are simply an apt shortening of the standard definitional trees. On the other hand, the incremental evaluation leads on its own to considerable speed-ups in program execution while still guaranteeing no appreciable slowdown.

Table 4.2 compares the fully optimized UPV-Curry implementation and other Curry interpreters. Namely, the figures in Table 4.2 correspond to the runtimes of the benchmarks for UPV-Curry, TasteCurry, and PAKCS-TasteCurry interpreters[2]. These results show that the incremental UPV-Curry implementation performs very well in comparison to the other Curry interpreters. In overall, it takes about 20% less than the time needed by PAKCS-TasteCurry and about 62% less than the time needed by TasteCurry to evaluate the queries. These results seem to substantiate the advantages of using the proposed incremental techniques.

Another interesting aspect for the comparison concerns the spatial savings due to the use of incremental definitional trees. For the comparison, we strictly consider the

[2] TasteCurry and PAKCS-TasteCurry are available from `http://www.informatik.uni-kiel.de/~curry/`.

Benchmark	Saving	Benchmark	Saving
`ackermann`	43%	`mergesort`	60%
`fibonacci`	25%	`iter`	65%
`horseman`	33%	`quicksort`	0%
`last`	33%		

Table 4.3: Spatial savings

information which is necessary for defining the needed narrowing strategy. Table 4.3 shows the ratio between the number of symbols of inner nodes of standard definitional trees and incremental definitional trees, respectively. The results of Table 4.3 clearly evidence that incremental definitional trees can also bring substantial savings in memory space (up to 65% in these tests).

4.5 Related Work

It has been proved that inductively sequential TRSs and Huet and Lévy's strongly sequential CSs coincide [Hanus *et al.*, 1998b]. [Hanus *et al.*, 1998b] also discuss the precise relation between definitional trees and Strandh's *index trees* [Strandh, 1989], a data structure which was proved equivalent by Durand [Durand, 1994] to Huet and Lévy's *matching dags* (i.e., the original data structure which serves to implement strongly needed reductions in strongly sequential TRSs [Huet and Lévy, 1979, 1992]).

Roughly speaking, an index tree for an orthogonal TRS $\mathcal{R}$ is (despite its name) a directed acyclic graph whose inner nodes are *index points*, i.e., pairs $\langle \pi, p \rangle$, where π is a strict prefix of a *lhs* of $\mathcal{R}$ (with the variables replaced by a special symbol Ω) and p is a strong index of π, which satisfy some suitable conditions (see [Durand, 1994]). There is a special root node $\langle \Omega, \Lambda \rangle$. Given a node $\langle \pi, p \rangle$ and a function symbol f, there is an (ordinary) *arc* from $\langle \pi, p \rangle$ to $\langle \pi[f(\overline{\Omega})]_p, q \rangle$ if either $\langle \pi[f(\overline{\Omega})]_p, q \rangle$ is an index point or $\pi[f(\overline{\Omega})]_p$ is a *lhs* of a rule whose variables have been replaced by Ω's (in the latter case, we discard q and just write $\pi[f(\overline{\Omega})]_p$ to denote this node at the end of an arc). There are also *failure arcs*, which are used to proceed after a failing partial matching without being compelled to start from scratch. Orthogonal TRSs such that a corresponding index tree exists are called *bounded* TRSs. Durand proved that the class of bounded TRSs and strongly sequential TRSs coincide [Durand, 1994]. The proof is based on the existence of a precise and immediate correspondence between Huet and Lévy's matching dags and Strandh's index trees.

According to the definition of index trees, some nodes might *not* be reachable from the initial node $\langle \Omega, \Lambda \rangle$ via ordinary arcs (failure arcs can be necessary to access them). Strandh defined the *forward branching* index trees, which are those index trees such that all nodes can be reached from the root node through an ordinary path

[Strandh, 1989], and introduced the class of forward branching TRSs (FB) to contain those TRSs which admit a forward branching index tree. Strandh proved (actually, that was the main aim of [Strandh, 1989]) that forward branching index trees are suitable for implementing incremental strongly needed reductions and provided an algorithm which does perform incrementally. Later on, and following a different line of work, Toyama et al. investigated the existence of the *transitive* strongly needed redexes, which can be reduced incrementally also [Toyama *et al.*, 1993]. They did not provide any particular data structure (comparable to forward branching index trees) to ease the search for the redexes but only defined the transitive TRSs (TR) as those which grant the transitive strongly needed reductions. Durand showed that FB = TR [Durand, 1994].

Huet and Lévy had previously defined a particular class of strongly sequential TRSs, namely the *simple* systems [Huet and Lévy, 1979, 1992], with a very simple associated matching dag; actually, they showed that the corresponding matching dag is *almost* a tree (that is, the ordinary arcs and the corresponding nodes make up a tree, see [Huet and Lévy, 1992], pages 439 – 440). Nevertheless, they did state nothing about the possibility of performing incremental reductions with simple systems; Huet and Lévy only noticed that there is a very simple algorithm for supplying a simple system with a matching dag. Durand and Salinier [Durand and Salinier, 1993] demonstrated that simple systems are forward branching and, correspondingly, the matching dag which is associated to a simple system exactly coincides with the forward branching index tree which would be constructed using the algorithm in [Durand, 1994]. As a consequence, it is possible to perform incremental reductions in simple systems.

By glueing together all these facts, and since strongly sequential CSs (hence inductively sequential ones) are simple systems [Huet and Lévy, 1979, 1992], we get that the forward branching index tree which corresponds to an inductively sequential TRS is 'almost' a tree. Actually, we can say that it *is* a tree since, as a benefit of the constructor discipline, failing arcs are *not* present in the index trees of inductively sequential TRSs (see Definition 2.21 of [Durand, 1994] for more details). Now, by using the connection between index trees and definitional trees given in [Hanus *et al.*, 1998b] we obtain, from the forward branching index tree associated to an inductively sequential system, a forest of definitional trees which consists of the immediate subtrees stemming from the special node $\langle \Omega, \Lambda \rangle$. On the other hand, the nonexistence of failing arcs (in CSs) allows us a further refinement in the representation, which immediately derives in our incremental definitional trees. This redeems definitional trees from unnecessarily mimicking the structure of index trees when we consider CSs. Nevertheless, for arbitrary (possible non-constructor) simple systems, the need to record failing arcs prevents one from considering an index tree as a simple collection of definitional trees.

On the other hand, Huet and Lévy and Strandh only considered rewriting computations, whereas our incremental trees do not only ease pattern matching but also support the variable instantiation in input terms which is necessary for the narrowing steps. Hence, our incremental definition of needed narrowing can be clearly thought of as the functional logic counterpart of Strandh's algorithm for the incremental reduction of terms in forward branching TRSs [Strandh, 1989].

4.6 Conclusions

We have introduced *incremental* definitional trees, a data structure which is a refinement of the usual definitional trees and redeems them from unnecessarily sticking to the traditional Huet and Lévy's matching dags [Huet and Lévy, 1979, 1992] or Strandh's index trees [Strandh, 1989]. We have shown that, in CSs, such a basic redefinition of definitional trees permits a more compact representation which can be used to substantially improve the evaluation of input expressions.

Inspired in the fact that inductively sequential TRSs are forward branching, we have developed an incremental description of needed narrowing which can be thought of as the functional logic counterpart of Strandh's algorithm for the incremental reduction of terms [Strandh, 1989]. By this means, we have implicitly demonstrated that (incremental) Curry implementations fit the original (nonincremental) semantics of Curry. We have also proved that our incremental needed narrower preserves the optimality properties and is more efficiently implementable than the original procedure thanks to its intrinsic incremental nature. We have shown that the proposed refinements can significantly improve the execution of functional logic languages based on needed narrowing. In particular, we have shown that they can be advantageously applied to optimize the implementation of the basic Curry machinery.

Compiling functional logic programs into another high-level language for which efficient implementations exist is a well known technique for the efficient implementation of integrated languages [Hanus, 1994]. Compilation into Prolog is the most popular option [Bosco *et al.*, 1989; Loogen *et al.*, 1993; Hanus, 1995a; Antoy and Hanus, 2000]. Prolog-based implementations implicitly reap the benefits of incrementality thanks to Prolog's SLD resolution mechanism. Unfortunately, no formal demonstration that these kinds of implementations fit the original (nonincremental) semantics of Curry had been provided before. By having proved the strong equivalence between the standard Curry semantics and the incremental one, our work can be thought of as a formal warranty that compiling functional logic programs into Prolog code is not only simple and practical but also reliable. A similar argument applies to those implementations based on abstract narrowing machines [Kuchen *et al.*, 1990], which also enjoy the incrementality implicitly which is embedded into standard compilation techniques.

On the other hand, programmers working on other implementation techniques can also rely on the present work for achieving incremental evaluation safely. Although it cannot compete with an implementation based on the compilation into low-level (abstract) machine code, our interpreter written in Prolog demonstrates that all optimizations are effective and could generate new insights for further developments in this field.

Appendix A Benchmarks Code

ackermann

```
data Nat = Z | S Nat

ackermann n = ack (S (S Z)) n

ack Z n = S n
ack (S m) Z = ack m (S Z)
ack (S m) (S n) = ack m (ack (S m) n)
```

fibonacci

```
data Nat = Z | S Nat

sum Z y = y
sum (S x) y = S (sum x y)

fib Z = S Z
fib (S x) = S x
fib (S (S x)) = sum (fib (S x)) (fib x)
```

iter

```
data Nat = Z | S Nat

iter Z f x = x
iter (S y) f x = f (iter y f x)

sub1 (S Z) = Z
sub1 (S (S Z)) = S Z
sub1 (S (S (S Z))) = S (S Z)
sub1 (S (S (S (S x)))) = S (S (S x))
```

last

```
append eval flex
append [] y = y
append (x:r) y = (x:append r y)

last l | append _ [_x] =:= l = _x
```

horseman

```
data Nat = Z | S Nat

int2nat   ::  Int -> Nat
int2nat n = if n<=Z then Z else S (int2nat (n-1))

add Z     n = n
add (S m) n = S (add m n)

horseman m h heads feet = heads =:= add m h & feet =:= add (add m m) (add (add h h) (add h h))
```

quicksort

```
-- quicksort using higher-order functions:
qsort ::  [Int] -> [Int]
qsort [] = []
qsort (x:l) = qsort (filter (<x) l) ++ x :  qsort (filter (>=x) l)
```

mergesort

```
-- merge sort:  sorting two list by merging the sorted first
-- and second half if the list
firsthalf xs = take (length xs `div` 2) xs

secondhalf xs = drop (length xs `div` 2) xs

sort ::  ([a] -> [a] -> [a] -> Constraint) -> [a] -> [a] -> Constraint
sort merge xs ys =
   if length xs < 2 then ys =:= xs
   else sort merge (firsthalf xs) _us
        & sort merge (secondhalf xs) _vs
        & (merge _us _vs ys)

intMerge ::  [Int] -> [Int] -> [Int] -> Constraint
intMerge [] ys zs = zs =:= ys
intMerge (x:xs) [] zs = zs =:= x:xs
intMerge (x:xs) (y:ys) zs =
   if (x > y) then intMerge (x:xs) ys _us & zs =:= y:_us
              else intMerge xs (y:ys) _vs & zs =:= x:_vs
```

Chapter 5

Natural Rewriting and Narrowing

This chapter provides a refinement of the demandedness notion associated to (weakly) outermost-needed rewriting and narrowing. Namely, we refine the demandedness notion associated to outermost-needed rewriting and narrowing including their weakly extensions. We introduce an extension of outermost-needed rewriting called *natural rewriting* and we extend it to narrowing. In Section 5.1, we motivate the convenience to refine outermost-needed rewriting and narrowing together with their weakly extensions. In Section 5.2, we present *natural rewriting*. Intuitively, natural rewriting always reduces the most often demanded position in a term. We demonstrate its optimality properties w.r.t. the length of the reduction sequences and we show that for the class of inductively sequential CSs, natural rewriting behaves even better than outermost-needed rewriting in the avoidance of failing computations. Regarding inductively sequential CSs, we introduce a bigger class of CSs called *inductively sequential preserving* where natural rewriting and narrowing preserve optimality for sequential parts of the program. In Section 5.3, we extend natural rewriting to narrowing with similar achievements. Section 5.4 summarizes our contributions and provides some related work. We also present a prototype interpreter of natural rewriting and narrowing.

A short version of this chapter appeared in [Escobar, 2003a,b]

5.1 Introduction

A challenging problem in modern programming languages is the discovery of sound and complete evaluation strategies which are 'optimal' w.r.t. some efficiency criterion (typically the number of evaluation steps and the avoidance of infinite, failing or redundant derivations) and which are easily implementable.

For orthogonal term rewriting systems (TRSs), Huet and Lévy's *needed rewriting* is optimal [Huet and Lévy, 1992]. However, automatic detection of needed redexes is difficult (or even impossible) and Huet and Lévy defined the notion of 'strong sequentiality', which provides a formal basis for the mechanization of sequential, normalizing rewriting computations [Huet and Lévy, 1992]. In [Huet and Lévy, 1992], Huet and Lévy defined the (computable) notion of strongly needed redexes and showed that, for the (decidable) class of strongly sequential orthogonal TRSs, the steady reduction of strongly needed redexes is normalizing. When strongly sequential TRSs are restricted to constructor systems (CSs), we obtain the class of inductively sequential CSs (see [Hanus *et al.*, 1998b]). Intuitively, a CS is inductively sequential if there exists some branching selection structure in the rules. Sekar and Ramakrishnan provided *parallel needed reduction* [Sekar and Ramakrishnan, 1993] as the extension of Huet and Lévy needed rewriting for non-inductively sequential CSs (namely, almost orthogonal CSs).

A sound and complete rewrite strategy for the class of inductively sequential CSs is *outermost-needed rewriting* [Antoy, 1992]. The extension to narrowing is called *outermost-needed narrowing* [Antoy *et al.*, 2000]. Its optimality properties and the fact that inductively sequential CSs are a subclass of strongly sequential programs explains why outermost-needed narrowing has become useful in functional logic programming as the functional logic counterpart of Huet and Lévy's strongly needed reduction. *Weakly outermost-needed rewriting* [Antoy *et al.*, 1997] is defined for non-inductively sequential CSs. The extension to narrowing is called *weakly outermost-needed narrowing* [Antoy *et al.*, 1997] and is considered as the functional logic counterpart of Sekar and Ramakrishnan's parallel needed reduction. See Chapter 3.

Unfortunately, weakly outermost-needed rewriting and narrowing do not work appropriately on non-inductively sequential CSs.

Example 5.1 *Consider Berry's program [Sekar and Ramakrishnan, 1993] where* `T` *and* `F` *are constructor symbols and* `X` *is a variable:*

```
B(T,F,X) = T     B(F,X,T) = T     B(X,T,F) = T
```

This CS is not inductively sequential since there is no branching selection structure in the rules. Although the CS is not inductively sequential, some terms can be reduced sequentially; where 'to be reduced sequentially' is understood as the property of reducing only positions which are unavoidable (or "needed") when attempting to obtain a normal form [Huet and Lévy, 1992]. For instance, the term `B(B(T,F,T),B(F,T,T),F)` *has a unique 'optimal' rewrite sequence which achieves its associated normal form* T*:*

$$\mathtt{B(B(T,F,T),}\underline{\mathtt{B(F,T,T)}}\mathtt{,F)} \to \underline{\mathtt{B(B(T,F,T),T,F)}} \to \mathtt{T}$$

However, weakly outermost-needed rewriting is not optimal since, besides the previous optimal sequence, the sequence

$$\mathtt{B(}\underline{\mathtt{B(T,F,T)}}\mathtt{,B(F,T,T),F)} \to \mathtt{B(T,}\underline{\mathtt{B(F,T,T)}}\mathtt{,F)} \to \underline{\mathtt{B(T,T,F)}} \to \mathtt{T}$$

is also obtained. The reason is that weakly outermost-needed rewriting partitions the CS into inductively sequential subsets $\mathcal{R}_1 = \{\mathtt{B(X,T,F) = T}\}$ *and* $\mathcal{R}_2 = \{\mathtt{B(T,F,X) = T}, \mathtt{B(F,X,T) = T}\}$ *in such a way that the first step of the former (optimal) rewriting sequence is obtained w.r.t. subset* $\mathcal{R}_1$ *whereas the first step of the latter (useless) rewriting sequence is obtained w.r.t. subset* $\mathcal{R}_2$*.*

Note that the problem also occurs in weakly outermost-needed narrowing. For instance, term B(X,B(F,T,T),F) *has the optimal narrowing sequence:*

$$\mathtt{B(X,}\underline{\mathtt{B(F,T,T)}}\mathtt{,F)} \leadsto_{id} \underline{\mathtt{B(X,T,F)}} \leadsto_{id} \mathtt{T}$$

whereas the non-optimal narrowing sequence produced by weakly outermost-needed narrowing is:

$$\mathtt{B(X,}\underline{\mathtt{B(F,T,T)}}\mathtt{,F)} \leadsto_{\{\mathtt{X}\mapsto\mathtt{T}\}} \underline{\mathtt{B(T,T,F)}} \leadsto_{id} \mathtt{T}$$

As [Sekar and Ramakrishnan, 1993] argued, strongly sequential systems have been widely used in programming languages and violations are not frequent. However, it is a cumbersome restriction and its relaxation is quite useful in some contexts. It would be convenient for programmers that such a restriction could be relaxed while optimal evaluation is preserved for strongly sequential parts of a program. The availability of optimal and effective evaluation strategies without such a restriction could encourage programmers to formulate non-inductively sequential programs, which could still be executed in an optimal way.

Example 5.2 *Consider the problem of coding the rather simple inequality*

$$x + y + z + w \leqslant h$$

for natural numbers x, y, z, w*, and* h*. A first (and naïve) approach could be to define an inductively sequential program:*

```
solve(X,Y,Z,W,H) = (((X + Y) + Z) + W) ⩽ H
0 ⩽ N = True               0 + N = N
s(M) ⩽ 0 = False           s(M) + N = s(M + N)
s(M) ⩽ s(N) = M ⩽ N
```

Consider the (auxiliary) factorial function $n!$*. The program is efficient for term*[1] $t_1 =$ solve(1,Y,0,10!,0) *since* False *is returned in* 5 *rewrite steps. However, it is not*

[1] Natural numbers 1,2,... are used as shorthand for s(s(... s(0))), where s is applied 1, 2, ... times.

very efficient for terms $t_2 =$ `solve(10!,0,1,0+1,0)` *or* $t_3 =$ `solve(10!,0+1,0+1,W,1)` *since several rewrite steps should be performed on* `10!` *before realizing that both terms rewrite to* `False`.

When a more effective program is desired for some input terms, an ordinary solution is to include some assertions by hand into the program while preserving determinism of computations. For instance, if terms t_1, t_2, *and* t_3 *are part of such input terms of interest, we could obtain the following program*

```
solve(X,s(Y),s(Z),W,s(H)) = solve(X,Y,s(Z),W,H)
solve(X,0,s(Z),W,s(H)) = solve(X,0,Z,W,H)
solve(s(X),Y,0,W,s(H)) = solve(X,Y,0,W,H)
solve(0,s(Y),0,W,s(H)) = solve(0,Y,0,W,H)
solve(0,0,0,0,H) = True
solve(s(X),Y,0,W,0) = False
solve(X,0,s(Z),W,0) = False
solve(0,s(Y),Z,0,0) = False
solve(X,s(Y),s(Z),s(W),0) = False
```

Note that this program is not inductively sequential but orthogonal, and some terms can still be reduced (or narrowed) sequentially. For instance, when we consider the previous term t_3 *for rewriting:*

$$\begin{array}{l}
\texttt{solve(10!,0+s(0),}\underline{\texttt{0+s(0)}}\texttt{,W,s(0))} \\
\quad \to \texttt{solve(10!,}\underline{\texttt{0+s(0)}}\texttt{,s(0),W,s(0))} \\
\quad \to \underline{\texttt{solve(10!,s(0),s(0),W,s(0))}} \\
\quad \to \underline{\texttt{solve(10!,0,s(0),W,0)}} \to \texttt{False}
\end{array}$$

or when we consider the term $t_4 =$ `solve(X,Y,0+2,W,1)` *for narrowing*[2]*:*

$$\begin{array}{l}
\texttt{solve(X,Y,}\underline{\texttt{0+s(s(0))}}\texttt{,W,s(0))} \\
\quad \leadsto_{id} \underline{\texttt{solve(X,Y,s(s(0)),W,s(0))}} \\
\quad \leadsto_{\{\texttt{Y}\mapsto\texttt{0}\}} \underline{\texttt{solve(X,0,s(0),W,0)}} \mid \leadsto_{\{\texttt{Y}\mapsto\texttt{s(Y')}\}} \underline{\texttt{solve(X,Y',s(s(0)),W,0)}} \\
\quad \leadsto_{id} \texttt{False} \mid \leadsto_{\{\texttt{Y'}\mapsto\texttt{0}\}} \texttt{False} \mid \leadsto_{\{\texttt{Y'}\mapsto\texttt{s(Y'')},\texttt{W}\mapsto\texttt{0},\texttt{X}\mapsto\texttt{0}\}} \texttt{False} \\
\qquad \mid \leadsto_{\{\texttt{Y'}\mapsto\texttt{s(Y'')},\texttt{W}\mapsto\texttt{s(W')}\}} \texttt{False}
\end{array}$$

Indeed, note that even in the case we add the following rule `solve(X,s(Y),0,W,0) = False`*, which makes the program almost orthogonal,* t_3 *is still sequentially rewritten and* t_4 *sequentially narrowed.*

Modern (multiparadigm) programming languages apply computational strategies which are based on some notion of demandedness of a position in a term by a rule (see [Antoy and Lucas, 2002] for a survey discussing this topic). Programs in these languages are commonly modeled by left-linear CSs and computational strategies take

[2] The symbol '|' is used as a separator to denote different narrowing steps from a similar term.

advantage of this constructor condition (see [Alpuente *et al.*, 1997; Moreno-Navarro and Rodríguez-Artalejo, 1992]). Furthermore, the semantics of these programs normally promotes the computation of values (or constructor head-normal forms) rather than head-normal forms. For instance, (weakly) outermost-needed rewriting and narrowing consider the constructor condition though some refinement is still possible, as shown by the following example.

Example 5.3 *Consider the following TRS from [Arts and Giesl, 2001] defining the symbol* $\div$*, which encodes the division function between natural numbers.*

```
0 ÷ s(N) = 0                      M − 0 = M
s(M) ÷ s(N) = s((M−N)÷s(N))  s(M) − s(N) = M−N
```

Consider the term t = `10!÷0`*, which is a (non-constructor) head-normal form. Outermost-needed rewriting forces*[3] *the reduction of the first argument and evaluates* `10!`*, which is useless. The reason is that outermost-needed rewriting uses a data structure called* definitional tree *which encodes the branching selection structure existing in the rules without testing whether the rules associated to each branch could ever be matched to the term or not. A similar problem occurs when narrowing the term* `X ÷ 0`*, since variable* `X` *is instantiated to* `0` *or* `s`*. However, no instantiation is really necessary.*

5.2 Natural Rewriting

Some rewrite strategies proposed to date are called demand driven and can be classified as call-by-need. Loosely speaking, *demandedness* is understood as follows: if possible, a term t is evaluated at the top; otherwise, some arguments of the root of t are (recursively) evaluated if they might promote the application of a rewrite rule at the top, i.e. if a rule "demands" the evaluation of these arguments. In [Antoy and Lucas, 2002], Antoy and Lucas show that modern functional (logic) languages include evaluation strategies which consider elaborated definitions of 'demandedness' which are, in fact, related [Alpuente *et al.*, 1997; Antoy *et al.*, 1997, 2000; Fokkink *et al.*, 2000; Lucas, 2001a, 2002a; Moreno-Navarro and Rodríguez-Artalejo, 1992].

In the following, we provide a notion of demandedness which gives us a useful algorithm to calculate the rules and "demanded" positions necessary in a rewriting sequence. In order to define in later sections natural rewriting in terms of narrowing, we consider also variables as disagreeing positions.

[3] Indeed, this specific fact depends on how a real implementation builds definitional trees. However, the problem cannot be avoided by selecting an alternative definitional tree when more than one exists; see Example 5.39 below.

Definition 5.4 (Disagreeing positions) *Given terms* $t, l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$*, we let* $\mathcal{P}os_{\neq}(t,l) = minimal_{\leq}(\{p \in \mathcal{P}os(t) \cap \mathcal{P}os_{\mathcal{F}}(l) \mid root(l|_p) \neq root(t|_p)\})$.

Definition 5.5 (Demanded positions) *We define the set of* demanded *positions of term* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *w.r.t.* $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *(a lhs of a rule defining* $root(t)$*), i.e. the set of (positions of) maximal disagreeing subterms as:*

$$DP_l(t) = \begin{cases} \mathcal{P}os_{\neq}(t,l) & \textit{if } \mathcal{P}os_{\neq}(t,l) \cap \mathcal{P}os_{\mathcal{C}}(t) = \varnothing \\ \varnothing & \textit{otherwise} \end{cases}$$

and the set of demanded positions of t *w.r.t. TRS* $\mathcal{R}$ *as* $DP_{\mathcal{R}}(t) = \cup\{DP_l(t) \mid l \to r \in \mathcal{R} \wedge root(t) = root(l)\}$.

Example 5.6 *Consider again Example 5.2 and terms* t_3 *and* t_4*. We have* $DP_{\mathcal{R}}(t_3) = DP_{\mathcal{R}}(t_4) = \{1,2,3,4\}$ *since (identically for* t_4*):*

$$\begin{array}{l}
DP_{\texttt{solve(X,s(Y),s(Z),W,s(H))}}(t_3) = \{2,3\} \\
DP_{\texttt{solve(X,0,s(Z),W,s(H))}}(t_3) = \{2,3\} \\
DP_{\texttt{solve(s(X),Y,0,W,s(H))}}(t_3) = \{1,3\} \\
DP_{\texttt{solve(0,s(Y),0,W,s(H))}}(t_3) = \{1,2,3\} \\
DP_{\texttt{solve(0,0,0,0,H)}}(t_3) = \{1,2,3,4\} \\
DP_{\texttt{solve(s(X),Y,0,W,0)}}(t_3) = \varnothing \\
DP_{\texttt{solve(X,0,s(Z),W,0)}}(t_3) = \varnothing \\
DP_{\texttt{solve(0,s(Y),Z,0,0)}}(t_3) = \varnothing \\
DP_{\texttt{solve(X,s(Y),s(Z),s(W),0)}}(t_3) = \varnothing
\end{array}$$

Note that the restriction of disagreeing positions to positions with non-constructor symbols (*defined* symbols in the case of rewriting) disables the evaluation of subterms which could never help to produce a redex (see [Alpuente *et al.*, 1997; Moreno-Navarro and Rodríguez-Artalejo, 1992]). Outermost-needed rewriting also considers this constructor-based condition though some refinement is still possible, as shown in Example 5.3.

Antoy and Lucas introduced the idea that some rewriting strategies select ‘popular’ demanded positions over the available set [Antoy and Lucas, 2002]. Here, we formalize a notion of popularity of demanded positions which makes the difference by counting the number of times a position is demanded by some rule and then we select the most (frequently) demanded positions. This information, i.e. the number of times a position is demanded, is crucial and is managed by using multisets instead of sets for representing demanded positions.

We redefine the set of demanded positions of t w.r.t. TRS $\mathcal{R}$, $DP_{\mathcal{R}}(t)$, as follows.

Definition 5.7 (Demanded positions) *We define the multiset of demanded positions of a term* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *w.r.t. TRS* $\mathcal{R}$ *as* $DP_{\mathcal{R}}(t) = \oplus\{DP_l(t) \mid l \to r \in \mathcal{R} \wedge root(t) = root(l)\}$ *where* $M_1 \oplus M_2$ *is the union of multisets* M_1 *and* M_2.

Example 5.8 *Continuing Example 5.6, we have* $DP_{\mathcal{R}}(t_3) = DP_{\mathcal{R}}(t_4) = \{4, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3\}$.

Now, one can simply think in a rewriting strategy which collects all demanded positions within a term w.r.t. the TRS and selects those which are "the most often demanded". For instance, we would select position 3 for terms t_3 and t_4 in Example 5.8. This idea works well when using inductively sequential CSs but does not work correctly with non-inductively sequential CSs, as the following example shows.

Example 5.9 *Consider the following parallel-or TRS* $\mathcal{R}$ *of Example 3.7:*

```
True ∨ X = True
X ∨ True = True
False ∨ False = False
```

Consider the term $t =$ `(True ∨ False)∨(True ∨ False)`. *Positions* 1 *and* 2 *of* t *would be selected as reducible by natural rewriting since* $DP_{\mathcal{R}}(t) = \{1, 2, 1, 2\}$. *However, if we consider the following slightly different TRS* $\mathcal{R}'$*:*

```
True ∨ X = True
X ∨ True = True
False ∨ X = X
```

only position 1 *would be reducible by natural rewriting since it is the most often demanded position, i.e.* $DP_{\mathcal{R}'}(t) = \{1, 2, 1\}$. *Thus, not all the necessary sequences are performed.*

The solution is to consider the set of most often demanded positions which covers all the rules involved in the term evaluation.

Definition 5.10 (Demanded positions) *We define the set of demanded positions of a term* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *w.r.t. TRS* $\mathcal{R}$ *as*

$$DP^{\mathtt{m}}_{\mathcal{R}}(t) = \{p \in DP_{\mathcal{R}}(t) \mid \exists l \in L(\mathcal{R}).p \in DP_l(t) \text{ and } \forall q \in DP_{\mathcal{R}}(t) : p <_{DP_{\mathcal{R}}(t)} q \Rightarrow l|_q \in \mathcal{X}\}$$

where $x <_M y$ *denotes that the number of occurrences of* x *in the multiset* M *is less than the number of occurrences of* y.

Note that the set of most often demanded positions is always included in $DP^{\mathtt{m}}_{\mathcal{R}}(t)$; in symbols $maximal_{<_{DP_{\mathcal{R}}(t)}}(DP_{\mathcal{R}}(t)) \subseteq DP^{\mathtt{m}}_{\mathcal{R}}(t)$.

Example 5.11 *Consider Example 5.9 again. Now we have* $DP^{\mathtt{m}}_{\mathcal{R}'}(t) = \{1, 2\}$ *since position* 1 *alone does not cover the rule* `X ∨ True = True`.

The following definition establishes the strategy used in natural rewriting for selecting demanded positions.

Definition 5.12 (Natural rewriting) *Given a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and a TRS $\mathcal{R}$, $\mathfrak{m}(t)$ is defined as the smallest set satisfying*

$$\mathfrak{m}(t) \ni \begin{cases} (\Lambda, l \to r) & \text{if } l \to r \in \mathcal{R} \text{ and } l \leq t \\ (p.q, l \to r) & \text{if } p \in DP^{\mathfrak{m}}_{\mathcal{R}}(t) \text{ and } (q, l \to r) \in \mathfrak{m}(t|_p) \end{cases}$$

We consider $\mathfrak{m}$ simply as a function returning positions if the rules selected for reduction are not relevant or directly deducible from the context. Roughly speaking, given a term, natural rewriting always reduces, in a non-deterministic way, the term itself or the most often demanded positions within the term covering all eventually applicable rules. We say term t reduces by *natural rewriting* to term s, denoted by $t \overset{\mathfrak{m}}{\to}_{\{p,l\to r\}} s$ (or simply $t \overset{\mathfrak{m}}{\to} s$) if $(p, l \to r) \in \mathfrak{m}(t)$.

Example 5.13 *Continuing Example 5.8. It is easy to see that $\mathfrak{m}(t_3) = \{3\}$ since positions 1, 2, and 4 are also demanded, but the redex at position 3 is the most often demanded (which also covers all eventually applicable rules). Then, the only possible natural rewriting step is:*

$$\texttt{solve(10!,0+s(0),}\underline{\texttt{0+s(0)}}\texttt{,W,s(0))} \overset{\mathfrak{m}}{\to} \texttt{solve(10!,0+s(0),s(0),W,s(0))}$$

Example 5.14 *Again, consider the TRS $\mathcal{R}'$ and the term t in Example 5.9. Example 5.11 showed that $\mathfrak{m}(t) = \{1, 2\}$ and, thus, there exist two possible natural rewriting steps:*

$$\underline{\texttt{(True} \vee \texttt{False)}} \vee \texttt{(True} \vee \texttt{False)} \overset{\mathfrak{m}}{\to} \texttt{True} \vee \texttt{(True} \vee \texttt{False)}$$
$$\texttt{(True} \vee \texttt{False)} \vee \underline{\texttt{(True} \vee \texttt{False)}} \overset{\mathfrak{m}}{\to} \texttt{(True} \vee \texttt{False)} \vee \texttt{True}$$

From now on, we investigate the properties of natural rewriting.

5.2.1 Neededness

For speaking about neededness of a natural rewriting step, we make use of the notions of an inductively sequential TRS and a definitional tree introduced in Section 3.2. However, we extend the notion of an inductively sequential symbol to terms. This is the key idea of the chapter for speaking about (optimal) sequential evaluation.

We denote the maximal set of rules which can eventually be matched to a term t (possibly after some evaluation steps) by $match_t(\mathcal{R}) = \{l \in L(\mathcal{R}) \mid DP_l(t) \neq \varnothing \text{ or } l \leq t\}$.

Definition 5.15 (Inductively sequential term) *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear CS and $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$. We say t is* inductively sequential *if for all $p \in \mathcal{P}os_{\mathcal{D}}(t)$, there exists a definitional tree $\mathcal{T}_p$ with root pattern π such that $\pi \leq t|_p$ and leaves of $\mathcal{T}_p$ are all and only instances of $match_{t|_p}(\mathcal{R})$.*

Note that inductive sequentiality of a term is preserved by instantiation since, given a term t with variables and a substitution σ, $match_{\sigma(t)}(\mathcal{R}) \subseteq match_t(\mathcal{R})$.

Example 5.16 *Consider Example 5.2. Terms t_3 and t_4 are inductively sequential since*

$$\begin{aligned} match_{t_3}(\mathcal{R}) = match_{t_4}(\mathcal{R}) = \{&\texttt{solve(X,s(Y),s(Z),W,s(H))}, \\ &\texttt{solve(X,0,s(Z),W,s(H))}, \texttt{solve(s(X),Y,0,W,s(H))}, \\ &\texttt{solve(0,s(Y),0,W,s(H))}, \texttt{solve(0,0,0,0,H)}\} \end{aligned}$$

and the definitional tree associated to both terms is depicted in Figure 5.17.

Example 5.18 *On the other hand, consider Example 5.2 again but term* $t' =$ `solve(10!,0+1,0+1,0+1,0+1)`. *This term is not inductively sequential since all lhs's for the symbol* `solve` *could be (potentially) matched and a rule partition is necessary, as shown in Figure 5.19. Then, optimal evaluation is not ensured. For instance, natural rewriting returns positions* 3 *and* 5 *as the most popular candidates for reduction, i.e.* $\mathtt{m}(t') = \{3, 5\}$ *where*

$$\begin{aligned}
&DP_{\texttt{solve(X,s(Y),s(Z),W,s(H))}}(t') = \{2,3,5\} \\
&DP_{\texttt{solve(X,0,s(Z),W,s(H))}}(t') = \{2,3,5\} \\
&DP_{\texttt{solve(s(X),Y,0,W,s(H))}}(t') = \{1,3,5\} \\
&DP_{\texttt{solve(0,s(Y),0,W,s(H))}}(t') = \{1,2,3,5\} \\
&DP_{\texttt{solve(0,0,0,0,H)}}(t') = \{1,2,3,4\} \\
&DP_{\texttt{solve(s(X),Y,0,W,0)}}(t') = \{1,3,5\} \\
&DP_{\texttt{solve(X,0,s(Z),W,0)}}(t') = \{2,3,5\} \\
&DP_{\texttt{solve(0,s(Y),Z,0,0)}}(t') = \{1,2,4,5\} \\
&DP_{\texttt{solve(X,s(Y),s(Z),s(W),0)}}(t') = \{2,3,4,5\}
\end{aligned}$$

The (computable) notion of an inductively sequential term is based on the idea of a definitional tree whose root pattern is not necessarily of the form $f(x_1, \ldots, x_k)$ where $x_1, \ldots, x_k$ are different variables. The following optimality results are consequences of Definition 5.15 above. It is worth noting that the Definition 2.7 of root-neededness of a rewriting step is only meaningful for orthogonal TRSs, since all redexes of a term correspond to non-overlapping rules and their descendants cannot be destroyed. However, this notion is well-defined when we consider inductively sequential terms because all redexes of an inductively sequential term correspond to non-overlapping rules.

Proposition 5.20 *Let $\mathcal{R}$ be a left-linear CS. If $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is an inductively sequential term, then for all $p \in \mathcal{P}os_{\mathcal{D}}(t)$, left-hand sides in $match_{t|_p}(\mathcal{R})$ are non-overlapping.*

- solve(X,Y,$\boxed{\texttt{Z}}$,W,s(0))
  - solve($\boxed{\texttt{X}}$,Y,0,W,s(0))
    - solve(0,$\boxed{\texttt{Y}}$,0,W,s(0))
      - solve(0,0,0,$\boxed{\texttt{W}}$,s(0))
        - solve(0,0,0,0,s(0))
      - solve(0,s(Y),0,W,s(0))
    - solve(s(X),Y,0,W,s(0))
  - solve(X,$\boxed{\texttt{Y}}$,s(Z),W,s(0))
    - solve(X,0,s(Z),W,s(0))
    - solve(X,s(Y),s(Z),W,s(0))

Figure 5.17: Definitional tree for the terms `solve(10!,0+1,0+1,W,1)` and `solve(X,Y,0+2,W,1)`.

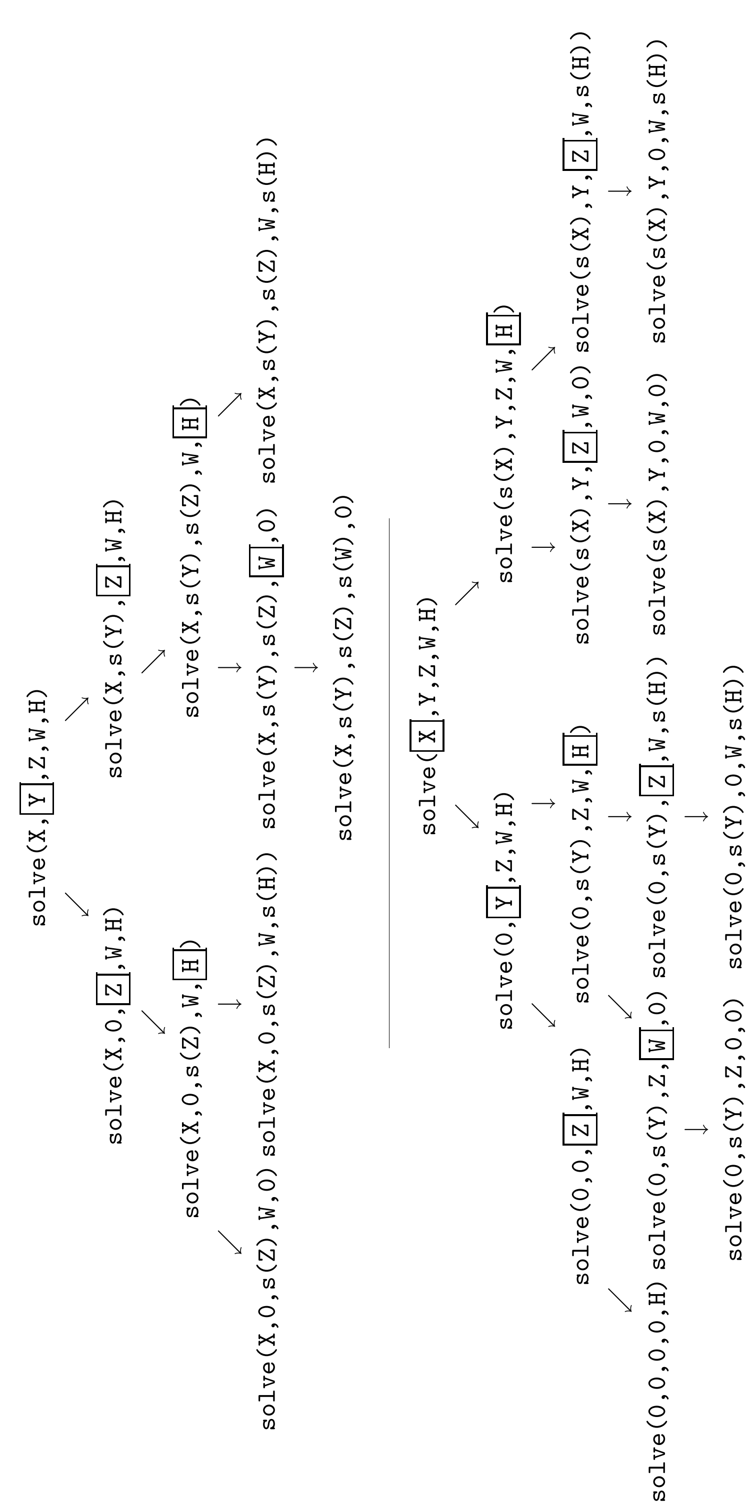

Figure 5.19: A partition of definitional trees for the symbol solve.

Proof. Immediate by the induction property of a definitional tree. □

The following lemma provides an essential result about left-linear CS which ensures that redexes cannot be disabled be further reductions.

Lemma 5.21 *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear CS. Let $t, s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $l \in L(\mathcal{R})$ such that $\exists\sigma : t = \sigma(l)$. If $t \stackrel{>\Lambda}{\longrightarrow}^* s$ then $\exists\sigma' : s = \sigma'(l)$.*

Proof. Standard, see for example [Middeldorp, 1997]. □

The following theorem shows that natural rewriting always performs root-needed rewriting steps when inductively sequential terms are considered.

Theorem 5.22 (Optimality) *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear CS and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be an inductively sequential term which is not root-stable. Then each $t \stackrel{\mathrm{m}}{\rightarrow}_{\{p,l\to r\}} s$ is root-needed.*

Proof. We prove it by induction on the position p.

- ($p = \Lambda$) By Proposition 5.20, $match_t(\mathcal{R})$ contains only one rule. By Lemma 5.21, redex t cannot be disabled by reduction in inner positions. Hence, since t is not root-stable, Λ is root-needed.

- ($p > \Lambda$) First, note that since t is not root-stable, $root(t) \in \mathcal{D}$. By Definition 5.15, there exists a definitional tree $\mathcal{T}$ and a branch pattern $\pi \in \mathcal{T}$ with inductive position p_π such that $\pi < t$ and $root(t|_{p_\pi}) \in \mathcal{D}$. Again by Definition 5.15, all leaves of $\mathcal{T}$ (i.e. all lhs's in $match_t(\mathcal{R})$) have some non-variable symbol at position p_π and, hence, $p_\pi \in DP^{\mathrm{m}}_{\mathcal{R}}(t)$. Moreover, since $root(t|_{p_\pi}) \in \mathcal{D}$, a position below or equal to p_π must be reduced to make t a redex.

  On the other hand, since $\mathcal{R}$ is a CS, we have that $\exists p_1, p_2$ s.t. $p = p_1.p_2$, $root(t|_{p_1}) \in \mathcal{D}$, and $\forall p'.(\Lambda < p' < p_1), root(t|_{p'}) \in \mathcal{C}$. Now, since subterm at position p_pi must be examined, $\forall p' \in DP_{\mathcal{R}}(t|_{p_\pi})$, $p_1.p_2 >_{DP_{\mathcal{R}}(t|_{p_\pi})} p_\pi.p'$. But this means that the number of rules demanding $p_1.p_2$ is greater or equal to the number of rules demanding $p_\pi.p'$ and, hence, all lhs's in $match_t(\mathcal{R})$ must have a non-variable symbol at position p_1 (i.e. $p_1 \in DP^{\mathrm{m}}_{\mathcal{R}}(t)$). Thus, a position below or equal to p_2 must be reduced to make t a redex. Finally, the conclusion follows by induction over position p_2. □

Note that since we do not restrict ourselves to orthogonal TRSs, later reduction steps in any derivation starting from the natural rewriting step may not hold the property of root-neededness. However, the idea behind this result is that even if a later step is not root-needed, the natural rewriting step (from the inductively sequential term) is still unavoidable to obtain a root-stable form, as shown by the following example.

Example 5.23 *Consider the following non-orthogonal TRS $\mathcal{R}$:*

$$\begin{array}{ll} \mathtt{f} \to \mathtt{g(h,h)} & \mathtt{g(x,0)} \to \mathtt{g(0,0)} \\ \mathtt{h} \to \mathtt{0} & \mathtt{g(0,x)} \to \mathtt{0} \end{array}$$

Term $\mathtt{f}$ *is inductively sequential since there is only one applicable rule. Natural rewriting performs the step* $\mathtt{f} \stackrel{\mathtt{m}}{\to} \mathtt{g(h,h)}$*. This natural rewriting step is root-needed since it is the unique reduction on term* $\mathtt{f}$ *which leads to the root-stable form* $\mathtt{0}$*. Here, the idea behind Theorem 5.22 shows up since positions* 1 *and* 2 *of* $\mathtt{g(h,h)}$ *are not root-needed because each one of them can be avoided depending on the chosen rule for* $\mathtt{g}$*. Indeed, the term* $\mathtt{g(h,h)}$ *is not inductively sequential because it cannot be given a definitional tree without a partition into inductively sequential subsets of rules.*

The condition $\mathcal{R}$ must be a constructor system is necessary, as shown by the following example.

Example 5.24 *Let* $\mathcal{R}$ *be the following non-CS:*

$$\mathtt{f(g)} \to \mathtt{f(g)} \qquad \mathtt{g} \to \mathtt{0}$$

Let $t = \mathtt{f(g)}$ *be an inductively sequential term. We have that* $\mathtt{m}(t) = \{\Lambda\}$*, but position* Λ *is not root-needed since* $\mathtt{f(\underline{g})} \to \mathtt{f(0)}$ *is the unique root-normalizing sequence.*

On the other hand, left-linearity is also necessary.

Example 5.25 *Consider the CS:*

$$\mathtt{f(x,x)} \to \mathtt{f(x,x)} \quad \mathtt{g} \to \mathtt{h} \quad \mathtt{h} \to \mathtt{a} \quad \mathtt{h} \to \mathtt{b}$$

The term $t = \mathtt{f(g,g)}$ *is inductively sequential and natural rewriting strategy yields* $\mathtt{m}(t) = \{\Lambda\}$*. However, position* Λ *is not root-needed since the following sequence, which does not reduce* Λ*, is root-normalizing:*

$$\mathtt{f(\underline{g},g)} \to \mathtt{f(\underline{h},g)} \to \mathtt{f(a,\underline{g})} \to \mathtt{f(a,b)}$$

Note that term $\mathtt{f(g,g)}$ *is inductively sequential while term* $\mathtt{f(h,g)}$ *is not since there exist overlapping rules for the symbol* $\mathtt{h}$*.*

We are also concerned with correctness and completeness of natural rewriting. We are able to prove that normal forms of $\stackrel{\mathtt{m}}{\to}$ are root-stable forms.

Theorem 5.26 (Correctness) *Let* $\mathcal{R} = (\mathcal{F}, R)$ *be a left-linear TRS. If* $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *is a* $\stackrel{\mathtt{m}}{\to}$*-normal form, then* s *is root-stable.*

Proof. By structural induction. If $s \in \mathcal{X}$ or $root(s) \in \mathcal{C}$, it is trivial. Consider $root(s) = f \in \mathcal{D}$ and $ar(f) = 0$. By Definition 5.12, there is no lhs rooted by symbol f. Consider $root(s) = f \in \mathcal{D}$ and $ar(f) > 0$. Since s is a $\stackrel{\mathtt{m}}{\to}$-normal form, $\mathtt{m}(s) = \varnothing$. By Definition 5.12 and left-linearity, s is not a redex and either $DP^{\mathtt{m}}_{\mathcal{R}}(t) = \varnothing$ or $\mathtt{m}(s|_p) = \varnothing$ for $p \in DP^{\mathtt{m}}_{\mathcal{R}}(t)$. By induction hypothesis, $t|_p$ is root-stable for $p \in DP^{\mathtt{m}}_{\mathcal{R}}(t)$ and no reduction on these positions can make s a redex. Thus, the only case is $DP^{\mathtt{m}}_{\mathcal{R}}(t) = \varnothing$. Then, $DP_{\mathcal{R}}(s) = \varnothing$ and by Definition 5.5, for all $l \in L(\mathcal{R})$ s.t. $root(l) = root(s)$, $\mathcal{P}os_{\neq}(s,l) \cap \mathcal{P}os_{\mathcal{C}}(s) \neq \varnothing$. And this implies lhs l is no applicable to s because of a constructor conflict. Hence, the conclusion follows. $\square$

The following example shows that left-linearity condition is necessary.

Example 5.27 *Consider the following TRS:*

$\mathtt{f(x,x)} \to \mathtt{0} \qquad \mathtt{g} \to \mathtt{0}$

For the term $t = \mathtt{f(g,0)}$, we obtain that $\mathtt{m}(t) = \varnothing$ since t is not a redex and there is no demanded position. However, t is not root-stable since there exists the following rewrite sequence: $\mathtt{f(\underline{g},0)} \to \underline{\mathtt{f(0,0)}} \to \mathtt{0}$.

And we can prove completeness of natural rewriting w.r.t. root-stable forms. First, we give some helpful definitions.

An elementary multiderivation simultaneously contracts a set U of disjoint occurrences of redexes (written $t \stackrel{U}{\to} s$) [Huet and Lévy, 1992]. We write $t \twoheadrightarrow\!\!\!\!\!\!\!\shortmid\;\; s$ when the information about the concrete contracted occurrences is not relevant. An elementary derivation $A : t \stackrel{p}{\to} t'$ can be seen as an elementary multiderivation $A : t \stackrel{\{p\}}{\to} t'$. Similarly, a multiderivation $A : t \xrightarrow{\{p_1,\ldots,p_n\}} t'$ can be simulated by a set of elementary derivations $A : t \stackrel{p_1}{\to} t_1 \cdots t_{n-1} \stackrel{p_n}{\to} t'$.

Proposition 5.28 *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear TRS and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. If $t \stackrel{p}{\to}_{l\to r} t' \stackrel{p'}{\to}_{l'\to r'} t''$, $p \notin \mathtt{m}(t)$, and $p' \in \mathtt{m}(t')$, then there exists $\exists s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ s.t. $t \stackrel{\mathtt{m}}{\to}_{\{p', l'\to r'\}} s \twoheadrightarrow\!\!\!\!\!\!\!\shortmid\;\; t''$.*

Proof. Immediate. Since $p \notin \mathtt{m}(t)$ and $\mathcal{R}$ is a left-linear CS, then this means that position p is under a variable position in all possible rules involved in the evaluation of t; i.e., $p \in \mathcal{P}os_{\mathcal{X}}(l')$. Hence, this rewriting step at position p can be reproduced using a multiderivation at those positions in r' of the variable occurring at position p in l'. $\square$

Theorem 5.29 (Completeness) *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear CS and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. If $t \to^* s$ and s is root-stable, then $\exists s' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ s.t. $t \stackrel{\mathtt{m}}{\to}^* s'$, $root(s') = root(s)$ and $s' \xrightarrow{>\Lambda}^* s$.*

Proof. (Sketch) By induction on the length n of the rewriting sequence $t \to^n s$. If $n = 0$, then it is straightforward since $t = s$. If $n > 0$, then we have $t \stackrel{p}{\to}_{l\to r} t' \to^{n-1} s$. Now we consider two cases. If $p \in \mathtt{m}(t)$, then this rewriting step is also produced by natural rewriting and the conclusion follows by induction hypothesis. If $p \notin \mathtt{m}(t)$, then by Proposition 5.28 we can move this rewriting step to the end of the derivation. Then, by induction hypothesis and the fact that a multiderivation can be seen as a normal derivation, the conclusion follows. $\square$

Here, Examples 5.24 and 5.27 can be used to show the necessity of CS and left-linearity conditions, respectively.

Besides the properties of natural rewriting w.r.t. inductively sequential terms, in the following section, we formalize a class of TRSs where natural rewriting preserves optimality for sequential parts of the program.

5.2.2 Preserving Sequentiality

We provide a class of TRSs called *inductively sequential preserving* with some interesting normalization properties. This class of programs is based on Definition 5.15 above.

Definition 5.30 (Inductively sequential preserving) *We say a left-linear CS $\mathcal{R}$ is* inductively sequential preserving *if for all rule $l \to r \in \mathcal{R}$, right-hand side r is an inductively sequential term.*

Example 5.31 *The program of Example 5.1 is trivially inductively sequential preserving. Consider now the program of Example 5.2. This CS is also inductively sequential preserving since rhs* `solve(X,0,Z,W,H)` *induces sequential evaluation in position* 3*, rhs* `solve(X,Y,0,W,H)` *induces sequential evaluation in position* 1*, and rhs* `solve(X,Y,s(Z),W,H)` *induces sequential evaluation in position* 2 *or* 5*. On the other hand, consider the CS of Example 5.25. This CS is clearly non-inductively sequential preserving since rhs* `h` *is a non-inductively sequential term because there exist overlapping rules for the defined symbol* `h`*.*

This class of TRSs is larger than the class of inductively sequential CSs.

Corollary 5.32 *Inductively sequential left-linear CSs are trivially inductively sequential preserving left-linear CSs.*

The following result asserts that natural rewriting steps in an *inductively sequential preserving* CS preserve inductive sequentiality of terms.

Theorem 5.33 (Preserving sequentiality) *Let $\mathcal{R} = (\mathcal{F}, R)$ be an inductively sequential preserving left-linear CS and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be an inductively sequential term. If $t \overset{\mathtt{m}}{\to}_{\{p,l\to r\}} s$, then s is inductively sequential.*

Proof. Immediate by Definition 5.15. Let $t|_p = \sigma(l)$, since t is inductively sequential, $\sigma(x)$ for $x \in \mathcal{V}ar(r)$ is also inductively sequential and thus $\sigma(r)$ is also inductively sequential. □

Specifically, this last theorem provides the following interesting property about natural rewriting.

Theorem 5.34 (Root-normalization) *Reduction of inductively sequential terms within the class of inductively sequential preserving left-linear CSs is root-normalizing.*

Proof. Immediate by Theorems 5.33 and 5.22. □

Example 5.35 *Consider Example 5.2 again. Since Example 5.31 shows CS of Example 5.2 is inductively sequential preserving and Example 5.16 proves term t_3 is inductively sequential, rewriting term t_3 through natural rewriting is optimal and provides appropriate root-stable forms.*

Now, we clarify the precise relation between outermost-needed rewriting and natural rewriting. Roughly speaking, natural rewriting is conservative w.r.t. outermost-needed rewriting for inductively sequential left-linear CSs.

5.2.3 (Weakly) Outermost-needed rewriting

Here, we relate natural rewriting with the weakly outermost-needed rewriting introduced in Definition 3.12. For instance, weakly outermost-needed rewriting behaves worse than natural rewriting with the program for symbol `solve` of Example 5.2.

Example 5.36 *Again, consider Example 5.2 and the definitional trees for the partition of the symbol* `solve` *in Figure 5.19. According to Definition 3.12, we have that* $\varphi(t_3, \mathcal{T}_{\texttt{solve}}) = \{1, 2\}$ *and that subterm* `10!` *is rewritten. This is because* `solve` *has two definitional trees, positions* 1 *and* 2 *are the inductive positions of the root branch pattern of each definitional tree, and term t_3 has defined symbols at these positions.*

The following theorem establishes that natural rewriting and outermost-needed rewriting coincide for inductively sequential CSs.

Theorem 5.37 *Let $\mathcal{R}$ be an inductively sequential left-linear CS and $t, s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that t is not root-stable. Then, $t \stackrel{\mathtt{n}}{\to}_{\{p,l\to r\}} s$ if and only if $t \stackrel{\mathtt{m}}{\to}_{\{p,l\to r\}} s$.*

Proof. By induction on position p.

- ($p = \Lambda$) There exists $l \to r \in \mathcal{R}$ and σ s.t. $t = \sigma(l)$ and $s = \sigma(r)$. Since $\mathcal{R}$ is orthogonal, $l \to r$ is the unique rule included in $match_t(\mathcal{R})$. Since $\mathcal{R}$ is inductively sequential, there exists a definitional tree for symbol $root(t)$ and a leaf pattern π such that $\pi \leq t$, i.e. π corresponds to lhs l. Hence, the conclusion follows.

- ($p > \Lambda$) First, since t is not a head normal form and $p > \Lambda$, $DP^{\mathtt{m}}_{\mathcal{R}}(t) \neq \varnothing$. Since $\mathcal{R}$ is inductively sequential, there exists a definitional tree $\mathcal{T}$ for symbol $root(t)$ and a branch pattern $\pi \in \mathcal{T}$ with inductive position p_π such that $\pi < t$ and

$$
\begin{tikzcd}[column sep=tiny]
& \boxed{\mathtt{M}} \div \mathtt{N} \arrow[dl] \arrow[dr] & \\
\mathtt{0} \div \boxed{\mathtt{N}} \arrow[d] & & \mathtt{s(M)} \div \boxed{\mathtt{N}} \arrow[d] \\
\mathtt{0} \div \mathtt{s(N)} & & \mathtt{s(M)} \div \mathtt{s(N)}
\end{tikzcd}
\qquad
\begin{tikzcd}[column sep=tiny]
& \mathtt{M} \div \boxed{\mathtt{N}} \arrow[d] & \\
& \boxed{\mathtt{M}} \div \mathtt{s(N)} \arrow[dl] \arrow[dr] & \\
\mathtt{0} \div \mathtt{s(N)} & & \mathtt{s(M)} \div \mathtt{s(N)}
\end{tikzcd}
$$

Definitional tree (a) Definitional tree (b)

Figure 5.38: The two possible definitional trees for the symbol $\div$.

$root(t|_{p_\pi}) \in \mathcal{D}$. Moreover, by Definition 5.12 and $\mathcal{T}_{root(t)} = \{\mathcal{T}\}$, $\exists p' : p = p_\pi.p'$. By definition, all leaves of subtree rooted by π have some non-variable symbol at position p_π. Hence, $p_\pi.DP^{\mathtt{m}}_{\mathcal{R}}(t|_{p_\pi}) \subseteq DP^{\mathtt{m}}_{\mathcal{R}}(t)$ and the conclusion follows by induction over position p'.

□

Note that the condition of non root-stability of t is necessary since natural rewriting performs a more exhaustive matching test.

Example 5.39 *Consider again Example 5.3 where the two possible definitional trees associated to symbol $\div$ are depicted in Figure 5.38. According to Definition 5.10, there is no demanded position for the term $t = \mathtt{10!} \div \mathtt{0}$, i.e. $DP^{\mathtt{m}}_{\mathcal{R}}(t) = \varnothing$. However, if definitional tree (a) of Figure 5.38 is used by outermost-needed rewriting, it cannot detect, according to Definition 3.12, that t is a head-normal form and forces the reduction of the first argument since it blindly follows the selected definitional tree.*

It is worth noting that if definitional tree (b) of Figure 5.38 is selected for use, this concrete behavior does not happen. However, the reader should note that the problem persists for other terms independently of the chosen definitional tree, i.e. it cannot be solved by simply selecting an alternative definitional tree. For instance, if we reconsider the set of constructor symbols in the program as $\mathcal{C} = \{\mathtt{0}, \mathtt{s}, \mathtt{pred}\}$ (instead of simply $\{\mathtt{0}, \mathtt{s}\}$), subterm $\mathtt{10!}$ in the term $t_1 = \mathtt{10!} \div \mathtt{pred(0)}$ is (uselessly) reduced when using definitional tree (a), whereas subterm $\mathtt{10!}$ in the term $t_2 = \mathtt{pred(0)} \div \mathtt{10!}$ is (uselessly) reduced when using definitional tree (b). Note that both terms t_1 and t_2 are detected as head-normal forms by natural rewriting.

In the following, we consider the complementary problem of whether natural rewriting is also optimal for non-inductively sequential terms.

5.2.4 Non-sequentiality

We are able to provide some results regarding reduction on non-inductively sequential terms. First, Definition 2.7 of neededness of a rewriting step is not applicable when non-inductively sequential CSs are considered (see [Sekar and Ramakrishnan, 1993; Middeldorp, 1997]) and the notion of a necessary set of redexes of Definition 2.5 is used instead. Natural rewriting always computes a root-necessary set of redexes.

Theorem 5.40 *Let $\mathcal{R}$ be a left-linear CS and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Then, $\mathfrak{m}(t)$ is a root-necessary set of redexes.*

Proof. Immediate by Definition 5.10 since all rules which could be involved in the evaluation of t are considered by demanded positions in $DP^{\mathfrak{m}}_{\mathcal{R}}(t)$. □

Note that for almost orthogonal TRSs, repeated contraction of root-necessary sets of redexes results in a root-stable form, whenever the latter exists (see [Middeldorp, 1997]). On the other hand, natural rewriting is more conservative than weakly outermost-needed rewriting, as shown by the following theorem.

Theorem 5.41 *Let $\mathcal{R}$ be a left-linear CS and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Then, $\mathfrak{m}(t) \subseteq \varphi(t, \mathcal{T}_{root(t)})$.*

Proof. Term t should be not inductively sequential, otherwise it is trivial by Theorem 5.37. Since t is not inductively sequential, we can make a partition of inductively sequential subsets such that positions in $\mathfrak{m}(t)$ are the inductive positions of the root nodes of each inductively sequential subset. □

From now on, we extend natural rewriting to narrowing with similar achievements.

5.3 Natural Narrowing

The extension of natural rewriting to narrowing is not difficult and stems from the idea that narrowing differs from rewriting only in the instantiation step prior to rewriting.

Definition 5.42 (Natural narrowing) *Given a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and a TRS $\mathcal{R}$, $\mathfrak{mn}(t)$ is defined as the smallest set satisfying*

$$\mathfrak{mn}(t) \ni \begin{cases} (\Lambda, id, l{\to}r) & \textit{if } l \to r \in \mathcal{R} \textit{ and } l \leq t \\ (p.q, \theta, l{\to}r) & \textit{if } p \in DP^{\mathfrak{m}}_{\mathcal{R}}(t) \cap \mathcal{P}os_{\mathcal{D}}(t) \textit{ and } (q, \theta, l{\to}r) \in \mathfrak{mn}(t|_p) \\ (p, \theta \circ \sigma, l{\to}r) & \textit{if } p \in DP^{\mathfrak{m}}_{\mathcal{R}}(t) \textit{ s.t. } t|_p = x \in \mathcal{X}, c \in \mathcal{C}, \sigma(x) = c(\overline{w}), \\ & \quad \overline{w} \textit{ are fresh variables, and } (p, \theta, l{\to}r) \in \mathfrak{mn}(\sigma(t)) \end{cases}$$

Intuitively, instantiations are performed only at those variables which are part of the most demanded positions, whereas reductions are only performed when no variable at the most demanded positions is available. We say term t narrows by *natural narrowing* to term s at position p using substitution σ, denoted by $t \overset{\mathfrak{m}}{\leadsto}_{\{p, l \to r, \sigma\}} s$ (or simply $t \overset{\mathfrak{m}}{\leadsto}_{\sigma} s$), if $(p, \sigma, l \to r) \in \mathfrak{mn}(t)$.

Example 5.43 *Consider TRS $\mathcal{R}$ and term t_4 of Example 5.2. Natural narrowing performs only the following narrowing sequences:*

$$\begin{array}{l}
\texttt{solve(X,Y,}\underline{\texttt{0+s(s(0))}}\texttt{,W,s(0))}\\
\quad \overset{\text{m}}{\rightsquigarrow}_{id}\ \underline{\texttt{solve(X,Y,s(s(0)),W,s(0))}}\\
\quad \overset{\text{m}}{\rightsquigarrow}_{\{\texttt{Y}\mapsto\texttt{0}\}}\ \underline{\texttt{solve(X,0,s(0),W,0)}} \ \mid\ \overset{\text{m}}{\rightsquigarrow}_{\{\texttt{Y}\mapsto\texttt{s(Y')}\}}\ \underline{\texttt{solve(X,Y',s(s(0)),W,0)}}\\
\quad \overset{\text{m}}{\rightsquigarrow}_{id}\ \texttt{False} \ \mid\ \overset{\text{m}}{\rightsquigarrow}_{\{\texttt{Y'}\mapsto\texttt{0}\}}\ \texttt{False} \ \mid\ \overset{\text{m}}{\rightsquigarrow}_{\{\texttt{Y'}\mapsto\texttt{s(Y'')},\texttt{W}\mapsto\texttt{0},\texttt{X}\mapsto\texttt{0}\}}\ \texttt{False}\\
\qquad\qquad \mid\ \overset{\text{m}}{\rightsquigarrow}_{\{\texttt{Y'}\mapsto\texttt{s(Y'')},\texttt{W}\mapsto\texttt{s(W')}\}}\ \texttt{False}
\end{array}$$

Note that these narrowing sequences coincide with the 'optimal' narrowing sequences associated to t_4 in Example 5.2.

5.3.1 Neededness and sequentiality

In Section 3.1, it is shown that neededness of a narrowing step does not stem in a simple way from its counterpart of rewriting (Definition 2.3) since some unifiers may not be strictly unavoidable to achieve a normal form. According to Definition 3.1, a narrowing step is needed no matter which unifiers might be used later in the derivation.

In the following, we prove that optimality w.r.t. inductively sequential terms, similar to Theorem 5.22, can be established for natural narrowing.

Theorem 5.44 (Optimality) *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear CS and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be an inductively sequential term which is not root-stable. Then each natural narrowing step $t \overset{\text{m}}{\rightsquigarrow}_{\{p,\sigma,l\to r\}} s$ is outermost-needed.*

Proof. Similar to proof of Theorem 5.22. □

Following the explanations given for Theorem 5.22, the notion of neededness of a narrowing step of Definition 3.1 is only meaningful for orthogonal TRSs though it is well-defined when we consider inductively sequential terms (see Proposition 5.20). Moreover, the idea behind the previous theorem is again similar to that of natural rewriting (see Example 5.23).

Example 5.45 *Consider Example 5.43. All the steps in the narrowing sequences from t_4 are outermost-needed since they are unavoidable when attempting to achieve the normal form* `False`*. Specifically, the initial rewrite step is necessary to achieve the normal form and the instantiation on* `Y` *is unavoidable if a normal form is desired.*

Moreover, root-stability of Theorem 5.26 is extensible to natural narrowing.

Theorem 5.46 (Correctness) *Let $\mathcal{R} = (\mathcal{F}, R)$ be a left-linear CS, If $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is a $\overset{\text{m}}{\rightsquigarrow}$-normal form, then s is root-stable.*

Proof. Similar to Theorem 5.26. □

Note that the CS condition is necessary here (in contrast to Theorem 5.26) due to variable instantiation, as shown by the following example.

Example 5.47 *Consider the following non-constructor system where* $\mathcal{C} = \{\mathtt{0}\}$ *and* $\mathcal{D} = \{\mathtt{f}, \mathtt{g}\}$*:*

$\mathtt{f(g)} \rightarrow \mathtt{0}$

Natural narrowing cannot narrow the term `f(x)` *to the root-stable form* `0` *since it is not able to build the substitution* $\{\mathtt{x} \mapsto \mathtt{g}\}$*.*

We can also prove completeness of natural narrowing w.r.t. root-stable forms.

Theorem 5.48 (Completeness) *Let* $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ *be a left-linear CS and* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$*. If* $t \leadsto^* s$ *and* s *is root-stable, then* $\exists s' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *s.t.* $t \overset{\mathtt{m}}{\leadsto}{}^* s'$*,* $root(s') = root(s)$ *and* $s' \overset{>\Lambda}{\leadsto}{}^* s$*.*

Proof. Similar to Theorem 5.29 but using the extension of multiderivation to the narrowing case given in [Antoy *et al.*, 2000]. □

Similarly, we provide the extension of root-normalization within the class of inductively sequential preserving CSs of Theorem 5.33 and 5.34.

Theorem 5.49 (Preserving sequentiality) *Let* $\mathcal{R}$ *be an inductively sequential preserving left-linear CS and* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *be an inductively sequential term. If* $t \overset{\mathtt{m}}{\leadsto}_{\{p,\sigma,l\to r\}} s$*, then* s *is inductively sequential.*

Proof. Similar to Theorem 5.33 since inductive sequentiality is closed under substitutions of inductively sequential terms. □

Theorem 5.50 (Root-normalization) *Narrowing of inductively sequential terms within the class of inductively sequential preserving left-linear CSs is root-normalizing.*

Proof. Immediate by Theorems 5.49 and 5.44. □

Now, we clarify the precise relation between outermost-needed narrowing and natural narrowing.

5.3.2 (Weakly) Outermost-needed narrowing

Similarly to Section 5.2.3, we relate natural narrowing with the weakly outermost-needed narrowing strategy introduced in Definition 3.9. For instance, weakly outermost-needed narrowing behaves worse than natural narrowing with the program for symbol `solve` of Example 5.2.

Example 5.51 *Consider the TRS and term t_4 of Example 5.2. Also consider the definitional trees for the partition of the symbol* `solve` *in Figure 5.19. Outermost-needed narrowing yields:*

$$\begin{aligned}\lambda(\texttt{solve(X,Y,0+2,W,1)}, \mathcal{T}_{\texttt{solve}}) = \{&(3,\{\texttt{Y}\mapsto\texttt{0}\}),(3,\{\texttt{Y}\mapsto\texttt{s(Y')}\}),\\ &(3,\{\texttt{X}\mapsto\texttt{0},\texttt{Y}\mapsto\texttt{0}\}),(3,\{\texttt{X}\mapsto\texttt{0},\texttt{Y}\mapsto\texttt{s(Y')}\}),\\ &(3,\{\texttt{X}\mapsto\texttt{s(X')}\})\}\end{aligned}$$

since `solve` *has two definitional trees, positions* 1 *and* 2 *are the inductive positions of the root branch pattern of each definitional tree, and subsequent matching or instantiation is made until a branch whose inductive position is* 3 *is reached.*

The following theorem establishes that natural narrowing is conservative w.r.t. outermost-needed narrowing for inductively sequential CSs.

Theorem 5.52 *Let $\mathcal{R}$ be an inductively sequential left-linear CS and $t, s \in \mathcal{T}(\mathcal{F},\mathcal{X})$ such that t is not root-stable. Then, $t \overset{\mathtt{n}}{\rightsquigarrow}_{\{p,\sigma,l\to r\}} s$ if and only if $t \overset{\mathtt{m}}{\rightsquigarrow}_{\{p,\sigma,l\to r\}} s$.*

Proof. Similar to proof of Theorem 5.37 but considering that instantiations are made when all rules in the definitional tree associated to t have a constructor symbol at position p. □

Again, condition of non root-stability is necessary (as in Theorem 5.37) since natural narrowing performs a more exhaustive unification process (see Examples 5.3 and 5.39).

5.3.3 Non-sequentiality

And finally, we are able to provide some results regarding narrowing of non-inductively sequential terms. Here, in contrast to Section 5.2.4, there is no a common notion of neededness of a narrowing step when non-inductively sequential CSs are considered. Thus, we only prove that natural narrowing is more conservative than weakly outermost-needed narrowing.

Theorem 5.53 *Let $\mathcal{R}$ be a left-linear CS and $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$. Then, $\mathtt{mn}(t) \subseteq \lambda(t, \mathcal{T}_{root(t)})$.*

Proof. Similar to proof of Theorem 5.41. □

5.4 Conclusions

We have provided a suitable refinement of the demandedness notion associated to *outermost-needed rewriting*. On behalf of this new demandedness notion, we have introduced an improvement of *outermost-needed rewriting* called *natural rewriting*, and we have extended it to narrowing.

We summarize the contributions:

1. We have shown that natural rewriting is a conservative extension of outermost-needed rewriting (see Theorem 5.37).

2. Similarly, we have shown that natural narrowing is a conservative extension of outermost-needed narrowing (see Theorem 5.52).

3. Moreover, natural rewriting behaves even better than outermost-needed rewriting for the class of inductively sequential left-linear CSs in the avoidance of failing computations (see Theorem 5.37 and Examples 5.3 and 5.39).

4. Also in the case of narrowing (see Theorem 5.52 and Examples 5.3 and 5.39).

5. On the other hand, correctness and completeness of natural rewriting and narrowing are guaranteed (see Theorems 5.26, 5.29, 5.46, and 5.48).

Moreover, we defined a new class of TRSs called *inductively sequential preserving* where natural rewriting and narrowing preserve optimality for sequential parts of a program. We provide the following contributions w.r.t. this new notion:

1. This new class of TRSs is based on the extension of the notion of inductive sequentiality from defined function symbols to terms (see Theorems 5.22 and 5.44).

2. Moreover, an interesting result is obtained w.r.t. this new class of programs: reduction (narrowing) of inductively sequential terms within the class of inductively sequential preserving left-linear CSs is root-normalizing (see Theorems 5.34 and 5.50).

3. Note that the formalization extends in general to left-linear CSs (see Theorems 5.41 and 5.53).

Finally, we have developed a prototype implementation of natural rewriting and natural narrowing (called Natur) which is publicly available at

`http://www.dsic.upv.es/users/elp/soft.html`

This prototype has been used to test the examples presented in this chapter. The interpreter is implemented in Haskell (using ghc 5.04.2) and accepts OBJ programs, which have a syntax that is similar to the syntax of modern functional (logic) programs, e.g. the functional syntax used in Haskell or Curry. Note that the prototype does not build any definitional tree and directly codifies Definitions 5.12 and 5.42.

As future work, we plan to develop better implementation techniques, which would be similar to the incremental finite state automata approach of [Sekar and Ramakrishnan, 1993]. On the other hand, in this chapter, we have not addressed complexity of natural rewriting and narrowing (or the prototype implementation). We have left this for future work.

5.4.1 Related work

This work is framed into the formal basis established by [Huet and Lévy, 1992] and [Sekar and Ramakrishnan, 1993] for optimal sequential reduction. It is worth noting that the idea of most often demanded positions can be recovered in a general way from the normalization algorithm $FindNS$ of [Sekar and Ramakrishnan, 1993], though the idea was hinted at [Antoy and Lucas, 2002].

On the one hand, note that a computable and optimal rewrite strategy based on tree automata exists for left-linear right-ground TRSs [Durand and Middeldorp, 1997]. This technique is able to manage Example 5.1 but cannot perform sequential evaluation in Example 5.2. On the other hand, there exist normalizing, sequential rewrite strategies for almost orthogonal TRSs [Antoy and Middeldorp, 1996]. However, efficiency (w.r.t. the number of reduction steps) is not their main objective and optimal evaluation is not ensured. For instance, when rewriting the following term $t =$ `solve(10!,0,0+10,0,1)` with the TRS of Example 5.2, the sequential strategy of [Antoy and Middeldorp, 1996] would select the subterm `10!` for reduction instead of subterm `0+10` since both subterms have the same maximal depth and leftmost redexes are preferred.

Finally, a simple but informal clarifying idea arises as a conclusion of this work: natural rewriting and narrowing dynamically select the rules involved in the evaluation of a term in order to ensure optimality. Indeed, natural rewriting and narrowing refine outermost-needed rewriting and narrowing, respectively, as strategies which simulate the dynamical computation of the definitional tree associated to each term found in an evaluation sequence. However, the reader should remember that no definitional tree is used in the formal definition of natural rewriting and narrowing, and this behavior is obtained using only the simple but natural and powerful idea of "most often demanded positions".

Part II

Strategies: Syntactic Annotations

Chapter 6

Syntactic Strategy Annotations

This chapter introduces different basic notions about strategy annotations in programming languages and recalls the most representative formalisms dealing with syntactic strategy annotations for restricting the rewriting relation. Section 6.1 briefly introduces the use of strategy annotations in programming languages. In Section 6.2, we recall *context-sensitive rewriting* (*CSR* [Lucas, 1998a]), which uses syntactic replacement maps for restricting the rewriting relation. In Section 6.3, we recall *on-demand rewriting* (*ODR* [Lucas, 2001a]), the natural extension of context-sensitive rewriting for dealing with on-demand syntactic annotations. In Section 6.4, we recall reduction strategies with *local strategies* (also called E-evaluation maps) [Nagaya, 1999], which is the evaluation model of Maude [Clavel *et al.*, 1996], OBJ2 [Futatsugi *et al.*, 1985], and OBJ3 [Goguen *et al.*, 2000]. In Section 6.5, we recall the extension of reduction strategies with E-evaluation maps to on-demand strategy annotations, which is the evaluation model of CafeOBJ [Futatsugi and Nakagawa, 1997]. Finally, in Section 6.6, we recall *lazy rewriting* (*LR* [Fokkink *et al.*, 2000]), which inspired the proposal of on-demand strategies such as on-demand E-evaluation [Futatsugi and Nakagawa, 1997] or *ODR* [Lucas, 2001a].

6.1 Introduction

In recent years, strategies whose eager/lazy behavior can be programmed by the user has deserved great interest. A number of (eager) programming languages (e.g., ASF+SDF [van Deursen *et al.*, 1996], CafeOBJ [Futatsugi and Nakagawa, 1997], ELAN [Borovanský *et al.*, 1998], Maude [Clavel *et al.*, 1996], OBJ2 [Futatsugi *et al.*, 1985], OBJ3 [Goguen *et al.*, 2000], or Stratego [Visser, 1999]) permit (to some extent) the explicit specification of strategies aimed at controlling the execution of programs; see [Visser, 2001] for a recent survey. Other (lazy) programming languages such as Clean [van Eekelen *et al.*, 1991], Curry [Hanus *et al.*, 1995], Gofer [Jones, 1992], Standard ML [Milner *et al.*, 1997], or Haskell [Hudak and Fasel, 1992] have a predefined computational strategy whose exact behavior can be modified (to some extent) by means of program annotations. Several formalisms to support this eager/lazy programmable behavior have been recently developed by means of syntactic strategy annotations [Lucas, 1998a,c, 2002a; Nagaya, 1999; Visser, 2001; Dolstra, 2001].

6.1.1 Syntactic Annotations

Context-sensitive rewriting (*CSR*) [Lucas, 1998a] is a rewriting restriction which can be associated to every term rewriting system (TRS). Given a signature $\mathcal{F}$, a mapping $\mu : \mathcal{F} \to \mathcal{P}(\mathbb{N})$, called the *replacement map*, discriminates some argument positions $\mu(f) \subseteq \{1, \ldots, k\}$ for each k-ary symbol f. Given a function call $f(t_1, \ldots, t_k)$, the replacements are allowed on arguments t_i such that $i \in \mu(f)$ and are forbidden for the other argument positions. Syntactic strategy annotations are usually provided to (hopefully) avoid non-termination of programs.

Example 6.1 [Lucas, 2002a] *Consider the TRS $\mathcal{R}$:*

```
sel(0,cons(x,y)) → x
sel(s(x),cons(y,z)) → sel(x,z)
from(x) → cons(x,from(s(x)))
first(0,x) → []
first(s(x),cons(y,z)) → cons(y,first(x,z))
```

together with the following replacement map which indicates that reduction is allowed on all symbol arguments of the signature except the second argument of symbol `cons`

$$\mu(\mathtt{s}) = \mu(\mathtt{cons}) = \mu(\mathtt{from}) = \{1\} \quad \textit{and} \quad \mu(\mathtt{sel}) = \mu(\mathtt{first}) = \{1, 2\}.$$

The following derivation is allowed with CSR *under μ (we underline the redex which is contracted in each μ-*rewriting *step):*

$\mathtt{sel(s(0),}\underline{\mathtt{from(0)}}\mathtt{)}$

$\quad \to \underline{\mathtt{sel(s(0),cons(0,from(s(0))))}}$

$$\begin{array}{l}
\to\ \texttt{sel(0,}\underline{\texttt{from(s(0))}}\texttt{)}\\
\to\ \underline{\texttt{sel(0,cons(s(0),from(s(s(0)))))}}\ \to\ \texttt{s(0)}
\end{array}$$

However, the infinite (meaningless) derivation

$$\begin{array}{l}
\texttt{sel(s(0),}\underline{\texttt{from(0)}}\texttt{)}\\
\quad\to\ \texttt{sel(s(0),cons(0,}\underline{\texttt{from(s(0))}}\texttt{))}\\
\quad\to\ \texttt{sel(s(0),cons(0,cons(s(0),}\underline{\texttt{from(s(s(0)))}}\texttt{)))}\\
\quad\to\ \cdots
\end{array}$$

is avoided since $\mu(\texttt{cons}) = \{1\}$ *(the second argument of* 'cons' *cannot be rewritten). That is, the second reduction step is not allowed with* CSR*.*

The search for complete implementations of eager languages such as Lisp [McCarthy, 1960, 1978] originated the first proposals for the use of syntactic replacement restrictions in programming [Friedman and Wise, 1976; Henderson and Morris, 1976]. Since list processing is prominent in the design of Lisp, the authors of these works studied implementations where the list constructor operator *cons* did not evaluate its arguments, during certain stages of the computation (*lazy cons*). As a motivating example, Friedman and Wise propose the following Lisp definition for computing an infinite sequence of fractions (by evaluating expression `(terms 1)`):

$$\frac{1}{1},\ \frac{1}{4},\ \frac{1}{9},\ \ldots,\ \frac{1}{n^2},\ \ldots$$

whose partial sums converge to $\pi^2/6$ (see page 265 of [Friedman and Wise, 1976]):

```
(terms n) ≡ (cons (reciprocal (square n)) (terms (add1 n)))
```

Their idea is to *'violate the data type of LISP 1.0'* in such a way that *'the evaluation* (of, e.g., `(terms 1)`) *does not immediately diverge; i.e. it results in a node referencing two suspensions'* [Friedman and Wise, 1976].

Example 6.2 [Lucas, 2002a] *With the TRS:*

```
sqr(0)      → 0                          0 + x     → x
sqr(s(x))   → s(sqr(x)+dbl(x))           s(x) + y  → s(x+y)
dbl(0)      → 0
dbl(s(x))   → s(s(dbl(x)))
terms(n)    → cons(recip(sqr(n)),terms(s(n)))
```

together with $\mu(\texttt{cons}) = \varnothing$ *(or even* $\mu(\texttt{cons}) = \{1\}$*) and* $\mu(f) = \{1, \ldots, k\}$ *for any other* k*-ary symbol* f*, we are able to obtain the desired restriction.*

Moreover, by adding the two rules for function `first` of Example 6.1:

```
first(0,x)            → []
first(s(x),cons(y,z)) → cons(y,first(x,z))
```

we can obtain the first n terms of the series that approximates $\pi^2/6$ (as the result of expression `first(`n`,terms(1)))`, thus obtaining an arbitrary precision for the approximation. Here, as usual, n abbreviates `s`^{n}`(0)`.

Friedman and Wise also use replacement restrictions to provide alternative (more efficient) definitions to the logical connectives `and`, `or`. In fact, they implement their 'short-cut' definitions of these boolean operators using lazy cons in such a way that the evaluation is suspended after the first argument which evaluates to `false` (resp. `true`) if connective `and` (resp. `or`) is considered (see page 277 of [Friedman and Wise, 1976]). This is implemented (without using lists) with the TRS:

`and(false,x)` $\to$ `false`	`or(true,x)` $\to$ `true`
`and(true,x)` $\to$ `x`	`or(false,x)` $\to$ `x`

together with $\mu($`and`$) = \mu($`or`$) = \{1\}$.

Syntactic replacement restrictions have been *explicitly* included in the design of several (eager) programming languages. For instance, the so called *strategy annotations* have been used in the OBJ family of languages[1]: OBJ2 [Futatsugi *et al.*, 1985], OBJ3 [Goguen *et al.*, 2000], CafeOBJ [Futatsugi and Nakagawa, 1997], and Maude [Clavel *et al.*, 1996] to introduce *replacement restrictions* aimed at improving the efficiency of computations (by reducing the number of attempted matchings) or (hopefully) avoid nontermination. Local strategies or E-strategy maps (specified as sequences of integers associated to function arguments) are used in OBJ programs for guiding the *evaluation strategy* (abbr. E-strategy): when considering a function call $f(t_1, \ldots, t_k)$, only the arguments whose indices are present as *positive* integers in the local strategy for f are evaluated (following the specified ordering). If 0 is found, then the evaluation of f is attempted. Its formalization is based on Nagaya's thesis [Nagaya, 1999].

Example 6.3 *Consider the TRS of Example 6.1. The associated* OBJ *program is:*

```
obj SEL-FIRST is
  sorts Nat LNat .
  op 0     : -> Nat .
  op s     : Nat -> Nat .
  op nil   : -> LNat .
  op cons  : Nat LNat -> LNat [strat (1)] .
  op from  : Nat -> LNat .
  op sel   : Nat LNat -> Nat .
  op first : Nat LNat -> LNat .
  vars X Y : Nat . var Z : LNat .
  eq sel(s(X),cons(Y,Z)) = sel(X,Z) .
```

[1] From now on, by OBJ we mean OBJ2, OBJ3, CafeOBJ, or Maude.

```
    eq sel(0,cons(X,Z)) = X .
    eq from(X) = cons(X,from(s(X))) .
    eq first(0,Z) = nil .
    eq first(s(X),cons(Y,Z)) = cons(Y,first(X,Z)) .
  endo
```

which specifies an explicit *strategy annotation for the list constructor* `cons` *which disables replacements on the second argument while the rest of symbols follow the* default *strategy*[2]*. The same evaluation of term* `sel(s(0),from(0))` *in Example 6.1 is obtained with this program.*

The usefulness of strategy annotations has been demonstrated in practice: in [Futatsugi *et al.*, 1985] the authors remark that, due to their use in OBJ2 programs, *'the ratio between attempted matches and successful matches is usually around* 2/3*, which is really impressive'*. For instance, OBJ's *built-in* conditional operator has the following (implicit) strategy annotation[3] ([Futatsugi *et al.*, 1985], Section 4.4, [Goguen *et al.*, 2000], Section 2.4.4)

```
op if_then_else_fi : Bool Int Int -> Int [strat (1 0)]
```

which asks for evaluating the first argument until it is reduced, and then apply rules at the top (indicated by '`0`'). Reductions on the second or third arguments of calls to `if_then_else_fi` are never attempted.

Other eager programming languages such as ELAN [Borovanský *et al.*, 1998] incorporate the specification of syntactic replacement restrictions as an ingredient of the definition of more complex rewriting strategies which can be used to guide the evaluation of expressions.

In (*lazy*) functional programming, different kinds of syntactic annotations on the program (such as strictness annotations [Peyton-Jones, 1987], or global and local annotations [Plasmeijer and Eekelen, 1993]) have been introduced in order to drive local changes in the basic underlying lazy evaluation strategy and obtain more efficient executions [Burn, 1991; Mycroft and Norman, 1992; Mycroft, 1980; Plasmeijer and Eekelen, 1993; Peyton-Jones, 1987]. In these languages, constructor symbols are lazy, i.e., their arguments are not evaluated until needed. This permits structures that contain elements which, if evaluated, would lead to an error or fail to terminate [Hudak *et al.*, 1999]. Since there are a number of overheads in the implementation of this feature (see [Peyton-Jones, 1987]), lazy functional languages like Gofer [Jones,

[2] The shape of the *default* strategy depends on the language considered. For instance, in Maude, the default local strategy associated to a k-ary symbol f is $(1\ 2\ \cdots k\ 0)$, see [Eker, 2000]. This is the default strategy considered in this thesis.

[3] A more precise and general definition can be found in OBJ3's standard prelude, see Appendix D.3 of [Goguen *et al.*, 2000].

1992] and Haskell [Hudak *et al.*, 1992] allow for syntactic annotations on the arguments of datatype constructors, thus allowing an immediate evaluation.

Example 6.4 *The following definition in* Haskell*:*

```
data List a = Nil | Cons !a (List a)
```

declares a (polymorphic) type `List a` *whose binary data constructor* `Cons` *evaluates the first argument (of type* `a`*). This is specified by using the symbol '*`!`*' in the first argument of* `Cons`*.*

Other lazy functional languages, such as Clean [van Eekelen *et al.*, 1991; Nöcker *et al.*, 1991; Plasmeijer and Eekelen, 1993], allow for more general annotations.

Example 6.5 *The following specification*

```
if :: !Bool a a -> a
if True  x y = x
if False x y = y
```

is an annotated definition of the function `if` *which forces the evaluation of the first argument of each* `if` *call (notice the mark '*`!`*' in the type declaration of* `if`*; this is called a* global annotation *[Plasmeijer and Eekelen, 1993]).*

The use of annotations of this kind can be understood as follows [Plasmeijer and Eekelen, 1993]:

> *A given lazy strategy indicates whether an argument t_i of a function call $f(t_1, \ldots, t_k)$ must be evaluated. However, we overcome this rule by evaluating t_i (up to a head-normal form) if the i-th argument is annotated in the profile of f.*

Thus, annotations play a *secondary* role in the global execution mechanism: an underlying strategy is assumed.

Program annotations are usually obtained from some kind of strictness[4] analysis. Strictness analyses are usually costly as they involve fixpoint computations [Clack and Peyton-Jones, 1985; Mycroft, 1980]. In this case, the safety of this deviation of the main strategy is ensured because strictness analyses are derived from the semantics of the functional language. Sometimes, the programmer is allowed (but discouraged) to annotate the program by himself. In this case, however, there is no way to determine what kind of modification of the semantics or computational behavior is introduced by the annotations. In general, strictness information is not adequate for defining a

[4] Let $D_1, \ldots, D_k, D$ be ordered sets with least elements $\perp_1, \ldots, \perp_k, \perp$ respectively, expressing undefinedness. A mapping $f : D_1 \times \cdots \times D_k \to D$ is said to be strict in its i-th argument if $f(d_1, \ldots, \perp_i, \ldots, d_k) = \perp$ for all $d_1 \in D_1, \ldots, d_k \in D_k$.

replacement map (for *CSR*) as there is no underlying (lazy) strategy which can be altered according to strictness annotations (see [Lucas, 1998a]), i.e., these annotations play a secondary role in the computation.

6.1.2 On-demand Syntactic Annotations

Unfortunately, using rewriting restrictions may cause *incompleteness*, i.e., normal forms of input expressions could be unreachable by such restricted computation. The limits of using strategy annotations regarding correctness and completeness of computations are discussed in [Lucas, 2001a, 2002b; Nakamura and Ogata, 2001; Ogata and Futatsugi, 2000]: the obvious problem is that the absence of some indices in the strategies can have a negative impact in the ability of such strategies to compute normal forms.

When the restriction of rewriting disables the consecution of normal forms and head normal forms (i.e. the restricted rewrite relation is not root-normalizing –see Section 2.5 for the definition of root-normalizing strategies–), the rather intuitive notion of demanded evaluation of an argument of a function call arises as a possible solution to this problem [Eker, 2000; Fokkink *et al.*, 2000; Goguen *et al.*, 2000; Lucas, 2001a; Nakamura and Ogata, 2001; Ogata and Futatsugi, 2000]. The following example explains the concept of demanded evaluation.

Example 6.6 *Consider the following* OBJ *program (borrowed from [Nakamura and Ogata, 2001]):*

```
obj 2ND is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (1)] .
  op 2nd  : LNat -> Nat .
  op from : Nat -> LNat .
  vars X Y : Nat . var Z : LNat .
  eq 2nd(cons(X,cons(Y,Z))) = Y .
  eq from(X) = cons(X,from(s(X))) .
endo
```

which codifies the function returning the second element of a list. The OBJ *evaluation of term* `2nd(from(0))` *is given by the sequence:*

$\texttt{2nd(}\underline{\texttt{from(0)}}\texttt{)} \rightarrow \texttt{2nd(cons(0,from(s(0))))}$

The evaluation stops here since reductions on the second argument of `cons` *are disallowed. Note that we cannot apply the rule defining* `2nd` *because the subterm* `from(s(0))` *should be further reduced. Thus, a further step is 'demanded' (by the rule of* `2nd`*) in order to obtain the desired outcome:*

`2nd(cons(0,from(s(0))))` $\rightarrow$ `2nd(cons(0,cons(s(0),from(s(s(0))))))`

Now, we do not *need to reduce the second argument of the inner occurrence of* `cons` *anymore since reducing at the root position yields the final value (i.e. position* 1.2.2 *is not demanded by the rule for* `2nd`*):*

`2nd(cons(0,cons(s(0),from(s(s(0))))))` $\rightarrow$ `s(0)`

On-demand syntactic strategy annotations are proposed to indicate those arguments that should be evaluated only 'on-demand', where the 'demand' is an attempt to match an argument term with the left-hand side of a rewrite rule [Eker, 2000; Goguen *et al.*, 2000; Ogata and Futatsugi, 2000]. In the computational model of [Fokkink *et al.*, 2000], there is a replacement map which indicates those arguments allowed for evaluation (as in *CSR*) and the rest of arguments are understood as allowed for evaluation 'on-demand'. In the *on-demand rewriting* [Lucas, 2001a], there exists a replacement map μ as in *CSR* and an additional replacement map μ_D denotes those arguments that should be evaluated 'on-demand'. And, in [Nakamura and Ogata, 2001; Ogata and Futatsugi, 2000], *negative* indices are proposed to indicate those arguments that should be evaluated 'on-demand' (in contrast to positive indices).

Example 6.7 *For instance, in [Nakamura and Ogata, 2001] the authors suggest* `(1 -2)` *as the "apt" local strategy for constructor* `cons` *in Example 6.6. Thus, we write:*

```
obj 2ND is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (1 -2)] .
  op 2nd  : LNat -> Nat .
  op from : Nat -> LNat .
  vars X Y : Nat . var Z : LNat .
  eq 2nd(cons(X,cons(Y,Z))) = Y .
  eq from(X) = cons(X,from(s(X))) .
endo
```

And now, only the following evaluation sequence is proven:

`2nd(cons(0,from(s(0))))`
  $\rightarrow$ `2nd(cons(0,cons(s(0),from(s(s(0))))))`
  $\rightarrow$ `2nd(cons(0,cons(s(0),from(s(s(0))))))`

$\to$ `s(0)`

This rather intuitive notion of demanded evaluation of an argument of a function call by a left hand side of a rule has also been explored as a technique to effectively implement lazy evaluation [Alpuente *et al.*, 1997; Antoy *et al.*, 1997, 2000; Fokkink *et al.*, 2000; Lucas, 2001a, 2002a; Moreno-Navarro and Rodríguez-Artalejo, 1992]; see [Antoy and Lucas, 2002] for a survey discussing this topic. It is worth noting that this was the main topic of Part I of this thesis.

In the following, we recall different formalisms dealing with (on-demand) syntactic strategy annotations.

6.2 Context-Sensitive Rewriting

In *context-sensitive rewriting* (*CSR* [Lucas, 1998a]), we rewrite subterms at positions specified by a replacement map for a signature $\mathcal{F}$. A replacement map μ specifies the *argument* positions which can be reduced for each *symbol* in $\mathcal{F}$. A mapping $\mu : \mathcal{F} \to \mathcal{P}(\mathbb{N})$ is a *replacement map* (or $\mathcal{F}$-map) if $\forall f \in \mathcal{F},\ \mu(f) \subseteq \{1, \ldots, ar(f)\}$ [Lucas, 1998a]. Let $M_{\mathcal{F}}$ be the set of all $\mathcal{F}$-maps. The ordering $\sqsubseteq$ on $M_{\mathcal{F}}$, the set of all $\mathcal{F}$-maps, is: $\mu \sqsubseteq \mu'$ if for all $f \in \mathcal{F}$, $\mu(f) \subseteq \mu'(f)$. Thus, $\mu \sqsubseteq \mu'$ means that μ considers less positions than μ' (for reduction), i.e. μ is more restrictive than μ'. According to $\sqsubseteq$, $\mu_\bot$ (resp. $\mu_\top$) given by $\mu_\bot(f) = \varnothing$ (resp. $\mu_\bot(f) = \{1, \ldots, ar(f)\}$) for all $f \in \mathcal{F}$, is the minimum (maximum) element of $M_{\mathcal{F}}$.

The set of *μ-replacing positions* $\mathcal{P}os^\mu(t)$ of $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is: $\mathcal{P}os^\mu(t) = \{\Lambda\}$, if $t \in \mathcal{X}$ and $\mathcal{P}os^\mu(t) = \{\Lambda\} \cup \bigcup_{i \in \mu(root(t))} i.\mathcal{P}os^\mu(t|_i)$, if $t \notin \mathcal{X}$. The set of positions of replacing redexes in t is $\mathcal{P}os^\mu_{\mathcal{R}}(t) = \mathcal{P}os_{\mathcal{R}}(t) \cap \mathcal{P}os^\mu(t)$. We (only) rewrite subterms at *replacing* positions: t μ-rewrites to s, written $t \hookrightarrow_{\mathcal{R}(\mu)} s$ (or simply $t \hookrightarrow_\mu s$ or $t \hookrightarrow s$), if $t \xrightarrow{p}_{\mathcal{R}} s$ and $p \in \mathcal{P}os^\mu(t)$. Obviously, a term t is a μ-normal form if and only if $\mathcal{P}os^\mu_{\mathcal{R}}(t) = \varnothing$.

Example 6.8 *[Lucas, 2002a] Consider the TRS $\mathcal{R}$ and the replacement map μ of Example 6.1. Now, we write:*

$$
\begin{array}{l}
\texttt{sel(s(0),}\underline{\texttt{from(0)}}\texttt{)} \\
\quad \hookrightarrow_\mu \underline{\texttt{sel(s(0),cons(0,from(s(0))))}} \\
\quad \hookrightarrow_\mu \texttt{sel(0,}\underline{\texttt{from(s(0))}}\texttt{)} \\
\quad \hookrightarrow_\mu \underline{\texttt{sel(0,cons(s(0),from(s(s(0)))))}} \\
\quad \hookrightarrow_\mu \texttt{s(0)}
\end{array}
$$

However,

$$
\begin{array}{l}
\texttt{sel(0,cons(s(0),}\underline{\texttt{from(s(s(0)))}}\texttt{))} \\
\quad \not\hookrightarrow_\mu \texttt{sel(0,cons(s(0),cons(s(s(0)),from(s(s(s(0)))))))}
\end{array}
$$

since the restriction $\mu(\texttt{cons}) = \{1\}$ *avoids reduction on redex* `from(s(s(0)))`. *Hence, non-termination is avoided.*

Let $\mu_{\mathcal{R}}^{can}$ be the canonical replacement map, i.e. *the most restrictive replacement map which ensures that the non-variable subterms of the left-hand sides of the rules of* $\mathcal{R}$ *are replacing*, which is easily obtained from $\mathcal{R}$: $\forall f \in \mathcal{F}$, $i \in \{1, \ldots, ar(f)\}$,

$$i \in \mu_{\mathcal{R}}^{can}(f) \quad \text{iff} \quad \exists l \in L(\mathcal{R}), p \in \mathcal{P}os_{\mathcal{F}}(l), (root(l|_p) = f \wedge p.i \in \mathcal{P}os_{\mathcal{F}}(l)).$$

Let $CM_{\mathcal{R}} = \{\mu \in M_{\mathcal{F}} \mid \mu_{\mathcal{R}}^{can} \sqsubseteq \mu\}$ be the set of replacement maps which are less or equally restrictive than $\mu_{\mathcal{R}}^{can}$.

Example 6.9 *[Lucas, 2002a] Consider the TRS* $\mathcal{R}$ *of Example 6.1. We have* $1 \in \mu_{\mathcal{R}}^{can}(\texttt{first})$ *since position* 1 *in lhs* `first(0,x)` *does not correspond to a variable. Also,* $2 \in \mu_{\mathcal{R}}^{can}(\texttt{first})$ *since position* 2 *in lhs* `first(s(x),cons(y,z))` *does not correspond to a variable. However,* $\mu_{\mathcal{R}}^{can}(\texttt{from}) = \varnothing$ *since position* 1 *in lhs* `from(x)` *corresponds to a variable.*

$\hookrightarrow_\mu$-normal forms are called μ-normal forms. A TRS $\mathcal{R}$ is μ-terminating if $\hookrightarrow_\mu$ is terminating (see [Lucas, 2001a]). Correctness and completeness of *CSR* is ensured under some conditions.

Theorem 6.10 (Correctness) [Lucas, 1998a] *Let* $\mathcal{R}$ *be a left-linear TRS and* $\mu \in CM_{\mathcal{R}}$. *Every* μ*-normal form is root-stable.*

Theorem 6.11 (Completeness) [Lucas, 1998a] *Let* $\mathcal{R}$ *be a left-linear TRS and* $\mu \in CM_{\mathcal{R}}$. *Let* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *and* s *be a root-stable term. If* $t \to^* s$, *then there exists* s' *such that* $t \hookrightarrow_\mu^* s'$, $root(s) = root(s')$, *and* $s' \xrightarrow{>\Lambda}{}^* s$.

6.3 On-demand Rewriting

A replacement map $\mu \in M_{\mathcal{F}}$ is aimed at specifying the *argument* positions which can be reduced for each symbol in $\mathcal{F}$. In *context-sensitive rewriting* (*CSR* [Lucas, 1998a]), we (only) rewrite subterms at *replacing* positions and the notion of demanded evaluation presented in Section 6.1.2 becomes useful as a possible extension of context-sensitive rewriting.

Example 6.12 *Consider the TRS* $\mathcal{R}$ *associated to the program of Example 6.6 and the replacement map* $\mu(\texttt{s}) = \mu(\texttt{2nd}) = \mu(\texttt{from}) = \mu(\texttt{cons}) = \{1\}$ *(which corresponds to the strategy map* φ *of Example 6.6). Then, we have:*

$$\texttt{2nd(}\underline{\texttt{from(0)}}\texttt{)} \hookrightarrow_\mu \texttt{2nd(cons(0,from(s(0))))}$$

where, since $\mu(\texttt{cons}) = \{1\}$, `2nd(cons(0,from(s(0))))` *cannot be further* μ*-rewritten.*

Given a pair $\langle\mu, \mu_D\rangle$ of replacement maps μ and μ_D, *on-demand rewriting* (*ODR* [Lucas, 2001a]) is defined as an extension of *CSR* (under μ), where reductions *on-demand* are also permitted according to μ_D. Given $f \in \mathcal{F}$, indices $j \in \mu_D(f)$ are aimed at enabling reductions on a subterm t_j of a function call $f(t_1, \ldots, t_j, \ldots, t_k)$ if they can eventually lead to matching a pattern of a rule defining f (i.e., $l \to r \in R$ such that $root(t) = f$). After its formal definition, we explain the notion and give an example. The chain of symbols lying on positions above/on $p \in \mathcal{P}os(t)$ is $prefix_t(\Lambda) = root(t)$, $prefix_t(i.p) = root(t).prefix_{t|_i}(p)$. The strict prefix *sprefix* is $sprefix_t(\Lambda) = \Lambda$, $sprefix_t(p.i) = prefix_t(p)$. Accordingly to the set of *μ-replacing* positions $\mathcal{P}os^\mu(t)$ of a term t, the *non-replacing* positions are $\overline{\mathcal{P}os^\mu}(t) = \mathcal{P}os(t) - \mathcal{P}os^\mu(t)$. We also use $\mathcal{L}azy_\mu(t) = minimal_\leq(\overline{\mathcal{P}os^\mu}(t))$ which covers the non-replacing positions of t, i.e., for all $p \in \overline{\mathcal{P}os^\mu}(t)$, there exists $q \in \mathcal{L}azy_\mu(t)$ such that $q \leq p$.

Definition 6.13 (On-demand rewriting) [Lucas, 2001a] *Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS and $\mu, \mu_D \in M_\mathcal{F}$. Then, $t \overset{p}{\hookrightarrow}_{\langle\mu,\mu_D\rangle} s$ (or simply $t \hookrightarrow_{\langle\mu,\mu_D\rangle} s$), if $t \overset{p}{\to} s$ and either*

1. *$p \in \mathcal{P}os^\mu(t)$, or*
2. *$p \in \mathcal{P}os^{\mu\sqcup\mu_D}(t) - \mathcal{P}os^\mu(t)$ and there exist $e \in \mathcal{P}os^\mu(t)$, $p_1, \ldots, p_n \in \mathcal{L}azy_{\langle\mu,\mu_D\rangle}(t)$, $r_1, \ldots, r_n, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \to r \in R$, and substitution σ such that*
   (1) $e \leq p$, $t' = t[r_1]_{p_1} \cdots [r_n]_{p_n}$, $t'|_e = \sigma(l)$ and
   (2) for all $q \in \mathcal{P}os(l)$ s.t. $sprefix_{t|_e}(q) = sprefix_l(q)$, whenever $e.q \leq p$, we have that $l|_q \notin \mathcal{X}$.

Here, $\mathcal{L}azy_{\langle\mu,\mu_D\rangle}(t) = \mathcal{L}azy_\mu(t) \cap \mathcal{P}os^{\mu\sqcup\mu_D}(t)$.

Therefore, given a term t, a rewriting step $t \overset{p}{\to} s$ is *on-demand* (w.r.t. μ and μ_D) if either

(i) $t \overset{p}{\hookrightarrow}_\mu s$, or

(ii) $t \overset{p}{\hookrightarrow}_{\mu\sqcup\mu_D} s$ and reducing $t|_p$ may contribute to a future μ-rewriting step at μ-replacing position e, using some rule $l \to r$.

Such a contribution is approximated by checking whether the replacement of some non-μ-replacing maximal subterms of t would eventually make the matching possible (condition 2(1) of Definition 6.13). On-demand indices in μ_D determine the positions (in $\mathcal{L}azy_{\langle\mu,\mu_D\rangle}(t)$) of the subterms of t that can be refined. Note that the position p on which the rewriting step is performed is always covered by some position $p_i \in \mathcal{L}azy_{\langle\mu,\mu_D\rangle}(t)$, i.e., $p_i \leq p$ and p_i is a position demanded by the lhs l, which is (possibly) applicable at position e. On the other hand, the position p is constrained to having a non-variable position of l covering p (condition 2(2) of Definition 6.13); otherwise, the reduction at $t|_p$ would not improve the matching.

Example 6.14 *Consider the TRS $\mathcal{R}$ of Example 6.7, the replacement map μ of Example 6.1 and the following on-demand replacement map μ_D(*`cons`*) = $\{2\}$ and μ(*`s`*) = μ(*`2nd`*) = μ(*`from`*) = $\varnothing$ (where the union of μ and μ_D corresponds to the "apt" strategy map φ of Example 6.7). Now we have:*

$$\begin{array}{ll} \mathtt{2nd(\underline{from(0)})} & \hookrightarrow_{\langle\mu,\mu_D\rangle}\ \mathtt{2nd(cons(0,\underline{from(s(0))}))} \\ & \hookrightarrow_{\langle\mu,\mu_D\rangle}\ \underline{\mathtt{2nd(cons(0,cons(s(0),from(s(s(0))))))}} \\ & \hookrightarrow_{\langle\mu,\mu_D\rangle}\ \mathtt{s(0)} \end{array}$$

but

$$\begin{array}{l} \mathtt{2nd(cons(0,cons(s(0),\underline{from(s(s(0)))})))} \\ \quad \not\hookrightarrow_{\langle\mu,\mu_D\rangle}\ \mathtt{2nd(cons(0,cons(s(0),cons(s(s(0)),from(s(s(s(0))))))))} \end{array}$$

since $1.2.2 \in \mathcal{P}os(l)$ and $sprefix_t(1.2.2) = sprefix_l(1.2.2)$, but $l|_{1.2.2} \in \mathcal{X}$ (where $l =$ `2nd(cons(x,cons(y,z)))`*).*

$\hookrightarrow_{\langle\mu,\mu_D\rangle}$-normal forms are called $\langle\mu,\mu_D\rangle$-normal forms. A TRS $\mathcal{R}$ is $\langle\mu,\mu_D\rangle$-terminating if $\hookrightarrow_{\langle\mu,\mu_D\rangle}$ is terminating. Thus, *ODR* seems to provide better opportunities for improving termination than *CSR*. In general, *CSR* and *ODR* are related as follows.

Theorem 6.15 [Lucas, 2001a] *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and $\mu \in M_\Sigma$. Then, $\hookrightarrow_\mu = \hookrightarrow_{\langle\mu,\mu_\perp\rangle}$.*

Theorem 6.16 [Lucas, 2001a] *Let $\mathcal{R} = (\Sigma, R)$ be a TRS and $\mu, \mu_D \in M_\Sigma$. Then, $\hookrightarrow_\mu \subseteq \hookrightarrow_{\langle\mu,\mu_D\rangle} \subseteq \hookrightarrow_{\mu\sqcup\mu_D}$.*

Completeness of *ODR* is ensured by Theorems 6.11 and 6.16. Correctness of *ODR* is also ensured (under some conditions).

Theorem 6.17 (Correctness) [Lucas, 2001a] *Let $\mathcal{R} = (\Sigma, R)$ be a left-linear CS and $\mu, \mu_D \in M_\Sigma$ be such that $\mu^{can}_{\mathcal{R}} \sqsubseteq \mu \sqcup \mu_D$. Every $\langle\mu,\mu_D\rangle$-normal form is a head-normal form.*

6.4 Rewriting with Positive Strategy Annotations

Algebraic languages, such as OBJ2 [Futatsugi *et al.*, 1985], OBJ3 [Goguen *et al.*, 2000], or Maude [Clavel *et al.*, 1996] admit the specification of *local strategies* associated to function symbols. A local strategy $\varphi : \mathcal{F} \to \mathbb{Z}^*$ for a k-ary symbol $f \in \mathcal{F}$ is a sequence $\varphi(f)$ of integers taken from $\{-k, \ldots, -1, 0, 1, \ldots, k\}$, which are given in parentheses. A positive local strategy is a local strategy restricted only to natural numbers. We define an ordering $\sqsubseteq$ between sequences of integers as: $nil \sqsubseteq L$, for all sequence L, $(i_1\ i_2\ \cdots\ i_m) \sqsubseteq (j_1\ j_2\ \cdots\ j_n)$ if $i_1 = j_1$ and $(i_2\ \cdots\ i_m) \sqsubseteq (j_2\ \cdots\ j_n)$, or

$(i_1\ i_2\ \cdots\ i_m) \sqsubseteq (j_1\ j_2\ \cdots\ j_n)$ if $i_1 \neq j_1$ and $(i_1\ i_2\ \cdots\ i_m) \sqsubseteq (j_2\ \cdots\ j_n)$; i.e. $L_1 \sqsubseteq L_2$ if L_2 has some additional integers. A mapping φ that associates a local strategy $\varphi(f)$ to every $f \in \mathcal{F}$ is called an E-strategy map [Nagaya, 1999; Nakamura and Ogata, 2001; Ogata and Futatsugi, 2000]. An ordering $\sqsubseteq$ between strategy maps is defined: $\varphi \sqsubseteq \varphi'$ if for all $f \in \mathcal{F}$, $\varphi(f) \sqsubseteq \varphi'(f)$. Roughly speaking, $\varphi \sqsubseteq \varphi'$ if for all $f \in \mathcal{F}$, $\varphi'(f)$ is $\varphi(f)$ where some additional indices have been included.

Semantics of rewriting under a given E-evaluation map φ is usually given by means of a mapping $eval_\varphi : \mathcal{T}(\mathcal{F},\mathcal{X}) \to \mathcal{P}(\mathcal{T}(\mathcal{F},\mathcal{X}))$ from terms to the set of its computed values (technically E-normal forms). Nagaya describes the mapping $eval_\varphi$ for *positive* E-strategy maps φ (i.e., E-strategy maps where only naturals are allowed) by using a reduction relation on pairs $\langle t, p\rangle$ of labeled terms t and positions p [Nagaya, 1999]. Let $\mathbb{L}$ be the set of all lists consisting of integers, $\mathbb{L}_n$ be the set of all lists of integers whose absolute value does not exceed $n \in \mathbb{N}$, and $\mathbb{L}_n^{\mathbb{N}}$ be the restriction to natural numbers. Given an E-strategy map φ for $\mathcal{F}$, we use the signature[5] $\mathcal{F}_\varphi^{\mathbb{N}} = \{f_L \mid f \in \mathcal{F}, L \in \mathbb{L}^{\mathbb{N}}_{ar(f)}$, and $L \sqsubseteq \varphi(f)\}$ and labeled variables $\mathcal{X}_\varphi^{\mathbb{N}} = \{x_{nil} \mid x \in \mathcal{X}\}$. An E-strategy map φ for $\mathcal{F}$ is extended to a mapping from $\mathcal{T}(\mathcal{F},\mathcal{X})$ to $\mathcal{T}(\mathcal{F}_\varphi^{\mathbb{N}},\mathcal{X}_\varphi^{\mathbb{N}})$ by introducing the local strategy associated to each symbol as a subscript of the symbol as follows:

$$\varphi(t) = \begin{cases} x_{nil} & \text{if } t = x \in \mathcal{X} \\ f_{\varphi(f)}(\varphi(t_1),\ldots,\varphi(t_k)) & \text{if } t = f(t_1,\ldots,t_k) \end{cases}$$

The mapping *erase* : $\mathcal{T}(\mathcal{F}_\varphi^{\mathbb{N}},\mathcal{X}_\varphi^{\mathbb{N}}) \to \mathcal{T}(\mathcal{F},\mathcal{X})$ removes labelings from symbols in the obvious way. Then, given a TRS $\mathcal{R} = (\mathcal{F}, R)$ and a positive E-strategy map φ for $\mathcal{F}$, $eval_\varphi$ is defined as

$$eval_\varphi(t) = \{erase(s) \in \mathcal{T}(\mathcal{F},\mathcal{X}) \mid \langle\varphi(t),\Lambda\rangle \stackrel{\mathbb{N}\,!}{\to}_\varphi \langle s,\Lambda\rangle\}$$

where the binary relation $\stackrel{\mathbb{N}}{\to}_\varphi$ on $\mathcal{T}(\mathcal{F}_\varphi^{\mathbb{N}},\mathcal{X}_\varphi^{\mathbb{N}}) \times \mathbb{N}^*$ behaves as follows [Nagaya, 1999; Nakamura and Ogata, 2001].

Definition 6.18 [Nagaya, 1999] *Given $\langle t, p\rangle$, where $t \in \mathcal{T}(\mathcal{F}_\varphi^{\mathbb{N}},\mathcal{X}_\varphi^{\mathbb{N}})$ and $p \in \mathcal{P}os(t)$, $\langle t,p\rangle \stackrel{\mathbb{N}}{\to}_\varphi \langle s,q\rangle$ if and only if $p \in \mathcal{P}os(t)$ and either*

1. *$root(t|_p) = f_{nil}$, $s = t$ and $p = q.i$ for some i; or*

2. *$t|_p = f_{i:L}(t_1,\ldots,t_k)$, with $i > 0$, $s = t[f_L(t_1,\ldots,t_k)]_p$ and $q = p.i$; or*

3. *$t|_p = f_{0:L}(t_1,\ldots,t_k)$, $erase(t|_p)$ is not a redex, $s = t[f_L(t_1,\ldots,t_k)]_p$, $q = p$; or*

[5] Note that Nakamura and Ogata's definition uses $\mathcal{F}_{\mathbb{L}}$ and $\mathcal{X}_{\mathbb{L}}$ instead of $\mathcal{F}_\varphi$ and $\mathcal{X}_\varphi$, where the restriction to $L \sqsubseteq \varphi(f)$ is not considered. However, using terms over $\mathcal{F}_\varphi$ does not entail loss of generality; furthermore, it actually provides a more accurate framework for formalizing and studying the strategy, since this is the only class of terms involved in the computations.

4. $t|_p = f_{0:L}(t_1, \ldots, t_k) = \sigma(l')$, *erase*$(l') = l$, $s = t[\sigma(\varphi(r))]_p$ *for some* $l \to r \in R$ *and substitution* σ, $q = p$.

Namely, if a positive index $i > 0$ is found in the list labeling the symbol at $t|_p$, then the index is removed from the list, the "target position" is moved from p to $p.i$, and the subterm $t|_{p.i}$ is next considered. If 0 is found, then the evaluation of $t|_p$ is attempted: if possible, a rewriting step is performed; otherwise, the 0 is removed from the list. In both cases for index 0, the evaluation continues at the same position p.

Example 6.19 *Consider the* OBJ *program of Example 6.3. The evaluation of the term* `sel(s(0),from(0))` *is represented in Figure 6.20, where we underline the redex which is contracted in each step and we surround by a box the index of a strategy list which is been considered in each step. Note that the infinite (meaningless) derivation of Example 6.3 is avoided.*

A TRS $\mathcal{R}$ is φ-terminating if, for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, there is no infinite $\stackrel{\mathbb{N}}{\to}_\varphi$-rewrite sequence starting from $\langle \varphi(t), \Lambda \rangle$. An OBJ program P is terminating if the corresponding TRS $\mathcal{R}$ is φ-terminating [Lucas, 2001a]. Correctness is ensured (under some conditions).

Theorem 6.21 (Correctness) [Nagaya, 1999] *Let* $\mathcal{R}$ *be a TRS and* φ *be an E-strategy map such that* $\varphi(f)$ *contains* $1, \ldots, ar(f)$ *for* $f \in \mathcal{F}$ *and the last element of* $\varphi(f)$ *is* 0 *if* $f \in \mathcal{D}$. *Let* $t, s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *such that* $s \in eval_\varphi(t)$, *then* s *is a normal form of* t.

6.5 Rewriting with On-demand Strategy Annotations

Ogata and Futatsugi [Ogata and Futatsugi, 2000] have provided an operational description of the *on-demand evaluation strategy* $eval_\varphi$ where negative integers are also allowed in local strategies to denote those arguments to be evaluated on-demand. Nakamura and Ogata [Nakamura and Ogata, 2001] have described the corresponding evaluation mapping $eval_\varphi$ by using a reduction relation. We recall here the latter one since the former one is not appropriate for our objectives in this thesis[6].

Given an E-strategy map φ, we use the signature[7] $\mathcal{F}_\varphi = \{f_L^b \mid f \in \mathcal{F}, L \in \mathbb{L}_{ar(f)}, L \sqsubseteq \varphi(f)$, and $b \in \{0,1\}\}$ and labeled variables $\mathcal{X}_\varphi = \{x_{nil}^0 \mid x \in \mathcal{X}\}$. An on-demand flag $b = 1$ indicates that the term may be reduced if demanded. An

[6] Actually, the operational models in [Ogata and Futatsugi, 2000] and [Nakamura and Ogata, 2001] differ and deliver different computations, see Example 7.5 in Chapter 7 below.

[7] Note that similarly to Section 6.4, Nakamura and Ogata's definition uses $\mathcal{F}_{\mathbb{L}}$ and $\mathcal{X}_{\mathbb{L}}$ instead of $\mathcal{F}_\varphi$ and $\mathcal{X}_\varphi$.

$$
\begin{array}{l}
\langle \mathtt{sel}_{(\boxed{1}\ 2\ 0)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}), \mathtt{from}_{(0)}(\mathtt{0}_{nil})), \Lambda\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(2\ 0)}(\mathtt{s}_{(\boxed{1})}(\mathtt{0}_{nil}), \mathtt{from}_{(0)}(\mathtt{0}_{nil})), 1\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(2\ 0)}(\mathtt{s}_{nil}(\mathtt{0}_{\boxed{nil}}), \mathtt{from}_{(0)}(\mathtt{0}_{nil})), 1.1\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(2\ 0)}(\mathtt{s}_{\boxed{nil}}(\mathtt{0}_{nil}), \mathtt{from}_{(0)}(\mathtt{0}_{nil})), 1\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(\boxed{2}\ 0)}(\mathtt{s}_{nil}(\mathtt{0}_{nil}), \mathtt{from}_{(0)}(\mathtt{0}_{nil})), \Lambda\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{s}_{nil}(\mathtt{0}_{nil}), \underline{\mathtt{from}_{(\boxed{0})}(\mathtt{0}_{nil})}), 2\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{s}_{nil}(\mathtt{0}_{nil}), \mathtt{cons}_{(\boxed{1})}(\mathtt{0}_{nil}, \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil})))), 2\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{s}_{nil}(\mathtt{0}_{nil}), \mathtt{cons}_{nil}(\mathtt{0}_{\boxed{nil}}, \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil})))), 2.1\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{s}_{nil}(\mathtt{0}_{nil}), \mathtt{cons}_{\boxed{nil}}(\mathtt{0}_{nil}, \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil})))), 2\rangle \\
\quad \to_\varphi \ \langle \underline{\mathtt{sel}_{(\boxed{0})}(\mathtt{s}_{nil}(\mathtt{0}_{nil}), \mathtt{cons}_{nil}(\mathtt{0}_{nil}, \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))))}, \Lambda\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(\boxed{1}\ 2\ 0)}(\mathtt{0}_{nil}, \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))), \Lambda\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(2\ 0)}(\mathtt{0}_{\boxed{nil}}, \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))), 1\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(\boxed{2}\ 0)}(\mathtt{0}_{nil}, \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))), \Lambda\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{0}_{nil}, \underline{\mathtt{from}_{(\boxed{0})}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))}), 2\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{0}_{nil}, \mathtt{cons}_{(\boxed{1})}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}), \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))))), 2\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{0}_{nil}, \mathtt{cons}_{nil}(\mathtt{s}_{(\boxed{1})}(\mathtt{0}_{nil}), \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))))), 2.1\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{0}_{nil}, \mathtt{cons}_{nil}(\mathtt{s}_{nil}(\mathtt{0}_{\boxed{nil}}), \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))))), 2.1.1\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{0}_{nil}, \mathtt{cons}_{nil}(\mathtt{s}_{\boxed{nil}}(\mathtt{0}_{nil}), \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))))), 2.1\rangle \\
\quad \to_\varphi \ \langle \mathtt{sel}_{(0)}(\mathtt{0}_{nil}, \mathtt{cons}_{\boxed{nil}}(\mathtt{s}_{nil}(\mathtt{0}_{nil}), \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil}))))), 2\rangle \\
\quad \to_\varphi \ \langle \underline{\mathtt{sel}_{(\boxed{0})}(\mathtt{0}_{nil}, \mathtt{cons}_{nil}(\mathtt{s}_{nil}(\mathtt{0}_{nil}), \mathtt{from}_{(0)}(\mathtt{s}_{(1)}(\mathtt{s}_{(1)}(\mathtt{0}_{nil})))))}, \Lambda\rangle \\
\quad \to_\varphi \ \langle \mathtt{s}_{nil}(\mathtt{0}_{nil}), \Lambda\rangle
\end{array}
$$

Figure 6.20: Evaluation of `sel(s(0),from(0))` using strategy annotations

E-strategy map φ for $\mathcal{F}$ is extended to a mapping from $\mathcal{T}(\mathcal{F},\mathcal{X})$ to $\mathcal{T}(\mathcal{F}_\varphi,\mathcal{X}_\varphi)$ as follows:

$$
\varphi(t) = \begin{cases} x^0_{nil} & \text{if } t = x \in \mathcal{X} \\ f^0_{\varphi(f)}(\varphi(t_1),\ldots,\varphi(t_k)) & \text{if } t = f(t_1,\ldots,t_k) \end{cases}
$$

On the other hand, the mapping $erase : \mathcal{T}(\mathcal{F}_\varphi,\mathcal{X}_\varphi) \to \mathcal{T}(\mathcal{F},\mathcal{X})$ removes labelings from symbols in the obvious way. Moreover, some auxiliary functions are provided to manage the demandedness flag b. The (partial) function $flag : \mathcal{T}(\mathcal{F}_\varphi,\mathcal{X}_\varphi) \times \mathbb{N}^* \to \{0,1\}$ returns the flag of the function symbol attached to a position of the term: $flag(t,p) = b$ if $root(t|_p) = f^b_L$. The map $up : \mathcal{T}(\mathcal{F}_\varphi,\mathcal{X}_\varphi) \to \mathcal{T}(\mathcal{F}_\varphi,\mathcal{X}_\varphi)$ (resp. $dn : \mathcal{T}(\mathcal{F}_\varphi,\mathcal{X}_\varphi) \to \mathcal{T}(\mathcal{F}_\varphi,\mathcal{X}_\varphi)$) raises (lowers) the on-demand flag of each function symbol in a term, i.e. $up(x^0_{nil}) = dn(x^0_{nil}) = x^0_{nil}$, $up(f^b_L(t_1,\ldots,t_k)) = f^1_L(up(t_1),\ldots,up(t_k))$, and $dn(f_L^b(t_1,\ldots,t_k)) = f_L^0(dn(t_1),\ldots,dn(t_k))$.

When it is examined whether a term t matches the left-hand side l of a rule in order to determine demandedness, a top-to-bottom and left-to-right pattern matching is performed.

Definition 6.22 *Let $\mathcal{P}os_{\neq}(t,l) = \{p \in \mathcal{P}os_{\mathcal{F}}(t) \cap \mathcal{P}os_{\mathcal{F}}(l) \mid root(l|_p) \neq root(t|_p)\}$ be the set of (common) positions of non-variable disagreeing symbols of terms t and l.*

Definition 6.23 *The map $df_l : \mathcal{T}(\mathcal{F},\mathcal{X}) \to \mathbb{N}^* \cup \{\top\}$ returns the first position where the term and the considered lhs differ (on some non-variable position) or $\top$ if each function symbol of the term coincides with l:*

$$df_l(t) = \begin{cases} min_{\leq_{lex}}(\mathcal{P}os_{\neq}(t,l)) & \text{if } \mathcal{P}os_{\neq}(t,l) \neq \varnothing \\ \top & \text{otherwise} \end{cases}$$

Definition 6.24 *Given a TRS $\mathcal{R}$, the map $DF_{\mathcal{R}} : \mathcal{T}(\mathcal{F},\mathcal{X}) \to \mathbb{N}^* \cup \{\top\}$ returns the first position where the term differs from all lhs's:*

$$DF_{\mathcal{R}}(t) = \begin{cases} \top & \text{if } df_l(t) = \top \text{ for some } l \to r \in \mathcal{R} \\ max_{<_{lex}}\{df_l(t) \mid l \to r \in \mathcal{R}\} & \text{otherwise} \end{cases}$$

Now, we provide the associated evaluation function $eval_\varphi(t)$.

Definition 6.25 [Nakamura and Ogata, 2001] *Given a TRS $\mathcal{R} = (\mathcal{F}, R)$ and an arbitrary E-strategy map φ for $\mathcal{F}$, $eval_\varphi : \mathcal{T}(\mathcal{F},\mathcal{X}) \to \mathcal{P}(\mathcal{T}(\mathcal{F},\mathcal{X}))$ is defined as $eval_\varphi(t) = \{erase(s) \in \mathcal{T}(\mathcal{F},\mathcal{X}) \mid \langle\varphi(t),\Lambda\rangle \to_\varphi^! \langle s,\Lambda\rangle\}$. The binary relation $\to_\varphi$ on $\mathcal{T}(\mathcal{F}_\varphi,\mathcal{X}_\varphi) \times \mathbb{N}^*$ is defined as follows: $\langle t,p\rangle \to_\varphi \langle s,q\rangle$ if and only if $p \in \mathcal{P}os(t)$ and either*

1. *$root(t|_p) = f_{nil}^b$, $s = t$ and $p = q.i$ for some i; or*
2. *$t|_p = f_{i:L}^b(t_1,\ldots,t_k)$, $i > 0$, $s = t[f_L^b(t_1,\ldots,t_k)]_p$ and $q = p.i$; or*
3. *$t|_p = f_{-i:L}^b(t_1,\ldots,t_k)$, $i > 0$, $s = t[f_L^b(t_1,\ldots,up(t_i),\ldots,t_k)]_p$ and $q = p$; or*
4. *$t|_p = f_{0:L}^b(t_1,\ldots,t_k)$, $s = t[t']_p$, $q = p$ where t' is a term such that*
   - (a) *$t' = \theta(\varphi(r))$ if $DF_{\mathcal{R}}(erase(t|_p)) = \top$, $t|_p = \theta(l')$, $erase(l') = l$ and $l \to r \in \mathcal{R}$.*
   - (b) *$t' = f_L^b(t_1,\ldots,t_k)$ if either $DF_{\mathcal{R}}(erase(t|_p)) = \top$ and $erase(t|_p)$ is not a redex, or $DF_{\mathcal{R}}(erase(t|_p)) = \Lambda$, or $DF_{\mathcal{R}}(erase(t|_p)) = p' \neq \Lambda$ and $flag(t,p.p') = 0$;*
   - (c) *$t' = f_L^b(t_1,\ldots,t_i[up(s)]_{p''},\ldots,t_k)$ if $DF_{\mathcal{R}}(erase(t|_p)) = p' = i.p''$, $flag(t,p.p') = 1$, $\langle dn(t|_{p.p'}),\Lambda\rangle \to_\varphi^! \langle s,\Lambda\rangle$, and $DF_{\mathcal{R}}(erase(t|_p[s]_{p'})) = p'$;*

(d) $t' = t|_p[up(s)]_{p'}$ *if* $DF_{\mathcal{R}}(erase(t|_p)) = p' \neq \Lambda$, $flag(t, p.p') = 1$, $\langle dn(t|_{p.p'}), \Lambda\rangle \to^!_\varphi \langle s, \Lambda\rangle$, *and either* $p' <_{lex} DF_{\mathcal{R}}(erase(t|_p[s]_{p'}))$ *or* $DF_{\mathcal{R}}(erase(t|_p[s]_{p'})) = \top$.

Case 1 means that no more annotations are provided and the evaluation is completed. In case 2, a positive argument index is found and the evaluation goes down to the subterm at such argument. In case 3, the subterm at the argument indicated by the (absolute value of the) negative index is completely marked with on-demand flags. Case 4 considers the attempt to match the term against the left-hand sides of the program rules. Case 4.a applies if the considered (unlabeled) subterm is a redex (which is, then, contracted). If the subterm is not a redex, cases 4.b, 4.c and 4.d are considered, possibly involving some on-demand evaluation steps on some subterm. The selected demanded position of a term t (w.r.t. program $\mathcal{R}$) is denoted as $DF_{\mathcal{R}}(t)$ (eventually, symbol $\top$ is returned if t matches the left-hand side of some rule of the TRS). According to $DF_{\mathcal{R}}(t)$, case 4.b applies if no demanded evaluation is allowed (or required). Cases 4.c and 4.d apply if the on-demand evaluation of the subterm $t|_{p.p'}$ is required, i.e. $DF_{\mathcal{R}}(t|_p) = p'$. In both cases, the evaluation is attempted; if it finishes, the evaluation of $t|_p$ continues accordingly to the computed value.

Example 6.26 *Following the Example 6.7. The on-demand evaluation of term* `2nd(from(0))` *is as follows:*

$$\langle \mathtt{2nd}^0_{(\boxed{1}\ 0)}(\mathtt{from}^0_{(0)}(\mathtt{0}^0_{nil})), \Lambda\rangle$$

$$\to_\varphi \langle \mathtt{2nd}^0_{(0)}(\underline{\mathtt{from}^0_{(\boxed{0})}(\mathtt{0}^0_{nil})}), 1\rangle$$

$$\to_\varphi \langle \mathtt{2nd}^0_{(0)}(\mathtt{cons}^0_{(\boxed{1}\ -2)}(\mathtt{0}^0_{nil}, \mathtt{from}^0_{(0)}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil})))), 1\rangle$$

$$\to_\varphi \langle \mathtt{2nd}^0_{(0)}(\mathtt{cons}^0_{(-2)}(\mathtt{0}^0_{\boxed{nil}}, \mathtt{from}^0_{(0)}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil})))), 1.1\rangle$$

$$\to_\varphi \langle \mathtt{2nd}^0_{(0)}(\mathtt{cons}^0_{(\boxed{-2})}(\mathtt{0}^0_{nil}, \mathtt{from}^0_{(0)}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil})))), 1\rangle$$

$$\to_\varphi \langle \mathtt{2nd}^0_{(0)}(\mathtt{cons}^0_{\boxed{nil}}(\mathtt{0}^0_{nil}, \mathtt{from}^1_{(0)}(\mathtt{s}^1_{(1)}(\mathtt{0}^1_{nil})))), 1\rangle$$

$$\to_\varphi \langle \mathtt{2nd}^0_{(\boxed{0})}(\mathtt{cons}^0_{nil}(\mathtt{0}^0_{nil}, \underline{\mathtt{from}^{\boxed{1}}_{(0)}(\mathtt{s}^1_{(1)}(\mathtt{0}^1_{nil}))})), \Lambda\rangle$$

$$\textit{subsequence: } \langle \underline{\mathtt{from}^0_{(\boxed{0})}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil}))}, \Lambda\rangle$$

$$\to_\varphi \langle \mathtt{cons}^0_{(\boxed{1})}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil}), \mathtt{from}^0_{(0)}(\mathtt{s}^0_{(1)}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil})))), \Lambda\rangle$$

$$\to_\varphi \langle \mathtt{cons}^0_{nil}(\mathtt{s}^0_{(\boxed{1})}(\mathtt{0}^0_{nil}), \mathtt{from}^0_{(0)}(\mathtt{s}^0_{(1)}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil})))), 1\rangle$$

$$\to_\varphi \langle \mathtt{cons}^0_{nil}(\mathtt{s}^0_{nil}(\mathtt{0}^0_{\boxed{nil}}), \mathtt{from}^0_{(0)}(\mathtt{s}^0_{(1)}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil})))), 1.1\rangle$$

$$\to_\varphi \langle \mathtt{cons}^0_{nil}(\mathtt{s}^0_{\boxed{nil}}(\mathtt{0}^0_{nil}), \mathtt{from}^0_{(0)}(\mathtt{s}^0_{(1)}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil})))), 1\rangle$$

$$\to_\varphi \langle \mathtt{cons}^0_{nil}(\mathtt{s}^0_{nil}(\mathtt{0}^0_{nil}), \mathtt{from}^0_{(0)}(\mathtt{s}^0_{(1)}(\mathtt{s}^0_{(1)}(\mathtt{0}^0_{nil})))), \Lambda\rangle$$

$$\to_\varphi \langle \underline{\mathtt{2nd}^0_{(\boxed{0})}(\mathtt{cons}^0_{nil}(\mathtt{0}^0_{nil}, \mathtt{cons}^1_{nil}(\mathtt{s}^1_{nil}(\mathtt{0}^1_{nil}), \mathtt{from}^1_{(0)}(\mathtt{s}^1_{(1)}(\mathtt{s}^1_{(1)}(\mathtt{0}^1_{nil}))))))}, \Lambda\rangle$$

$$\to_\varphi \langle \mathtt{s}^1_{nil}(\mathtt{0}^1_{nil}), \Lambda\rangle$$

Note that the computational description of on-demand strategy annotations above involves recursive steps. A single reduction step on a (labeled) term t may involve the application of more than one reduction step on subterms of t (as shown by the previous example when considering steps $4(c)$ and $4(d)$ of Definition 6.25). In fact, the definition of a single rewriting step may depend on the possibility of evaluating some arguments of the considered function call. This implies that the one-step reduction relation proposed by Nakamura and Ogata is generally undecidable.

6.6 Lazy rewriting

Lazy rewriting [Fokkink *et al.*, 2000; Lucas, 2002b] is a popular, demand-driven technique to perform lazy functional computations which inspired the development of local strategies with on-demand strategy annotations in CafeOBJ [Futatsugi and Nakagawa, 1997] as well as the development of on-demand rewriting [Lucas, 2001a].

In lazy rewriting [Fokkink *et al.*, 2000], there is a replacement map which indicates those arguments enabled for evaluation (as in *CSR*) and the rest of arguments are understood as allowed for evaluation 'on-demand'. Reductions are issued on a different kind of *labeled terms*. Nodes (or positions) of a term t are labeled with 'e' for the so-called *eager* positions or 'ℓ' for so-called *lazy* ones: Let $\mathcal{F}$ be a signature and $\mathcal{L} = \{e, \ell\}$; then, $\mathcal{F} \times \mathcal{L}$ (or $\mathcal{F}_\mathcal{L}$) is a new signature of labeled symbols. The labeling of a symbol $f \in \mathcal{F}$ is denoted f^e or f^ℓ rather than $\langle f, e\rangle$ or $\langle f, \ell\rangle$. Labeled terms are terms in $\mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$. Given $t \in \mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$ and $p \in \mathcal{P}os(t)$, if $root(t|_p) = x^e$ $(= x^\ell)$ or $root(t|_p) = f^e$ $(= f^\ell)$, then we say that p is an *eager* (resp. *lazy*) position of t.

Given a replacement map $\mu \in M_\mathcal{F}$ and $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $label_\mu(s)$ denotes the following *intended* labeling of s: the topmost position Λ of $label_\mu(s)$ is eager; given a position $p \in \mathcal{P}os(label_\mu(s))$ and $i \in \{1, \ldots, ar(root(s|_p))\}$, position $p.i$ of $label_\mu(s)$ is lazy if and only if $i \notin \mu(root(s|_p))$; otherwise, it is eager. In other words, the replacement map μ denotes those positions allowed for reduction (as in *CSR*) and the rest of positions in a term are considered to be evaluated only on-demand.

Example 6.27 *Consider the program of Example 6.7 (as a TRS) and the replacement map μ given by $\mu(\mathtt{2nd}) = \mu(\mathtt{from}) = \mu(\mathtt{cons}) = \mu(\mathtt{s}) = \{1\}$. Then, the labeling of s = $\mathtt{2nd(cons(0,from(s(0))))}$ is $t = label_\mu(s) =$ $\mathtt{2nd}^e\mathtt{(cons}^e\mathtt{(0}^e\mathtt{,from}^\ell\mathtt{(s}^e\mathtt{(0}^e\mathtt{))))}$. Thus, $\Lambda, 1, 1.1, 1.2.1$, and $1.2.1.1$ are eager positions; position 1.2 is lazy.*

Given $t \in \mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$, *erase*$(t)$ is the term in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ that (obviously) corresponds to t after removing labels.

As remarked above, given $t \in \mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$, a position $p \in \mathcal{P}os(t)$ is eager (resp. lazy) if $root(t|_p)$ is labeled with e (resp. ℓ). The so-called *active* positions of a labeled term

$t \in \mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$, denoted by $\mathcal{A}ct(t)$, are those positions which are always reachable from the root of the term via a path consisting of eager positions. For instance, positions $\Lambda, 1$, and 1.1 are active in term t of Example 6.27; positions 1.2.1 and 1.2.1.1 are eager but *not* active, since position 1.2 below is lazy in t. In lazy rewriting, *the set of active nodes may increase as reduction of labeled terms proceeds.* Each lazy reduction step on labeled terms may have two different effects:

1. changing the "activation" status of a given position within a term, or
2. performing a rewriting step (always on an active position).

The *activation* status of a lazy position immediately below an active position within a (labeled) term can be modified if the position is 'essential', i.e. *'its contraction may lead to new redexes at active nodes'* [Fokkink *et al.*, 2000].

Definition 6.28 (Matching modulo laziness) *[Fokkink et al., 2000] Let $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be linear, $t \in \mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$, and p be an active position of t. Then, l matches modulo laziness $s = t|_p$ if either $l \in \mathcal{X}$, or $l = f(l_1, \ldots, l_k)$, $s = f^e(s_1, \ldots, s_k)$ and, for all $i \in \{1, \ldots, k\}$, if $p.i$ is eager, then l_i matches modulo laziness s_i. If position $p.i$ is lazy and $l_i \notin \mathcal{X}$, then position $p.i$ is called* essential.

If p is an active position in $t \in \mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$ and $l \to r$ is a rewrite rule of a left-linear TRS $\mathcal{R}$ such that l matches modulo laziness $t|_p$ giving rise to an essential position q of t and $t|_q = f^\ell(t_1, \ldots, t_k)$, then we write $t \overset{A}{\to}_\mu t[f^e(t_1, \ldots, t_k)]_q$ for denoting the activation of position p.

Lazy rewriting reduces active positions: let p be an active position of $t \in \mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$, $u = t|_p$ and $l \to r$ be a rule of a left-linear TRS $\mathcal{R}$ such that l matches *erase*(u) using substitution σ, then, $t \overset{R}{\to}_\mu s$, where s is obtained from t by replacing $t|_p$ in t by $label_\mu(r)$ with all its variables instantiated according to σ but preserving the label of the variable in $label_\mu(r)$ (see [Lucas, 2002b] for a formal definition).

Example 6.29 *Consider the program of Example 6.7 (as a TRS) and the term t of Example 6.27. The reduction step that corresponds to such term is (we surround by a box those annotations which change by activation in an activation step, and we underline the contracted redex in an evaluation step):*

$$
\begin{array}{l}
\mathtt{2nd}^e\mathtt{(cons}^e\mathtt{(0}^e\mathtt{,from}^{\boxed{\ell}}\mathtt{(s}^e\mathtt{(0}^e\mathtt{))))} \\
\quad \overset{A}{\to}_\mu \underline{\mathtt{2nd}^e\mathtt{(cons}^e\mathtt{(0}^e\mathtt{,from}^e\mathtt{(s}^e\mathtt{(0}^e\mathtt{))))}} \\
\quad \overset{R}{\to}_\mu \underline{\mathtt{2nd}^e\mathtt{(cons}^e\mathtt{(0}^e\mathtt{,cons}^e\mathtt{(s}^e\mathtt{(0}^e\mathtt{),from}^\ell\mathtt{(s}^e\mathtt{(s}^e\mathtt{(0}^e\mathtt{))))))}} \\
\quad \overset{R}{\to}_\mu \mathtt{s}^e\mathtt{(0}^e\mathtt{)}
\end{array}
$$

Note that term $\mathtt{2nd}^e\mathtt{(cons}^e\mathtt{(0}^e\mathtt{,cons}^e\mathtt{(s}^e\mathtt{(0}^e\mathtt{),from}^\ell\mathtt{(s}^e\mathtt{(s}^e\mathtt{(0}^e\mathtt{))))))}$ *is a $\overset{A}{\to}_\mu$-normal form.*

The lazy term rewriting relation on labeled *terms* (LR) is $\stackrel{\mathsf{LR}}{\to}_\mu = \stackrel{A}{\to}_\mu \cup \stackrel{R}{\to}_\mu$ and the evaluation $LR\text{-}eval_\mu(t)$ of a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ using LR is given by $LR\text{-}eval_\mu(t) = \{erase(s) \in \mathcal{T}(\mathcal{F},\mathcal{X}) \mid label_\mu(t) \stackrel{\mathsf{LR}}{\longrightarrow}^{!}_\mu s\}$. We say that a TRS is $LR(\mu)$-terminating if, for all $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, there is no infinite $\stackrel{\mathsf{LR}}{\to}_\mu$-rewrite sequence starting from $label_\mu(t)$ [Lucas, 2002b].

Chapter 7

On-demand Strategy Annotations

This chapter provides a suitable extension of the positive E-evaluation strategy of OBJ-like languages to general (positive as well as negative) annotations. Such an extension is conservative, i.e., programs which only use positive strategy annotations and that are executed under our strategy behave exactly as if they were executed under the standard OBJ evaluation strategy. In Section 7.1, we give some brief motivation. In Section 7.2, we discuss some drawbacks for dealing with on-demand strategy annotations regarding the treatment of demandedness in the current proposals in the literature as well as in current implementations. In Section 7.3, we (re-)formulate the computational model of *on-demand strategy annotations* by handling demandedness in a different way. The new definition provides a corrected and well-defined approach to demandedness via E-strategies, which solves the problems detected in the previous proposals. Such a new on-demand strategy also holds better computational properties. In Section 7.4, we show that our definition behaves better than lazy rewriting (LR) [Fokkink *et al.*, 2000] and on-demand rewriting (ODR) [Lucas, 2001a] regarding the ability to compute (head-)normal forms. Section 7.5 introduces a transformation which can be used to formally prove termination of programs that use our computational model for implementing arbitrary strategy annotations. In order to prove the practicality of our ideas, we present in Section 7.6 an interpreter of our on-demand evaluation strategy introduced in this chapter, together with some promising experiments. Section 7.7 concludes and summarizes our contributions.

A short version of this chapter appeared in [Alpuente et al., 2002b]

7.1 Introduction

Eager rewriting-based programming languages such as Lisp, OBJ*, CafeOBJ, ELAN, or Maude evaluate expressions by innermost rewriting. Since nontermination is a known problem of innermost reduction, *syntactic annotations* (generally specified as sequences of integers associated to function arguments, called *local strategies*) have been used in OBJ2 [Futatsugi *et al.*, 1985], OBJ3 [Goguen *et al.*, 2000], CafeOBJ [Futatsugi and Nakagawa, 1997], and Maude [Clavel *et al.*, 1996] to improve efficiency and (hopefully) avoid nontermination. Local strategies are used in OBJ programs[1] for guiding the *evaluation strategy* (abbr. E-strategy): when considering a function call $f(t_1, \ldots, t_k)$, only the arguments whose indices are present as *positive* integers in the local strategy for f are evaluated (following the specified ordering). If 0 is found, then the evaluation of f is attempted. See Section 6.4 for details.

The limits of using only positive annotations regarding correctness and completeness of computations are discussed in [Lucas, 2001a, 2002b; Nakamura and Ogata, 2001; Ogata and Futatsugi, 2000]: the obvious problem is that the absence of some indices in the local strategies can have a negative impact in the ability of such strategies to compute normal forms. See Section 6.1 for details.

Example 7.1 *Consider the following* OBJ *program from Example 6.6:*

```
obj 2ND is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (1)] .
  op 2nd  : LNat -> Nat .
  op from : Nat -> LNat .
  vars X Y : Nat . var Z : LNat .
  eq 2nd(cons(X,cons(Y,Z))) = Y .
  eq from(X) = cons(X,from(s(X))) .
endo
```

The OBJ *evaluation of* `2nd(from(0))` *is given by the sequence:*

$\texttt{2nd(}\underline{\texttt{from(0)}}\texttt{)} \to \texttt{2nd(cons(0,from(s(0))))}$

The evaluation stops here since reductions on the second argument of `cons` *are disallowed (index* 2 *is not included in the strategy for* `cons`*). Note that we cannot apply the rule defining* `2nd` *because the subterm* `from(s(0))` *should be further reduced. Thus, a further step is* demanded *(by the rule of* `2nd`*) in order to obtain the desired outcome:*

[1] By OBJ we mean OBJ2, OBJ3, CafeOBJ, or Maude.

`2nd(cons(0,from(s(0))))` $\rightarrow$ `2nd(cons(0,cons(s(0),from(s(s(0))))))`

Now, we do not *need to reduce the second argument of the inner occurrence of* `cons` *anymore, since reducing at the root position yields the final value:*

`2nd(cons(0,cons(s(0),from(s(s(0))))))` $\rightarrow$ `s(0)`

Therefore, the rather intuitive notion of demanded evaluation of an argument of a function call arises as a possible solution to this problem (see Section 6.1.2). In [Nakamura and Ogata, 2001; Ogata and Futatsugi, 2000], *negative* indices are proposed to indicate those arguments that should be evaluated only 'on-demand', where the 'demand' is an attempt to match an argument term with the left-hand side of a rewrite rule [Eker, 2000; Goguen *et al.*, 2000; Ogata and Futatsugi, 2000]. For instance, as introduced in Example 6.7, in [Nakamura and Ogata, 2001] the authors suggest `(1 -2)` as the "apt" local strategy for `cons` in Example 6.6.

The inspiration for the local strategies of OBJ comes from *lazy rewriting* (*LR*) [Fokkink *et al.*, 2000], introduced in Section 7.4.1, the demand-driven technique where syntactic annotations enable the eager evaluation of function arguments and the default strategy is some kind of (on-demand) lazy evaluation. However, the extended, on-demand E-strategy of [Nakamura and Ogata, 2001; Ogata and Futatsugi, 2000] presents a number of drawbacks, which we formally address in this chapter. The following example illustrates that the notion of demandedness which is formalized in [Nakamura and Ogata, 2001] needs to be revisited.

Example 7.2 *Consider the following* OBJ *program:*

```
obj LENGTH is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat  [strat (1)] .
  op from : Nat -> LNat .
  op length : LNat -> Nat [strat (0)] .
  op length' : LNat -> Nat [strat (-1 0)] .
  var X : Nat . var Z : LNat .
  eq from(X) = cons(X,from(s(X))) .
  eq length(nil) = 0 .
  eq length(cons(X,Z)) = s(length'(Z)) .
  eq length'(Z) = length(Z) .
endo
```

The expression `length'(from(0))` *is rewritten (in one step) to* `length(from(0))`. *No evaluation is demanded on the argument of* `length'` *for enabling this step and no further evaluation on* `length(from(0))` *should be performed due to the local strategy* `(0)` *of* `length`. *However, the annotation list* `(-1 0)` *of function* `length'` *is treated in such a way that the on-demand evaluation of the expression* `length'(from(0))` *yields an infinite sequence (whether we use the operational model in [Ogata and Futatsugi, 2000] or whether we use [Nakamura and Ogata, 2001]) of Section 6.5. For instance,* CafeOBJ *ends with a stack overflow*[2]*:*

```
Ex2> red length'(from(0)) .
-- reduce in Ex2 : length'(from(0))
Error: Stack overflow (signal 1000)
```

This is because the negative annotations are implemented as marks on terms which can (improperly) enable reductions later on; see Example 7.3 below.

7.2 Drawbacks of the On-demand Evaluation Strategy

Note that, as pointed out in Section 6.5, the computational description of on-demand strategy annotations given in Definition 6.25 involves recursive steps. That is, a single reduction step on a (labeled) term t may involve the application of more than one reduction step on subterms of t (as it was shown by the application of steps $4(c)$ and $4(d)$ of Definition 6.25 in Example 6.26). In fact, the definition of a single rewriting step may depend on the possibility of evaluating some arguments of the considered function call. This implies that the one-step reduction relation proposed by Nakamura and Ogata is generally undecidable. Therefore, associated notions such as normal form (w.r.t. their reduction relation) are also undecidable.

Furthermore, as remarked in our introduction (Example 7.2 above), the notion of demandedness formalized in [Nakamura and Ogata, 2001] needs to be revisited.

Example 7.3 *Following the Example 7.2. The on-demand evaluation of term* `length'(from(0))` *following Definition 6.25 yields the infinite sequence of Figure 7.4.*

In the first reduction step, annotation -1 of symbol `length'` *is consumed according to case 3 of Definition 6.25 and, thus, subterm* $\mathtt{from}^{0}_{(1\ 0)}(\mathtt{0}^{0}_{nil})$ *is marked with the on-demand (superscript) flag. Annotation* `0` *of* `length'` *is reached and the whole term is*

[2] Negative annotations are (syntactically) accepted in current OBJ implementations, namely OBJ3, Maude, and CafeOBJ, but they have no effect over the computations of OBJ3 and Maude, whereas CafeOBJ follows the computational model of [Ogata and Futatsugi, 2000; Nakamura and Ogata, 2001] which is not much better in practice, as we discuss in this chapter.

$$
\begin{array}{l}
\langle \texttt{length'}^{0}_{(\boxed{\text{-1}}\ 0)}(\texttt{from}^{0}_{(1\ 0)}(\texttt{0}^{0}_{nil})), \Lambda\rangle \\
\quad \to_{\varphi} \ \langle \underline{\texttt{length'}^{0}_{(\boxed{0})}(\texttt{from}^{1}_{(1\ 0)}(\texttt{0}^{1}_{nil}))}, \Lambda\rangle \\
\quad \to_{\varphi} \ \langle \texttt{length}^{0}_{(\boxed{0})}(\underline{\texttt{from}^{\boxed{1}}_{(1\ 0)}(\texttt{0}^{1}_{nil})}), \Lambda\rangle \\
\qquad\qquad \text{subsequence: } \langle \texttt{from}^{0}_{(\boxed{1}\ 0)}(\texttt{0}^{0}_{nil}), \Lambda\rangle \\
\qquad\qquad\qquad\qquad \to_{\varphi} \ \langle \texttt{from}^{0}_{(0)}(\texttt{0}^{0}_{\boxed{nil}}), 1\rangle \\
\qquad\qquad\qquad\qquad \to_{\varphi} \ \langle \underline{\texttt{from}^{0}_{(\boxed{0})}(\texttt{0}^{0}_{nil})}, \Lambda\rangle \\
\qquad\qquad\qquad\qquad \to_{\varphi} \ \langle \texttt{cons}^{0}_{(\boxed{1})}(\texttt{0}^{0}_{nil}, \texttt{from}^{0}_{(0)}(\texttt{s}^{0}_{(1)}(\texttt{0}^{0}_{nil}), \Lambda\rangle \\
\qquad\qquad\qquad\qquad \to_{\varphi} \ \langle \texttt{cons}^{0}_{nil}(\texttt{0}^{0}_{\boxed{nil}}, \texttt{from}^{0}_{(0)}(\texttt{s}^{0}_{(1)}(\texttt{0}^{0}_{nil}), 1\rangle \\
\qquad\qquad\qquad\qquad \to_{\varphi} \ \langle \texttt{cons}^{0}_{nil}(\texttt{0}^{0}_{nil}, \texttt{from}^{0}_{(0)}(\texttt{s}^{0}_{(1)}(\texttt{0}^{0}_{nil})), \Lambda\rangle \\
\quad \to_{\varphi} \ \langle \underline{\texttt{length}^{0}_{(\boxed{0})}(\texttt{cons}^{1}_{nil}(\texttt{0}^{1}_{nil}, \texttt{from}^{1}_{(0)}(\texttt{s}^{1}_{(1)}(\texttt{0}^{1}_{nil}))))}, \Lambda\rangle \\
\quad \to_{\varphi} \ \langle \texttt{s}^{0}_{(\boxed{1})}(\texttt{length'}^{0}_{(-1\ 0)}(\texttt{from}^{1}_{(1\ 0)}(\texttt{s}^{1}_{(1)}(\texttt{0}^{1}_{nil})))), \Lambda\rangle \\
\quad \to_{\varphi} \ \langle \texttt{s}^{0}_{nil}(\texttt{length'}^{0}_{(-1\ 0)}(\texttt{from}^{1}_{(1\ 0)}(\texttt{s}^{1}_{(1)}(\texttt{0}^{1}_{nil})))), 1\rangle \\
\quad \to_{\varphi} \ \cdots
\end{array}
$$

Figure 7.4: On-demand evaluation of term `length'(from(0))`

rewritten using rule `length'(Z) = length(Z)`*, according to case* 4(*a*) *of Definition 6.25. Then, annotation* `0` *of* `length` *is reached but the whole term is not a redex, and an on-demand position has to be looked for. Here, the function* $DF_{\mathcal{R}}$ *for calculating demanded positions returns position* 1*, i.e.* $DF_{\mathcal{R}}(\texttt{length(from(0))}) = 1$*, and this position is marked with the on-demand flag, thus, case* 4(*c*) *or* 4(*d*) *is applied and position* 1 *is reduced using a different evaluation (sub-)sequence. Then,* `length` *is rewritten to* `length'` *and the cycle is repeated.*

Note that, within the labeled term $\texttt{length}^{0}_{(0)}(\texttt{from}^{1}_{(1\ 0)}(\texttt{0}^{1}_{nil}))$*, the strategy does not recognize that the (activated) on-demand flags on symbols* `from` *and* `0` *do* not *come from the local annotation for* `length`*. That is, the strategy does not maintain any kind of record about the origin of on-demand flags. Hence, it (unnecessarily) evaluates the argument of* `length`*. Moreover, at this point, this evaluation does* not *correspond to the 'intended' meaning of the strategy annotations that the programmer may have in mind (since the specific annotation* `(0)` *for* `length` *forbids reductions on its argument).*

On the other hand, the two existing definitions for the on-demand E-strategy (namely Nakamura and Ogata's [Nakamura and Ogata, 2001] and Ogata and Futatsugi's [Ogata and Futatsugi, 2000]) sensibly differ. For instance, Nakamura and Ogata select a demanded position for evaluating a given term t by taking the maximum of all positions demanded on t by each rule of the TRS (according to the lexicographic ordering on positions). On the other hand, in Ogata and Futatsugi's selection of demanded positions, the ordering of the rules in the program is extremely important.

Example 7.5 *Consider the* OBJ *program of Example 7.2 with the strategy* `(1 0)` *for* `length`*, together with the function* `geq` *defined by the following module*[3]*:*

```
obj LENGTH-GEQ is
  protecting LENGTH .
  sorts Bool .
  op true  : -> Bool .
  op false : -> Bool .
  op geq   : Nat Nat -> Bool    [strat (-1 -2 0)] .
  vars X Y : Nat .
  eq geq(s(X),s(Y)) = geq(X,Y) .
  eq geq(X,0) = true .
endo
```

Consider the expression `geq(length(from(0)),length(nil))`*. According to Ogata and Futatsugi's definition of the on-demand E-strategy, an infinite reduction sequence is overtaken since position* 1 *is selected as demanded and, thus, its (non-terminating) evaluation attempted. For instance,* CafeOBJ *ends with a stack overflow:*

```
Ex3> red geq(length(from(0)),length(nil)) .
-- reduce in Ex3 : geq(length(from(0)),length(nil))
Error: Stack overflow (signal 1000)
```

However, Nakamura and Ogata's definition of on-demand E-strategy selects position 2 *as demanded and, after the evaluation, the second rule is applied, thus obtaining the expected outcome* `true`*.*

We claim that it is possible to provide a more practical framework for implementing and studying OBJ computations, which may integrate the most interesting features of modern evaluation strategies with on-demand syntactic annotations. This is made precise in the sequel.

7.3 Improving Rewriting under On-demand Strategy Annotations

Let us summarize the drawbacks of existing operational models for arbitrary strategy annotations as follows: (1) the one-step reduction relation is, in general, undecidable; (2) the implementation of demandedness by using negative annotations (via the

[3] The reserved word `protecting` can be understood as *module reuse* in the OBJ syntax.

marking of terms with flag 0 or flag 1) enables evaluation steps that shouldn't be allowed, since (3) it does not properly keep track of the origin of the marks (lack of memory, see Example 7.3). Here, we want to introduce an additional consideration which can be used for improving the previous definitions. Let us show it up by means of an example.

Example 7.6 *Consider the* OBJ *program of Example 7.5, together with the following function* `lt`*:*

```
obj LENGTH-GEQ-LT is
  protecting LENGTH-GEQ .
  op lt : Nat Nat -> Bool    [strat (-1 -2 0)] .
  vars X Y : Nat .
  eq lt(0,s(Y)) = true .
  eq lt(s(X),s(Y)) = lt(X,Y) .
endo
```

Consider the expression $t =$ `lt(length(from(0)),0)`*, which is a* head-normal form *since no possible evaluation can lead the expression to match the left-hand side of a rule due to subterm* `0` *at position* 2*. Neither Nakamura and Ogata's nor Ogata and Futatsugi's formulations are able to avoid evaluations on* t*. For instance,* CafeOBJ *ends with a stack overflow:*

```
Ex4> red lt(length(from(0)),0).
-- reduce in Ex4 : lt(length(from(0)),0)
Error: Stack overflow (signal 1000)
```

Nevertheless, by exploiting the standard distinction between constructor and defined symbols of a signature in the presence of a TRS, it is easy to detect that no rule could ever be matched. Indeed, `0` *is a constructor symbol in the input term* t *and, hence, it cannot be reduced for improving the matching of* t *against the left-hand side of the rule for* `lt`*. See [Alpuente et al., 1997; Antoy and Lucas, 2002; Moreno-Navarro and Rodríguez-Artalejo, 1992] for a more detailed motivation and formal discussion of the use of these ideas for defining and using optimized demand-driven strategies.*

In the following, we propose a revised and refined definition of the on-demand E-strategy which takes into account all previous considerations. The two important points are the use of two lists of annotations for each symbol (instead of using only one as in the on-demand evaluation of Section 6.5) and a kind of flag for avoiding recursive definitions of the strategy (as it happens in [Nakamura and Ogata, 2001]).

Given a E-strategy map φ, we use the signature[4] $\mathcal{F}^{\sharp}_{\varphi} = \cup\{f_{L_1|L_2}, \overline{f}_{L_1|L_2} \mid f \in \mathcal{F} \wedge L_1, L_2 \in \mathbb{L}_{ar(f)}.(L_1{+\!+}L_2 \sqsubseteq \varphi(f))\}$ and labeled variables $\mathcal{X}^{\sharp}_{\varphi} = \{x_{nil|nil} \mid x \in \mathcal{X}\}$

[4] The function ++ defines the concatenation of two sequences of integers.

for marking ordinary terms $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ as terms $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi,\mathcal{X}^\sharp_\varphi)$. Overlining the root symbol of a subterm means that no evaluation is required for that subterm, and the control goes back to the parent; the auxiliary list L_1 in the subscript $L_1 \mid L_2$ is interpreted as a kind of memory of previously considered annotations. We use $f^\sharp$ to denote f or $\overline{f}$ for a symbol $f \in \mathcal{F}$. We define the list of active indices of a labeled symbol $f^\sharp_{L_1|L_2}$ as $active(f^\sharp_{L_1|L_2}) = \begin{cases} L_1 & \text{if } L_1 \neq nil \\ L_2 & \text{if } L_1 = nil \end{cases}$. The operator φ is extended to a mapping from $\mathcal{T}(\mathcal{F},\mathcal{X})$ to $\mathcal{T}(\mathcal{F}^\sharp_\varphi,\mathcal{X}^\sharp_\varphi)$ as follows:

$$\varphi(t) = \begin{cases} x_{nil|nil} & \text{if } t = x \in \mathcal{X} \\ f_{nil|\varphi(f)}(\varphi(t_1),\ldots,\varphi(t_k)) & \text{if } t = f(t_1,\ldots,t_k) \end{cases}$$

Also, the operator $erase : \mathcal{T}(\mathcal{F}^\sharp_\varphi,\mathcal{X}^\sharp_\varphi) \to \mathcal{T}(\mathcal{F},\mathcal{X})$ removes labelings from terms.

Definition 7.7 *We define the set of demanded positions of $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ w.r.t. l (a lhs of a rule defining $root(t)$), i.e. the set of (positions of) maximal disagreeing subterms as:*

$$DP_l(t) = \begin{cases} minimal_\leq(\mathcal{P}os_{\neq}(t,l)) & \textit{if } minimal_\leq(\mathcal{P}os_{\neq}(t,l)) \subseteq \mathcal{P}os_{\mathcal{D}}(t) \\ \varnothing & \textit{otherwise} \end{cases}$$

where $\mathcal{P}os_{\neq}(t,l)$ was given in Definition 6.22.

Note that the problem described in Example 7.6 is solved by restricting our attention to disagreeing positions that correspond only to defined symbols (by using $\mathcal{P}os_{\mathcal{D}}(t)$), since only these positions can produce a redex (see [Alpuente *et al.*, 1997; Loogen *et al.*, 1993; Moreno-Navarro and Rodríguez-Artalejo, 1992]).

Example 7.8 *Consider again Example 7.1 and lhs $l =$ `2nd(cons(X,cons(Y,Ys)))`. Let $t_1 =$ `2nd(cons(0,nil))`, we have $DP_l(t_1) = \varnothing$, i.e. no position is demanded by l due to a constructor clash with subterm `nil` at position 1.2. Let $t_2 =$ `2nd(cons(0,from(s(0))))`, we have $DP_l(t_2) = \{1.2\}$, i.e. position 1.2 is demanded by l because it is function-rooted.*

Example 7.9 *Continuing Example 7.6 where $l_1 =$ `lt(0,s(Y))` and $l_2 =$ `lt(s(X),s(Y))` are the lhs's of the rules. Let $t_1 =$ `lt(length(from(0)),0)`, we have $DP_{l_1}(t_1) = \varnothing$ and $DP_{l_2}(t_1) = \varnothing$, i.e. no position is demanded by l_1 or l_2 due to a constructor clash with subterm `0` at position 2. Let $t_2 =$ `lt(length(from(0)),length(nil))`, we have $DP_{l_1}(t_2) = \{1,2\}$ and $DP_{l_2}(t_2) = \{1,2\}$, i.e. positions 1 and 2 are demanded by l_1 and l_2 since both positions are function-rooted. And, let $t_3 =$ `lt(0,length(nil))`, we have $DP_{l_1}(t_3) = \{2\}$ but $DP_{l_2}(t_3) = \varnothing$, i.e. position 2 is demanded by l_1 but not by l_2 because of a constructor conflict with l_2.*

We define the set of *positive* positions of a term $s \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ as $\mathcal{P}os_P(s) = \{\Lambda\} \cup \{i.\mathcal{P}os_P(s|_i) \mid i > 0 \text{ and } i \in \mathit{active}(\mathit{root}(s))\}$ and the set of *active* positions as $\mathcal{P}os_A(s) = \{\Lambda\} \cup \{i.\mathcal{P}os_A(s|_i) \mid i > 0 \text{ and } i \in \mathit{active}(\mathit{root}(s)) \text{ or } -i \in \mathit{active}(\mathit{root}(s))\}$. We also define the set of positions with *empty* annotation list as $\mathcal{P}os_{nil}(s) = \{p \in \mathcal{P}os(s) \mid \mathit{root}(s|_p) = f_{L|nil}\}$. Then, the set of *active demanded* positions of a term $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ w.r.t. l, where l is the lhs of a rule defining $\mathit{root}(\mathit{erase}(t))$, is defined as follows:

$$\mathit{ADP}_l(t) = \begin{cases} \mathit{DP}_l(\mathit{erase}(t)) \cap \mathcal{P}os_A(t) & \text{if } \mathit{DP}_l(\mathit{erase}(t)) \not\subseteq \mathcal{P}os_P(t) \cup \mathcal{P}os_{nil}(t) \\ \varnothing & \text{otherwise} \end{cases}$$

and we define the set of active demanded positions of $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ w.r.t. TRS $\mathcal{R}$ as $\mathit{ADP}_\mathcal{R}(t) = \cup\{\mathit{ADP}_l(t) \mid l \to r \in \mathcal{R} \wedge \mathit{root}(\mathit{erase}(t)) = \mathit{root}(l)\}$.

Note that the restriction of active demanded positions to non-positive and non-empty positions is consistent w.r.t the intended meaning of strategy annotations since positive or empty positions should not be evaluated on-demand.

Example 7.10 *Let us consider the lhs* $l = \texttt{2nd(cons(X,cons(Y,Yz)))}$*. Let*

$$t_1 = \texttt{2nd}_{(1)|(0)}(\texttt{cons}_{(1\ 2)|nil}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(1\ 0)}(\texttt{s}_{nil|(1)}(\texttt{0}_{nil|nil}))))$$

we have $\mathit{DP}_l(\mathit{erase}(t_1)) = \{1.2\}$ *but* $\mathit{ADP}_l(t_1) = \varnothing$*, i.e. position* 1.2 *is demanded by* l *but it is a positive position. Let*

$$t_2 = \texttt{2nd}_{(1)|(0)}(\texttt{cons}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|nil}(\texttt{s}_{nil|(1)}(\texttt{0}_{nil|nil}))))$$

we have $\mathit{ADP}_l(t_2) = \varnothing$*, i.e. position* 1.2 *is still demanded by* l *but it is rooted by a symbol with an empty annotation list. Let*

$$t_3 = \texttt{2nd}_{(1)|(0)}(\texttt{cons}_{(1)|nil}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(1\ 0)}(\texttt{s}_{nil|(1)}(\texttt{0}_{nil|nil}))))$$

we have $\mathit{ADP}_l(t_3) = \varnothing$*, i.e. position* 1.2 *is again demanded by* l *but it is not an active position. Finally, let*

$$t_4 = \texttt{2nd}_{(1)|(0)}(\texttt{cons}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(1\ 0)}(\texttt{s}_{nil|(1)}(\texttt{0}_{nil|nil}))))$$

we have $\mathit{ADP}_l(t_4) = \{1.2\}$*.*

Example 7.11 *(Continuing Example 7.2) For* $t_1 = \texttt{length}_{(1)|(0)}(\texttt{from}_{nil|(1\ 0)}(\texttt{0}_{nil|nil}))$*, we have* $\mathit{DP}_l(\mathit{erase}(t_1)) = \{1\}$ *but* $\mathit{ADP}_l(t_1) = \varnothing$*, i.e. the (positive) position* 1 *is demanded by* l*. For* $t_2 = \texttt{length}_{(-1)|(0)}(\texttt{from}_{nil|nil}(\texttt{0}_{nil|nil}))$*, we have* $\mathit{ADP}_l(t_2) = \varnothing$*, i.e. position* 1 *is still demanded by* l *but it is rooted by a symbol with an empty annotation list. For* $t_3 = \texttt{length}_{nil|(0)}(\texttt{from}_{nil|(1\ 0)}(\texttt{0}_{nil|nil}))$*, we have* $\mathit{ADP}_l(t_3) = \varnothing$*, i.e. position* 1 *is still demanded by* l *but it is not active. Finally, for* $t_4 = \texttt{length}_{(-1)|(0)}(\texttt{from}_{nil|(1\ 0)}(\texttt{0}_{nil|nil}))$*, we have* $\mathit{ADP}_l(t_4) = \{1\}$*.*

Given a term $s \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$, the total ordering $\leq_s$ between active positions of s is defined as (1) $\Lambda \leq_s p$ for all $p \in \mathcal{P}os_A(s)$; (2) if $i.p, i.q \in \mathcal{P}os_A(s)$ and $p \leq_{s|_i} q$, then $i.p \leq_s i.q$; and (3) if $i.p, j.q \in \mathcal{P}os_A(s)$, $i \neq j$, and i (or $-i$) appears before j (or $-j$) in $active(root(s))$, then $i.p \leq_s j.q$. The ordering $\leq_s$ allows us to choose a position from the set of active demanded positions in s, which is consistent with user's annotations (see $min_{\leq_s}$ below). We define the set $OD_\mathcal{R}(s)$ of on-demand positions of a term $s \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ w.r.t. TRS $\mathcal{R}$ as follows:

$$\text{if } ADP_\mathcal{R}(s) = \varnothing \text{ then } OD_\mathcal{R}(s) = \varnothing \text{ else } OD_\mathcal{R}(s) = \{min_{\leq_s}(ADP_\mathcal{R}(s))\}$$

Example 7.12 *(Continuing Example 7.10) Given the term*

$$t_5 = \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(-1\ -2)|nil}(\mathtt{2nd}_{nil|(10)}(\mathtt{nil}_{nil|nil}), \mathtt{from}_{nil|(1\ 0)}(\mathtt{0}_{nil|nil})))$$

we have $ADP_l(t_5) = \{1.1, 1.2\}$ *whereas* $OD_{\{l\}}(t_5) = \{1.1\}$ *since annotation* `-1` *appears before* `-2` *in the memoizing list for symbol* `cons`.

Given a term $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and position $p \in \mathcal{P}os(t)$, $mark(t,p)$ is the term s with all symbols occurring at positions above p (except the root) marked as non-evaluable terms, in symbols $\mathcal{P}os(s) = \mathcal{P}os(t)$ and $\forall q \in \mathcal{P}os(t)$, if $\Lambda < q < p$ and $root(t|_q) = f_{L_1|L_2}$, then $root(s|_q) = \overline{f}_{L_1|L_2}$, otherwise $root(s|_q) = root(t|_q)$. This is useful for avoiding recursive invocations of the evaluation strategy, as it is shown below.

Example 7.13 *Continuing Example 7.10 and term*

$$t_4 = \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil}))))$$

We have that $mark(t_4, 1.2) =$

$$\mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil}))))$$

We formulate a binary relation $\stackrel{\sharp}{\to}_\varphi$ on the set $\mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi) \times \mathbb{N}^*_+$, such that a single reduction step on a (labeled) term t does not involve the recursive application of reduction steps on t. In the following definition, the symbol @ denotes appending an element at the end of a list.

Definition 7.14 *Given a TRS* $\mathcal{R} = (\mathcal{F}, R)$ *and an arbitrary E-strategy map* φ *for* $\mathcal{F}$*,* $eval_\varphi : \mathcal{T}(\mathcal{F},\mathcal{X}) \to \mathcal{P}(\mathcal{T}(\mathcal{F},\mathcal{X}))$ *is defined as*

$$eval_\varphi(t) = \{erase(s) \in \mathcal{T}(\mathcal{F},\mathcal{X}) \mid \langle\varphi(t), \Lambda\rangle \stackrel{\sharp}{\to}^!_\varphi \langle s, \Lambda\rangle\}$$

The binary relation $\stackrel{\sharp}{\to}_\varphi$ *on* $\mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi) \times \mathbb{N}^*_+$ *is defined as follows:* $\langle t, p\rangle \stackrel{\sharp}{\to}_\varphi \langle s, q\rangle$ *if and only if* $p \in \mathcal{P}os(t)$ *and either*

1. $t|_p = f_{L|nil}(t_1, \ldots, t_k)$, $s = t$ *and* $p = q.i$ *for some* i*; or*

2. $t|_p = f_{L_1|i:L_2}(t_1, \ldots, t_k)$, $i > 0$, $s = t[f_{L_1 @ i|L_2}(t_1, \ldots, t_k)]_p$ *and* $q = p.i$*; or*

3. $t|_p = f_{L_1|-i:L_2}(t_1, \ldots, t_k)$, $i > 0$, $s = t[f_{L_1 @ -i|L_2}(t_1, \ldots, t_k)]_p$ *and* $q = p$*; or*

4. $t|_p = f_{L_1|0:L_2}(t_1, \ldots, t_k) = \sigma(l')$, $erase(l') = l$, $s = t[\sigma(\varphi(r))]_p$ *for some* $l \to r \in R$ *and substitution* σ, $q = p$*; or*

5. $t|_p = f_{L_1|0:L_2}(t_1, \ldots, t_k)$, $erase(t|_p)$ *is not a redex,* $OD_{\mathcal{R}}(t|_p) = \varnothing$, $s = t[f_{L_1|L_2}(t_1, \ldots, t_k)]_p$, *and* $q = p$*; or*

6. $t|_p = f_{L_1|0:L_2}(t_1, \ldots, t_k)$, $erase(t|_p)$ *is not a redex,* $OD_{\mathcal{R}}(t|_p) = \{p'\}$, $s = t[mark(t|_p, p')]_p$, $q = p.p'$*; or*

7. $t|_p = \overline{f}_{L_1|L_2}(t_1, \ldots, t_k)$, $s = t[f_{L_1|L_2}(t_1, \ldots, t_k)]_p$ *and* $p = q.i$ *for some* i.

Cases 1 and 2 of Definition 7.14 essentially correspond to cases 1 and 2 of Definitions 6.18 and 6.25; that is, (1) no more annotations are pending and the evaluation is completed, or (2) a positive argument index is provided and the evaluation goes down to the subterm corresponding to such argument (note that the index is stored here, in contrast to Definitions 6.18 and 6.25). Case 3 just stores the negative index for further use. Cases 4, 5, and 6 correspond to the attempt to match the term against the left-hand sides of the rules of the program. Case 4 applies if the considered (unlabeled) subterm is a redex (which is, then, contracted). If the subterm is not a redex, cases 5 and 6 are considered (possibly involving some on-demand evaluation). We use the lists of indices labeling the symbols for fixing the concrete positions on which on-demand evaluations are allowed; in particular, the first (memoizing) list is crucial for achieving this (by means of the function *active* and the order $\leq_s$ used in the definition of the set $OD_{\mathcal{R}}(s)$ of on-demand positions of a term s). Case 5 applies if no demanded evaluation is allowed (or required). Case 6 applies if the on-demanded evaluation of the subterm $t|_{p.p'}$ is required, i.e. $OD_{\mathcal{R}}(t|_p) = \{p'\}$. In this case, the symbols lying along the path from $t|_p$ to $t|_{p.p'}$ (excluding the border ones) are overlined. Then, the evaluation process continues on $t|_{p.p'}$. Once the evaluation of $t|_{p.p'}$ has finished, the only possibility is the repeated (but possibly empty) application of steps issued according to the last case 7 which causes the evaluation control to come back to position p (which originated the on-demand evaluation).

Example 7.16 *Following the Example 7.1, the appropriate evaluation sequence for the term* `2nd(from(0))` *via* $eval^{\varphi}_{\mathcal{R}}$ *is depicted in Figure 7.15, which is similar to that shown in Example 6.7.*

$$
\begin{array}{l}
\langle \texttt{2nd}_{nil|(\boxed{1}\ 0)}(\texttt{from}_{nil|(1\ 0)}(\texttt{0}_{nil|nil})), \Lambda\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\texttt{from}_{nil|(\boxed{1}\ 0)}(\texttt{0}_{nil|nil})), 1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\texttt{from}_{(1)|(0)}(\texttt{0}_{\boxed{nil}|nil})), 1.1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\underline{\texttt{from}_{(1)|(\boxed{0})}(\texttt{0}_{nil|nil})}), 1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\texttt{cons}_{nil|(\boxed{1}\ -2)}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(1\ 0)}(\texttt{s}_{nil|(1)}(\texttt{0}_{nil|nil})))), 1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\texttt{cons}_{(1)|(\boxed{-2})}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(1\ 0)}(\texttt{s}_{nil|(1)}(\texttt{0}_{nil|nil})))), 1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\texttt{cons}_{(1\ -2)|\boxed{nil}}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(1\ 0)}(\texttt{s}_{nil|(1)}(\texttt{0}_{nil|nil})))), 1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\texttt{cons}_{(1\ \boxed{-2})|nil}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(1\ 0)}(\texttt{s}_{nil|(1)}(\texttt{0}_{nil|nil})))), \Lambda\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(\boxed{1}\ 0)}(\texttt{s}_{nil|(1)}(\texttt{0}_{nil|nil})))), 1.2\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(0)}(\texttt{s}_{nil|(\boxed{1})}(\texttt{0}_{nil|nil})))), 1.2.1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(0)}(\texttt{s}_{(1)|nil}(\texttt{0}_{nil|\boxed{nil}})))), 1.2.1.1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{from}_{nil|(0)}(\texttt{s}_{(1)|\boxed{nil}}(\texttt{0}_{nil|nil})))), 1.2.1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \underline{\texttt{from}_{(1)|(\boxed{0})}(\texttt{s}_{(1)|nil}(\texttt{0}_{nil|nil}))})), 1.2\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{cons}_{nil|(\boxed{1}\ -2)}(\texttt{s}_{(1)|nil}(\texttt{0}_{nil|nil}), \\
\qquad\qquad \texttt{from}_{nil|(1\ 0)}(\texttt{s}_{nil|(1)}(\texttt{s}_{(1)|nil}(\texttt{0}_{nil|nil})))))), 1.2\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{cons}_{(1)|(-2)}(\texttt{s}_{(1)|\boxed{nil}}(\texttt{0}_{nil|nil}), \cdots))), 1.2.1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{cons}_{(1)|(\boxed{-2})}(\texttt{s}_{(1)|nil}(\texttt{0}_{nil|nil}), \cdots))), 1.2\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{cons}_{(1\ -2)|\boxed{nil}}(\texttt{s}_{(1)|nil}(\texttt{0}_{nil|nil}), \cdots))), 1.2\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{2nd}_{(1)|(0)}(\overline{\texttt{cons}}_{(1\ -2)|\boxed{nil}}(\texttt{0}_{nil|nil}, \texttt{cons}_{(1\ -2)|nil}(\texttt{s}_{(1)|nil}(\texttt{0}_{nil|nil}), \cdots))), 1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \underline{\texttt{2nd}_{(1)|(\boxed{0})}(\texttt{cons}_{(1\ -2)|nil}(\texttt{0}_{nil|nil}, \texttt{cons}_{(1\ -2)|nil}(\texttt{s}_{(1)|nil}(\texttt{0}_{nil|nil}), \cdots)))}, \Lambda\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{s}_{(1)|nil}(\texttt{0}_{nil|nil}), \Lambda\rangle
\end{array}
$$

Figure 7.15: On-demand evaluation of term `2nd(from(0))`

Example 7.17 *(Continuing Examples 7.2 and 7.3). The on-demand evaluation of expression* `length'(from(0))` *under the refined on-demand strategy is the following:*

$$
\begin{array}{l}
\langle \texttt{length'}_{nil|(\boxed{-1}\ 0)}(\texttt{from}_{nil|(1\ 0)}(\texttt{0}_{nil|nil})), \Lambda\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \underline{\texttt{length'}_{(-1)|(\boxed{0})}(\texttt{from}_{nil|(1\ 0)}(\texttt{0}_{nil|nil}))}, \Lambda\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{length}_{nil|(\boxed{0})}(\texttt{from}_{nil|(1\ 0)}(\texttt{0}_{nil|nil})), \Lambda\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \texttt{length}_{nil|nil}(\texttt{from}_{nil|(1\ 0)}(\texttt{0}_{nil|nil})), \Lambda\rangle
\end{array}
$$

The evaluation clearly terminates and the result coincides with the intended behavior. Namely, in the first step, the negative annotation -1 *of* `length'` *is stored for further use according to case* 3 *of Definition 7.14. Annotation* `0` *of* `length'` *is reached, and the whole term is rewritten using rule* `length'(Z) = length(Z)`*, according to case* 4 *of Definition 7.14. Then, annotation* `0` *of* `length` *is reached but the whole term cannot be rewritten since it is not a redex, and demanded positions have to be looked for. However, no demanded position arises since the memoizing list of strategy* *annotations for* `length` *is empty (see* $ADP_{\mathcal{R}}(t_3)$ *in Example 7.11 above). Therefore, we obtain* `length(from(0))` *as the computed value of the evaluation, according to case* 5 *of Definition 7.14.*

In the following, we study important properties of our on-demand evaluation strategy.

7.3.1 Properties of the Refined On-demand Strategy

The following theorem shows that, for positive strategy annotations, each reduction step with $\overset{\sharp}{\to}_{\varphi}$ exactly corresponds to the original Nagaya's relation $\overset{\mathbb{N}}{\to}_{\varphi}$ of Definition 6.18 [Nagaya, 1999; Nakamura and Ogata, 2001]. For an E-strategy map φ and a term $t \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$, we define $positive : \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi}) \to \mathcal{T}(\mathcal{F}^{\mathbb{N}}_{\varphi}, \mathcal{X}^{\mathbb{N}}_{\varphi})$ as $positive(x_{nil|nil}) = x_{nil}$ for $x \in \mathcal{X}$, and $positive(f^{\sharp}_{L_1|L_2}(t_1, \ldots, t_n)) = f_{L'_2}(positive(t_1), \ldots, positive(t_n))$, where L'_2 is L_2 without the negative indices.

Theorem 7.18 *Let* $\mathcal{R}$ *be a TRS and* φ *be a positive* E*-strategy map. Let* $t, s \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$ *and* $p \in \mathcal{P}os(t)$*. Then,* $\langle t, p\rangle \overset{\sharp}{\to}_{\varphi} \langle s, q\rangle$ *if and only if* $\langle positive(t), p\rangle \overset{\mathbb{N}}{\to}_{\varphi} \langle positive(s), q\rangle$*.*

Proof. It is trivial according to Definition 6.18. □

Sometimes, it is interesting to get rid of the ordering of indices in local strategies. Then, we use *replacement maps* ($\mu \in M_{\mathcal{F}}$). Given an E-strategy map φ, μ^{φ} is the following replacement map $\mu^{\varphi}(f) = \{|i| \mid i \in \varphi(f) \wedge i \neq 0\}$. If $\mu^{\varphi} \in CM_{\mathcal{R}}$, we say that φ is a *canonical* E-strategy map (and, by abuse, we write $\varphi \in CM_{\mathcal{R}}$); see Section 6.2 for details. Given an E-strategy map φ, φ_+ denotes the E-strategy map obtained by taking away all negative indices for each symbol $f \in \mathcal{F}$. Note that $\varphi_+ \sqsubseteq \varphi$, for every E-strategy map φ.

In the following, we show that, for E-strategy maps φ whose positive part φ_+ is canonical, extra negative annotations can be completely disregarded. This means that negative annotations are only useful if the positive indices do not collect all indices in the canonical replacement map of the TRS. We introduce some auxiliary notation in order to prove Theorem 7.20. Let φ_1, φ_2 be two E-strategy maps such that $\varphi_1 \sqsubseteq \varphi_2$ and $t \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi_2}, \mathcal{X}^{\sharp}_{\varphi_2})$. We define the adaptation of term t from the strategy list φ_2 to the strategy list φ_1 as $<t>_{\varphi_1} = t' \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi_1}, \mathcal{X}^{\sharp}_{\varphi_1})$ such that, for all $p \in \mathcal{P}os(t)$, $root(t|_p) = f^{\sharp}_{L_1|L_2}$ implies that $root(t'|_p) = f^{\sharp}_{L'_1|L'_2}$ and L'_1 and L'_2 are the maximal sequences such that $L'_1 \sqsubseteq L_1$, and $L'_2 \sqsubseteq L_2$.

Lemma 7.19 *Let* $\mathcal{R}$ *be a TRS and* φ *be an* E*-strategy map such that* $\varphi_+ \in CM_{\mathcal{R}}$*. If* $t \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$*, then* $OD_{\mathcal{R}}(t) = \varnothing$*.*

Proof. Immediate. By property $\varphi_+ \in CM_{\mathcal{R}}$, $DP_{\mathcal{R}}(erase(t)) \subseteq Paths_P(<t>_{\varphi_+}) \subseteq Paths_P(t)$. Thus, $OD_{\mathcal{R}}(t) = OD_{\mathcal{R}}(<t>_{\varphi_+}) = \varnothing$. □

Theorem 7.20 *Let $\mathcal{R}$ be a TRS and φ be an E-strategy map such that $\varphi_+ \in CM_{\mathcal{R}}$. Let $t, s \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and $p \in \mathcal{P}os(t)$. Then, $\langle t, p\rangle \stackrel{\sharp}{\to}_\varphi \langle s, q\rangle$ if and only if $\langle positive(t), p\rangle \stackrel{\mathbb{N}}{\to}_{\varphi_+} \langle positive(s), q\rangle$.*

Proof. Immediate, since term $positive(t)$ does not contain negative indices and, by Lemma 7.19, $OD_{\mathcal{R}}(t) = \varnothing$. □

Example 7.17 shows that evaluating terms by using on-demand strategy annotations can even lead to terms which are not head-normal forms. The following result establishes conditions ensuring that the normal forms of our refined on-demand strategy are ordinary head–normal forms (w.r.t. the TRS).

Theorem 7.21 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a left-linear CS, $\varphi \in CM_{\mathcal{R}}$ and $\varphi(f)$ ends in 0 for all $f \in \mathcal{D}$. Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. If $s \in eval_\varphi(t)$, then s is a head-normal form of t.*

Proof. First, note that it is not possible to have that $root(s) = \overline{f}_{L|nil}$ for $f \in \mathcal{F}$ since non-evaluable flags are raised only when a position is demanded and for the symbols occurring at positions between the root and the demanded position (excluding both).

We prove it by structural induction on s. If $s \in \mathcal{C}$ or $s \in \mathcal{X}$, it is trivial. Consider $s = f \in \mathcal{D}$ with $ar(f) = 0$. By assumption, $\varphi(f)$ ends with 0, thus the last rewriting step was $\langle f_{nil|(0)}, \Lambda\rangle \stackrel{\sharp}{\to}_\varphi \langle f_{nil|nil}, \Lambda\rangle$. The only case when this can happen is when $erase(f_{nil|(0)}) = f$ is not a redex and $OD_{\mathcal{R}}(f_{nil|(0)}) = \varnothing$. But this can happen only if $\nexists l \in L(\mathcal{R}).root(l) = f$. Hence, s is a head-normal form.

For the induction case, we omit the case $root(s) \in \mathcal{C}$ which is trivial. Consider $root(s) = f \in \mathcal{D}$. By assumption, $\varphi(f)$ ends with 0, thus, there are terms $t', s' \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ such that the last rewriting step was $\langle t', \Lambda\rangle \stackrel{\sharp}{\to}_\varphi \langle s', \Lambda\rangle$, $s = erase(s')$, $root(t') = f_{L|(0)}$ and $s' = f_{L|nil}(t'|_1, \ldots, t'|_{ar(f)})$. This can happen only when $erase(t')$ is not a redex and $OD_{\mathcal{R}}(t') = \varnothing$.

If $erase(t')$ is not a redex, then s is not a redex either and, by left-linearity, $\nexists l \in L(\mathcal{R})$ and $\sigma \in Subst(\mathcal{T}(\mathcal{F}, \mathcal{X})) : s = \sigma(l)$, i.e. $Pos_{\neq}(s, l) \neq \varnothing$ for all $l \in L(\mathcal{R})$. Moreover, if $DP_{\mathcal{R}}(s) = \varnothing$, s is a head-normal form because either $\nexists l \in L(\mathcal{R}).root(s) = root(l)$ (and then there is no rule which can be applied to s) or $\exists p \in \mathcal{P}os(s).\forall l \in L(\mathcal{R})$, $p \in \mathcal{P}os_{\neq}(s, l)$ and $root(s|_p) \notin \mathcal{D}$ (and then, by CS property, we know that the symbol at this position will never change by further reductions on the term). On the contrary, if $\exists p \in DP_{\mathcal{R}}(s)$, then exists $l \in L(\mathcal{R}).p \in DP_l(s)$. In this case, since $OD_{\mathcal{R}}(t') = \varnothing$, either $p \notin Paths_A(t')$ or $p \in Paths_P(t')$. But, since $\varphi \in CM_{\mathcal{R}}$, all symbols in l at positions which are above p have an index ($-i$ or i) in φ; thus, it is $p \in Paths_A(t')$ and $p \in Paths_P(t')$. Then, since all indices from Λ down to p are positive, position p was previously reduced and now, $root(t'|_p) = g_{L'|nil}$ for $g \in \mathcal{F} \cup \mathcal{X}$. Thus, by induction hypothesis, $s|_p$ is a head-normal form and, by CS property, the symbol g at this position will never disappear. Hence, s is also a head-normal form. $\square$

Left-linearity and CS conditions cannot be dropped, as [Lucas, 2001a] has shown for on-demand rewriting. The following two counterexamples are an adaptation of the ones in [Lucas, 2001a].

Example 7.22 *Consider the following TRS $\mathcal{R}$ from [Lucas, 2001a] which is not a CS:*

$$\mathtt{f(g(x,a))} \rightarrow \mathtt{a} \qquad \mathtt{g(a,b)} \rightarrow \mathtt{g(b,a)}$$

Let $\varphi(\mathtt{f}) = \mathtt{(-1\ 0)}$, $\varphi(\mathtt{g}) = \mathtt{(1\ 2\ 0)}$, *and* $\varphi(\mathtt{a}) = \varphi(\mathtt{b}) = nil$. *Term* $t = \mathtt{f(g(a,b))}$ *is not a head-normal form since* $\mathtt{f(\underline{g(a,b)})} \rightarrow \underline{\mathtt{f(g(b,a))}} \rightarrow \mathtt{a}$. *However, the head-normal form of t cannot be computed by* $\stackrel{\sharp}{\rightarrow}_\varphi$*:*

$$\begin{array}{l} \langle \mathtt{f}_{nil|(-1\ \ 0)}(\mathtt{g}_{nil|(1\ \ 2\ \ 0)}(\mathtt{a}_{nil|nil}, \mathtt{b}_{nil|nil})), \Lambda\rangle \\ \quad \stackrel{\sharp}{\rightarrow}_\varphi\ \langle \mathtt{f}_{(-1)|(0)}(\mathtt{g}_{nil|(1\ \ 2\ \ 0)}(\mathtt{a}_{nil|nil}, \mathtt{b}_{nil|nil})), \Lambda\rangle \\ \quad \stackrel{\sharp}{\rightarrow}_\varphi\ \langle \mathtt{f}_{(-1)|nil}(\mathtt{g}_{nil|(1\ \ 2\ \ 0)}(\mathtt{a}_{nil|nil}, \mathtt{b}_{nil|nil})), \Lambda\rangle \end{array}$$

Note that $1 \notin \mathcal{P}os_{\neq}(\mathtt{f(g(a,b))}, \mathtt{f(g(x,a))})$, *i.e. position* 1 *of t is not demanded by lhs* $\mathtt{f(g(x,a))}$ *since the symbols occurring at that positions in the two terms coincide.*

Example 7.23 *Consider the following TRS $\mathcal{R}$ from [Lucas, 2001a] which is not left-linear:*

$$\mathtt{f(x,x)} \rightarrow \mathtt{x} \qquad \mathtt{a} \rightarrow \mathtt{b}$$

Let $\varphi(\mathtt{f}) = \mathtt{(-1\ -2\ 0)}$, $\varphi(\mathtt{a}) = \mathtt{(0)}$, *and* $\varphi(\mathtt{b}) = nil$. *Term* $t = \mathtt{f(a,b)}$ *is not a head-normal form since* $\mathtt{f(\underline{a},b)} \rightarrow \underline{\mathtt{f(b,b)}} \rightarrow \mathtt{b}$. *However, the head-normal form of t cannot be computed by* $\stackrel{\sharp}{\rightarrow}_\varphi$*:*

$$\begin{array}{l} \langle \mathtt{f}_{nil|(-1\ \ -2\ \ 0)}(\mathtt{a}_{nil|nil}, \mathtt{b}_{nil|nil}), \Lambda\rangle \\ \quad \stackrel{\sharp}{\rightarrow}_\varphi\ \langle \mathtt{f}_{(-1)|(-2\ \ 0)}(\mathtt{a}_{nil|nil}, \mathtt{b}_{nil|nil}), \Lambda\rangle \\ \quad \stackrel{\sharp}{\rightarrow}_\varphi\ \langle \mathtt{f}_{(-1\ \ -2)|(0)}(\mathtt{a}_{nil|nil}, \mathtt{b}_{nil|nil}), \Lambda\rangle \\ \quad \stackrel{\sharp}{\rightarrow}_\varphi\ \langle \mathtt{f}_{(-1\ \ -2)|nil}(\mathtt{a}_{nil|nil}, \mathtt{b}_{nil|nil}), \Lambda\rangle \end{array}$$

Note that $1.1 \notin \mathcal{P}os_{\neq}(\mathtt{f(a,b)}, \mathtt{f(x,x)})$, *i.e. position* 1.1 *of t is not demanded by lhs* $\mathtt{f(x,x)}$ *since this position is under a variable in the lhs.*

Theorem 7.21 suggests the following *normalization via φ-normalization procedure* to obtain normal forms of a term t: given an E-strategy map φ and $s = f(s_1, \ldots, s_k) \in eval_\varphi(t)$, we evaluate t by (recursively) normalizing $s_1, \ldots, s_k$ using $eval_\varphi$. It is not difficult to see that confluence and φ-termination of the TRS guarantee that this procedure actually describes a normalizing strategy (see [Lucas, 2001a, 2002a] for the application of the same idea to on-demand rewriting and *CSR*, respectively).

In the next section, we show that our on-demand strategy improves *lazy rewriting* as well as *on-demand rewriting*.

7.4 Comparison with Other Techniques dealing with On-demand Annotations

7.4.1 Lazy Rewriting

In the following, we show that each evaluation step of our refined on-demand strategy is subsumed by some evaluation steps of lazy rewriting of Section 6.6. First, we give some preparatory definitions and results for Theorem 7.26 below.

Lemma 7.24 *Let $\mathcal{R}$ be a left-linear TRS and φ be an E-strategy map. Let $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$. If $OD_\mathcal{R}(t) = \{p\}$, then $\exists l \in L(\mathcal{R})$ such that l matches modulo laziness $label_{\mu^{\varphi+}}(erase(t))$; $root(erase(t)|_p) \neq root(l|_p) \notin \mathcal{X}$; for all $p' : \Lambda \leq p' < p$, $root(erase(t)|_{p'}) = root(l|_{p'})$; and at least one position $p' : \Lambda < p' \leq p$ is declared essential.*

Proof. By definition, there exists $l \in L(\mathcal{R})$ such that $p \in \mathcal{P}os_{\neq}(erase(t), l)$ and $root(erase(t)|_p) \neq root(l|_p) \notin \mathcal{X}$. By minimality, for all $p' : \Lambda \leq p' < p$, $root(erase(t)|_{p'}) = root(l|_{p'})$. Moreover, we have that $OD_\mathcal{R}(erase(t)) \cap Paths_P(t) = \varnothing$. That is, for all $q \in DP_\mathcal{R}(erase(t))$, $q \notin Paths_P(t)$ and then, $\exists q' \leq q$ such that $q' \in Paths_A(t)$, $root(label_{\mu^{\varphi+}}(erase(t))|_{q'}) = f^\ell$ and $\forall q'' : \Lambda \leq q'' < q'$, $root(label_{\mu^{\varphi+}}(erase(t))|_{q''}) = g^e$ where g is distinct for each q''. Hence, l matches modulo laziness t and the conclusion follows. $\square$

Lemma 7.25 *Let $\mathcal{R}$ be a left-linear TRS and φ be an E-strategy map. If $t, l', r' \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and $l \to r \in \mathcal{R}$ such that $t = \sigma(l')$, $erase(l') = l$, and $r' = \sigma(\varphi(r))$, then l matches $erase(t)$ and, let $t^\varphi = label_{\mu^{\varphi+}}(erase(t))$, there exists θ for* LR *such that $label_{\mu^{\varphi+}}(erase(r')) = \theta(label_{\mu^{\varphi+}}(r))$.*

Proof. Note that all variables of l' have the same labeling, i.e. $\mathcal{V}ar(l') = \{x_{nil|nil} \mid x \in \mathcal{V}ar(l)\}$. Note also that, if $t = \sigma(l')$, then l matches $erase(t)$, and there exists θ for LR such that $erase(\sigma(x')) = erase(\theta(x''))$ for $erase(x') = erase(x'') = x \in \mathcal{V}ar(l)$. Finally, $\theta(label_{\mu^{\varphi+}}(r)) = label_{\mu^{\varphi+}}(erase(\sigma(r)))$, i.e. $label_{\mu^{\varphi+}}(erase(r')) = \theta(label_{\mu^{\varphi+}}(r))$. $\square$

The following theorem establishes that each evaluation step of our refined on-demand strategy corresponds to some evaluation steps of lazy rewriting. Also, it shows that lazy rewriting (potentially) activates as many symbols (within a term) as our strategy does (we use the ordering $\leq_{lazy}$ for expressing this fact). Given a term $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and $p \in \mathcal{P}os(t)$, we translate the labeling of terms in $\mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ into the labeling of $\mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$ (see Section 6.6) by considering only positive annotations and transforming overlined symbols, as well as the symbols at the position under consideration, into eager symbols, as follows:

$$\begin{array}{ll} \multicolumn{2}{l}{lazy^p_\varphi(t) = \rho''_p(\rho'_t(label_{\mu^{\varphi+}}(erase(t)))) \text{ where}} \\ (1)\ \rho'_t(f^b(t_1,\dots,t_n)) = f^e(\rho'_{s_1}(t_1),\dots,\rho'_{s_n}(t_n)) & \text{if } t = \overline{f}_{L_1|L_2}(s_1,\dots,s_n), \\ (2)\ \rho'_t(f^b(t_1,\dots,t_n)) = f^b(\rho'_{s_1}(t_1),\dots,\rho'_{s_n}(t_n)) & \text{if } t = f_{L_1|L_2}(s_1,\dots,s_n), \\ (3)\ \rho''_p(s) = s[f^e(s_1,\dots,s_k)]_p & \text{for } s|_p = f^b(s_1,\dots,s_k) \end{array}$$

We define the ordering $\leq_{lazy}$ between terms $\mathcal{T}(\mathcal{F}_\mathcal{L},\mathcal{X}_\mathcal{L})$ by extending the ordering $f^e \leq_{lazy} f^e$ and $f^\ell \leq_{lazy} f^e$, for all $f \in \mathcal{F}$, to terms.

Theorem 7.26 *Let $\mathcal{R}$ be a left-linear TRS and φ be an E-strategy map. Let $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi,\mathcal{X}^\sharp_\varphi)$, $p \in \mathcal{P}os(t)$ and $\mu = \mu^{\varphi+}$. If $\langle t,p\rangle \xrightarrow{\sharp}_\varphi \langle s,q\rangle$ and $p \in \mathcal{A}ct(lazy^p_\varphi(t))$, then $q \in \mathcal{A}ct(lazy^q_\varphi(s))$ and $lazy^p_\varphi(t) \xrightarrow{\mathsf{LR}}{}^*_\mu s'$ for $s' \in \mathcal{T}(\mathcal{F}_\mathcal{L},\mathcal{X}_\mathcal{L})$ such that $lazy^q_\varphi(s) \leq_{lazy} s'$.*

Proof. We consider the different cases of Definition 7.14 separately.

1. If $t|_p = f_{L|nil}(t_1,\dots,t_k)$, $s = t$ and $p = q.i$ for some i, then $lazy^q_\varphi(s) \leq_{lazy} lazy^p_\varphi(t)$ and $lazy^p_\varphi(t) \xrightarrow{\mathsf{LR}}{}^=_\mu lazy^p_\varphi(t)$. Note that $q \in \mathcal{A}ct(lazy^q_\varphi(s))$ since $p = q.i \in \mathcal{A}ct(lazy^p_\varphi(t))$.

2. If $t|_p = f_{L_1|i:L_2}(t_1,\dots,t_k)$ with $i > 0$, $s = t[f_{L_1@i|L_2}(t_1,\dots,t_k)]_p$ and $q = p.i$, then $lazy^p_\varphi(t) = lazy^q_\varphi(s)$ and, since $i \in \mu(f)$, $q \in \mathcal{A}ct(lazy^q_\varphi(s))$, and $lazy^p_\varphi(t) \xrightarrow{\mathsf{LR}}{}^=_\mu lazy^q_\varphi(s)$.

3. If $t|_p = f_{L_1|-i:L_2}(t_1,\dots,t_k)$ with $i > 0$, $s = t[f_{L_1@-i|L_2}(t_1,\dots,t_k)]_p$ and $q = p$, then $lazy^p_\varphi(t) = lazy^q_\varphi(s)$, $q \in \mathcal{A}ct(lazy^q_\varphi(s))$ and $lazy^p_\varphi(t) \xrightarrow{\mathsf{LR}}{}^=_\mu lazy^q_\varphi(s)$.

4. If $t|_p = f_{L_1|0:L_2}(t_1,\dots,t_k) = \sigma(l')$, $erase(l') = l$, $s = t[\sigma(\varphi(r))]_p$ for some $l \to r \in R$ and substitution σ, and $q = p$, then by Lemma 7.25, there exists θ for LR such that $lazy^p_\varphi(t) \xrightarrow{R}_\mu lazy^p_\varphi(t)[\theta(label_\mu(r))]_p$. By Lemma 7.25, we also have that $lazy^p_\varphi(t)[\theta(label_\mu(r))]_p = lazy^p_\varphi(t)[label_\mu(erase(\sigma(\varphi(r))))]_p$. Since $t|_p$ contains no overlined symbol and $p \in \mathcal{A}ct(lazy^p_\varphi(t))$, then $label_\mu(erase(\sigma(\varphi(r)))) = lazy^p_\varphi(\sigma(\varphi(r)))$, and thus $lazy^p_\varphi(t)[\theta(label_\mu(r))]_p = lazy^p_\varphi(t)[lazy^p_\varphi(\sigma(\varphi(r)))]_p = lazy^q_\varphi(s)$.

5. If $t|_p = f_{L_1|0:L_2}(t_1,\dots,t_k)$, $erase(t|_p)$ is not a redex, $OD_\mathcal{R}(t|_p) = \varnothing$, $s = t[f_{L_1|L_2}(t_1,\dots,t_k)]_p$, and $q = p$, then $lazy^p_\varphi(t) = lazy^q_\varphi(s)$, $q \in \mathcal{A}ct(lazy^q_\varphi(s))$ and $lazy^p_\varphi(t) \xrightarrow{\mathsf{LR}}{}^=_\mu lazy^q_\varphi(s)$.

6. If $t|_p = f_{L_1|0:L_2}(t_1,\dots,t_k)$, $erase(t|_p)$ is not a redex, $OD_\mathcal{R}(t|_p) = \{p'\}$, $s = t[mark(t|_p,p')]_p$, and $q = p.p'$, then, for all $p'' : p \leq p'' \leq p'$,

$root(lazy^q_\varphi(s)|_{p''}) = f^e$ and $q \in \mathcal{A}ct(lazy^q_\varphi(s))$. Thus, it is not difficult to see that, since $root(erase(t)|_{p''}) = root(l|_{p''})$ for all $p'' : \Lambda \leq p'' < p'$, many activation steps of lazy rewriting are available, by successive applications of Lemma 7.24, in order to activate the necessary symbols from the root down to q, i.e. $lazy^p_\varphi(t) \xrightarrow{A}{}^*_\mu lazy^q_\varphi(s)$.

7. If $t|_p = \overline{f}_{L_1|L_2}(t_1, \ldots, t_k)$, $s = t[f_{L_1|L_2}(t_1, \ldots, t_k)]_p$, and $p = q.i$ for some i, then, since $lazy^q_\varphi(s) \leq_{lazy} lazy^p_\varphi(t)$, we have $lazy^p_\varphi(t) \xrightarrow{\mathsf{LR}}{}^=_\mu lazy^p_\varphi(t)$. Note that $q \in \mathcal{A}ct(lazy^q_\varphi(t))$ since $q.i \in \mathcal{A}ct(lazy^p_\varphi(t))$.

□

In general, our strategy is strictly more restrictive than *LR* as the following example shows.

Example 7.27 *Consider the program $\mathcal{R}$ (as a TRS) and the E-strategy map φ of Example 7.2. Consider the replacement map $\mu = \mu^\varphi$. In Example 7.36 below, we prove that $\mathcal{R}$ is φ–terminating. However,* LR *enters an infinite reduction sequence steaming from the expression $label_\mu$*(`length'(from(0))`)*:*

$$
\begin{array}{l}
\mathtt{length'}^e(\mathtt{from}^\ell(\mathtt{0}^e)) \\
\quad \xrightarrow{R}_\mu \mathtt{length}^e(\mathtt{from}^\ell(\mathtt{0}^e)) \\
\quad \xrightarrow{A}_\mu \mathtt{length}^e(\mathtt{from}^e(\mathtt{0}^e)) \\
\quad \xrightarrow{R}_\mu \mathtt{length}^e(\mathtt{cons}^e(\mathtt{0}^e, \mathtt{from}^\ell(\mathtt{s}^e(\mathtt{0}^e)))) \\
\quad \xrightarrow{R}_\mu \mathtt{s}^e(\mathtt{length'}^e(\mathtt{from}^\ell(\mathtt{s}^e(\mathtt{0}^e)))) \\
\quad \xrightarrow{\mathsf{LR}}_\mu \cdots
\end{array}
$$

Note that if no positive annotation is provided for an argument of a symbol, LR *freely demands reduction on this argument. Then, in contrast to φ (where $\varphi(\mathtt{length}) = (0)$),* LR *can evaluate position* 1 *of the considered expression* `length(from(0))`.

7.4.2 On-demand Rewriting

In the following, we show that each evaluation step of our refined on-demand strategy is contained into (at most) one evaluation step of on-demand rewriting (see Section 6.3). We introduce some auxiliary definitions and results in order to prove Theorem 7.30 below. Given a term $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and a position $p \in \mathcal{P}os_A(t)$, we say the tuple $\langle t, p\rangle$ is *consistent* w.r.t. TRS $\mathcal{R}$ and strategy map φ (or simply *consistent*) if $\exists s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ s.t. $\langle \varphi(s), \Lambda\rangle \xrightarrow{\sharp}{}^*_\varphi \langle t, p\rangle$.

Given a term $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and a position $p \in \mathcal{P}os_A(t)$, we say that p is a *stop* position if there is no sequence $\langle t, p\rangle \xrightarrow{\sharp}{}^*_\varphi \langle t', q\rangle$ such that $p \leq q$, $erase(t) = erase(t')$, and $erase(t'|_q)$ is a redex.

Lemma 7.28 *Let $\mathcal{R}$ be a TRS and φ be an E-strategy map such that $\mu^{\varphi+}(c) = \varnothing$ for $c \in \mathcal{C}$. Let $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and $p \in \mathcal{P}os_A(t)$. If $root(erase(t|_p)) \in \mathcal{C}$, then p is a stop position.*

Proof. Immediate since $\mu^{\varphi+}(c) = \varnothing$ for $root(erase(t|_p)) = c$ and it has no effect, even if annotation 0 is included into $\varphi(c)$. □

Lemma 7.29 *Let $\mathcal{R}$ be a left-linear CS and φ be an E-strategy map such that $\mu^{\varphi+}(c) = \varnothing$ for $c \in \mathcal{C}$. Let $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and $p \in \mathcal{P}os_A(t)$. If $\langle t, p\rangle$ is consistent, then either $p \in \mathcal{P}os_P(t)$ or $\exists q \in \mathcal{P}os_P(t)$ s.t. $q < p$ and either $OD_{\mathcal{R}}(t|_q) \neq \varnothing$ or, otherwise, if $erase(t|_q)$ is a redex, then $\forall q < w \leq p$, w is a stop position.*

Proof. By induction on the length n of the evaluation sequence $\langle\varphi(s), \Lambda\rangle \xrightarrow{\sharp}{}^n_\varphi \langle t, p\rangle$ for $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

- $(n = 0)$ It is immediate to see that $p = \Lambda$ and $\Lambda \in \mathcal{P}os_P(t)$.
- $(n > 0)$ Let us consider $\langle\varphi(s), \Lambda\rangle \xrightarrow{\sharp}{}^{n-1}_\varphi \langle t', p'\rangle \xrightarrow{\sharp}_\varphi \langle t, p\rangle$. The induction hypothesis is: $p' \in \mathcal{P}os_P(t')$ or $p' \in \mathcal{P}os_A(t') - \mathcal{P}os_P(t')$ and $\exists q' \in \mathcal{P}os_P(t')$ s.t. $q' < p'$ and either $OD_{\mathcal{R}}(t'|_{q'}) \neq \varnothing$ or, otherwise, if $erase(t'|_{q'})$ is a redex, then for all q' s.t. $q' < w \leq p'$, w is a stop position. We consider the different cases of Definition 7.14 separately:

  1. Let $t'|_{p'} = f_{L|nil}(t_1, \ldots, t_k)$, $t = t'$ and $p' = p.i$ for some i. If $p' \in \mathcal{P}os_P(t')$, then $p \in \mathcal{P}os_P(t)$. If $p' \in \mathcal{P}os_A(t') - \mathcal{P}os_P(t')$ and $q' = p$, then $p \in \mathcal{P}os_P(t)$. If $p' \in \mathcal{P}os_A(t') - \mathcal{P}os_P(t')$, $q' < p$, and $OD_{\mathcal{R}}(t|_{q'}) \neq \varnothing$, then the conclusion follows since no symbol occurring above or at position p' has been changed. If $p' \in \mathcal{P}os_A(t') - \mathcal{P}os_P(t')$, $q' < p$, and $OD_{\mathcal{R}}(t|_{q'}) = \varnothing$, then the conclusion follows by induction since p is also a stop position.
  2. Let $t'|_{p'} = f_{L_1|i:L_2}(t_1, \ldots, t_k)$, $i > 0$, $t = t'[f_{L_1@i|L_2}(t_1, \ldots, t_k)]_{p'}$ and $p = p'.i$. If $p' \in \mathcal{P}os_P(t')$, then $p \in \mathcal{P}os_P(t)$. If $p' \in \mathcal{P}os_A(t') - \mathcal{P}os_P(t')$, $q' < p$, and $OD_{\mathcal{R}}(t'|_{q'}) \neq \varnothing$, then the conclusion follows since no symbol occurring above or at position p has been changed. If $p' \in \mathcal{P}os_A(t') - \mathcal{P}os_P(t')$, $q' < p$, $OD_{\mathcal{R}}(t'|_{q'}) = \varnothing$, and $erase(t'|_{q'})$ is a redex, then p is a stop position if p' is.
  3. Let $t'|_{p'} = f_{L_1|-i:L_2}(t_1, \ldots, t_k)$, $i > 0$, $t = t'[f_{L_1@-i|L_2}(t_1, \ldots, t_k)]_{p'}$ and $p = p'$. This case is straightforward since no symbol changes above (or at) position p'.
  4. Let $t'|_{p'} = f_{L_1|0:L_2}(t_1, \ldots, t_k) = \sigma(l')$, $erase(l') = l$, $t = t'[\sigma(\varphi(r))]_{p'}$ for some $l \to r \in R$ and substitution σ, and $p = p'$. If $p' \in \mathcal{P}os_P(t')$, then $p \in \mathcal{P}os_P(t)$. Otherwise, $p' \in \mathcal{P}os_A(t') - \mathcal{P}os_P(t')$ and $q' < p$. If $OD_{\mathcal{R}}(t|_{q'}) \neq$

$\varnothing$ or $OD_{\mathcal{R}}(t|_{q'}) = \varnothing$ and $erase(t|_{q'})$ is not a redex, then the conclusion follows. Otherwise, $erase(t|_{q'})$ is a redex.

Here, note that it is impossible that $OD_{\mathcal{R}}(t'|_{q'}) = \varnothing$ because in such case, either $erase(t'|_{q'})$ was not a redex and, since $\mathcal{R}$ is a left-linear CS, it is impossible that $erase(t|_{q'})$ becomes a redex; or $erase(t'|_{q'})$ was a redex and p' was a stop position, but, then, no reduction can be performed at position p'. Thus, $OD_{\mathcal{R}}(t'|_{q'}) \neq \varnothing$, $OD_{\mathcal{R}}(t|_{q'}) = \varnothing$, and $erase(t|_{q'})$ is a redex. Now, since $\mathcal{R}$ is a CS, for all q' s.t. $q' < w \leq p$, $root(erase(t|_w)) \in \mathcal{C}$ and, by Lemma 7.28, w is a stop position.

5. Let $t'|_{p'} = f_{L_1|0:L_2}(t_1, \ldots, t_k)$, $erase(t'|_{p'})$ is not a redex, $OD_{\mathcal{R}}(t'|_{p'}) = \varnothing$, $t = t'[f_{L_1|L_2}(t_1, \ldots, t_k)]_{p'}$, and $p = p'$. This case is straightforward since no symbol changes above (or at) position p'.

6. Let $t'|_{p'} = f_{L_1|0:L_2}(t_1, \ldots, t_k)$, $erase(t'|_{p'})$ is not a redex, $OD_{\mathcal{R}}(t'|_{p'}) = \{p''\}$, $t = t'[mark(t'|_{p'}, p'')]_{p'}$, and $p = p'.p''$. If $p' \in \mathcal{P}os_P(t')$, then $p' < p$, $OD_{\mathcal{R}}(t|_{p'}) \neq \varnothing$, and the conclusion follows. If $p' \in \mathcal{P}os_A(t') - \mathcal{P}os_P(t')$, $q' < p$, and $OD_{\mathcal{R}}(t'|_{q'}) \neq \varnothing$, then the conclusion follows since no symbol above p has been changed. If $p' \in \mathcal{P}os_A(t') - \mathcal{P}os_P(t')$, $q' < p$, $OD_{\mathcal{R}}(t'|_{q'}) = \varnothing$, and $erase(t'|_{q'})$ is a redex, then p is a stop position if p' is.

7. Let $t'|_{p'} = \overline{f}_{L_1|L_2}(t_1, \ldots, t_k)$, $t = t'[f_{L_1|L_2}(t_1, \ldots, t_k)]_{p'}$ and $p' = p.i$ for some i. This case is similar to case 1 above.

□

Theorem 7.30 *Let $\mathcal{R}$ be a left-linear CS and φ be an E-strategy map such that $\mu^{\varphi+}(c) = \varnothing$ for $c \in \mathcal{C}$. Let $\mu, \mu_D \in M_{\mathcal{F}}$ be such that $\mu = \mu^{\varphi+}$ and $\mu \sqcup \mu_D = \mu^{\varphi}$. Let $t \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$ and $p \in \mathcal{P}os_A(t)$. If $\langle t, p\rangle \overset{\sharp}{\to}_{\varphi} \langle s, q\rangle$ and $\langle t, p\rangle$ is consistent, then $erase(t) \overset{p}{\hookrightarrow}{}^{=}_{\langle \mu, \mu_D\rangle} erase(s)$.*

Proof. We consider only case 4 of Definition 7.14 since the other cases only manipulate annotations on symbols or compute the next position q to be considered, which implies a reflexive on-demand rewriting step. Then, by Lemma 7.29, case 4 can only occur under the following conditions:

1. If $p \in \mathcal{P}os_P(t)$, then it is trivial.

2. If $p \in \mathcal{P}os_A(t) - \mathcal{P}os_P(t)$, $\exists q \in \mathcal{P}os_P(t)$ s.t. $q < p$ and $OD_{\mathcal{R}}(t|_q) \neq \varnothing$, then it is easy to prove that there exist $p_1, \ldots, p_n \in \mathcal{L}azy_{\langle \mu, \mu_D\rangle}(erase(t))$, $r_1, \ldots, r_n, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \to r \in R$, and substitution σ such that $t' = erase(t)[r_1]_{p_1} \cdots [r_n]_{p_n}$, $t'|_q = \sigma(l)$ and, for all $w \in \mathcal{P}os(l)$ s.t. $sprefix_{erase(t)|_q}(w) = sprefix_l(w)$ whenever $q.w \leq p$, hence we have that $l|_w \notin \mathcal{X}$.

3. Otherwise, $p \in \mathcal{P}os_A(t) - \mathcal{P}os_P(t)$, and there is no $q \in \mathcal{P}os_P(t)$ s.t. $q < p$, $OD_{\mathcal{R}}(t|_q) = \varnothing$, and $erase(t|_q)$ is a redex. Note that, by Definition 7.14, there should exist $q \in \mathcal{P}os_P(t)$ and $l \in L(\mathcal{R})$ s.t. $q < p$ and $\mathcal{P}os_{\neq}(erase(t|_q), l) \neq \varnothing$ because, otherwise, it is impossible that position $p \notin \mathcal{P}os_P(t)$ is used for reduction. Thus, it is easy to prove that there exist $p_1, \ldots, p_n \in \mathcal{L}azy_{\langle\mu,\mu_D\rangle}(erase(t))$, $r_1, \ldots, r_n, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \to r \in R$, and substitution σ such that $t' = erase(t)[r_1]_{p_1} \cdots [r_n]_{p_n}$, $t'|_q = \sigma(l)$ and for all $w \in \mathcal{P}os(l)$ s.t. $sprefix_{erase(t)|_q}(w) = sprefix_l(w)$, whenever $q.w \leq p$, we have that $l|_w \notin \mathcal{X}$.

□

Similarly to what happened w.r.t. *LR*, our strategy is strictly more restrictive than *ODR* as the following example shows.

Example 7.31 *Consider the following* OBJ *program and its strategy map φ:*

```
obj EX is
  sorts Nat LNat .
  op 0    : -> Nat .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat  [strat (-2)] .
  op from : Nat -> LNat       [strat (-1 0)] .
  op f    : LNat -> Nat       [strat (-1 0)] .
  op g    : Nat -> Nat        [strat (-1 0)] .
  vars X : Nat .
  eq from(X) = cons(X,from(s(X))) .
  eq f(X) = 0 .
  eq g(0) = 0 .
endo
```

Consider also the TRS associated to this program and the replacement maps $\mu(\mathtt{g}) = \mu(\mathtt{f}) = \mu(\mathtt{from}) = \mu(\mathtt{cons}) = \varnothing$, $\mu_D(\mathtt{g}) = \mu_D(\mathtt{f}) = \mu_D(\mathtt{from}) = \{1\}$, and $\mu_D(\mathtt{cons}) = \{2\}$, whose union corresponds to strategy map φ. The expression $t = \mathtt{g(f(from(0)))}$ yields a finite evaluation sequence using $\xrightarrow{\sharp}$:

$$
\begin{array}{l}
\langle \mathtt{g}_{nil|(-1\ 0)}(\mathtt{f}_{nil|(-1\ 0)}(\mathtt{from}_{nil|(-1\ 0)}(\mathtt{0}_{nil|nil}))), \Lambda\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{g}_{(-1)|(0)}(\mathtt{f}_{nil|(-1\ 0)}(\mathtt{from}_{nil|(-1\ 0)}(\mathtt{0}_{nil|nil}))), \Lambda\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{g}_{(-1)|(0)}(\mathtt{f}_{nil|(-1\ 0)}(\mathtt{from}_{nil|(-1\ 0)}(\mathtt{0}_{nil|nil}))), 1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{g}_{(-1)|(0)}(\mathtt{f}_{(-1)|(0)}(\mathtt{from}_{nil|(-1\ 0)}(\mathtt{0}_{nil|nil}))), 1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{g}_{(-1)|(0)}(\mathtt{0}_{nil|nil}), 1\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{g}_{(-1)|(0)}(\mathtt{0}_{nil|nil}), \Lambda\rangle \\
\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{0}_{nil|nil}, \Lambda\rangle
\end{array}
$$

However, even though ODR *is able to reproduce the previous terminating reduction sequence:*

$$\mathtt{g(}\underline{\mathtt{f(from(0))}}\mathtt{)} \hookrightarrow_{\langle\mu,\mu_D\rangle} \underline{\mathtt{g(0)}} \hookrightarrow_{\langle\mu,\mu_D\rangle} \mathtt{0}$$

the following non-terminating reduction sequence is also proven:

$$\mathtt{g(f(}\underline{\mathtt{from(0)}}\mathtt{))} \hookrightarrow_{\langle\mu,\mu_D\rangle} \mathtt{g(f(cons(0,}\underline{\mathtt{from(s(0))}}\mathtt{)))} \hookrightarrow_{\langle\mu,\mu_D\rangle} \cdots$$

Note that Λ *is a positive position which is a eventual redex of* `g(0)` *and positions* $1.1, 1.1.2, \ldots$ *are positions under a non-variable subterm w.r.t. the lhs* `g(0)`*.*

Moreover, the condition in Theorem 7.30 that φ be an E-strategy map such that $\mu^{\varphi_+}(c) = \varnothing$ for $c \in \mathcal{C}$ cannot be dropped, as witnessed by the following example.

Example 7.32 *Consider the following* OBJ *program and its strategy map* φ*:*

```
obj EX is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat  [strat (2)] .
  op from : Nat -> LNat       [strat (-1 0)] .
  op f    : LNat -> Nat       [strat (-1 0)] .
  var X : Nat . var Z : LNat .
  eq from(X) = cons(X,from(s(X))) .
  eq f(cons(X,Z)) = 0 .
endo
```

Consider also the TRS associated to this program and the replacement maps $\mu(\mathtt{f}) = \mu(\mathtt{from}) = \varnothing$, $\mu(\mathtt{cons}) = \{2\}$, *and* $\mu_D(\mathtt{f}) = \mu_D(\mathtt{from}) = \{1\}$, *whose union corresponds to the strategy map* φ*. Consider the term* $t = \mathtt{f(from(0))}$*. This term admits a unique normalizing evaluation sequence under* ODR*:*

$$\mathtt{f(}\underline{\mathtt{from(0)}}\mathtt{)} \hookrightarrow_{\langle\mu,\mu_D\rangle} \underline{\mathtt{f(cons(0,from(s(0))))}} \hookrightarrow_{\langle\mu,\mu_D\rangle} \mathtt{0}$$

Position 1.2 *of* `f(cons(0,from(s(0))))` *is under a variable of lhs* `f(cons(X,Z))` *and the subterm* `from(s(0))` *is not further evaluated. However, the evaluation sequence for* $\stackrel{\sharp}{\to}$ *is non-terminating:*

$$\begin{array}{l}
\langle \mathtt{f}_{nil|(-1\ 0)}(\mathtt{from}_{nil|(-1\ 0)}(\mathtt{0}_{nil|nil})), \Lambda\rangle \\
\quad \stackrel{\sharp}{\to}_{\varphi} \langle \mathtt{f}_{(-1)|(0)}(\mathtt{from}_{nil|(-1\ 0)}(\mathtt{0}_{nil|nil})), \Lambda\rangle \\
\quad \stackrel{\sharp}{\to}_{\varphi} \langle \mathtt{f}_{(-1)|(0)}(\mathtt{from}_{nil|(-1\ 0)}(\mathtt{0}_{nil|nil})), 1\rangle \\
\quad \stackrel{\sharp}{\to}_{\varphi} \langle \mathtt{f}_{(-1)|(0)}(\mathtt{from}_{(-1)|(0)}(\mathtt{0}_{nil|nil})), 1\rangle \\
\quad \stackrel{\sharp}{\to}_{\varphi} \langle \mathtt{f}_{(-1)|(0)}(\mathtt{cons}_{nil|(2)}(\mathtt{0}_{nil|nil}, \mathtt{from}_{(-1)|(0)}(\mathtt{s}_{nil|nil}(\mathtt{0}_{nil|nil})))), 1\rangle \\
\quad \stackrel{\sharp}{\to}_{\varphi} \langle \mathtt{f}_{(-1)|(0)}(\mathtt{cons}_{(2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{(-1)|(0)}(\mathtt{s}_{nil|nil}(\mathtt{0}_{nil|nil})))), 2\rangle \\
\quad \stackrel{\sharp}{\to}_{\varphi} \cdots
\end{array}$$

Note that since $\varphi(\texttt{cons}) = \texttt{(2)}$*, each term* $\texttt{from}(w)$ *has associated a non-terminating reduction sequence.*

Condition of a program being a *CS* in Theorem 7.30 cannot be dropped; a similar argument to that of Example 7.32 can be used if we consider a defined symbol in a non-root position of a lhs whose strategy map associates a positive argument.

In the following section, we consider other aspects of the definition of a suitable on-demand evaluation strategy and formulate effective methods for proving termination of our on-demand strategy.

7.5 Proving Termination of Programs with On-demand Annotations by Transformation

In [Lucas, 2002b] a method for proving termination of *LR* as termination of *context-sensitive rewriting* (*CSR* [Lucas, 1998a]) is described. In contrast to *LR*, context-sensitive rewriting forbids *any* reduction on the arguments not included into $\mu(f)$ for a given function call $f(t_1, \ldots, t_k)$. A TRS $\mathcal{R}$ is μ-terminating if the context-sensitive rewrite relation associated to $\mathcal{R}$ and μ is terminating. The idea of the aforementioned method is simple: given a TRS $\mathcal{R}$ and a replacement map μ, a new TRS $\mathcal{R}'$ and a replacement map μ' can be obtained in such a way that μ'-termination of $\mathcal{R}'$ implies $LR(\mu)$-termination of $\mathcal{R}$. Fortunately, there are a number of different techniques for proving termination of *CSR* (see [Giesl and Middeldorp, 2003; Lucas, 2002c] for recent surveys) which provide a formal framework for proving termination of lazy rewriting. A simple modification of such transformation provides a sound technique for proving φ-termination of TRSs for arbitrary strategy annotations by taking into account that only those symbols which have associated a negative index may be activated on demand. Similarly to [Lucas, 2001a, 2002b], by φ-termination of a TRS $\mathcal{R}$ we mean the absence of infinite $\stackrel{\sharp}{\to}_\varphi$-sequences of terms starting from $\langle \varphi(t), \Lambda \rangle$.

As for the transformation in [Lucas, 2002b], the idea is to encode into new symbols and rules (together with the appropriate modification/extension of φ) the demandedness information expressed by the rules of the TRS $\mathcal{R}$ together with the (negative) annotations of the *E*-strategy map φ; in such a way that φ-termination is preserved in the new TRS and *E*-strategy map, and the negative indices are finally suppressed (by removing from the lhs of the rules the parts that introduce on-demand computations). We iterate on these basic transformation steps until obtaining a canonical *E*-strategy map. At this shape, we can simply stop the transformation and use the existing methods for proving termination of *CSR*. Let φ be an arbitrary *E*-strategy map. Given $l \to r \in R$ and $p \in \mathcal{P}os(l)$, we let

$$\mathcal{I}(l, p) = \{ i > 0 \mid p.i \in \mathcal{P}os_{\mathcal{F}}(l) \text{ and } -i \in \varphi(root(l|_p)) \}$$

Assume that $\mathcal{I}(l,p) = \{i_1,\ldots,i_n\}$ for some $n > 0$ (i.e., $\mathcal{I}(l,p) \neq \varnothing$) and let $f = root(l|_p)$. Then, $\mathcal{R}^\diamond = (\mathcal{F}^\diamond, R^\diamond)$ and $\varphi^\diamond$ are as follows: $\mathcal{F}^\diamond = \mathcal{F} \cup \{f_j \mid 1 \leq j \leq n\}$, where each f_j is a new symbol of arity $ar(f_j) = ar(f)$, and

$$R^\diamond = R - \{l \to r\} \cup \{l'_j \to r, l[x]_{p.i_j} \to l'_j[x]_{p.i_j} \mid 1 \leq j \leq n\}$$

where $l'_j = l[f_j(l|_{p.1},\ldots,l|_{p.k})]_p$ if $ar(f) = k$, and x is a new variable. We let $\varphi^\diamond(f_j) = (i_j\ 0)$ for $1 \leq j \leq n$ and $f \in \mathcal{D}$, $\varphi^\diamond(f_j) = (i_j)$ for $1 \leq j \leq n$ and $f \in \mathcal{C}$, and $\varphi^\diamond(g) = \varphi(g)$ for all $g \in \mathcal{F}.(g \neq f)$. Moreover, we let $\varphi^\diamond(f) = \varphi_+(f)$ if $\mu^{can}_{\mathcal{R}^\diamond}(f) \subseteq \mu^{\varphi_+}(f)$, or $\varphi^\diamond(f) = \varphi(f)$ otherwise. Informally, if p is a position in a lhs l defining a symbol f with a negative annotation $-i$, and position $p.i$ is non-variable in l, then we transform rule $l \to r$ into $l[x]_p \to l'[x]_p$ and $l' \to r$, where l' is l with a new symbol f' at position p such that annotation $-i$ is converted to positive in the strategy for f' and removed from the strategy for f.

The transformation proceeds in this way (starting from $\mathcal{R}^\diamond$ and $\mu^\diamond$) until obtaining $\mathcal{R}^\natural = (\mathcal{F}^\natural, R^\natural)$ and $\varphi^\natural$ such that $\varphi^\natural = {\varphi^\natural}^{\mathbb{N}}$. If $\varphi = \varphi_+$, then $\mathcal{R}^\natural = \mathcal{R}$ and $\varphi^\natural = \varphi$. Finally, we can state a sufficient condition for φ-termination as termination of *CSR* for the transformed TRS. We use the following result for $\varphi^\natural$-termination by *CSR* termination.

Theorem 7.33 [Lucas, 2001b] *Let $\mathcal{R}$ be a TRS, φ be a positive E-strategy map. If $\mathcal{R}$ is μ^φ-terminating, then $\mathcal{R}$ is φ-terminating.*

Theorem 7.34 *Let $\mathcal{R}$ be a TRS, φ be an E-strategy map. If $\mathcal{R}^\natural$ is $\mu^{\varphi^\natural}$-terminating, then $\mathcal{R}$ is φ-terminating.*

Proof. By induction on the number n of transformation steps $\langle \mathcal{R}_0, \varphi_0\rangle, \ldots, \langle \mathcal{R}_n, \varphi_n\rangle$, where $\mathcal{R}_0 = \mathcal{R}$ and $\varphi_0 = \varphi$. If $n = 0$, then $\varphi^\natural = {\varphi^\natural}^{\mathbb{N}}$ and, since $\mathcal{R}$ is $\mu^{\varphi^\natural}$-terminating, by Theorem 7.33, $\mathcal{R}$ is φ-terminating. If $n > 0$, then, it is not difficult to see that, for all $t, s \in \mathcal{T}(\mathcal{F}^\sharp_{\varphi_0}, \mathcal{X}^\sharp_{\varphi_0})$ and $p \in \mathcal{P}os(t)$, if $\langle t, p\rangle \xrightarrow{\sharp}_{\varphi_0} \langle s, q\rangle$, we have that $\langle {<}t{>}_{\varphi_1}, p\rangle \xrightarrow{\sharp}{}^{*}_{\varphi_1} \langle {<}s{>}_{\varphi_1}, q\rangle$. Hence, since $\mathcal{R}_1$ is φ_{n-1}-terminating, $\mathcal{R}_0$ is φ_n-terminating. □

In the following, we are able to demonstrate the termination of some examples appearing in this thesis.

Example 7.35 *Consider the TRS $\mathcal{R}$ associated to Example 7.1 where φ(*`cons`*) = *`(1 -2)`*. Then, $\mathcal{R}^\natural$ is:*

```
2nd(cons'(x,cons(y,z))) → y
2nd(cons(x,y))          → 2nd(cons'(x,y))
from(x)                 → cons(x,from(s(x)))
```

and $\varphi^\natural$ *is given by* $\varphi^\natural$(`2nd`) $=$ $\varphi^\natural$(`from`) $=$ `(1 0)`, $\varphi^\natural$(`cons`) $=$ `(1)`, *and* $\varphi^\natural$(`cons'`) $=$ `(2)`. *The* $\mu^{\varphi^\natural}$*-termination of* $\mathcal{R}^\natural$ *is proved by using Zantema's transformation for proving termination of* CSR *[Zantema, 1997]: the derived TRS*

```
2nd(cons'(x,cons(y,z))) → y
2nd(cons(x,y))          → 2nd(cons'(x,activate(y)))
from(x)                 → cons(x,from'(s(x)))
activate(from'(x))      → from(x)
from(x)                 → from'(x)
activate(x)             → x
```

(where `activate` *and* `from'` *are new symbols introduced by Zantema's transformation [Zantema, 1997]) is terminating*[5].

Example 7.36 *Consider the TRS* $\mathcal{R}$ *and the E-strategy map* φ *that correspond to the* OBJ *program of Example 7.2. Our transformation returns the original TRS, i.e.,* $\mathcal{R}^\natural$ *is:*

```
from(x)            → cons(x,from(s(x)))
length(nil)        → 0
length(cons(x,z))  → s(length'(z))
length'(z)         → length(z)
```

together with the simplified E-strategy $\varphi^\natural$(`s`) $=$ $\varphi^\natural$(`cons`) $= (1)$, $\varphi^\natural$(`from`) $= (1\ 0)$ *and* $\varphi^\natural$(`length`) $=$ $\varphi^\natural$(`length'`) $= (0)$. *The* $\mu^{\varphi^\natural}$*-termination of* $\mathcal{R}$ *can be automatically proved by splitting up the rules of the program into two modules* $\mathcal{R}_1$ *(consisting of the rule for* `from`*) and* $\mathcal{R}_2$ *(consisting of the rules for* `length` *and* `length'`*). The* $\mu^{\varphi^\natural}$*-termination of* $\mathcal{R}_1$ *can easily be proved by using Zantema's transformation (in fact, the proof can be extracted from that of Example 7.35). The* $\mu^{\varphi^\natural}$*-termination of* $\mathcal{R}_2$ *is easily proved: in fact,* $\mathcal{R}_2$ *can be proved terminating (regarding standard rewriting) by using a polynomial ordering*[6]*. Now,* $\mu^{\varphi^\natural}$*-termination of* $\mathcal{R}$ *follows by applying the modularity results of [Gramlich and Lucas, 2002].*

7.6 Experiments

In order to demonstrate the practicality of the on-demand evaluation strategy proposed in this chapter, a prototype interpreter has been implemented in Haskell (using ghc 5.04.2). The system is called OnDemandOBJ and is publicly available at

`http://www.dsic.upv.es/users/elp/soft.html`

[5] This is proven, e.g. using the C*i*ME 2.0 system (available at `http://cime.lri.fr`).
[6] C*i*ME 2.0 can also be used for issuing this proof.

ms./rewrites	`pi`
OnDemandOBJ	25/364
CafeOBJ	30/364
OBJ3	*unavailable*
Maude	*unavailable*

Table 7.1: Execution of call `pi(square(square(3)))`

ms./rewrites	`msquare_eager`	`msquare_apt`	`msquare_neg`
OnDemandOBJ	33/ 715	62/ 1640	0/ 1
	40/ 914	78/ 1992	80/ 1992
CafeOBJ	40/ 715	50/ 715	0/ 1
	50/ 914	60/ 914	60/ 914
OBJ3	20/ 715	*overflow*	*unavailable*
	30/ 914	*overflow*	*unavailable*
Maude	0/ 715	0/ 1640	*unavailable*
	0/ 914	3/ 1992	*unavailable*

Table 7.2: Execution of terms `minus(0,square(square(5)))` and `minus(square(square(5)),square(square(3)))`

Tables 7.1, 7.2, and 7.3 show the runtimes[7] and the number of rewrite steps of the benchmarks for the different OBJ-family systems (source programs are included in Appendix B). CafeOBJ[8] (we use version 1.4.6) is developed in Lisp at the Japan Advanced Inst. of Science and Technology (JAIST); OBJ3[9] (we use version 2.0), also written in Lisp, is maintained by the University of California at San Diego; Maude[10] (we use version 1.0.5) is developed in C++ and maintained by the University of Illinois at Urbana-Champaign. OBJ3 and Maude provide only computations with positive annotations whereas CafeOBJ provides computations with negative annotations as well, using the on-demand evaluation of [Nakamura and Ogata, 2001; Ogata and Futatsugi, 2000]. OnDemandOBJ computes with negative annotations using the on-demand evaluation strategy provided in this chapter. Note that CafeOBJ and OBJ3 implement sharing of variables whereas Maude and OnDemandOBJ do not.

The benchmark `pi` codifies the well-known infinite series expansion to approximate number π:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

[7] The average of 10 executions measured in a Pentium III machine running RedHat 7.2.
[8] Available at `http://www.ldl.jaist.ac.jp/Research/CafeOBJ/system.html`.
[9] Available at `http://www.kindsoftware.com/products/opensource/obj3/OBJ3/`.
[10] Available at `http://maude.cs.uiuc.edu/`.

ms./rewrites	quicksort	minsort	mod	mod'	average
OnDemandOBJ	55/1373	87/1649	540/13661	135/3117	70/1399
CafeOBJ	42/ 658	*overflow*	180/ 3117	175/3117	130/1399

Table 7.3: Comparison of CafeOBJ and OnDemandOBJ

and uses negative annotations to obtain a terminating and complete example, which can not be obtained using only positive annotations (see its program code in Appendix B). Termination of the program can be formally proved using the technique of Section 7.5 (see Appendix C below). Table 7.1 compares the evaluation of expression `pi(square(square(3)))` using existing OBJ implementations. It witnesses that negative annotations are actually useful in practice and that the implementation of the on-demand evaluation strategy in other systems is really promising.

On the other hand, Table 7.2 illustrates the interest of using negative annotations to improve the behavior of programs: The benchmark `msquare_eager` codifies the functions `square`, `minus`, `times`, and `plus` over natural numbers using only positive annotations. Every k-ary symbol f is given a strategy $(1\ 2\ \cdots k\ 0)$ (this corresponds to *default* strategies in Maude). Note that the program is terminating as a TRS (i.e., without any annotation). The benchmark `msquare_apt` is similar to `msquare_eager`, but *canonical* positive strategies are provided: the i-th argument of a symbol f is annotated if there is an occurrence of f in the left-hand side of a rule having a non-variable i-th argument; otherwise, the argument is not annotated (see [Antoy and Lucas, 2002]). The benchmark `msquare_neg` is similar to `msquare_apt`, though *canonical arbitrary* strategies are provided: now (from left-to-right), the i-th argument of a defined symbol f is annotated if all occurrences of f in the left-hand side of the rules contain a non-variable i-th argument; if all occurrences of f in the left-hand side of the rules have a variable i-th argument, then the argument is not annotated; in any other case, annotation $-i$ is given to f (see [Antoy and Lucas, 2002]). Then, for instance, program `msquare_neg` runs in less time and requires a smaller number of rewrite steps than `msquare_eager` or `msquare_apt`, which do not include negative annotations.

Finally, Table 7.3 compares the execution of typical functional programs with canonical arbitrary strategies in OnDemandOBJ and in CafeOBJ, and demonstrates that there are clear advantages in using our implementation of the on-demand evaluation. We have used benchmarks `quicksort`, `minsort`, `mod`, and `average` which are borrowed from [Arts and Giesl, 2001], and use canonical arbitrary strategies. Benchmark `mod'` is similar to `mod` but extra annotations are provided in order to avoid differences due to sharing.

7.7 Conclusions

We have provided a suitable extension of the positive E-evaluation strategy of OBJ-like languages to general (positive as well as negative) annotations. Such an extension is conservative, i.e., programs which only use positive strategy annotations and that are executed under our strategy behave exactly as if they were executed under the standard OBJ evaluation strategy (Theorems 7.18, 7.20, and 7.30). The main contributions of this chapter are:

(a) the definition of a revised and well-defined approach to demandedness via E-strategies (see Examples 7.2, 7.3, 7.17, and 7.27 for motivation regarding some of the problems detected on the previous proposals),

(b) the better computational properties associated to such a new on-demand strategy (Theorems 7.21 and 7.26),

(c) the definition of techniques for analyzing computational properties such as termination (Theorem 7.34), and that

(d) our approach is better suited for implementation (experimental results of Section 7.6).

We also state that this novel on-demand strategy improves the two most important evaluation strategies dealing with on-demand annotations: *lazy rewriting* (LR) [Fokkink *et al.*, 2000] and *on-demand rewriting* (ODR) [Lucas, 2001a], which does not directly apply to OBJ nor is comparable to LR.

Appendix B Benchmarks Code

B-1 Program pi

This program codifies the well-known infinite series expansion to approximate the π number: $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$. To make the program terminating and complete, the strategy for symbol `cons` must include annotation `-2`. The remainder of strategy annotations are positive since termination of the whole program is ensured (see termination proof in Appendix C). Note that the auxiliary functions `plus`, `times` and `square` for natural numbers are also included.

```
obj PI is
  sorts Nat LNat Recip LRecip .
  op 0 : -> Nat .
  op s : Nat -> Nat                          [strat (1)] .
  op posrecip : Nat -> Recip                 [strat (1)] .
  op negrecip : Nat -> Recip                 [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat                 [strat (1 -2)] .
  op rnil  : -> LRecip .
  op rcons : Recip LRecip -> LRecip          [strat (1 2)] .
  op from : Nat -> LNat                      [strat (1 0)] .
  op 2ndspos : Nat LNat -> LRecip            [strat (1 2 0)] .
  op 2ndsneg : Nat LNat -> LRecip            [strat (1 2 0)] .
  op pi : Nat -> LRecip                      [strat (1 0)] .
  op plus : Nat Nat -> Nat                   [strat (1 2 0)] .
  op times : Nat Nat -> Nat                  [strat (1 2 0)] .
  op square : Nat -> Nat                     [strat (1 0)] .
  vars N X Y : Nat . var Z : LNat .
  eq from(X) = cons(X,from(s(X))) .
  eq 2ndspos(0,Z) = rnil .
  eq 2ndspos(s(N),cons(X,cons(Y,Z))) = rcons(posrecip(Y),2ndsneg(N,Z)) .
  eq 2ndsneg(0,Z) = rnil .
  eq 2ndsneg(s(N),cons(X,cons(Y,Z))) = rcons(negrecip(Y),2ndspos(N,Z)) .
  eq pi(X) = 2ndspos(X,from(0)) .
  eq plus(0,Y) = Y .
  eq plus(s(X),Y) = s(plus(X,Y)) .
  eq times(0,Y) = 0 .
  eq times(s(X),Y) = plus(Y,times(X,Y)) .
  eq square(X) = times(X,X) .
endo
```

B-2 Program msquare_eager

This program uses functions `minus`, `square`, `times`, and `plus` over natural numbers; they are common to several examples included in the Appendix. The key point of this program is that it is terminating using only positive annotations and including the indices of all symbols.

```
obj MINUS-SQUARE is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat         [strat (1)] .
  op plus : Nat Nat -> Nat  [strat (1 2 0)] .
  op times : Nat Nat -> Nat [strat (1 2 0)] .
  op square : Nat -> Nat    [strat (1 0)] .
  op minus : Nat Nat -> Nat [strat (1 2 0)] .
  vars M N : Nat .
  eq plus(0,N) = N .
  eq plus(s(M),N) = s(plus(M,N)) .
  eq times(0,N) = 0 .
  eq times(s(M),N) = plus(N,times(M,N)) .
  eq square(N) = times(N,N) .
  eq minus(0,N) = 0 .
  eq minus(s(M),0) = s(M) .
  eq minus(s(M),s(N)) = minus(M,N) .
endo
```

B-3 Program msquare_apt

This program is identical to `msquare_eager` but only the annotations which are necessary to make the program complete are included, i.e. we use only canonical positive strategies.

```
obj MINUS-SQUARE is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat         [strat (1)] .
  op plus : Nat Nat -> Nat  [strat (1 0)] .
  op times : Nat Nat -> Nat [strat (1 0)] .
  op square : Nat -> Nat    [strat (0)] .
  op minus : Nat Nat -> Nat [strat (1 2 0)] .
  vars M N : Nat .
  eq plus(0,N) = N .
  eq plus(s(M),N) = s(plus(M,N)) .
  eq times(0,N) = 0 .
```

```
  eq times(s(M),N) = plus(N,times(M,N)) .
  eq square(N) = times(N,N) .
  eq minus(0,N) = 0 .
  eq minus(s(M),0) = s(M) .
  eq minus(s(M),s(N)) = minus(M,N) .
endo
```

B-4 Program msquare_neg

This program is identical to `msquare_apt` except that negative annotations are included, i.e. we consider canonical *arbitrary* strategies.

```
obj MINUS-SQUARE is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat         [strat (1)] .
  op plus : Nat Nat -> Nat  [strat (1 0)] .
  op times : Nat Nat -> Nat [strat (1 0)] .
  op square : Nat -> Nat    [strat (0)] .
  op minus : Nat Nat -> Nat [strat (1 -2 0)] .
  vars M N : Nat .
  eq plus(0,N) = N .
  eq plus(s(M),N) = s(plus(M,N)) .
  eq times(0,N) = 0 .
  eq times(s(M),N) = plus(N,times(M,N)) .
  eq square(N) = times(N,N) .
  eq minus(0,N) = 0 .
  eq minus(s(M),0) = s(M) .
  eq minus(s(M),s(N)) = minus(M,N) .
endo
```

B-5 Program quicksort

This program is borrowed from Example 3.11 of [Arts and Giesl, 2001]. Note that auxiliary functions `from` and `take` for constructing lists are included, as well as two predicates `nfLNat` and `nfNat` to normalize terms, and the connective `and`. The term used for evaluation is: `nfLNat(quicksort(take(10,from(0))))`

```
obj Quicksort is
   sorts Nat LNat Bool2 .
   op 0 :  -> Nat .
   op s : Nat -> Nat                    [strat (1)] .
   op nil :  -> LNat .
   op cons : Nat LNat -> LNat           [strat (1)] .
```

```
   op true2 :  -> Bool2 .
   op false2 :  -> Bool2 .
   op le : Nat Nat -> Bool2            [strat (1 -2 0)] .
   op app : LNat LNat -> LNat          [strat (1 0)] .
   op low : Nat LNat -> LNat           [strat (2 0)] .
   op high : Nat LNat -> LNat          [strat (2 0)] .
   op ifLNat : Bool2 LNat LNat -> LNat [strat (1 0)] .
   op quicksort : LNat -> LNat         [strat (1 0)] .
   op and : Bool2 Bool2 -> Bool2       [strat (1 0)] .
   op nfLNat : LNat -> Bool2           [strat (1 0)] .
   op nfNat : Nat -> Bool2             [strat (1 0)] .
   op from : Nat -> LNat               [strat (0)] .
   op take : Nat LNat -> LNat          [strat (1 -2 0)] .
   vars X Y : Nat . vars Z W : LNat . vars A B : Bool2 .
   eq le(0,Y) = true2 .
   eq le(s(X),0) = false2 .
   eq le(s(X),s(Y)) = le(X,Y) .
   eq app(nil,Z) = Z .
   eq app(cons(X,Z),W) = cons(X,app(Z,W)) .
   eq low(X,nil) = nil .
   eq low(X,cons(Y,Z)) = ifLNat(le(Y,X),cons(Y,low(X,Z)),low(X,Z)) .
   eq high(X,nil) = nil .
   eq high(X,cons(Y,Z)) = ifLNat(le(Y,X),high(X,Z),cons(Y,high(X,Z))) .
   eq ifLNat(true2,Z,W) = Z .
   eq ifLNat(false2,Z,W) = W .
   eq quicksort(nil) = nil .
   eq quicksort(cons(X,Z)) = app(quicksort(low(X,Z)),
                                 cons(X,quicksort(high(X,Z)))) .
   eq from(X) = cons(X,from(s(X))) .
   eq take(0,Z) = nil .
   eq take(s(X),cons(Y,Z)) = cons(Y,take(X,Z)) .
   eq nfLNat(nil) = true2 .
   eq nfLNat(cons(X,Z)) = and(nfNat(X),nfLNat(Z)) .
   eq nfNat(0) = true2 .
   eq nfNat(s(X)) = nfNat(X) .
   eq and(true2,A) = A .
   eq and(false2,A) = false2 .
endo
```

B-6 Program `minsort`

This program is borrowed from Example 3.10 of [Arts and Giesl, 2001]. The call considered for evaluation is: `nfLNat(minsort(take(10,from(0)),nil))`

```
obj Minsort is
  sorts Nat LNat Bool2 .
  op 0 :  -> Nat .
  op s : Nat -> Nat                   [strat (1)] .
  op nil :  -> LNat .
  op cons : Nat LNat -> LNat          [strat (1 -2)] .
  op true2 :  -> Bool2 .
  op false2 :  -> Bool2 .
  op le : Nat Nat -> Bool2            [strat (1 2 0)] .
  op app : LNat LNat -> LNat          [strat (1 0)] .
  op rm : Nat LNat -> LNat            [strat (2 0)] .
  op min : LNat -> Nat                [strat (1 0)] .
  op eqNat : Nat Nat -> Bool2         [strat (1 2 0)] .
  op ifNat : Bool2 Nat Nat -> Nat     [strat (1 0)] .
  op ifLNat : Bool2 LNat LNat -> LNat [strat (1 0)] .
  op and : Bool2 Bool2 -> Bool2       [strat (1 0)] .
  op minsort : LNat LNat -> LNat      [strat (1 2 0)] .
  op nfLNat : LNat -> Bool2           [strat (1 0)] .
  op nfNat : Nat -> Bool2             [strat (1 0)] .
  op take : Nat LNat -> LNat          [strat (1 2 0)] .
  op from : Nat -> LNat               [strat (0)] .
  vars X Y : Nat . vars Z W : LNat . vars A B : Bool2 .
  eq le(0,Y) = true2 .
  eq le(s(X),0) = false2 .
  eq le(s(X),s(Y)) = le(X,Y) .
  eq app(nil,Z) = Z .
  eq app(cons(X,Z),W) = cons(X,app(Z,W)) .
  eq rm(X,nil) = nil .
  eq rm(X,cons(Y,Z)) = ifLNat(eqNat(X,Y),rm(X,Z),cons(Y,rm(X,Z))) .
  eq min(cons(X,nil)) = X .
  eq min(cons(X,cons(Y,Z))) = ifNat(le(Y,X),min(cons(Y,Z)),min(cons(X,Z))) .
  eq eqNat(0,0) = true2 .
  eq eqNat(s(X),0) = false2 .
  eq eqNat(0,s(X)) = false2 .
  eq eqNat(s(X),s(Y)) = eqNat(X,Y) .
  eq ifLNat(true2,Z,W) = Z .
  eq ifLNat(false2,Z,W) = W .
  eq ifNat(true2,X,Y) = X .
  eq ifNat(false2,X,Y) = Y .
  eq minsort(nil,nil) = nil .
  eq minsort(cons(X,Z),W) = ifLNat(eqNat(X,min(cons(X,Z))),
                                  cons(X,minsort(app(rm(X,Z),W),nil)),
                                  minsort(Z,cons(X,W))) .
```

```
   eq from(X) = cons(X,from(s(X))) .
   eq take(0,W) = nil .
   eq take(s(X),cons(Y,Z)) = cons(Y,take(X,Z)) .
   eq nfLNat(nil) = true2 .
   eq nfLNat(cons(X,Z)) = and(nfNat(X),nfLNat(Z)) .
   eq nfNat(0) = true2 .
   eq nfNat(s(X)) = nfNat(X) .
   eq and(true2,A) = A .
   eq and(false2,A) = false2 .
endo
```

B-7 Program `mod`

This program is borrowed from Example 3.5 of [Arts and Giesl, 2001]. Auxiliary functions for natural numbers are included, namely `fact`, `times`, and `plus`. The call considered for evaluation is: `mod(fact(fact(3)),2)`

```
obj MOD is
   sorts Nat Bool2 .
   op 0 :  -> Nat .
   op s : Nat -> Nat               [strat (1)] .
   op true2 : -> Bool2 .
   op false2 : -> Bool2 .
   op minus : Nat Nat -> Nat       [strat (1 -2 0)] .
   op mod : Nat Nat -> Nat         [strat (1 -2 0)] .
   op le : Nat Nat -> Bool2        [strat (1 -2 0)] .
   op ifNat : Bool2 Nat Nat -> Nat [strat (1 0)] .
   op plus : Nat Nat -> Nat        [strat (1 0)] .
   op times : Nat Nat -> Nat       [strat (1 0)] .
   op fact : Nat -> Nat            [strat (1 0)] .
   vars M N : Nat .
   eq le(0,M) = true2 .
   eq le(s(N),0) = false2 .
   eq le(s(N),s(M)) = le(N,M) .
   eq minus(0,N) = 0 .
   eq minus(s(M),0) = s(M) .
   eq minus(s(M),s(N)) = minus(M,N) .
   eq mod(0,M) = 0 .
   eq mod(s(N),0) = 0 .
   eq mod(s(N),s(M)) = ifNat(le(M,N),mod(minus(N,M),s(M)),s(N)) .
   eq ifNat(true2,N,M) = N .
   eq ifNat(false2,N,M) = M .
   eq plus(0,N) = N .
```

```
  eq plus(s(M),N) = s(plus(M,N)) .
  eq times(0,N) = 0 .
  eq times(s(M),N) = plus(N,times(M,N)) .
  eq fact(0) = s(0) .
  eq fact(s(N)) = times(s(N),fact(N)) .
endo
```

B-8 Program mod'

This program is similar to program mod but positive annotations are provided for symbols times and plus in order to avoid differences due to sharing of variables. The call considered for evaluation is: mod(fact(fact(3)),2)

```
obj MOD is
  sorts Nat Bool2 .
  op 0 :  -> Nat [strat ()] .
  op s : Nat -> Nat [strat (1)] .
  op true2 : -> Bool2 .
  op false2 : -> Bool2 .
  op minus : Nat Nat -> Nat       [strat (1 -2 0)] .
  op mod : Nat Nat -> Nat         [strat (1 -2 0)] .
  op le : Nat Nat -> Bool2        [strat (1 -2 0)] .
  op ifNat : Bool2 Nat Nat -> Nat [strat (1 0)] .
  op plus : Nat Nat -> Nat        [strat (1 2 0)] .
  op times : Nat Nat -> Nat       [strat (1 2 0)] .
  op fact : Nat -> Nat            [strat (1 0)] .
  vars M N : Nat .
  eq le(0,M) = true2 .
  eq le(s(N),0) = false2 .
  eq le(s(N),s(M)) = le(N,M) .
  eq minus(0,N) = 0 .
  eq minus(s(M),0) = s(M) .
  eq minus(s(M),s(N)) = minus(M,N) .
  eq mod(0,M) = 0 .
  eq mod(s(N),0) = 0 .
  eq mod(s(N),s(M)) = ifNat(le(M,N),mod(minus(N,M),s(M)),s(N)) .
  eq ifNat(true2,N,M) = N .
  eq ifNat(false2,N,M) = M .
  eq plus(0,N) = N .
  eq plus(s(M),N) = s(plus(M,N)) .
  eq times(0,N) = 0 .
  eq times(s(M),N) = plus(N,times(M,N)) .
  eq fact(0) = s(0) .
```

```
   eq fact(s(N)) = times(s(N),fact(N)) .
endo
```

B-9 Program `average`

This program is borrowed from Example 3.15 of [Arts and Giesl, 2001]. Auxiliary functions for natural numbers are included, namely `fact`, `times`, and `plus`. The call considered for evaluation is: `average(square(square(4)),square(square(4)))`

```
obj AVERAGE is
   sort Nat .
   op 0 :  -> Nat .
   op s : Nat -> Nat           [strat (1)] .
   op average : Nat Nat -> Nat [strat (-1 -2 0)] .
   op plus : Nat Nat -> Nat    [strat (1 0)] .
   op times : Nat Nat -> Nat   [strat (1 0)] .
   op fact : Nat -> Nat        [strat (1 0)] .
   op square : Nat -> Nat      [strat (1 0)] .
   vars M N : Nat .
   eq average(0,0) = 0 .
   eq average(0,s(0)) = 0 .
   eq average(0,s(s(0))) = s(0) .
   eq average(s(M),N) = average(M,s(N)) .
   eq average(M,s(s(s(N)))) = s(average(s(M),N)) .
   eq plus(0,N) = N .
   eq plus(s(M),N) = s(plus(M,N)) .
   eq times(0,N) = 0 .
   eq times(s(M),N) = plus(N,times(M,N)) .
   eq square(N) = times(N,N) .
   eq fact(0) = s(0) .
   eq fact(s(N)) = times(s(N),fact(N)) .
endo
```

Appendix C Proof of termination of program pi

Consider the program of Appendix B-1. After applying the transformation included in Section 7.5 for proving termination, we obtain the following program:

```
obj PI4tr is
   sorts Nat LNat Recip LRecip .
   op 0 : -> Nat .
   op s : Nat -> Nat                          [strat (1)] .
   op posrecip : Nat -> Recip                 [strat (1)] .
   op negrecip : Nat -> Recip                 [strat (1)] .
   op nil  : -> LNat .
   op cons : Nat LNat -> LNat                 [strat (1)] .
   op cons2 : Nat LNat -> LNat                [strat (2)] .
   op rnil  : -> LRecip .
   op rcons : Recip LRecip -> LRecip          [strat (1 2)] .
   op from : Nat -> LNat                      [strat (1 0)] .
   op 2ndspos : Nat LNat -> LRecip            [strat (1 2 0)] .
   op 2ndsneg : Nat LNat -> LRecip            [strat (1 2 0)] .
   op pi : Nat -> LRecip                      [strat (1 0)] .
   op plus : Nat Nat -> Nat                   [strat (1 2 0)] .
   op times : Nat Nat -> Nat                  [strat (1 2 0)] .
   op square : Nat -> Nat                     [strat (1 0)] .
   vars N X Y : Nat . var Z : LNat .
   eq from(X) = cons(X,from(s(X))) .
   eq 2ndspos(0,Z) = rnil .
   eq 2ndspos(s(N),cons(X,Z)) = 2ndspos(s(N),cons2(X,Z)) .
   eq 2ndspos(s(N),cons2(X,cons(Y,Z))) = rcons(posrecip(Y),2ndsneg(N,Z)) .
   eq 2ndsneg(0,Z) = rnil .
   eq 2ndsneg(s(N),cons(X,Z)) = 2ndsneg(s(N),cons2(X,Z)) .
   eq 2ndsneg(s(N),cons2(X,cons(Y,Z))) = rcons(negrecip(Y),2ndspos(N,Z)) .
   eq pi(X) = 2ndspos(X,from(0)) .
   eq plus(0,Y) = Y .
   eq plus(s(X),Y) = s(plus(X,Y)) .
   eq times(0,Y) = 0 .
   eq times(s(X),Y) = plus(Y,times(X,Y)) .
   eq square(X) = times(X,X) .
endo
```

Following [Lucas, 2001a], in order to prove termination of PI4tr (which only contains positive annotations), we can use the techniques for proving termination of context-sensitive rewriting (see [Lucas, 2002c] for a survey of these techniques). The application of Zantema's transformation ([Zantema, 1997]) to remove positive annotations, yields the following TRS (in a generic syntax not bound to OBJ programs):

```
from(X) → cons(X,n__from(s(X)))
2ndspos(0,Z) → rnil
2ndspos(s(N),cons(X,Z)) → 2ndspos(s(N),cons2(X,activate(Z)))
2ndspos(s(N),cons2(X,cons(Y,Z))) → rcons(posrecip(Y),2ndsneg(N,activate(Z)))
2ndsneg(0,Z) → rnil
2ndsneg(s(N),cons(X,Z)) → 2ndsneg(s(N),cons2(X,activate(Z)))
2ndsneg(s(N),cons2(X,cons(Y,Z))) → rcons(negrecip(Y),2ndspos(N,activate(Z)))
pi(X) → 2ndspos(X,from(0))
plus(0,Y) → Y
plus(s(X),Y) → s(plus(X,Y))
times(0,Y) → 0
times(s(X),Y) → plus(Y,times(X,Y))
square(X) → times(X,X)
from(X) → n__from(X)
activate(n__from(X)) → from(X)
activate(X) → X
```

Termination of this program can be proved with the CiME 2.0 system (available at `http://cime.lri.fr/`) by using *dependency graphs* and *simple-mixed* interpretations:

```
CiME> termination R;
Entering the termination expert. Verbose level = 0
checking each of the 3 strongly connected components :
checking component 1 (disjunction of 1 constraints)
[rnil] = 0;
[0] = 0;
[activate](X0) = X0;
[n__from](X0) = 0;
[square](X0) = X02;
[pi](X0) = 0;
[negrecip](X0) = 0;
[posrecip](X0) = 0;
[s](X0) = X0 + 1;
[from](X0) = 0;
[times](X0,X1) = X1*X0;
[plus](X0,X1) = X1 + X0;
[2ndsneg](X0,X1) = 0;
[rcons](X0,X1) = 0;
[cons2](X0,X1) = 0;
[2ndspos](X0,X1) = 0;
[cons](X0,X1) = 0;
['plus`](X0,X1) = X0;
```

```
checking component 2 (disjunction of 1 constraints)
[rnil] = 0;
[0] = 0;
[activate](X0) = X0;
[n__from](X0) = 0;
[square](X0) = X02;
[pi](X0) = 0;
[negrecip](X0) = 0;
[posrecip](X0) = 0;
[s](X0) = X0 + 1;
[from](X0) = 0;
[times](X0,X1) = X1*X0;
[plus](X0,X1) = X1 + X0;
[2ndsneg](X0,X1) = 0;
[rcons](X0,X1) = 0;
[cons2](X0,X1) = 0;
[2ndspos](X0,X1) = 0;
[cons](X0,X1) = 0;
['times`](X0,X1) = X0;

checking component 3 (disjunction of 2 constraints)
[rnil] = 0;
[0] = 0;
[activate](X0) = X0;
[n__from](X0) = 0;
[square](X0) = X02;
[pi](X0) = 0;
[negrecip](X0) = 0;
[posrecip](X0) = 0;
[s](X0) = X0 + 1;
[from](X0) = 0;
[times](X0,X1) = X1*X0;
[plus](X0,X1) = X1 + X0;
[2ndsneg](X0,X1) = 0;
[rcons](X0,X1) = 0;
[cons2](X0,X1) = 0;
[2ndspos](X0,X1) = 0;
[cons](X0,X1) = 0;
['2ndsneg`](X0,X1) = X0;
['2ndspos`](X0,X1) = X0;

Termination proof found.
Execution time: 4.200000 sec
```

```
- : unit = ()
```

Chapter 8

Correct and Complete Strategy Annotations by Program Transformation

This chapter addresses how to perform correct and complete computations (or evaluations) using restrictions of rewriting such as *CSR* [Lucas, 1998a] or E-evaluation [Nagaya, 1999]. In Section 8.1, we give a brief introduction to the problem of performing correct and complete computations using restrictions of rewriting. In Section 8.2, we recall some results comparing the two restrictions of rewriting considered in this thesis: E-strategies and *CSR*. Due to the lack of a standard, commonly accepted terminology, we clarify in Section 8.3 our notion of correct and complete computations with (positive) strategy annotations. We also demonstrate that previously known approaches for computing normal forms with (non-terminating) OBJ programs using positive strategy annotations are not completely satisfactory. In Section 8.4, we formalize a program transformation which enables correct and complete computations by using appropriate restrictions of rewriting. Moreover, we have implemented such program transformation into the OnDemandOBJ prototype of Section 7.6. Finally, in Section 8.5, we discuss some related work and present our conclusions.

A short version of this chapter appeared in [Alpuente et al., 2002c, 2003b]

8.1 Introduction

Strategy annotations are used in the OBJ family of languages (OBJ2 [Futatsugi *et al.*, 1985], OBJ3 [Goguen *et al.*, 2000], CafeOBJ [Futatsugi and Nakagawa, 1997], and Maude [Clavel *et al.*, 1996]) to *avoid nontermination* ([Goguen *et al.*, 2000], Section 2.4.4).

Example 8.1 *The following* OBJ *program from Example 6.3:*

```
obj EXAMPLE is
  sorts Nat LNat .
  op 0     : -> Nat .
  op s     : Nat -> Nat .
  op nil   : -> LNat .
  op cons  : Nat LNat -> LNat [strat (1)] .
  op from  : Nat -> LNat .
  op sel   : Nat LNat -> Nat .
  op first : Nat LNat -> LNat .
  vars X Y : Nat .
  var Z : LNat .
  eq sel(s(X),cons(Y,Z)) = sel(X,Z) .
  eq sel(0,cons(X,Z)) = X .
  eq from(X) = cons(X,from(s(X))) .
  eq first(0,Z) = nil .
  eq first(s(X),cons(Y,Z)) = cons(Y,first(X,Z)) .
endo
```

specifies an explicit *strategy annotation for the list constructor* `cons` *which disables replacements on the second argument. In this way, we can ensure that computations with this program are terminating (see Example 8.6 below for a formal justification of this claim).*

Termination of rewriting under strategy annotations has been investigated in a number of papers [Fissore *et al.*, 2001; Lucas, 2001a,b]. Unfortunately, using rewriting restrictions may cause *incompleteness*, i.e., normal forms of input expressions could be unreachable by restricted computation.

Example 8.2 . *Consider Example 8.1. The evaluation of expression* $t =$ `first(s(0),from(0))` *stops yielding the term* `cons(0,first(0,from(s(0))))` *since reduction on the second argument of* `cons` *is forbidden:*

first(s(0),<u>from(0)</u>)
  → <u>first(s(0),cons(0,from(s(0))))</u>

$\to$ `cons(0,first(0,from(s(0))))`

Note that `cons(0,first(0,from(s(0))))` *is not a normal form. However, the intended normal form* `cons(0,nil)` *associated to t, i.e. $t \to^! $* `cons(0,nil)`*, cannot be obtained by restrictions of the rewriting relation. For instance, in order to obtain the intended normal form of term t, position 2 of term* `cons(0,first(0,from(s(0))))` *should be evaluated.*

Note that from the user's point of view, this must be thought of as a kind of *incorrect* evaluation, whenever normal forms are expected as the result of a computation. This problem can also be solved by implementing the naïve extension of root-normalizing strategies to normalizing ones described in Definition 2.1. However, in this chapter, we are able to develop a program transformation which solves the incorrectness problem while is still able to preserve termination of computations. This enables correct and complete computations for current OBJ implementations.

In order to set the stage for the program transformation, we first clarify our notion of correct and complete computations with (positive) strategy annotations. As there is no standard, commonly accepted terminology, current definitions are rather misleading and we think this may cause an erroneous understanding. Then, we demonstrate that previously known approaches for computing normal forms with (non-terminating) OBJ programs using positive strategy annotations are not completely satisfactory in practice since they do ensure correctness but not the desired 'definedness' (completeness). Finally, we formalize the program transformation using an incremental presentation, i.e. we first show a naïve approach and point out some drawbacks, then we present the final satisfactory program transformation.

8.2 E-strategies and Context-sensitive Rewriting

Rewriting with strategy annotations is closely related to *CSR*. Given an E-strategy map φ for $\mathcal{F}$, we can define $\mu^\varphi \in M_\mathcal{F}$ as in Section 7.3.1: $\mu^\varphi(f) = \{i \in \varphi(f) \mid i \neq 0\}$ for all $f \in \mathcal{F}$, where $e \in L$ means that item e appears somewhere within the list L. We will drop superscript φ from μ^φ if no confusion arises. Moreover, we also write $\varphi \in CM_\mathcal{R}$ meaning that $\mu^\varphi \in CM_\mathcal{R}$.

Example 8.3 *The TRS $\mathcal{R}$:*

```
sel(0,cons(x,z))         → x
sel(s(x),cons(y,z))      → sel(x,z)
first(0,z)               → nil
first(s(x),cons(y,z))    → cons(y,first(x,z))
from(x)                  → cons(x,from(s(x)))
```

together with the replacement map

$$\mu(\texttt{s}) = \mu(\texttt{cons}) = \mu(\texttt{from}) = \{1\} \quad \textit{and} \quad \mu(\texttt{sel}) = \mu(\texttt{first}) = \{1, 2\}$$

correspond to the OBJ *program in Example 8.1.*

Every $\to_\varphi$-reduction step issued from $\langle t, p\rangle$ correspond to a μ^φ-rewriting step on the unlabelled version $erase(t)$ of t (or $erase(t)$ just remains unchanged).

Theorem 8.4 [Lucas, 2001a] *Let $\mathcal{R}$ be a TRS and φ be an E-strategy map. Let $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$, and $p \in \mathcal{P}os^\mu(erase(t))$ be s.t. $root(t,p) = f_L$ for some suffix L of $\varphi(f)$. If $\langle t, p\rangle \to_\varphi \langle s, q\rangle$, then $q \in \mathcal{P}os^\mu(erase(s))$ and $erase(t) \hookrightarrow_\mu^= erase(s)$.*

Termination of OBJ programs and termination of *CSR* are also related.

Theorem 8.5 [Lucas, 2001a] *An* OBJ *program* P *with E-strategy map φ is terminating if the corresponding TRS $\mathcal{R}$ is μ^φ-terminating.*

Termination of *CSR* has been studied in a number of papers, see [Lucas, 2002c] for an overview of the different methods for proving termination of *CSR*.

Example 8.6 *Consider $\mathcal{R}$ and μ as in Example 8.3. The μ-termination of (a superset of) $\mathcal{R}$ is demonstrated in Example 7 of [Borralleras et al., 2002]. Hence, by Theorem 8.5, the* OBJ *program in Example 8.1 is terminating.*

8.3 Correctness and Completeness

A *rewriting semantics* for a TRS $\mathcal{R} = (\mathcal{F}, R)$ is a mapping $\mathsf{S} : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{P}(\mathcal{T}(\mathcal{F}, \mathcal{X}))$ such that, for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $s \in \mathsf{S}(t)$, $t \to^*_\mathcal{R} s$ [Lucas, 2001c]. Note that, given a TRS $\mathcal{R}$ and an E-strategy map φ, $eval_\varphi$ is a rewriting semantics for $\mathcal{R}$. A semantics S is *deterministic* if for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $|\mathsf{S}(t)| \leq 1$. On the other hand, a semantics S is *defined* if for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $|\mathsf{S}(t)| \geq 1$. In general, $eval_\varphi$ is neither deterministic nor defined. Note that φ-termination of $\mathcal{R}$ implies definedness of $eval_\varphi$.

The semantics which is most commonly considered in functional programming is the set of constructor terms that $\mathcal{R}$ is able to produce in a finite number of rewriting steps ($\mathsf{eval}(t) = \{s \in \mathcal{T}(\mathcal{C}, \mathcal{X}) \mid t \to^*_\mathcal{R} s\}$). Other kinds of semantics often considered for $\mathcal{R}$ are, e.g., the set of all possible reducts of a term which are head-normal forms ($\mathsf{hnf}(t) = \{s \in \mathsf{HNF}_\mathcal{R} \mid t \to^*_\mathcal{R} s\}$), or normal forms ($\mathsf{nf}(t) = \mathsf{hnf}(t) \cap \mathsf{NF}_\mathcal{R}$). Thus, given a semantics S for $\mathcal{R}$ (e.g., $\mathsf{S} \in \{\mathsf{eval}, \mathsf{hnf}, \mathsf{nf}\}$), a different rewriting semantics for $\mathcal{R}$ (e.g., $eval_\varphi$) is:

correct (w.r.t. S) if $eval_\varphi(t) \subseteq \mathsf{S}(t)$ for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and

complete (w.r.t. S) if, $\mathsf{S}(t) \subseteq eval_\varphi(t)$ for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

Computations with OBJ programs produce expressions (by means of $eval_\varphi$) called *E-normal forms* (*ENF*s). Such terms are not generally normal forms (i.e., terms without redexes). Therefore, $eval_\varphi$ is not guaranteed to be either correct or complete w.r.t. nf. In fact, we have the following:

Theorem 8.7 [Lucas, 2001a] *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS and φ be a E-strategy map such that for all $f \in \mathcal{D}$, $\varphi(f)$ ends in 0. If $s \in eval_\varphi(t)$, then s is a μ-normal form of t.*

Requiring that $\varphi(f)$ ends in 0 for all $f \in \mathcal{D}$ is essential in our development (see also [Eker, 2000] for a thorough analysis of the relevance of this requirement). Thus, we say that a E-strategy map φ is regular[1] if this condition holds.

If the strategy annotations are 'compatible' with the canonical replacement map, we can ensure that the E-strategy is correct w.r.t. hnf.

Theorem 8.8 [Lucas, 2001a] *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ be a left-linear TRS and φ be a regular E-strategy map such that $\varphi \in CM_\mathcal{R}$. If $s \in eval_\varphi(t)$, then s is a head-normal form.*

If we restrict our attention to the computation of values (i.e., constructor terms), then *CSR* is powerful enough to compute the pursued outcome. Given TRS $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ and $\mathcal{B} \subseteq \mathcal{C}$, we let $\mu^\mathcal{B}_\mathcal{R}(c) = \{1, \ldots, ar(c)\}$ for all $c \in \mathcal{B}$, and $\mu^\mathcal{B}_\mathcal{R}(f) = \mu^{can}_\mathcal{R}(f)$ if $f \in \mathcal{F} - \mathcal{B}$. Note that $\mu^\mathcal{B}_\mathcal{R} \in CM_\mathcal{R}$.

Theorem 8.9 [Lucas, 1998a] *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a left-linear TRS, $\mathcal{B} \subseteq \mathcal{C}$ and $\mu \in M_\mathcal{F}$ be such that $\mu^\mathcal{B}_\mathcal{R} \sqsubseteq \mu$. Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and $\delta \in \mathcal{T}(\mathcal{B}, \mathcal{X})$. Then, $t \to^* \delta$ iff $t \hookrightarrow^*_\mu \delta$.*

Theorem 8.9 is very easy to use in sorted signatures (as in OBJ programs), since, given a term t (of sort τ), we are able to establish the set of constructors $\mathcal{B} \subseteq \mathcal{C}$ which should be considered (namely, the constructor symbols of sort τ). Unfortunately, Theorem 8.9 does not directly apply to OBJ computations, as they must obey the order of evaluation expressed by the strategy annotations. Nevertheless, we still have the following.

Theorem 8.10 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a left-linear, confluent TRS and $\mathcal{B} \subseteq \mathcal{C}$. Let φ be a regular E-strategy map such that $\mathcal{R}$ is φ-terminating. Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $\delta \in \mathcal{T}(\mathcal{B}, \mathcal{X})$. If $\mu^\mathcal{B}_\mathcal{R} \sqsubseteq \mu^\varphi$, then $t \to^! \delta$ iff $\delta \in eval_\varphi(t)$.*

[1] This terminology is used in [Ogata and Futatsugi, 1997], with a slightly different meaning.

Proof. If $\delta \in eval_\varphi(t)$, then, by Theorem 8.7, $t \hookrightarrow^!_\mu \delta$. Since δ is a normal form, we have $t \to^! \delta$.

Assume that $t \to^! \delta$. By φ-termination, $eval_\varphi(t) \neq \varnothing$. Let $s \in eval_\varphi(t)$. By Theorem 8.7, $t \hookrightarrow^*_\mu s$ and s is a μ-normal form. Assume that $s \neq \delta$. By confluence $s \to^* \delta$ and by Theorem 8.9, $s \hookrightarrow^*_\mu \delta$. Hence, s is not a μ-normal form, a contradiction. Therefore, $s = \delta \in eval_\varphi(t)$. □

For instance, φ can be used to compute the *value* of every expression of the sort `Nat` in the OBJ program in Example 8.1 (since $\mu_{\mathcal{R}}^{\{\mathtt{0},\mathtt{s}\}} \sqsubseteq \mu^\varphi$). This is not true for expressions of the sort `LNat` as the following example shows.

Example 8.11 *The evaluation of expression* $t =$ `first(s(0),from(0))` *of sort* `LNat` *using the program in Example 8.1 yields (we use the version 1.0.5 of the* Maude *interpreter*[2] *but other interpreters behave similarly*[3]*):*

```
==========================================
obj EXAMPLE
==========================================
reduce in EXAMPLE : first(s(0), from(0)) .
rewrites: 2 in -10ms cpu (0ms real) (~ rewrites/second)
result LNat: cons(0, first(0, from(s(0))))
```

Note that `cons(0,first(0,from(s(0))))` *is not a normal form. However,* $t \to^*$ `cons(0,nil)` $\in \mathcal{T}(\mathcal{C},\mathcal{X})$*, i.e.,* `cons(0,nil)` *is a value of* t *which cannot be obtained by using the* OBJ *interpreter.*

Correctness of OBJ computations w.r.t. nf can also be achieved:

1. Nagaya shows that if $\varphi(f)$ contains all indices $0, 1, \ldots, ar(f)$ for each symbol $f \in \mathcal{F}$, and $\varphi(f)$ ends in 0 for defined symbols $f \in \mathcal{D}$, then $eval_\varphi$ is correct w.r.t. nf (Theorem 6.1.12 in [Nagaya, 1999] –Theorem 6.21 in this thesis–).

2. Nakamura and Ogata show that given a strategy map φ, if $eval_\varphi$ is correct w.r.t. hnf, then $eval_{\varphi'}$ is correct w.r.t. nf for any φ' given by $\varphi'(f) = \varphi(f){+}{+}(i_1 \cdots i_n)$ for all symbol $f \in \mathcal{F}$ (where '++' appends two lists, and $\{i_1, \ldots, i_n\} = \{1, \ldots, ar(f)\} - \mu^\varphi(f)$) (Theorem 3.2 in [Nakamura and Ogata, 2001]).

For instance, when φ is defined as in Example 8.1, $eval_\varphi$ is correct w.r.t. hnf (use Theorem 8.8). Moreover, since the OBJ program in Example 8.1 is φ-terminating, $eval_\varphi$ is defined. Thus, the evaluation of every term t yields a head-normal form of

[2] Available at `http://maude.cs.uiuc.edu/`.

[3] We have reproduced all our experiments using the OBJ3 interpreter v. 2.0 (available at `http://www.kindsoftware.com/products/opensource/obj3/OBJ3/`) and the CafeOBJ interpreter v. 1.3.1 (available at `http://www.ipa.go.jp/STC/CafeP/cafe.html`).

t (i.e., φ can be thought of as being *head-normalizing*). Unfortunately $eval_{\varphi'}$ is not defined anymore: the head-normalization capability of φ gets lost.

Example 8.12 *Consider the program in Example 8.1 with* $\varphi'(\mathtt{cons}) = (1\ 2)$ *and* $\varphi'(f) = \varphi(f)$ *for every other symbol* f*. Consider again the evaluation of* $t =$ `first(s(0),from(0))`*:*

```
==========================================
obj EXAMPLE-INF
==========================================
reduce in EXAMPLE : first(s(0), from(0)) .
Segment violation
```

The problem is that the evaluation of t*, i.e., the evaluation of*

$$\varphi'(t) = \mathtt{first}_{(1\ \ 2\ \ 0)}\mathtt{(s}_{(1)}\mathtt{(0}_{nil}\mathtt{)\,,from}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{))}$$

using $\to_{\varphi'}$ *does not terminate (we underline the contracted redexes):*

$$\begin{aligned}
&\langle \mathtt{first}_{(1\ 2\ 0)}\mathtt{(s}_{(1)}\mathtt{(0}_{nil}\mathtt{),from}_{(1\ 0)}\mathtt{(0}_{nil}\mathtt{))}, \Lambda\rangle \\
&\quad\to_{\varphi'} \langle \mathtt{first}_{(2\ 0)}\mathtt{(s}_{(1)}\mathtt{(0}_{nil}\mathtt{),from}_{(1\ 0)}\mathtt{(0}_{nil}\mathtt{))}, 1\rangle \\
&\quad\to_{\varphi'} \langle \mathtt{first}_{(2\ 0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),from}_{(1\ 0)}\mathtt{(0}_{nil}\mathtt{))}, 1.1\rangle \\
&\quad\to_{\varphi'} \langle \mathtt{first}_{(2\ 0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),from}_{(1\ 0)}\mathtt{(0}_{nil}\mathtt{))}, 1\rangle \\
&\quad\to_{\varphi'} \langle \mathtt{first}_{(2\ 0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),from}_{(1\ 0)}\mathtt{(0}_{nil}\mathtt{))}, \Lambda\rangle \\
&\quad\to_{\varphi'} \langle \mathtt{first}_{(0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),from}_{(1\ 0)}\mathtt{(0}_{nil}\mathtt{))}, 2\rangle \\
&\quad\to_{\varphi'} \langle \mathtt{first}_{(0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),from}_{(0)}\mathtt{(0}_{nil}\mathtt{))}, 2.1\rangle \\
&\quad\to_{\varphi'} \langle \mathtt{first}_{(0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),}\underline{\mathtt{from}_{(0)}\mathtt{(0}_{nil}\mathtt{)}}\mathtt{)}, 2\rangle \\
&\quad\to_{\varphi'} \langle \mathtt{first}_{(0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),cons}_{(1\ 2)}\mathtt{(0}_{nil}\mathtt{,from}_{(1\ 0)}\mathtt{(s}_{(1)}\mathtt{(0}_{nil}\mathtt{))))}, 2\rangle \\
&\quad\to^{+}_{\varphi'} \langle \mathtt{first}_{(0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),cons}_{(2)}\mathtt{(0}_{nil}\mathtt{,from}_{(1\ 0)}\mathtt{(s}_{(1)}\mathtt{(0}_{nil}\mathtt{))))}, 2\rangle \\
&\quad\to_{\varphi'} \langle \mathtt{first}_{(0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),cons}_{nil}\mathtt{(0}_{nil}\mathtt{,from}_{(1\ 0)}\mathtt{(s}_{(1)}\mathtt{(0}_{nil}\mathtt{))))}, 2.2\rangle \\
&\quad\to^{+}_{\varphi'} \langle \mathtt{first}_{(0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{),cons}_{nil}\mathtt{(0}_{nil}\mathtt{,}\underline{\mathtt{from}_{(0)}\mathtt{(s}_{nil}\mathtt{(0}_{nil}\mathtt{))}}\mathtt{))}, 2.2\rangle \\
&\quad\to_{\varphi'} \cdots
\end{aligned}$$

The Maude *interpreter 'shows' this infinite sequence as a 'segment violation'.*

Thus, the φ-termination of $\mathcal{R}$ (see Example 8.6) does *not* ensure definedness of $eval_{\varphi'}$ as the previous results by Nagaya, and Nakamura-Ogata may suggest. Moreover, $eval_{\varphi}$ was able to obtain head-normal forms that $eval_{\varphi'}$ does *not* attain (compare the evaluation of t in Examples 8.11 and 8.12). Example 8.12 also shows that requiring φ-termination in Theorem 8.10 is essential for ensuring correct and complete evaluations (note that $\mathcal{R}$ and φ' in Example 8.12 fulfill all requirements for Theorem 8.10, except for φ'-termination).

In the following section, we propose a solution to (partially) overcome this problem which is based on program transformation.

8.4 Program Transformations for Complete Evaluations

The discussion and examples in the previous section suggest to isolate the replacement restrictions which are needed to achieve the head-evaluation of a term t (which requires, at least, considering the map $\mu^{can}_{\mathcal{R}}$, see Theorem 8.8) from the restrictions which are needed to get them within a constructor context $C[\,] \in \mathcal{T}(\mathcal{B} \cup \{\square\}, \mathcal{X})$ for some $\mathcal{B} \subseteq \mathcal{C}$ (which, at least, requires $\mu^{\mathcal{B}}_{\mathcal{R}}$, see Theorem 8.10). In practice, we only need (and want) to ascertain the sort τ of input expressions we want to evaluate in order to fix the 'interesting' constructor terms. Assume that symbols $f \in \mathcal{F}$ are sorted by: $f : \tau_1 \times \cdots \times \tau_k \to \tau$. The (output) *sort* of f is $sort(f) = \tau$. Variables $x \in \mathcal{X}$ also have a sort, $sort(x)$. We also assume that all terms are well sorted everywhere. The sort of a term t is the sort of its outermost symbol. Given a sort τ, let $\mathcal{C}^*_\tau \subseteq \mathcal{C}$ be the set of constructor symbols that can be found in the constructor terms of sort τ. For instance, $\mathcal{C}^*_{\mathtt{Nat}} = \{\mathtt{0}, \mathtt{s}\}$ and $\mathcal{C}^*_{\mathtt{LNat}} = \{\mathtt{0}, \mathtt{s}, \mathtt{nil}, \mathtt{cons}\}$. We introduce a set $\mathcal{C}'$ of fresh constructor symbols: they are renamed versions c' of the original constructors $c \in \mathcal{C}^*_\tau$ that give value to all the arguments.

The renaming of constructor symbols $c \in \mathcal{C}^*_\tau$ into new constructor symbols $c' \in \mathcal{C}'$ is performed by the rules

$$\mathtt{quote}_\tau(c(x_1, \ldots, x_k)) \to c'(\mathtt{quote}_{\tau_1}(x_1), \ldots, \mathtt{quote}_{\tau_k}(x_k))$$

where $c, c' : \tau_1 \times \cdots \times \tau_k \to \tau$. Let *Quote* be the set containing all these symbols: $Quote = \{\mathtt{quote}_\tau \mid \exists c \in \mathcal{C}^*_\tau, sort(c) = \tau\}$. The evaluation of a term t would proceed by reducing $\mathtt{quote}_{sort(t)}(t)$. The obtained value is built by using only symbols in $\mathcal{C}'$. After the evaluation, new symbols $\mathtt{unquote}_\tau : \tau \to \tau$ are used to reverse the renamings. For each constant $b \in \mathcal{C}^*_\tau$, we add a rule

$$\mathtt{unquote}_{sort(b)}(b') \to b$$

For each $c \in \mathcal{C}^*_\tau$ such that $c : \tau_1 \times \cdots \times \tau_k \to \tau$, $k > 0$, and $\mu^\varphi(c) = \{1, \ldots, k\}$, we add a rule

$$\mathtt{unquote}_\tau(c'(x_1, \ldots, x_k)) \to c(\mathtt{unquote}_{\tau_1}(x_1), \ldots, \mathtt{unquote}_{\tau_k}(x_k))$$

Finally, for each $c \in \mathcal{C}^*_\tau$ such that $c : \tau_1 \times \cdots \times \tau_k \to \tau$, $k > 0$, and $\mu^\varphi(c) \neq \{1, \ldots, k\}$, we consider a new symbol $f_c : \tau_1 \times \cdots \times \tau_k \to \tau$; i.e. we add two rules

$$\begin{array}{rcl} \mathtt{unquote}_\tau(c'(x_1, \ldots, x_k)) & \to & f_c(\mathtt{unquote}_{\tau_1}(x_1), \ldots, \mathtt{unquote}_{\tau_k}(x_k)) \\ f_c(x_1, \ldots, x_k) & \to & c(x_1, \ldots, x_k) \end{array}$$

We collect these new symbols together in a new set *Unquote*. We denote $\mathsf{E}_\tau(\mathcal{R})$ as the TRS obtained from joining these rules together with those of $\mathcal{R}$. The transformed

TRS $\mathsf{E}_\tau(\mathcal{R})$ includes the rules of the original TRS $\mathcal{R}$. The added rules manage the appropriate quoting and unquoting of constructor symbols: quoted constructors enable the evaluation of all their arguments; after evaluating them, defined symbol `unquote` restores the original constructors. Therefore, we also extend the (original) E-strategy φ: let $\varphi' = Emap_\tau(\varphi)$ be defined as follows: $\varphi'(f) = \varphi(f)$ if $f \in \mathcal{F}$, $\varphi'(c') = \varphi(c)$ if $c' \in \mathcal{C}'$, $\varphi'(\mathtt{quote}_\tau) = \varphi'(\mathtt{unquote}_\tau) = (1\ 0)$ for all sort τ, and $\varphi(f_c) = (1\ \cdots\ ar(c)\ 0)$ for each $c \in \mathcal{C}^*_\tau$ such that $\mu^\varphi(c) \neq \{1, \ldots, ar(c)\}$. In the following results, $eval_{\varphi'}$ uses φ' and $\mathsf{E}_\tau(\mathcal{R})$ to evaluate terms ($eval_\varphi$ uses φ and $\mathcal{R}$, as above). Our transformation is correct[4] in a very general sense. In the following, we give an auxiliary result in order to prove Theorem 8.14 below.

Lemma 8.13 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS. Let φ be a positive E-strategy map. Let $\mathcal{R}' = \mathsf{E}_\tau(\mathcal{R})$ for a given sort τ and $\varphi' = Emap_\tau(\varphi)$. Let $\delta \in \mathcal{T}(\mathcal{C})$ and δ' be δ with all constructor symbols c renamed to their alias c'. For all term s of sort τ, $\delta \in eval_{\varphi'}(\mathtt{unquote}_\tau(s))$ if and only if $\delta' \in eval_{\varphi'}(s)$.*

Proof. We proceed by induction on δ. If δ is a constant b (which necessarily belongs to $\mathcal{T}(\mathcal{C}^*_\tau)$), then it is immediate. If $\delta = c(\delta_1, \ldots, \delta_k)$ (and $\delta' = c'(\delta'_1, \ldots, \delta'_k)$), then whenever $\delta \in eval_{\varphi'}(\mathtt{unquote}_\tau(s))$ (i.e., we first consider the 'only if' part), we have:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\langle \mathtt{unquote}_{\mathtt{(1\ 0)}}(\varphi'(s)), \Lambda\rangle} \\
\to_{\varphi'} & \mathtt{unquote}_{\mathtt{(0)}}(\varphi'(s)), 1\rangle \\
\to^*_{\varphi'} & \mathtt{unquote}_{\mathtt{(0)}}(c'_{nil}(\delta''_1, \ldots, \delta''_k)), 1) \\
\to^*_{\varphi'} & \mathtt{unquote}_{\mathtt{(0)}}(c'_{nil}(\delta''_1, \ldots, \delta''_k)), \Lambda\rangle \\
\to_{\varphi'} & h^c_{(1\ 2\cdots k)++L}(\mathtt{unquote}_{\mathtt{(1\ 0)}}(\delta''_1), \ldots, \mathtt{unquote}_{\mathtt{(1\ 0)}}(\delta''_k))), 1\rangle
\end{array}
$$

where h^c is c' if $\mu^\varphi(c) = \{1, \ldots, k\}$ and h^c is f_c otherwise. Note that, in any case, we can write $\varphi'(h^c) = (1\ 2 \cdots k) + +L$ where $L = (0)$ if $h^c = f_c$ and L is the empty list if $h^c = c$. We also have that $erase(\delta'') = \delta'$ for $\delta'' = c'_{nil}(\delta''_1, \ldots, \delta''_k)$. Even though $\varphi'(\delta')$ can be different from δ'' (the strategy labels of constructor symbols within δ'' are empty), since $erase(\varphi'(\delta')) = \delta' = erase(\delta'')$ is a constructor term, this is not a problem for applying the induction hypothesis which concludes:

$$
\begin{array}{l}
\langle h^c_{(1\ 2\cdots k)++L}(\mathtt{unquote}_{\mathtt{(1\ 0)}}(\delta''_1), \ldots, \mathtt{unquote}_{\mathtt{(1\ 0)}}(\delta''_k))), 1\rangle \\
\qquad \to^*_{\varphi'} \langle h^c_L(\delta'''_1, \ldots, \delta'''_k)), \Lambda\rangle
\end{array}
$$

where $erase(\delta'''_i) = \delta_i$. Now, we consider two cases depending on h^c.

1. If $h^c = c$, then L is the empty list, $h^c_L(\delta'''_1, \ldots, \delta'''_k) = c_{nil}(\delta'''_1, \ldots, \delta'''_k)$ and $erase(c_{nil}(\delta'''_1, \ldots, \delta'''_k)) = \delta$.

[4] In this section we do not strictly use 'correct' and 'complete' in the sense defined in Section 8.3 because we need to consider two rewrite systems rather than only one.

2. If $h^c = f_c$, then $L = (0)$ and

$$\begin{array}{rcl}\langle h^c_L(\delta'''_1,\ldots,\delta'''_k)),\Lambda\rangle & \to_{\varphi'} & \langle c_{\varphi(c)}(\delta'''_1,\ldots,\delta'''_k),\Lambda\rangle \\ & \to^*_{\varphi'} & \langle c_{nil}(\delta'''_1,\ldots,\delta'''_k),\Lambda\rangle\end{array}$$

and, again, $erase(c_{nil}(\delta'''_1,\ldots,\delta'''_k)) = \delta$.

The 'if' part easily follows from the previous considerations. □

Theorem 8.14 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS. Let φ be a regular E-strategy map. Let $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ be such that $sort(t) = \tau$ and $\delta \in \mathcal{T}(\mathcal{C})$. Let $\mathcal{R}' = \mathsf{E}_\tau(\mathcal{R})$ and $\varphi' = Emap_\tau(\varphi)$. If $\delta \in eval_{\varphi'}(\mathtt{unquote}_\tau(\mathtt{quote}_\tau(t)))$, then $t \to^*_{\mathcal{R}} \delta$.*

Proof. By Lemma 8.13, $\delta \in eval_{\varphi'}(\mathtt{unquote}_\tau(\mathtt{quote}_\tau(t)))$ if and only if $\delta' \in eval_{\varphi'}(\mathtt{quote}_\tau(t))$, where δ' is δ with all constructor symbols c renamed to their alias c'. Thus, we prove that $t \to^*_{\mathcal{R}} \delta$ if $\delta' \in eval_{\varphi'}(\mathtt{quote}_\tau(t))$. We proceed by induction on δ. If δ is a constant b, then the evaluation of $\mathtt{quote}_\tau(t)$ according to φ' is (note that, since $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $\varphi(t) = \varphi'(t)$; we left the indexing sort of symbols quote unspecified):

$$\begin{array}{rll}\langle \mathtt{quote}_{(1\ 0)}(\varphi(t)),\Lambda\rangle & \to_{\varphi'} \langle \mathtt{quote}_{(0)}(\varphi(t)),1\rangle & (*) \\ & \to_{\varphi'} & \\ & \vdots & \\ & \to_{\varphi'} \langle \mathtt{quote}_{(0)}(b_{nil}),\Lambda\rangle & (**) \\ & \to_{\varphi'} \langle b'_{nil},\Lambda\rangle & \end{array}$$

Thus, from the evaluation steps between $(*)$ and $(**)$ (which only involve rules in $\mathcal{R}$ and annotations issued from φ on symbols in $\mathcal{F}$) we conclude that $b \in eval_\varphi(t)$. Thus, by Theorem 8.7, $t \hookrightarrow^*_{\mathcal{R}} b$, i.e., $t \to^*_{\mathcal{R}} b$.

If $\delta = c(\delta_1,\ldots,\delta_k)$, then similarly:

$$\begin{array}{rll}\langle \mathtt{quote}_{(1\ 0)}(\varphi(t)),\Lambda\rangle & \to_{\varphi'} \langle \mathtt{quote}_{(0)}(\varphi(t)),1\rangle & (*) \\ & \to_{\varphi'} & \\ & \vdots & \\ & \to_{\varphi'} \langle \mathtt{quote}_{(0)}(c_{nil}(s'_1,\ldots,s'_k)),\Lambda\rangle & (**) \\ & \to_{\varphi'} \langle c'_{(1\ 2\ \cdots\ k)}(\mathtt{quote}_{(1\ 0)}(s'_1),\ldots,\mathtt{quote}_{(1\ 0)}(s'_k)),\Lambda\rangle & \end{array}$$

where, again, the evaluation steps between $(*)$ and $(**)$ only involve rules in $\mathcal{R}$, positions $p \geq 1$ and annotations issued from φ on symbols in $\mathcal{F}$ (in particular, $s = c(erase(s'_1),\ldots,erase(s'_k)) \in \mathcal{T}(\mathcal{F},\mathcal{X})$), and the presence of c is needed if the evaluation yields δ because the rule

$$\mathtt{quote}(c(x_1,\ldots,x_k)) \to c'(\mathtt{quote}(x_1),\ldots,\mathtt{quote}(x_k))$$

must be applied in order to allow the evaluation to progress towards δ'. Thus, we can say that $s \in eval_\varphi(t)$, and by Theorem 8.7, $t \hookrightarrow^*_\mathcal{R} s$, i.e., $t \to^*_\mathcal{R} s$. The previous evaluation sequence continues as follows:

$$\begin{array}{ll}\langle c'_{(1\ 2\ \cdots\ k)}(\texttt{quote}_{(1\ 0)}(s'_1),\ldots,\texttt{quote}_{(1\ 0)}(s'_k)),\Lambda\rangle & \\ \quad \to_{\varphi'} & \langle c'_{(2\ \cdots\ k)}(\texttt{quote}_{(1\ 0)}(s'_1),\ldots,\texttt{quote}_{(1\ 0)}(s'_k)),1\rangle \\ \quad \to_{\varphi'} & \\ \quad \vdots & \\ \quad \to_{\varphi'} & \langle c'_{nil}(\delta''_1,\ldots,\delta''_k),\Lambda\rangle\end{array}$$

where $erase(\delta''_i) = \delta'_i$ for $1 \leq i \leq k$. Obviously, $\delta'_i \in eval_\varphi(\texttt{quote}(erase(s'_i)))$ for $1 \leq i \leq k$. By the induction hypothesis, $erase(s'_i) \to^*_\mathcal{R} \delta_i$ for $1 \leq i \leq k$, hence $t \to^*_\mathcal{R} c(erase(s'_1),\ldots,erase(s'_k)) \to^*_\mathcal{R} \delta$. □

Thus, no 'unexpected' value can be obtained when evaluating $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ of sort τ as $\texttt{unquote}_\tau(\texttt{quote}_\tau(t))$. Moreover, no constructor term (of sort τ) obtained by using φ and $\mathcal{R}$ gets lost when $Emap_\tau(\varphi)$ and $\mathsf{E}_\tau(\mathcal{R})$ are used instead.

Corollary 8.15 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS. Let φ be a regular E-strategy map. Let $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ be such that $sort(t) = \tau$ and $\delta \in \mathcal{T}(\mathcal{C})$. Let $\mathcal{R}' = \mathsf{E}_\tau(\mathcal{R})$ and $\varphi' = Emap_\tau(\varphi)$. If $\delta \in eval_\varphi(t)$, then $\delta \in eval_{\varphi'}(\texttt{unquote}_\tau(\texttt{quote}_\tau(t)))$.*

Proof. Implicit in the proof of Theorem 8.14. □

Completeness of the transformation (regarding the computation of constructor terms) requires some additional conditions.

Theorem 8.16 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a left-linear, confluent TRS. Let φ be a regular E-strategy map such that $\varphi \in CM_\mathcal{R}$, and $\mathcal{R}$ is φ-terminating. Let $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ be such that $sort(t) = \tau$ and $\delta \in \mathcal{T}(\mathcal{C})$. Let $\mathcal{R}' = \mathsf{E}_\tau(\mathcal{R})$ and $\varphi' = Emap_\tau(\varphi)$. If $t \to^*_\mathcal{R} \delta$, then $\delta \in eval_{\varphi'}(\texttt{unquote}_\tau(\texttt{quote}_\tau(t)))$.*

Proof. As in the proof of Theorem 8.14, we only need to prove that, whenever $t \to^*_\mathcal{R} \delta$, then $\delta' \in eval_{\varphi'}(\texttt{quote}_{sort(t)}(t)))$, where δ' is δ with all constructor symbols c renamed to their alias c'. We proceed by induction on δ.

- If δ is a constant b, then, by taking $\mathcal{B} = \{b\}$, we have that $\mu^\mathcal{B}_\mathcal{R} = \mu^{can}_\mathcal{R}$, and by Theorem 8.10, $b \in eval_\varphi(t)$, i.e.,

$$\langle \varphi(t), \Lambda\rangle \to^*_\varphi \langle b_{nil}, \Lambda\rangle$$

In fact, we can use such $\to_\varphi$-sequence for building a corresponding $\to_{\varphi'}$ sequence issued from $\varphi'(\texttt{unquote}_\tau(\texttt{quote}_\tau(t)))$, where $\tau = sort(t)$ (note that, since $t \in$

$\mathcal{T}(\mathcal{F},\mathcal{X})$, $\varphi(t)=\varphi'(t)$):

$$\begin{array}{rcl}
\langle \texttt{quote}_{\texttt{(1 0)}}(\varphi(t)),\Lambda\rangle & \to_{\varphi'} & \texttt{quote}_{\texttt{(0)}}(\varphi(t)),1\rangle \\
 & \to_{\varphi'} & \\
 & \vdots & \\
 & \to_{\varphi'} & \langle \texttt{quote}_{\texttt{(0)}}(b_{nil}),1\rangle \\
 & \to_{\varphi'} & \langle b'_{nil},\Lambda\rangle
\end{array}$$

- If δ is not a constant, then let $\delta = c(\delta_1,\ldots,\delta_k)$. Again, by φ-termination, $eval_\varphi(t)$ is not empty and, by Theorem 8.7 every $s \in eval_\varphi(t)$ is a μ^φ-normal form which, by Theorem 8 in [Lucas, 1998a] is also a head-normal form and, by confluence of $\mathcal{R}$, it is rooted by a symbol c. Then, let $s = c(s_1,\ldots,s_k)$. We have:

  $$\langle t,\Lambda\rangle \to^!_\varphi \langle s',\Lambda\rangle$$

  where $erase(s') = s$. By confluence of $\mathcal{R}$, $s_i \to^* \delta_i$ for $1 \le i \le k$, and by the induction hypothesis, $\delta'_i \in eval_{\varphi'}(\texttt{quote}_{sort(s_i)}(s_i)))$ for $1 \le i \le k$. By reasoning as in the base case, this derivation induces a corresponding $\to_{\varphi'}$-derivation:

  $$\begin{array}{rl}
\langle \texttt{quote}_{\texttt{(1 0)}}(\varphi(t)),\Lambda\rangle & \to_{\varphi'} \texttt{quote}_{\texttt{(0)}}(\varphi(t)),1\rangle \\
 & \to_{\varphi'} \\
 & \;\vdots \\
 & \to_{\varphi'} \langle \texttt{quote}_{\texttt{(0)}}(c_{nil}(s'_1,\ldots,s'_k)),1\rangle \\
 & \to_{\varphi'} \langle \texttt{quote}_{\texttt{(0)}}(c_{nil}(s'_1,\ldots,s'_k)),\Lambda\rangle \\
 & \to_{\varphi'} \langle c'_{(1\ 2\cdots\ k)}(\texttt{quote}_{\texttt{(1 0)}}(s'_1),\ldots,\texttt{quote}_{\texttt{(1 0)}}(s'_k))),1\rangle \\
 & \to^*_{\varphi'} \langle c'_{nil}(\delta''_1,\ldots,\delta''_k)),\Lambda\rangle
\end{array}$$

  where $erase(c'_{nil}(\delta''_1,\ldots,\delta''_k)) = c'(erase(\delta''_1),\ldots,erase(\delta''_k)) = \delta'$.

□

In contrast to Theorem 8.10, note that we can now use any E-strategy map $\varphi \in CM_{\mathcal{R}}$.

Example 8.17 *The following* OBJ *program:*

```
obj EXAMPLE-STR is
  sorts Nat LNat .
  ops 0 0' : -> Nat .
  ops s s' : Nat -> Nat [strat (1)] .
  ops nil nil' : -> LNat .
  op cons : Nat LNat -> LNat [strat (1)] .
  op cons' : Nat LNat -> LNat [strat (1 2)] .
  op fcons : Nat LNat -> LNat [strat (1 2 0)] .
```

```
    op from : Nat -> LNat [strat (1 0)] .
    op sel : Nat LNat -> Nat [strat (1 2 0)] .
    op first : Nat LNat -> LNat [strat (1 2 0)] .
    ops quote unquote : Nat -> Nat [strat (1 0)] .
    ops quote' unquote' : LNat -> LNat [strat (1 0)] .
    vars X Y : Nat .
    var Z : LNat .
    eq sel(s(X),cons(Y,Z)) = sel(X,Z) .
    eq sel(0,cons(X,Z)) = X .
    eq first(0,Z) = nil .
    eq first(s(X),cons(Y,Z)) = cons(Y,first(X,Z)) .
    eq from(X) = cons(X,from(s(X))) .
    eq quote(0) = 0' .
    eq quote'(cons(X,Z)) = cons'(quote(X),quote'(Z)) .
    eq quote'(nil) = nil' .
    eq quote(s(X)) = s'(quote(X)) .
    eq unquote(0') = 0 .
    eq unquote(s'(X)) = s(unquote(X)) .
    eq unquote'(nil') = nil .
    eq unquote'(cons'(X,Z)) = fcons(unquote(X),unquote'(Z)) .
    eq fcons(X,Z) = cons(X,Z) .
  endo
```

is the transformed version of the OBJ *program in Example 8.1. Now, the evaluation of term* `unquote'(quote'(first(s(0),from(0))))` *yields:*

```
==========================================
obj EXAMPLE-STR
==========================================
reduce in EXAMPLE-STR : unquote'(quote'(first(s(0), from(0)))) .
rewrites: 11 in -10ms cpu (0ms real) (~ rewrites/second)
result LNat: cons(0, nil)
```

Note the difference between 'unquoting' rules for symbols `s'` and `cons'`. The unquoting of `cons'` is indirect; the obvious short-cut:

```
unquote(cons'(X,Z)) = cons(unquote(X),unquote'(Z))
```

in the program in Example 8.17 does not work: the reason is that, after applying this rule, *the second argument of* `cons` *remains non-replacing.* For instance, by using such a rule (instead of the last two rules of the program in Example 8.17) the evaluation of `unquote'(quote'(first(s(0),from(0))))` would yield

```
cons(0,unquote'(nil'))
```

This is solved by introducing the intermediate defined symbol `fcons` which first evaluates its arguments (thus performing the renaming) and then reduces to term rooted by `cons`. In this sense, the *explicit* annotation `(1 2 0)` is also crucial for symbol `fcons`; otherwise, the interpreter could associate a default strategy which does not permit the renamings (for instance, OBJ3 associates the strategy `(0 1 2 0)` to `fcons`; with this default annotation, we would still obtain `cons(0,unquote'(nil'))` instead of the desired value).

Unfortunately, the previous transformation does not preserve termination of the original program (proved in Example 8.6).

Example 8.18 *The evaluation of* $t =$ `quote'(from(0))` *yields:*

```
==========================================
obj EXAMPLE-STR
==========================================
reduce in EXAMPLE-STR : quote'(from(0)) .

ADVISORY: closing open files.
Debug(1)> Bye.
```

where we were forced to abort the non-terminating execution. Again, the problem is that the evaluation of t*, i.e., the evaluation of*

$$\varphi(t) = \mathtt{quote'}_{(1\ \ 0)}\mathtt{(from}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{))}$$

does not terminate:

$$\begin{array}{l}
\langle \mathtt{quote'}_{(1\ \ 0)}\mathtt{(from}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{))}, \Lambda\rangle \\
\quad \to_\varphi \langle \mathtt{quote'}_{(0)}\mathtt{(from}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{))}, 1\rangle \\
\quad \to_\varphi \langle \mathtt{quote'}_{(0)}\mathtt{(from}_{(0)}\mathtt{(0}_{nil}\mathtt{))}, 1.1\rangle \\
\quad \to_\varphi \langle \mathtt{quote'}_{(0)}\mathtt{(}\underline{\mathtt{from}_{(0)}\mathtt{(0}_{nil}\mathtt{)}}\mathtt{)}, 1\rangle \\
\quad \to_\varphi \langle \mathtt{quote'}_{(0)}\mathtt{(}\overline{\mathtt{cons}_{(1)}\mathtt{(0}_{nil}\mathtt{,from}_{(1\ \ 0)}\mathtt{(s}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{)))}}, 1\rangle \\
\quad \to_\varphi \langle \mathtt{quote'}_{(0)}\mathtt{(cons}_{nil}\mathtt{(0}_{nil}\mathtt{,from}_{(1\ \ 0)}\mathtt{(s}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{)))}, 1.1\rangle \\
\quad \to_\varphi \langle \mathtt{quote'}_{(0)}\mathtt{(cons}_{nil}\mathtt{(0}_{nil}\mathtt{,from}_{(1\ \ 0)}\mathtt{(s}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{)))}, 1\rangle \\
\quad \to_\varphi \langle \underline{\mathtt{quote'}_{(0)}\mathtt{(cons}_{nil}\mathtt{(0}_{nil}\mathtt{,from}_{(1\ \ 0)}\mathtt{(s}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{)))}}, \Lambda\rangle \\
\quad \to_\varphi \langle \overline{\mathtt{cons'}_{(1\ \ 2)}\mathtt{(quote}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{),quote'}_{(1\ \ 0)}\mathtt{(from}_{(1\ \ 0)}\mathtt{(s}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{))))}}, \Lambda\rangle \\
\quad \to_\varphi^+ \langle \mathtt{cons'}_{(2)}\mathtt{(0'}_{nil}\mathtt{,quote'}_{(1\ \ 0)}\mathtt{(from}_{(1\ \ 0)}\mathtt{(s}_{(1\ \ 0)}\mathtt{(0}_{nil}\mathtt{))))}, 2\rangle \\
\quad \to_\varphi \cdots
\end{array}$$

8.4.1 Preserving Completeness and Termination

Example 8.18 shows that the annotation $\varphi'(\mathtt{quote}_\tau) =$ `(1 0)` may cause non-termination. We can try to avoid this problem by restricting the E-strategy for $\mathtt{quote}_\tau$ to `(0)`.

In this case, however, we need to add new rules to enable the evaluation in some alternative way. In [Lucas, 1997], a program transformation which is able to achieve the desired effect was introduced. In the following, by an outermost (occurrence of a) defined symbol in a term t, we mean a defined symbol which has constructor symbols above it in t. The new constructors are now introduced in computations by the contraction of redexes of outermost defined symbols f. Thus, we add new defined symbols f', which will show up when these outermost defined f symbols emerge, and new rules for defining these symbols. The new rules $f'(l_1, \ldots, l_k) \to r'$ come from the original ones $f(l_1, \ldots, l_k) \to r$ as follows: occurrences of outermost defined symbols g in r are renamed in r' as g'; occurrences of constructor symbols c above those g in r are renamed in r' as c'; occurrences of variables x in r which only have constructor symbols above them are marked as $\mathtt{quote}_{sort(x)}(x)$ in r'. Now (in contrast to the previous transformation) symbols $\mathtt{quote}_\tau$ are also intended to rename outermost defined symbols f (of sort τ) as their alias f' (of the same sort). In order to simplify the transformation, it is tempting not to take into account the number of extra rules which are added to the transformed TRS and introduce new rules $f'(l_1, \ldots, l_k) \to r'$ for each defined symbol f. Unfortunately, this may unnecessarily cause non-termination.

Example 8.19 *Consider the rule*

`from(x)` $\to$ `cons(x,from(s(x)))`

of our running example. We then introduce the rule:

`from'(x)` $\to$ `cons'(quote(x),from'(s(x)))`

For example, in the evaluation of $t =$ `first(s(0),from(0))` *in Example 8.11, the symbol* `from` *does* not *emerge as outermost: roughly speaking, the only possibility is that either the right-hand side of a rule defining* `first` *contains a variable of sort* `LNat` *having only constructor symbols above, or that* `from` *is outermost in some right-hand side of the rule. This does not happen in our example. Thus, we do not need the rule which would introduce* non-termination *since reductions are allowed on both arguments of* `cons'`*. For this reason, we perform a more accurate analysis of the required additional rules by carefully identifying the outermost defined symbols that can emerge during the evaluation of a given expression.*

The following notations are auxiliary [Lucas, 1997]: Given $f : \tau_1 \times \ldots \times \tau_k \to \tau$, the sorts of arguments of f are gathered in the set[5] $\mathit{sortarg}(f) = \{\tau_1, \ldots, \tau_k\}$. Given a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,

- $\mathit{CVar}(t) = \{x \in \mathcal{V}ar(t) \mid \exists p \in \mathcal{P}os(t),\ t|_p = x \wedge \forall\, q < p,\ q \in \mathcal{P}os_{\mathcal{C}}(t)\}$ is the set of *constructor variables* of t, i.e., variables of t having a maximal proper prefix

[5] Here, we disregard from the ordering of the argument sorts (i.e., we do not use a list of sorts) since it is not important for our purposes.

which only points to constructor symbols. We also use $\mathcal{C}_\tau = \{c \in \mathcal{C} \mid \mathit{sort}(c) = \tau\}$.

- The set of possible sorts for symbols arising by instantiation of a constructor variable x is $\mathit{CVSort}(\mathit{sort}(x))$ where, given a sort τ,

  $\mathit{CVSort}(\tau) = \{\tau\} \cup \bigcup_{c \in \mathcal{C}_\tau, \tau \in \mathit{sortarg}(c)} \mathit{CVSort}(\tau)$

- $\mathit{Vouter}(t) = \bigcup_{x \in \mathit{CVar}(t)} \{f \in \mathcal{D} \mid \mathit{sort}(f) \in \mathit{CVSort}(\mathit{sort}(x))\}$ are the defined symbols which can root the subterms introduced in t by instantiation of constructor variables of t (that is, which emerge as outermost in t after instantiation).

Example 8.20 *Consider the term* $t = \texttt{cons(y,first(x,z))}$, *where* $\mathit{sort}(\texttt{y}) = \texttt{Nat}$ *and* $\texttt{first} : \texttt{Nat} \times \texttt{LNat} \to \texttt{LNat}$. *Then,*

- $\mathit{CVar}(t) = \{\texttt{y}\}$*; note that* $\mathit{sort}(\texttt{y}) = \texttt{Nat}$.

- $\mathit{CVSort}(\texttt{Nat}) = \{\texttt{Nat}\}$ *and*

$$\begin{array}{rcl} \mathit{CVSort}(\texttt{LNat}) & = & \{\texttt{LNat}\} \cup \bigcup_{c \in \{\texttt{nil},\texttt{cons}\}, \tau \in \mathit{sortarg}(c)} \mathit{CVSort}(\tau) \\ & = & \{\texttt{LNat}\} \cup \mathit{CVSort}(\texttt{Nat}) \cup \mathit{CVSort}(\texttt{LNat}) \\ & = & \{\texttt{LNat}, \texttt{Nat}\} \end{array}$$

- $\mathit{Vouter}(t) = \{f \in \mathcal{D} \mid \mathit{sort}(f) \in \mathit{CVSort}(\mathit{sort}(\texttt{y}))\} = \{f \in \mathcal{D} \mid \mathit{sort}(f) \in \{\texttt{Nat}\}\} = \{\texttt{sel}\}$.

Given a TRS $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ and $f \in \mathcal{D}$,

- $\mathit{outrhs}_\mathcal{R}(f) \subseteq \mathcal{D}$ contains the outermost defined symbols in the *rhs*'s of the f-rules: $\mathit{outrhs}_\mathcal{R}(f) = \cup_{f(l_1,\ldots,l_k) \to r \in R} \{\mathit{root}(r,p) \mid p \in \mathcal{P}os_\mathcal{D}(r) \wedge \forall\, q < p.\ q \in \mathcal{P}os_\mathcal{C}(r)\}$.

- $\mathit{Vrhs}_\mathcal{R}(f) \subseteq \mathcal{D}$ is the set of outermost defined symbols which can appear by instantiation of constructor variables in the *rhs*'s of the f-rules: $\mathit{Vrhs}_\mathcal{R}(f) = \cup_{f(l_1,\ldots,l_k) \to r \in R} \mathit{Vouter}(r)$.

- $\mathit{newouter}_\mathcal{R}(f) = \mathit{outrhs}_\mathcal{R}(f) \cup \mathit{Vrhs}_\mathcal{R}(f)$.

Example 8.21 *Consider the TRS $\mathcal{R}$ in Example 8.3 (assume the sorts as given in the signature of the original* OBJ *program in Example 8.1). We have:*

- $\mathit{outrhs}_\mathcal{R}(\texttt{sel}) = \{\texttt{sel}\}$ *and* $\mathit{outrhs}_\mathcal{R}(\texttt{first}) = \{\texttt{first}\}$. *Let us develop the first transformation: the rules defining* `sel` *are*

  $\texttt{sel(0,cons(x,z))} \to \texttt{x}$ *and* $\texttt{sel(s(x),cons(y,z))} \to \texttt{sel(x,z)}$.

The rhs of the first rule is a variable; hence it does not contribute to $outrhs_{\mathcal{R}}(\mathtt{sel})$. *On the other hand, the only outermost defined symbol in the rhs of the second rule is* `sel`; *hence,* $outrhs_{\mathcal{R}}(\mathtt{sel}) = \{\mathtt{sel}\}$.

- $Vrhs_{\mathcal{R}}(\mathtt{sel}) = Vouter(\mathtt{x}) \cup Vouter(\mathtt{sel(x,z)}) = Vouter(\mathtt{x}) = \{\mathtt{sel}\}$ *(note that* $sort(\mathtt{x}) = \mathtt{Nat}$*), and according to Example 8.20:*

$$\begin{array}{rcl} Vrhs_{\mathcal{R}}(\mathtt{first}) & = & Vouter(\mathtt{nil}) \cup Vouter(\mathtt{cons(y,first(x,z))}) \\ & = & Vouter(\mathtt{cons(y,first(x,z))}) \\ & = & \{\mathtt{sel}\} \end{array}$$

- *Finally,* $newouter_{\mathcal{R}}(\mathtt{sel}) = outrhs_{\mathcal{R}}(\mathtt{sel}) \cup Vrhs_{\mathcal{R}}(\mathtt{sel}) = \{\mathtt{sel}\}$, *and* $newouter_{\mathcal{R}}(\mathtt{first}) = outrhs_{\mathcal{R}}(\mathtt{first}) \cup Vrhs_{\mathcal{R}}(\mathtt{first}) = \{\mathtt{first}, \mathtt{sel}\}$.

In contrast to transformation E_τ, here we are mainly interested in evaluating the term $f(t_1, \ldots, t_k)$ for a given defined symbol $f \in \mathcal{D}$. Given $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ and $f \in \mathcal{D}$, we let $\mathcal{D}^f_{\mathcal{R}} \subseteq \mathcal{D}$ be:

$$\mathcal{D}^f_{\mathcal{R}} = \{f\} \cup \bigcup_{g \in newouter_{\mathcal{R}}(f)} \mathcal{D}^g_{\mathcal{R}}$$

$\mathcal{D}^f_{\mathcal{R}}$ contains the outermost defined symbols which arise when a (well sorted) f-rooted term $f(t_1, \ldots, t_k)$ is arbitrarily rewritten. In practice, since the definition of $\mathcal{D}^f_{\mathcal{R}}$ has induced recursivity, we must consider all possible equations

$$\begin{array}{rcl} \mathcal{D}^{f_1}_{\mathcal{R}} & = & \{f_1\} \cup \bigcup_{g \in newouter_{\mathcal{R}}(f_1)} \mathcal{D}^g_{\mathcal{R}} \\ & \vdots & \\ \mathcal{D}^{f_n}_{\mathcal{R}} & = & \{f_n\} \cup \bigcup_{g \in newouter_{\mathcal{R}}(f_n)} \mathcal{D}^g_{\mathcal{R}} \end{array}$$

(where $f_1 = f$ and $f_2, \ldots, f_n$ are all the defined symbols successively occurring in $newouter_{\mathcal{R}}(f_1) \cup \cdots \cup newouter_{\mathcal{R}}(f_n)$), and compute the (least) solutions $\mathcal{D}^{f_1}_{\mathcal{R}}, \ldots, \mathcal{D}^{f_n}_{\mathcal{R}}$ by using fixpoint techniques (see [Lucas, 1997, 1998b]).

Example 8.22 *(Continuing Example 8.21) Since* $newouter_{\mathcal{R}}(\mathtt{sel}) = \{\mathtt{sel}\}$ *and* $newouter_{\mathcal{R}}(\mathtt{first}) = \{\mathtt{first}, \mathtt{sel}\}$, *we have the system:*

$$\begin{array}{rcl} \mathcal{D}^{\mathtt{first}}_{\mathcal{R}} & = & \{\mathtt{first}\} \cup \mathcal{D}^{\mathtt{sel}}_{\mathcal{R}} \cup \mathcal{D}^{\mathtt{first}}_{\mathcal{R}} \\ \mathcal{D}^{\mathtt{sel}}_{\mathcal{R}} & = & \{\mathtt{sel}\} \cup \mathcal{D}^{\mathtt{sel}}_{\mathcal{R}} \end{array}$$

which has a simple solution: $\mathcal{D}^{\mathtt{first}}_{\mathcal{R}} = \{\mathtt{first}, \mathtt{sel}\}$ *and* $\mathcal{D}^{\mathtt{sel}}_{\mathcal{R}} = \{\mathtt{sel}\}$. *Note that* $\mathtt{from} \notin \mathcal{D}^{\mathtt{first}}_{\mathcal{R}}$ *and* $\mathtt{from} \notin \mathcal{D}^{\mathtt{sel}}_{\mathcal{R}}$

The set $ev^f(\mathcal{F}, \mathcal{X})$ of terms is given as follows: (1) $\mathcal{X} \subseteq ev^f(\mathcal{F}, \mathcal{X})$, (2) $g(\overline{t}) \in ev^f(\mathcal{F}, \mathcal{X})$ if $g \in \mathcal{D}^f_{\mathcal{R}}$, and (3) $c(t_1, \ldots, t_k) \in ev^f(\mathcal{F}, \mathcal{X})$ if $c \in \mathcal{C}^*_{sort(f)}$ and $t_1, \ldots, t_k \in$

$ev^f(\mathcal{F}, \mathcal{X})$. If we do not require (1) (and change the inductive case (3) to be $c(t_1, \ldots, t_k) \in gev^f(\mathcal{F}, \mathcal{X})$ if $c \in \mathcal{C}^*_{sort(f)}$ and $t_1, \ldots, t_k \in gev^f(\mathcal{F}, \mathcal{X})$), then we are defining the set $gev^f(\mathcal{F}, \mathcal{X})$. Roughly speaking, if we rewrite a term $t = g(\overline{t})$ for some $g \in \mathcal{D}^f_{\mathcal{R}}$, then every possible reduct of t belongs to $ev^f(\mathcal{F}, \mathcal{X})$. If t is ground, then we only need to consider $gev^f(\mathcal{F}, \mathcal{X})$.

We now formalize the program transformation. First, we give the new signature. Note that the transformation is parametric w.r.t. a TRS $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ and a defined symbol $f \in \mathcal{D}$.

Definition 8.23 *Given a TRS $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ and $f \in \mathcal{D}$, we let $\mathcal{F}^f = \mathcal{F} \uplus \mathcal{C}' \uplus \mathcal{D}' \uplus$ Quote $\uplus$ Unquote, where: $c' \in \mathcal{C}' \Leftrightarrow c \in \mathcal{C}^*_{sort(f)} \wedge ar(c') = ar(c)$ and $g' \in \mathcal{D}' \Leftrightarrow g \in \mathcal{D}^f_{\mathcal{R}} \wedge ar(g') = ar(g)$. Quote and Unquote are defined as before.*

The following transformation introduces rules to deal with the different symbols that we consider, according to the informal description above.

Definition 8.24 (Transformation V) *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS and $f \in \mathcal{D}$. We let $\mathsf{V}^f(\mathcal{R}) = (\mathcal{F}^f, R \cup S \cup Q \cup U)$, where:*

- $S = \{g'(\overline{l}) \to \kappa^f(r) \mid g(\overline{l}) \to r \in R \wedge g \in \mathcal{D}^f_{\mathcal{R}}\}$, *where*

  $\kappa^f(x) = \mathtt{quote}_{sort(x)}(x)$, *for* $x \in \mathcal{X}$, $\kappa^f(g(\overline{t})) = g'(\overline{t})$ *if* $g \in \mathcal{D}^f_{\mathcal{R}}$, *and* $\kappa^f(c(\overline{t})) = c'(\kappa^f(\overline{t}))$ *if* $c \in \mathcal{C}$.

- *Rules in Q define symbols $\mathtt{quote}_\tau$, in order to rename external constructors $c \in \mathcal{C}^*_\tau$ (where $\tau = sort(f)$) to constructors $c' \in \mathcal{C}'$ where $c, c' : \tau_1 \times \cdots \times \tau_k \to \tau$, and outermost application of $g \in \mathcal{D}^f_{\mathcal{R}}$ to outermost applications of the corresponding $g' \in \mathcal{D}'$.*

$$\begin{aligned} Q = \; & \{\mathtt{quote}_\tau(c(x_1, \ldots, x_k)) \to c'(\mathtt{quote}_{\tau_1}(x_1), \ldots, \mathtt{quote}_{\tau_k}(x_k)) \mid c \in \mathcal{C}^*_\tau\} \\ & \cup \{\mathtt{quote}_{sort(g)}(g(x_1, \ldots, x_k)) \to g'(x_1, \ldots, x_k) \mid g \in \mathcal{D}^f_{\mathcal{R}}\} \end{aligned}$$

- *Rules in U define symbols in Unquote exactly as in the previous transformation E_τ.*

Given an E-strategy map φ, we define the new E-strategy map φ'; we define $\varphi' = Emap^f(\varphi)$ as follows: $\varphi'(g) = \varphi(f)$ if $g \in \mathcal{D}$, $\varphi(g') = \varphi(g)$ if $g \in \mathcal{D}^f_{\mathcal{R}}$, $\varphi'(c) = \varphi(c)$ if $c \in \mathcal{C}$, and $\varphi(c') = (1 \; \cdots \; ar(c'))$ if $c' \in \mathcal{C}'$, $\varphi(\mathtt{quote}_\tau) = (0)$ and $\varphi(\mathtt{unquote}_\tau) = (1 \; 0)$ for each sort τ; and $\varphi(f_c) = (1 \; \cdots \; ar(c) \; 0)$ for each $c \in \mathcal{C}^*_{sort(f)}$ such that $\mu^\varphi(c) \neq \{1, \ldots, ar(c)\}$.

For the new transformation, we have similar results as for the simpler one.

Lemma 8.25 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS. Let φ be a positive E-strategy map. Let $\mathcal{R}' = \mathsf{V}^f(\mathcal{R})$ for a given $f \in \mathcal{D}$ and $\varphi' = Emap^f(\varphi)$. Let $\delta \in \mathcal{T}(\mathcal{C})$ and δ'*

be δ with all constructor symbols c renamed to their alias c'. For all term s of sort τ, $\delta \in eval_{\varphi'}(\mathtt{unquote}_\tau(s))$ if and only if $\delta' \in eval_{\varphi'}(s)$.

Proof. Since the strategy annotation for $\mathtt{unquote}_\tau$ is the same for both $Emap^f(\varphi)$ and $\varphi' = Emap_{sort(f)}(\varphi)$, the proof is analogous to that of Lemma 8.13. □

Theorem 8.26 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS. Let φ be a regular E-strategy map. Let $f \in \mathcal{D}$, $t \in ev^f(\mathcal{F}, \mathcal{X})$, and $\delta \in \mathcal{T}(\mathcal{C})$. Let $\mathcal{R}' = \mathsf{V}^f(\mathcal{R})$ and $\varphi' = Emap^f(\varphi)$. If $\delta \in eval_{\varphi'}(\mathtt{unquote}_{sort(t)}(\mathtt{quote}_{sort(t)}(t)))$, then $t \to^*_{\mathcal{R}} \delta$.*

Proof. (Sketch) Similarly to the proof of Theorem 8.14 (now using Lemma 8.25), we only need to prove that $t \to^*_{\mathcal{R}} \delta$ if $\delta' \in eval_{\varphi'}(\mathtt{quote}_\tau(t))$. We proceed by induction on δ. If δ is a constant b, then either $t = b$ or $root(t) = g \in \mathcal{F}^f_{\mathcal{R}}$, and the evaluation of $\mathtt{quote}_\tau(t)$ according to φ' is (note that, since $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\varphi(t) = \varphi'(t)$; we left the indexing sort of symbols $\mathtt{quote}$ unspecified):

$$\begin{array}{rll}
\langle \mathtt{quote}_{\mathtt{(0)}}(g_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k))), \Lambda\rangle & \to_{\varphi'} \langle g'_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k)), \Lambda\rangle & (*) \\
& \to_{\varphi'} & \\
& \vdots & \\
& \to_{\varphi'} h'_{\varphi(h)}(s_1, \ldots, s_m), \Lambda\rangle & \\
& \to_{\varphi'} \langle \delta''_{nil}, \Lambda\rangle & (**)
\end{array}$$

where $h' \in \mathcal{F}'$ is the renamed version of a defined symbol $h \in \mathcal{F}$. Note that the evaluation steps between $(*)$ and $(**)$ only involve rules in $\mathcal{R}'$ that either are in R (if the rewriting steps are performed below position 1) or in S (if the rewriting steps are performed at position 1). Thus, since the rules in S are modified versions of rules in $\mathcal{R}$, we can write $t \to^*_{\mathcal{R}} b$.

If $\delta = c(\delta_1, \ldots, \delta_k)$, then either $t = c(t_1, \ldots, t_k)$ or $root(t) = g \in \mathcal{F}^f_{\mathcal{R}}$. In the first case, we have:

$$\begin{array}{l}
\langle \mathtt{quote}_{\mathtt{(0)}}(c_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k))), \Lambda\rangle \\
\quad \to_{\varphi'} \langle c'_{(1\ 2 \cdots k)}(\mathtt{quote}_{\mathtt{(0)}}(\varphi(t_1)), \ldots, \mathtt{quote}_{\mathtt{(0)}}(\varphi(t_k))), \Lambda\rangle \\
\quad \to_{\varphi'} \\
\quad \vdots \\
\quad \to_{\varphi'} c'_{nil}(\delta''_1, \ldots, \delta''_k), \Lambda\rangle
\end{array}$$

where $erase(\delta_i'') = \delta_i'$ for $1 \leq i \leq k$, and the conclusion easily follows by the induction hypothesis. In the second case, let $t = g(t_1, \ldots, t_k)$. We have,

$$\begin{array}{ll}\langle \mathtt{quote}_{(0)}(g_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k))), \Lambda\rangle & \\ \quad \to_{\varphi'} \langle g'_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k)), \Lambda\rangle & (*) \\ \quad \to_{\varphi'} & \\ \quad\ \ \vdots & \\ \quad \to_{\varphi'} h'_{\varphi(h)}(s_1, \ldots, s_m), \Lambda\rangle & \\ \quad \to_{\varphi'} \langle s, \Lambda\rangle & (**)\end{array}$$

where $h' \in \mathcal{F}'$, and $s = C[s_1, \ldots, s_n]$ is such that $erase(C[\,])$ is the maximal constructor prefix of s which is shared with δ' (i.e., there are $\delta_1'', \ldots, \delta_n''$ such that $\delta' = erase(C[\delta_1'', \ldots, \delta_n''])$. Thus, we can say again that $t \to_{\mathcal{R}}^* s$. Note also that for all $1 \leq i \leq k$, either $s_i' = \mathtt{quote}_{(0)}(s_i'')$, or $erase(s_i) = h'(u_1, \ldots, u_m)$ for $h' \in \mathcal{F}'$, and $u_j \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ for $1 \leq j \leq m$ (i.e., $h(u_1, \ldots, u_m)$ belongs to the 'base' of the inductive set $ev^f(\mathcal{F}, \mathcal{X})$). Moreover, every constructor symbol c' in $C[\,]$ is labelled with $\varphi(c') = \{1, \ldots, ar(c')\}$. Thus, by using the induction hypothesis (for quote-rooted terms s_i is obvious; for the other terms, it follows from the fact that terms $h(u_1, \ldots, u_m)$ in the 'base' of $ev^f(\mathcal{F}, \mathcal{X})$ are transformed into $h'(u_1, \ldots, u_m)$ by a single rewriting step issued on $\mathtt{quote}(h(u_1, \ldots, u_m))$), we have that $erase(s_i) \to_{\mathcal{R}}^* \delta_i$ for $1 \leq i \leq n$ and the conclusion follows. □

Theorem 8.27 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS. Let φ be a regular E-strategy map. Let $f \in \mathcal{D}$, $t \in ev^f(\mathcal{F}, \mathcal{X})$, and $\delta \in \mathcal{T}(\mathcal{C})$. Let $\mathcal{R}' = \mathsf{V}^f(\mathcal{R})$ and $\varphi' = Emap^f(\varphi)$. If $\delta \in eval_\varphi(t)$, then $\delta \in eval_{\varphi'}(\mathtt{unquote}_{sort(t)}(\mathtt{quote}_{sort(t)}(t)))$.*

Proof. Similar to the proof of Theorem 8.26. □

Theorem 8.28 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a left-linear, confluent TRS. Let φ be a regular E-strategy map such that $\varphi \in CM_{\mathcal{R}}$ and $\mathcal{R}$ is φ-terminating. Let $f \in \mathcal{D}$, $t \in ev^f(\mathcal{F}, \mathcal{X})$, and $\delta \in \mathcal{T}(\mathcal{C})$. Let $\mathcal{R}' = \mathsf{V}^f(\mathcal{R})$ and $\varphi' = Emap^f(\varphi)$. If $t \to_{\mathcal{R}}^* \delta$, then $\delta \in eval_{\varphi'}(\mathtt{unquote}_{sort(t)}(\mathtt{quote}_{sort(t)}(t)))$.*

Proof. (Sketch) We proceed by induction on the structure of terms in $ev^f(\mathcal{F}, \mathcal{X})$ (again, we use Lemma 8.25). For the base case, we consider terms $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $root(t) \in \mathcal{D}_{\mathcal{R}}^f$ (if $t \in \mathcal{X}$, it is immediate). Let $t = g(\overline{t})$ for $g \in \mathcal{F}_{\mathcal{R}}^f$. Now we proceed by induction on δ.

- If δ is a constant or a variable b, then by taking $\mathcal{B} = \{b\}$, we have that $\mu_{\mathcal{R}}^{\mathcal{B}} = \mu_{\mathcal{R}}^{can}$

and by Theorem 8.10, $b \in eval_\varphi(t)$, i.e.,

$$\begin{array}{rl} \langle g_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k)), \Lambda\rangle & \to_\varphi \\ & \vdots \\ & \to_\varphi \quad \langle h_{\varphi(h)}(s_1, \ldots, s_k), \Lambda\rangle \\ & \to_\varphi \quad \langle b_{nil}, \Lambda\rangle \end{array}$$

In fact, we can use such $\to_\varphi$-sequence for building a corresponding $\to_{\varphi'}$ sequence issued from $\varphi'(\mathtt{quote}_{sort(t)}(t))$:

$$\begin{array}{rl} \langle \mathtt{quote}_{\mathtt{(0)}}(g_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k))), \Lambda\rangle & \to_{\varphi'} \quad \langle g'_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k)), \Lambda\rangle \\ & \to_{\varphi'} \\ & \vdots \\ & \to_{\varphi'} \quad h'_{\varphi(h)}(s_1, \ldots, s_k), \Lambda\rangle \\ & \to_{\varphi'} \quad \langle b'_{nil}, \Lambda\rangle \end{array}$$

Thus, $b' \in eval_{\varphi'}(\mathtt{quote}_{sort(t)}(t))$.

- If δ is not a constant, then let $c = root(\delta)$. Again, by φ-termination, $eval_\varphi(t)$ is not empty, and by Theorem 8.7, every $s \in eval_\varphi(t)$ is a μ^φ-normal form which, by Theorem 8 in [Lucas, 1998a], is a head-normal form and by confluence of $\mathcal{R}$, it is rooted by a symbol c. Then, we have an initial prefix of the φ-evaluation sequence for t which is as follows:

$$\langle t, \Lambda\rangle \to_\varphi^* \cdots \to_\varphi \langle h(s_1, \ldots, s_k), \Lambda\rangle \to_\varphi \langle C[u_1, \ldots, u_n], \Lambda\rangle$$

where $h \in \mathcal{D}$, $C[\,,\ldots,\,] \in \mathcal{T}(\mathcal{C}^*_{sort(f)} \cup \{\Box\}, \mathcal{X})$, and $root(u_i) \in \mathcal{D} \cup \mathcal{X}$ for $1 \leq i \leq n$. By confluence, we can write $\delta = C[\delta_1, \ldots, \delta_n]$ and $u_i \to^* \delta_i$ for $1 \leq i \leq n$. By reasoning as in the base case, this derivation induces a corresponding $\to_{\varphi'}$-derivation:

$$\begin{array}{rl} \langle \mathtt{quote}_{\mathtt{(0)}}(g_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k))), \Lambda\rangle & \to_{\varphi'} \quad \langle g'_{\varphi(g)}(\varphi(t_1), \ldots, \varphi(t_k)), \Lambda\rangle \\ & \to_{\varphi'} \\ & \vdots \\ & \to_{\varphi'} \quad h'_{\varphi(h)}(s_1, \ldots, s_k), \Lambda\rangle \\ & \to_{\varphi'} \quad \langle C'[u'_1, \ldots, u'_n], \Lambda\rangle \end{array}$$

where $root(C'[u'_1, \ldots, u'_n]) = c'_{(1\ 2\cdots ar(c'))}$, and, for all $1 \leq i \leq n$, either $u'_i = \mathtt{quote}_{\mathtt{(0)}}(u_i)$ or $root(erase(u'_i)) = f' \in \mathcal{D}'$ (and $f = root(erase(u_i))$). Thus, we can apply the (second) induction hypothesis. Therefore, $\kappa^f(\delta_i) \in eval_{\varphi'}(\mathtt{quote}_{sort(u_i)}(erase(u_i))$ for $1 \leq i \leq n$. Now, we easily conclude that $\delta \in eval_{\varphi'}(\mathtt{quote}_{sort(t)}(t))$ (note that $\langle \mathtt{quote}_{(0)}(\varphi'(u_i)), \Lambda\rangle \to_{\varphi'} \langle u'_i, \Lambda\rangle$).

If $t = c(t_1, \ldots, t_k)$ for some constructor symbol c, then, by confluence of $\mathcal{R}$, we can write $\delta = c(\delta_1, \ldots, \delta_k)$ and $t_i \to^* \delta_i$ for $1 \leq i \leq k$. By induction hypothesis, $\kappa^f(\delta_i) \in eval_{\varphi'}(\mathtt{quote}_{sort(t_i)}(t_i))$. Thus, since

$$\begin{array}{l}\langle \mathtt{quote}_{(0)}(c_{\varphi'(c)}(\varphi'(t_1), \ldots, \varphi'(t_k))), \Lambda\rangle \\ \qquad \to_{\varphi'} \langle c'_{(1\ 2\cdots k)}(\mathtt{quote}_{(0)}(\varphi(t_1)), \ldots, \mathtt{quote}_{(0)}(\varphi(t_k))), \Lambda\rangle\end{array}$$

and $\kappa^f(c(\delta_1, \ldots, \delta_k)) = c'(\kappa^f(\delta_1), \ldots, \kappa^f(\delta_k))$, the conclusion follows. $\square$

Example 8.29 *The following* OBJ3 *program is the new transformed version of the* OBJ *program in Example 8.1:*

```
obj EXAMPLE-TR is
  sorts Nat LNat .
  ops 0 0' : -> Nat .
  ops s s' : Nat -> Nat [strat (1)] .
  ops nil nil' : -> LNat .
  op cons : Nat LNat -> LNat [strat (1)] .
  op cons' : Nat LNat -> LNat [strat (1 2)] .
  op fcons : Nat LNat -> LNat [strat (1 2 0)] .
  op from : Nat -> LNat [strat (1 0)] .
  ops sel sel' : Nat LNat -> Nat [strat (1 2 0)] .
  ops first first' : Nat LNat -> LNat [strat (1 2 0)] .
  op quote : Nat -> Nat [strat (0)] .
  op unquote : Nat -> Nat [strat (1 0)] .
  op quote' : LNat -> LNat [strat (0)] .
  op unquote' : LNat -> LNat [strat (1 0)] .
  vars X Y : Nat .
  var Z : LNat .
  eq sel(s(X),cons(Y,Z)) = sel(X,Z) .
  eq sel(0,cons(X,Z)) = X .
  eq first(0,Z) = nil .
  eq first(s(X),cons(Y,Z)) = cons(Y,first(X,Z)) .
  eq from(X) = cons(X,from(s(X))) .
  eq sel'(s(X),cons(Y,Z)) = sel'(X,Z) .
  eq sel'(0,cons(X,Z)) = quote(X) .
  eq first'(0,Z) = nil' .
  eq first'(s(X),cons(Y,Z)) = cons'(quote(Y),first'(X,Z)) .
  eq quote(0) = 0' .
  eq quote'(cons(X,Z)) = cons'(quote(X),quote'(Z)) .
  eq quote'(nil) = nil' .
```

```
  eq quote(s(X)) = s'(quote(X)) .
  eq quote(sel(X,Z)) = sel'(X,Z) .
  eq quote'(first(X,Z)) = first'(X,Z) .
  eq unquote(0') = 0 .
  eq unquote(s'(X)) = s(unquote(X)) .
  eq unquote'(nil') = nil .
  eq unquote'(cons'(X,Z)) = fcons(unquote(X),unquote'(Z)) .
  eq fcons(X,Z) = cons(X,Z) .
endo
```

Now, the evaluation of `unquote'(quote'(first(s(0),from(0))))` *delivers:*

```
==========================================
obj EXAMPLE-TR
==========================================
reduce in EXAMPLE-TR : unquote'(quote'(first(s(0), from(0)))) .
rewrites: 10 in -10ms cpu (0ms real) (~ rewrites/second)
result LNat: cons(0, nil)
```

This program transformation has been implemented in the OnDemandOBJ prototype of Section 7.6 by using command `eval` which applies the transformation and automatically quotes/unquotes the input expression.

```
EXAMPLE> eval first(s(0),from(0)).
Normal form: cons(0, nil)
{ 0.0000 sec., 10 rewrites }
```

Moreover, we can prove termination of the program in Example 8.29 by using the context-sensitive recursive path ordering (CSRPO) of [Borralleras *et al.*, 2002].

Example 8.30 *Consider again the evaluation of the non-terminating expression* `from(0)` *using the program in Example 8.29. Now, we obtain:*

```
==========================================
obj EXAMPLE-TR
==========================================
reduce in EXAMPLE-TR : unquote'(quote'(from(0))) .
rewrites: 0 in -10ms cpu (0ms real) (~ rewrites/second)
result LNat: unquote'(quote'(from(0)))
```

Conditions under which this second transformation preserve termination of the original program should be further investigated.

8.5 Conclusions and Related Work

We summarize the contributions of this chapter as follows:

- We first introduce an accurate and clear notion of correct and complete computations with (positive) strategy annotations, which overcomes the ambiguity and inconsistency of current terminology (e.g., compare the mix of different concepts for the notion of correctness/completeness in [Nakamura and Futatsugi, 2001; Nakamura and Ogata, 2001; van de Pol, 2001]).

- We demonstrate that previously known approaches for computing normal forms with (non-terminating) OBJ programs using positive strategy annotations (e.g., Nakamura and Ogata's technique of 'completing' head-normalizing E-strategy maps φ for obtaining a normalizing one φ') are not completely satisfactory in practice: they do ensure correctness (that is, that computed E-normal forms are normal forms) but are not able to ensure definedness of the considered semantics.

- We ascertain the conditions (on φ) ensuring that OBJ programs using (positive) strategy annotations do compute the value of any given expression (Theorem 8.10). As shown in Example 8.12, termination of the program (under φ) is essential for achieving correct (and complete) computations.

- Theorem 8.10 requires that all arguments of constructor symbols be replacing. This may incur in unnecessary nontermination. Thus, we have formalized a transformation which can achieve (correct and) complete computations without worsening the termination behavior. Our technique differs from Nakamura and Ogata's (or Nagaya's) approach: we only relax the replacement restrictions associated to the (constructor) symbols after a thorough analysis of their role in the computation.

The only work addressing completeness of the E-strategy (w.r.t. normalization) is Nagaya's thesis (although completeness is called 'normalizability' in Nagaya's terminology). Nagaya establishes conditions (both on the TRS and the E-strategy φ) ensuring that φ is normalizing, i.e., it is able to compute a normal form of a term whenever it exists [Nagaya, 1999]. However, these results only apply to a rather restricted subclass of orthogonal TRSs. In this chapter, by focusing on the functional evaluation semantics (i.e., computations leading to constructor terms or values), we are able to deal with more general programs (represented by left-linear and confluent TRSs); as a counterpart, the termination of the program must be proved either before or after transforming it in order to ensure correctness and completeness (regarding functional evaluation).

In *CSR*, normal forms of a term t can be obtained by successively computing its μ-normal forms s, and continuing the evaluation of t by (recursively) normalizing the maximal non-replacing subterms of s (normalization via μ-normalization [Lucas, 2002a,c]). In OBJ programs, we could proceed in a similar way provided that E-normal forms are μ-normal forms. Unfortunately, we would need a 'meta-process' that uses $eval_\varphi$ to obtain partially evaluated results (i.e., E-normal forms) and then 'jumps' into the non-replacing parts of them in order to obtain normal forms. Of course, this procedure is not directly available in current OBJ implementations. The possibility of achieving a similar effect by using program transformation is a subject of future work.

Chapter 9

On-demand Evaluation by Program Transformation

This chapter presents how the on-demand evaluation of Chapter 7 can be made available to languages that do not provide it (e.g., Maude and OBJ3), thereby introducing a flavour of laziness into such languages. Our proposal is based on a program transformation for OBJ programs which achieves correctness and works well in current OBJ interpreters. In Section 9.1, we motivate the extension of evaluation on demand for functional languages of the OBJ family since on-demand strategy annotations have not been properly implemented to date. In Section 9.2, we introduce an automatic, semantics–preserving program transformation which produces a program (without negative annotations) which can be then correctly executed by typical OBJ interpreters. Moreover, to demonstrate the practicality of our ideas, the program transformation has been implemented into the OnDemandOBJ prototype of Section 7.6. In Section 9.3, we compare the evaluation of transformed programs with the original ones on a set of representative benchmarks. Section 9.4 concludes and summarizes our contributions.

A short version of this chapter appeared in [Alpuente et al., 2002d,f, 2003a,b]

9.1 Introduction

Eager rewriting-based programming languages such as Lisp, OBJ*, CafeOBJ, ELAN, or Maude evaluate expressions by innermost rewriting. Since nontermination is a known problem of innermost reduction, *syntactic annotations* (generally specified as sequences of integers associated to function arguments, called *local strategies*) have been used in OBJ2 [Futatsugi *et al.*, 1985], OBJ3 [Goguen *et al.*, 2000], CafeOBJ [Futatsugi and Nakagawa, 1997], and Maude [Clavel *et al.*, 1996] to improve efficiency and (hopefully) avoid nontermination. Local strategies are used in OBJ programs[1] for guiding the *evaluation strategy* (abbr. E-strategy): when considering a function call $f(t_1, \ldots, t_k)$, only the arguments whose indices are present as *positive* integers in the local strategy for f are evaluated (following the specified ordering). If 0 is found, then the evaluation of f is attempted. Unfortunately, this restriction of rewriting can have a negative impact in the ability to compute normal forms. Whenever the user provides no local strategy for a given symbol, the (Maude, OBJ*, CafeOBJ) interpreter automatically assigns a *default* E-strategy. For instance, the default local strategy of Maude associates the list $(1\ 2 \cdots k\ 0)$ to each k-ary symbol f having no explicit strategy, i.e. all arguments are marked as evaluable.

Example 9.1 *Consider the following* OBJ *program from Example 6.6:*

```
obj Ex1 is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (1)] .
  op from : Nat -> LNat [strat (1 0)] .
  op 2nd  : LNat -> Nat [strat (1 0)] .
  vars X Y : Nat . var Ys : LNat .
  eq 2nd(cons(X,cons(Y,Ys))) = Y .
  eq from(X) = cons(X,from(s(X))) .
endo
```

The evaluation of the term `2nd(from(0))` *as performed by a typical* OBJ *interpreter*[2] *is:*

```
Maude> red 2nd(from(0)) .
reduce in Ex1 : 2nd(from(0)) .
rewrites: 1 in -10ms cpu (0ms real) (~ rewrites/second)
```

[1] By OBJ we mean OBJ2, OBJ3, CafeOBJ, or Maude.
[2] We use the SRI's Maude interpreter (version 1.0.5) available at: `http://maude.cs.uiuc.edu/`.

```
result Nat: 2nd(cons(0, from(s(0))))
```

This corresponds to the following reduction sequence:

$\texttt{2nd(}\underline{\texttt{from(0)}}\texttt{)} \to \texttt{2nd(cons(0,from(s(0))))}$

The evaluation stops at this point since reductions on the second argument of `cons` *are disallowed due to the local strategy* `(1)`*.*

The handicaps of using only positive annotations regarding correctness and completeness of computations are discussed in Section 6.1 and Chapter 8 of this thesis as well as [Lucas, 2001a, 2002b; Ogata and Futatsugi, 2000; Nakamura and Ogata, 2001]: the problem is that the absence of some indices in the local strategies can have a negative impact in the ability of such strategies to compute normal forms. For instance, a further step is required (*demanded* by the rule of `2nd`) in order to obtain the desired outcome in Example 9.1:

$\texttt{2nd(cons(0,}\underline{\texttt{from(s(0))}}\texttt{))} \to \texttt{2nd(cons(0,cons(s(0),from(s(s(0))))))}$

At this stage, it is not necessary to perform any reduction on symbol `from`, since reducing at the root position yields the final value:

$\underline{\texttt{2nd(cons(0,cons(s(0),from(s(s(0))))))}} \to \texttt{s(0)}$

In [Ogata and Futatsugi, 2000; Nakamura and Ogata, 2001], *negative* indices are proposed to indicate those arguments that should be evaluated only 'on-demand', where the 'demand' is an attempt to match an argument term with the left-hand side of a rewrite rule [Eker, 2000; Goguen *et al.*, 2000; Ogata and Futatsugi, 2000]. For instance, in [Nakamura and Ogata, 2001] the authors suggest that `(1 -2)` is the apt local strategy for `cons` in Example 9.1; i.e. the first argument is always evaluated but the second argument is evaluated only "on-demand". Then, the evaluation with strategy `(1 -2)` for `cons` is able to reduce `2nd(from(0))` to `s(0)` without entering in a non-terminating evaluation, whereas evaluation only with positive annotations enters in an infinite derivation or does not provide the associated normal form. It is worthy to note that the calculus is simpler than typical functional lazy rewriting and the appropriate (on-demand) strategy annotations for achieving suitable normal forms can be inferred from the program (see [Lucas, 2001a, 2002b; Ogata and Futatsugi, 2000; Nakamura and Ogata, 2001]).

On-demand strategy annotations have not been properly implemented to date: even if negative annotations are (syntactically) accepted in current OBJ implementations, namely OBJ3 and Maude, unfortunately they do not have the expected (on-demand) effect over the computations.

Example 9.2 *Consider the following* OBJ *program from Example 6.7, which is similar to Example 9.1 except the on-demand strategy annotation for symbol* `cons`*:*

```
obj Ex2 is
```

```
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (1 -2)] .
  op from : Nat -> LNat [strat (1 0)] .
  op 2nd  : LNat -> Nat [strat (1 0)] .
  vars X Y : Nat . var Ys : LNat .
  eq 2nd(cons(X,cons(Y,Ys))) = Y .
  eq from(X) = cons(X,from(s(X))) .
endo
```

The OBJ3 *interpreter does not implement negative (on-demand) annotations though does* accept *this program and the evaluation of* `2nd(from(0))` *surprisingly delivers the very same result as in Example 9.1. That is, the negative annotation is just disregarded by the* OBJ3 *interpreter (which, in this case, causes loss of completeness). On the other hand, the* Maude *interpreter neither implements negative (on-demand) annotations though does also accept this program and the evaluation of the same expression diverges. This is because the negative annotation is interpreted by* Maude *as a positive one thus resulting in non-termination.*

On the other hand, CafeOBJ is able to deal with negative annotations using the on-demand evaluation model of [Nakamura and Ogata, 2001]. For instance, the CafeOBJ interpreter is able to compute the intended value `s(0)` of Example 9.1. However, in Chapter 7 we discussed a number of problems of the on-demand evaluation model of [Nakamura and Ogata, 2001; Ogata and Futatsugi, 2000], as shown in the following example.

Example 9.3 *Consider the following* OBJ *program of Example 7.2:*

```
obj LENGTH is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat  [strat (1)] .
  op from : Nat -> LNat .
  op length : LNat -> Nat     [strat (0)] .
  op length' : LNat -> Nat    [strat (-1 0)] .
  vars X Y : Nat . var Z : LNat .
  eq from(X) = cons(X,from(s(X))) .
  eq length(nil) = 0 .
```

```
  eq length(cons(X,Z)) = s(length'(Z)) .
  eq length'(Z) = length(Z) .
endo
```

The expression `length'(from(0))` *is rewritten (in one step) to the expression* `length(from(0))`*. No evaluation is demanded on the argument of* `length'` *for enabling this step (i.e. the negative annotation* `-1` *is included for* `length'` *but the corresponding rule includes a variable at the first argument of* `length'`*) and no further evaluation on* `length(from(0))` *should be performed (due to the local strategy* `(0)` *of* `length` *which forbids evaluation on any argument of* `length`*). However, the annotation* `-1` *of function* `length'` *is treated in such a way by the operational model of [Ogata and Futatsugi, 2000; Nakamura and Ogata, 2001] that the on-demand evaluation of the expression* `length'(from(0))` *yields an infinite evaluation sequence (see Example 7.2 for a more detailed explanation).*

We proposed in Chapter 7 a solution to these problems which is based on a suitable extension of the E-evaluation strategy of OBJ-like languages (that only considers annotations given as natural numbers) to cope with on-demand strategy annotations. Our strategy incorporates a better treatment of demandedness and also enjoys good computational properties; in particular, we show how to use it for computing (head-)normal forms and we prove it is conservative w.r.t. other on-demand strategies: *lazy rewriting* [Fokkink *et al.*, 2000] and *on-demand rewriting* [Lucas, 2001a]. A program transformation for proving termination of the on-demand evaluation strategy was also formalized, which relies on standard techniques. Furthermore, a *direct* implementation of such on-demand evaluation strategy has been developed.

In this chapter, we show how OBJ programs that use local strategies containing negative annotations (and hence could be correctly executed by using the evaluation strategy proposed in Chapter 7) can be (also) executed in the existing OBJ implementations which only admit positive annotations (e.g. Maude). This is done by means of an automatic program transformation which encodes the 'on-demand' strategy instrumented by the negative annotations within new function symbols (and corresponding program rules) that only use positive strategy annotations. Before entering in too technical details, we give an example which illustrates the power of our transformation.

Example 9.4 *The program of Example 9.2 is transformed by using our method into the following* OBJ *program* without *negative annotations.*

```
obj Ex3 is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat           [strat (1)] .
```

```
    op nil  : -> LNat .
    op cons : Nat LNat -> LNat      [strat ()] .
    op cons_root : Nat LNat -> LNat [strat (1 0)] .
    op cons_+2 : Nat LNat -> LNat   [strat (2)] .
    op from : Nat -> LNat           [strat (1 0)] .
    op 2nd  : LNat -> Nat           [strat (1 0)] .
    op 2nd_+1  : LNat -> Nat        [strat (1 0)] .
    op quoteNat : Nat -> Nat        [strat (0)] .
    op quoteLNat : LNat -> LNat     [strat (0)] .
    vars X Y : Nat . vars Xs : LNat .
    eq 2nd(cons(X,Xs)) = 2nd_+1(cons_+2(X,Xs)) .
    eq 2nd_+1(cons_+2(X,cons(Y,Xs))) = quoteNat(Y) .
    eq from(X) = quoteLNat(cons(X,from(s(X)))) .
    eq quoteNat(2nd(Xs)) = 2nd(quoteLNat(Xs)) .
    eq quoteNat(2nd_+1(Xs)) = 2nd(Xs) .
    eq quoteNat(s(X)) = s(quoteNat(X)) .
    eq quoteNat(0) = 0 .
    eq quoteLNat(from(X)) = from(quoteNat(X)) .
    eq quoteLNat(cons(X,Xs)) = cons_root(quoteNat(X),Xs) .
    eq quoteLNat(cons_+2(X,Xs)) = cons(X,Xs) .
    eq quoteLNat(nil) = nil .
    eq cons_root(X,Xs) = cons(X,Xs) .
  endo
```

where $\mathtt{cons}_{root}$ *and* $\mathtt{cons}_{+2}$ *are new symbols introduced by the transformation which perform the pattern matching in a stepwise manner, and* quoteNat *and* quoteLNat *are auxiliary symbols which help to perform a correct evaluation to head normal forms. Roughly speaking, for each constructor symbol* c *with a negative annotation* $-i$*, we remove all strategy annotations for* c *and introduce an auxiliary constructor symbol* $\mathtt{c}_{+i}$ *with positive annotation* i*. Also, we introduce a defined symbol* $\mathtt{c}_{root}$ *without the negative annotations but with the positive ones plus* 0 *and we introduce a new rule which is used to translate the new symbol* $\mathtt{c}_{root}$ *back to* c*. Then, we add new rules which re–define symbols using constructor* c *in terms of* c*,* $\mathtt{c}_{root}$*, and* $\mathtt{c}_{+i}$*. Finally, for each program rule* $l \to r$*, we introduce a symbol* quote *in* r *(specialized for each sort) and add a number of new rules for symbols* quote *which transform* c *into* $\mathtt{c}_{root}$ *and help to appropriately (head)-normalize terms.*

Now, the evaluation of 2nd(from(0)) *using* Maude *yields:*

```
Maude> red quoteNat(2nd(from(0))) .
reduce in Ex3 : quoteNat(2nd(from(0))) .
rewrites: 16 in -10ms cpu (0ms real) (~ rewrites/second)
```

```
result Nat: s(0)
```

which is the desired result. Note that for evaluating expression e*, we only need to call* `quote`$_\tau(e)$ *for the appropriate sort* τ *of* e*.*

9.2 The Program Transformation

In the following, we formalize a program transformation which translates OBJ programs with arbitrary indices into OBJ programs with positive indices alone. We first explain the awkward points associated to the evaluation with negative indices in order to discern how to transform these indices into positive ones.

Example 9.5 *Consider the following* OBJ *program which is mainly borrowed from a* CafeOBJ *program in [Ogata and Futatsugi, 2000] where negative indices are considered for* `cons`*:*

```
obj Ex3rd is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat       [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (1 -2)] .
  op from : Nat -> LNat      [strat (1 0)] .
  op 3rd  : LNat -> Nat      [strat (1 0)] .
  vars X Y Z : Nat . var Zs : LNat .
  eq 3rd(cons(X,cons(Y,cons(Z,Zs)))) = Z .
  eq from(X) = cons(X,from(s(X))) .
endo
```

Let us introduce an auxiliary function symbol `inf`*, which returns an infinite sequence of symbols* `s` *(note that the evaluation of the call* `inf` *is non-terminating since the strategy annotation for the constructor* `s` *above is* `1`*):*

```
op inf : -> Nat .
eq inf = s(inf) .
```

Consider the following term $t =$ `3rd(cons(0,cons(inf,cons(s(0),nil))))`*. The on-demand E-evaluation of* t *terminates and returns* `s(0)`*, since the term* `inf` *is not under a positive (reducible) position nor is demanded by the rule defining* `3rd`*.*

Let us consider a naïve approach for transforming negative indices into positive indices which duplicates the symbols containing negative indices into new symbols containing the positive counterparts (as in the program transformation showed in Section 7.5 for approximating termination). The raw application of such program transformation to the previous example delivers the following rules:

```
eq 3rd(cons(X,Zs)) = 3rd(cons+2(X,Zs)) .
eq 3rd(cons+2(X,cons(Y,Zs))) = 3rd(cons+2(X,cons+2(Y,Zs))) .
eq 3rd(cons+2(X,cons+2(Y,cons(Z,Zs)))) = Z .
```

together with the following definitions for symbols `cons` *and* $\texttt{cons}_{+2}$*:*

```
op cons : Nat LNat -> LNat [strat (1)] .
op cons+2 : Nat LNat -> LNat [strat (2)] .
```

However, the evaluation of the previous call t *w.r.t. this new program enters in an infinite reduction sequence, since the term* `inf` *will be under a positive (reducible) position after transforming the leftmost symbol* `cons` *of* t *into* $\texttt{cons}_{+2}$.

In order to avoid this problem, the solution is to remove all positive annotations of symbol `cons`*, i.e. we obtain:*

```
op cons : Nat LNat -> LNat [strat ()] .
op cons+2 : Nat LNat -> LNat [strat (2)] .
```

However, occurrences of symbol `cons` *appearing at positive positions of the original program rules do not behave correctly now, e.g.* `3rd(cons(inf,from(s(0)))` *does not have an infinite reduction sequence as in the original program because the subterm* `inf` *does not appear under a positive position.*

Nevertheless, we can define a new symbol $\texttt{cons}_{root}$ *which inherits the behavior of the original symbol* `cons` *(i.e. the positive indices of* `cons`*), and consistently rename the symbols in the TRS through a special symbol* `quote` *before evaluating each term (note that* `quote` *is specialized to sorts). Moreover, we introduce a new rule for translate* $\texttt{cons}_{root}$ *back to* `cons`*. We finally obtain the program:*

```
obj Ex3rdA is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat          [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat    [strat ()] .
  op consroot : Nat LNat -> LNat [strat (1 0)] .
  op cons+2 : Nat LNat -> LNat  [strat (2)] .
  op from : Nat -> LNat         [strat (1 0)] .
  op 3rd  : LNat -> Nat         [strat (1 0)] .
  op quoteNat : Nat -> Nat      [strat (0)].
  op quoteLNat : LNat -> LNat   [strat (0)].
  vars X Y Z : Nat . var Zs : LNat .
  eq 3rd(cons(X,Zs)) = 3rd(cons+2(X,Zs)) .
  eq 3rd(cons+2(X,cons(Y,Zs))) = 3rd(cons+2(X,cons+2(Y,Zs))) .
  eq 3rd(cons+2(X,cons+2(Y,cons(Z,Zs)))) = quoteNat(Z) .
```

```
    eq from(X) = quoteLNat(cons(X,from(s(X)))) .
    eq quoteNat(3rd(Zs)) = 3rd(quoteLNat(Zs)) .
    eq quoteNat(s(X)) = s(quoteNat(X)) .
    eq quoteNat(0) = 0 .
    eq quoteLNat(from(X)) = from(quoteNat(X)) .
    eq quoteLNat(cons(X,Zs)) = cons_root(quoteNat(X),Zs) .
    eq quoteLNat(nil) = nil .
    eq cons_root(X,Zs) = cons(X,Zs) .
  endo
```

However, in order to speed up the evaluation, we use more f_{+i} symbols from the root of the left hand side traversing the on-demand path (see Section 9.2.2).

We define the complete transformation for a TRS by two different transformers which tackle the two difficulties described above. That is, the transformation starts by applying the first transformer which introduces symbols f_{root} and specialized symbols `quote` in order to set the stage for the second transformer. Then, the second transformer is applied iteratively, which turns the negative indices into positive ones by introducing symbols f_{+i}. This transformer finally removes all negative annotations together with positive annotations of conflictive symbols (such as symbol `cons` in the previous example).

9.2.1 Transformation for fixing rules

Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS, and φ be an E-strategy map. We define the set of positions which are active (but not reducible) of a term $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ as $\mathcal{P}os_{A-P}(t) = \mathcal{P}os_A(t) - \mathcal{P}os_P(t)$. Given a strategy map φ, in Section 7.3.1 we defined φ_+ as the result of removing all negative indices from φ.

Let us define the set of symbols of the original TRS at positions active but not reducible which have positive indices which can be lost as $\mathcal{F}^\varphi_\mathcal{R} = \{f \in \mathcal{F} \mid ar(f) > 0 \wedge \varphi_+(f) \neq nil \wedge \exists l \in L(\mathcal{R}) : \mathcal{P}os_f(l) \cap \mathcal{P}os_{A-P}(\varphi(l)) \neq \varnothing\}$. Note that if $\mathcal{R}$ is a CS, then $\mathcal{F}^\varphi_\mathcal{R} \subseteq \mathcal{C}$. In the following, we define the set of auxiliary rules $Quote_{\mathcal{R}^\mathbb{I},\mathcal{F}^\varphi_\mathcal{R}}$ to appropriately (head)-normalize terms. Intuitively, `quote` translates a symbol $f \in \mathcal{F}^\varphi_\mathcal{R}$ at a positive position into the new symbol f_{root}.

$$Quote_{\mathcal{R}^\mathbb{I},\mathcal{F}^\varphi_\mathcal{R}} = \bigcup_{f\in\mathcal{F}} \begin{cases} \mathtt{quote}(f(\overline{x})) \to f_{root}(\rho_f(\overline{x})) & \text{if } f \in \mathcal{F}^\varphi_\mathcal{R} \\ \mathtt{quote}(f(\overline{x})) \to f(\rho_f(\overline{x})) & \text{if } f \notin \mathcal{F}^\varphi_\mathcal{R} \end{cases}$$

$$\text{where } \rho_f(x_i) = \begin{cases} \mathtt{quote}(x_i) & \text{if } \mathtt{(i)} \sqsubseteq \varphi^\mathbb{I}(f) \\ x_i & \text{if } \mathtt{(i)} \not\sqsubseteq \varphi^\mathbb{I}(f) \end{cases}$$

We define the *first transformer* for fixing strategy annotations $\mathcal{R}^\mathbb{I} = (\mathcal{F}^\mathbb{I}, R^\mathbb{I})$ and

$\varphi^{\mathbb{I}}$ as follows: $\mathcal{F}^{\mathbb{I}} = \mathcal{F} \cup \{f_{root} \mid f \in \mathcal{F}^{\varphi}_{\mathcal{R}}\}$, and

$$R^{\mathbb{I}} = \{f(\overline{t}) \to \mathtt{quote}(r) \mid f(\overline{t}) \to r \in R\} \cup \{f_{root}(\overline{x}) \to f(\overline{x}) \mid f \in \mathcal{F}^{\varphi}_{\mathcal{R}}\} \\ \cup\, Quote_{\mathcal{R}^{\mathbb{I}},\mathcal{F}^{\varphi}_{\mathcal{R}}}$$

Also, $\varphi^{\mathbb{I}}(f) = \varphi(f)$ for all $f \in \mathcal{F}$, $\varphi^{\mathbb{I}}(f_{root}) = \varphi(f){+}{+}\mathtt{(0)}$ for all $f \in \mathcal{F}^{\varphi}_{\mathcal{R}}$, and $\varphi^{\mathbb{I}}(\mathtt{quote}) = \mathtt{(0)}$.

Example 9.6 *Consider the TRS $\mathcal{R}$ and the E-strategy map φ of Example 9.5. The TRS $\mathcal{R}^{\mathbb{I}}$ together with $\varphi^{\mathbb{I}}$ is:*

```
obj Ex3rdI is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat           [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat     [strat (1 -2)] .
  op consroot : Nat LNat -> LNat  [strat (1 -2 0)] .
  op from : Nat -> LNat          [strat (1 0)] .
  op 3rd  : LNat -> Nat          [strat (1 0)] .
  op quoteNat : Nat -> Nat       [strat (0)] .
  op quoteLNat : LNat -> LNat    [strat (0)] .
  vars X Y Z : Nat . var Zs : LNat .
  eq 3rd(cons(X,cons(Y,cons(Z,Zs)))) = quoteNat(Z) .
  eq from(X) = quoteLNat(cons(X,from(s(X)))) .
  eq quoteNat(3rd(Zs)) = 3rd(quoteLNat(Zs)) .
  eq quoteNat(s(X)) = s(quoteNat(X)) .
  eq quoteNat(0) = 0 .
  eq quoteLNat(from(X)) = from(quoteNat(X)) .
  eq quoteLNat(cons(X,Zs)) = consroot(quoteNat(X),Zs) .
  eq quoteLNat(nil) = nil .
  eq consroot(X,Zs) = cons(X,Zs) .
endo
```

9.2.2 Transformation for eliminating negative indices

We formulate the *transformer* for switching negative indices into positive ones. Intuitively, we transform a rule $l \to r$ with a position p which is active but not reducible into the rules $l[x]_p \to l'[x]_p$ and $l' \to r$, where each symbol f in l from the root to p has been replaced in l' by the new symbol f_{+i} and its negative indices are replaced by their positive counterparts.

Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS, and φ be an E-strategy map. Given $l \in L(\mathcal{R})$, we define the set of position of a lhs l which are active but not reducible as $\mathcal{I}^\varphi(l) = \mathcal{P}os_{A-P}(\varphi(l)) \cap \mathcal{P}os_{\mathcal{F}}(l)$. Assume that $\mathcal{I}^\varphi(l) \neq \varnothing$ for a rule $l \to r \in \mathcal{R}$, the position to be transformed is $p.i = max_{\leq_{\varphi(l)}}(\mathcal{I}^\varphi(l))$ for $p.i \in \mathcal{P}os(l)$ and $i \in \mathbb{N}$ (note that $\Lambda \notin \mathcal{I}^\varphi(l)$ by definition), and the set of symbols involved in the transformation are $f^\Lambda, \ldots, f^p$ such that $root(l|_q) = f^q \in \mathcal{F}$ for $q \leq p$. Then, the *transformer for eliminating negative indices* $\mathcal{R}^{neg} = (\mathcal{F}^{neg}, R^{neg})$ and φ^{neg} is as follows: $\mathcal{F}^{neg} = \mathcal{F} \cup \{f^\Lambda_{+j_\Lambda}, \ldots, f^p_{+j_p}\}$ where $j_\Lambda, \ldots, j_p \in \mathbb{N}$ and $q.j_q \leq p.i$ for $q \leq p$; and

$$R^{neg} = R - \{l \to r\} \cup \{l[y]_{p.i} \to l'[y]_{p.i}, l' \to r\} \cup \{\mathtt{quote}(f^q_{+j_q}(\overline{x})) \to f^q(\overline{x})\}$$

where y is a fresh variable and l' is obtained from l such that $\forall q \in \mathcal{P}os(l')$:

$$root(l'|_q) = \begin{cases} f^q_{+j_q} & \text{if } q \leq p \\ root(l|_q) & \text{otherwise} \end{cases}$$

We let $\varphi^{neg}(f) = \varphi(f)$ for $f \in \mathcal{F}$ and

$$\varphi^{neg}(f_{+j}) = \begin{cases} \mathtt{(j\ 0)} & \text{if } \mathtt{(-j\ 0)} \sqsubseteq \varphi(f) \vee \mathtt{(j\ 0)} \sqsubseteq \varphi(f) \\ \mathtt{(j)} & \text{if } \mathtt{(-j\ 0)} \not\sqsubseteq \varphi(f) \wedge \mathtt{(j\ 0)} \not\sqsubseteq \varphi(f) \end{cases}$$

Example 9.7 *Consider the TRS $\mathcal{R} = \mathcal{R}^{\mathbb{I}}$ and the E-strategy map $\varphi = \varphi^{\mathbb{I}}$ in Example 9.6. The application of the transformer, the TRS $\mathcal{R}^{neg}$ together with φ^{neg}, is:*

```
obj Ex3rdINeg is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat            [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat      [strat (1 -2)] .
  op cons_root : Nat LNat -> LNat  [strat (1 -2 0)] .
  op cons_+2 : Nat LNat -> LNat    [strat (2)] .
  op from : Nat -> LNat           [strat (1 0)] .
  op 3rd  : LNat -> Nat           [strat (1 0)] .
  op 3rd_+1  : LNat -> Nat        [strat (1 0)] .
  op quoteNat : Nat -> Nat        [strat (0)] .
  op quoteLNat : LNat -> LNat     [strat (0)] .
  vars X Y Z : Nat . var Zs : LNat .
  eq 3rd(cons(X,cons(Y,Zs))) = 3rd_+1(cons_+2(X,cons_+2(Y,Zs))) .
  eq 3rd_+1(cons_+2(X,cons_+2(Y,cons(Z,Zs)))) = quoteNat(Z) .
  eq from(X) = quoteLNat(cons(X,from(s(X)))) .
  eq quoteNat(3rd(Zs)) = 3rd(quoteLNat(Zs)) .
```

```
    eq quoteNat(3rd+1(Zs)) = 3rd(Zs) .
    eq quoteNat(s(X)) = s(quoteNat(X)) .
    eq quoteNat(0) = 0 .
    eq quoteLNat(from(X)) = from(quoteNat(X)) .
    eq quoteLNat(cons(X,Zs)) = consroot(quoteNat(X),Zs) .
    eq quoteLNat(cons+2(X,Zs)) = cons(X,Zs) .
    eq quoteLNat(nil) = nil .
    eq consroot(X,Zs) = cons(X,Zs) .
  endo
```

Note the relevant changes in the rule for symbol `3rd`*:*

```
    eq 3rd(cons(X,cons(Y,Zs))) = 3rd+1(cons+2(X,cons+2(Y,Zs))) .
    eq 3rd+1(cons+2(X,cons+2(Y,cons(Z,Zs)))) = quoteNat(Z) .
```

The second transformation process starts from $\mathcal{R}^{\mathbb{I}}$ and $\varphi^{\mathbb{I}}$ and applies as many transformation steps $\mathcal{R}^{neg}$ and φ^{neg} for removing negative indices as necessary to obtain $\mathcal{R}' = (\mathcal{F}', R')$ and φ' such that no negative index is necessary, i.e. $\mathcal{I}^{\varphi'}(l) = \varnothing$ for all $l \in L(\mathcal{R}')$.

The final TRS $\mathcal{R}^{\mathbb{II}} = (\mathcal{F}^{\mathbb{II}}, R^{\mathbb{II}})$ and $\varphi^{\mathbb{II}}$ is obtained as $\mathcal{F}^{\mathbb{II}} = \mathcal{F}'$, $R^{\mathbb{I}} = R'$, and $\varphi^{\mathbb{II}}(f) = \varphi'_+(f)$ for $f \in \mathcal{F}' - \mathcal{F}^{\varphi}_{\mathcal{R}}$ and $\varphi^{\mathbb{II}}(f) = nil$ for $f \in \mathcal{F}^{\varphi}_{\mathcal{R}}$.

Example 9.8 *Continuing with Example 9.7. The final TRS $\mathcal{R}^{\mathbb{II}}$ together with $\varphi^{\mathbb{II}}$ is:*

```
  obj Ex3rdII is
    sorts Nat LNat .
    op 0    : -> Nat .
    op s    : Nat -> Nat            [strat (1)] .
    op nil  : -> LNat .
    op cons : Nat LNat -> LNat      [strat ()] .
    op consroot : Nat LNat -> LNat  [strat (1 0)] .
    op cons+2 : Nat LNat -> LNat    [strat (2)] .
    op from : Nat -> LNat           [strat (1 0)] .
    op 3rd  : LNat -> Nat           [strat (1 0)] .
    op 3rd+1  : LNat -> Nat         [strat (1 0)] .
    op quoteNat : Nat -> Nat        [strat (0)] .
    op quoteLNat : LNat -> LNat     [strat (0)] .
    vars X Y Z : Nat . var Zs : LNat .
    eq 3rd(cons(X,Zs)) = 3rd+1(cons+2(X,Zs))) .
    eq 3rd+1(cons+2(X,cons(Y,Zs))) = 3rd+1(cons+2(X,cons+2(Y,Zs))) .
    eq 3rd+1(cons+2(X,cons+2(Y,cons(Z,Zs)))) = quoteNat(Z) .
    eq from(X) = quoteLNat(cons(X,from(s(X)))) .
```

```
  eq quoteNat(3rd(Zs)) = 3rd(quoteLNat(Zs)) .
  eq quoteNat(3rd+1(Zs)) = 3rd(Zs) .
  eq quoteNat(s(X)) = s(quoteNat(X)) .
  eq quoteNat(0) = 0 .
  eq quoteLNat(from(X)) = from(quoteNat(X)) .
  eq quoteLNat(cons(X,Zs)) = cons_root(quoteNat(X),Zs) .
  eq quoteLNat(cons+2(X,Zs)) = cons(X,Zs) .
  eq quoteLNat(nil) = nil .
  eq cons_root(X,Zs) = cons(X,Zs) .
endo
```

We would like to emphasize that the transformation defined in this chapter is able to deal with the general case where more than one negative annotation exist for the same function symbol and different demandedness paths are possible; which are just ordered using the ordering $\leq_s$ between positive and negative annotations and managed by using symbols f_{+i} which traverse the path from the root. This is not illustrated in our main example due to space restrictions, though we give some hints in the following example.

Example 9.9 *Consider the following program rule:*

```
min(s(X),0,s(0),s(s(Y))) = 0
```

with the strategy map $\varphi(\texttt{min}) = \texttt{(-3 -2 1 -4 0)}$*,* $\varphi(\texttt{s}) = \texttt{(-1)}$*, and* $\varphi(\texttt{0}) = nil$*. The transformation of this rule produces (without considering symbol* `quote`*):*

```
min(s(X),Z,W,Y) = min+3(s(X),Z,W,Y)
min+3(s(X),Z,s(W),Y) = min+3(s(X),Z,s+1(W),Y)
min+3(s(X),Z,s+1(0),Y) = min+2(s(X),Z,s(0),Y)
min+2(s(X),0,s(0),Y) = min+4(s(X),0,s(0),Y)
min+4(s(X),0,s(0),s(Y)) = min+4(s(X),0,s(0),s+1(Y))
min+4(s(X),0,s(0),s+1(s(Y))) = 0
```

and the strategy map $\varphi^{\mathbb{II}}(\texttt{min}) = \texttt{(1 0)}$*,* $\varphi^{\mathbb{II}}(\texttt{min}_{+2}) = \texttt{(2 0)}$*,* $\varphi^{\mathbb{II}}(\texttt{min}_{+3}) = \texttt{(3 0)}$*,* $\varphi^{\mathbb{II}}(\texttt{min}_{+4}) = \texttt{(4 0)}$*,* $\varphi^{\mathbb{II}}(\texttt{s}) = nil$*,* $\varphi^{\mathbb{II}}(\texttt{s}_{+1}) = \texttt{(1)}$*, and* $\varphi^{\mathbb{II}}(\texttt{0}) = nil$*. This transformed program reproduces the ordering between the different on-demand paths of the left-hand side.*

9.2.3 Properties

In the following, we establish the main results of the program transformation. We first introduce some helpful definitions and results.

Given two strategy lists $L, L' \in \mathbb{L}$, the restriction of L to L', denoted as $L_{\downarrow L'}$, is the maximal sublist L'' such that $L'' \sqsubseteq L$ and $L'' \sqsubseteq L'$. In the following, we define three different translations of terms in $\mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ into terms of $\mathcal{T}({\mathcal{F}^{\mathbb{II}}}^\sharp_{\varphi^{\mathbb{II}}}, \mathcal{X}^\sharp_{\varphi^{\mathbb{II}}})$: without considering extra symbols f_{root} and f_{+i} (translation $t_{\downarrow\varphi^{\mathbb{II}}}$), considering the insertion of symbols f_{root} (translation $pos_p(t)$), or considering the insertion of symbols f_{+i} (translation $neg_p(t)$). Note that for all $f \in \mathcal{F}$, $\varphi^{\mathbb{II}}(f) \sqsubseteq \varphi(f)$.

Definition 9.10 *Let φ be a strategy map over signature $\mathcal{F}$. Let $\varphi^{\mathbb{II}}$ be a strategy map over signature $\mathcal{F}^{\mathbb{II}}$. Let $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$. We define the translation of t into terms $\mathcal{T}({\mathcal{F}^{\mathbb{II}}}^\sharp_{\varphi^{\mathbb{II}}}, \mathcal{X}^\sharp_{\varphi^{\mathbb{II}}})$ as $t_{\downarrow\varphi^{\mathbb{II}}} = s$ where $\mathcal{P}os(s) = \mathcal{P}os(t)$ and $\forall q \in \mathcal{P}os(t).root(t|_q) = f_{L_1|L_2}$, $root(s|_q) = f_{L'_1|L'_2}$ where $L'_1 = L_{1\downarrow\varphi^{\mathbb{II}}(f)}$ and $L'_2 = L_{2\downarrow\varphi^{\mathbb{II}}(f)}$.*

Definition 9.11 *Let φ be a strategy map over signature $\mathcal{F}$ and $\varphi^{\mathbb{II}}$ be a strategy map over signature $\mathcal{F}^{\mathbb{II}}$. Let $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and $p \in \mathcal{P}os_A(t)$. We define the translation of t into terms $\mathcal{T}({\mathcal{F}^{\mathbb{II}}}^\sharp_{\varphi^{\mathbb{II}}}, \mathcal{X}^\sharp_{\varphi^{\mathbb{II}}})$ as $pos_p(t) = s$ where $\mathcal{P}os(s) = \mathcal{P}os(t)$ and $\forall q \in \mathcal{P}os(t).root(t|_q) = f_{L_1|L_2}$, we have:*

$$root(s|_q) = \begin{cases} f_{root\,L'_1|L'_2} & \text{if } p \in \mathcal{P}os_P(t), q \leq p, \text{ and } f \in \mathcal{F}^\varphi_\mathcal{R}; \\ & \quad \text{where } L'_1 = L_{1\downarrow\varphi^{\mathbb{II}}(f_{root})}, \text{ and } L'_2 = L_{2\downarrow\varphi^{\mathbb{II}}(f_{root})} \\ f_{L'_1|L'_2} & \text{otherwise; where } L'_1 = L_{1\downarrow\varphi^{\mathbb{II}}(f)}, \text{ and } L'_2 = L_{2\downarrow\varphi^{\mathbb{II}}(f)} \end{cases}$$

Definition 9.12 *Let φ be a strategy map over signature $\mathcal{F}$ and $\varphi^{\mathbb{II}}$ be a strategy map over signature $\mathcal{F}^{\mathbb{II}}$. Let $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ and $p \in \mathcal{P}os_A(t)$. We define the translation of t into terms $\mathcal{T}({\mathcal{F}^{\mathbb{II}}}^\sharp_{\varphi^{\mathbb{II}}}, \mathcal{X}^\sharp_{\varphi^{\mathbb{II}}})$ as $neg_p(t) = s$ where $\mathcal{P}os(s) = \mathcal{P}os(t)$ and $\forall q \in \mathcal{P}os(t).root(t|_q) = f_{L_1|L_2}$, we have:*

$$root(s|_q) = \begin{cases} f_{+i\,L'_1|L'_2} & \text{if } p \in \mathcal{P}os_{A-P}(t) \text{ and } q.i \leq p; \text{ where } L'_1 = (\texttt{i}) \text{ and} \\ & L'_2 = \begin{cases} (\texttt{0}) & \text{if } (\texttt{-i 0}) \sqsubseteq \varphi(f) \vee (\texttt{i 0}) \sqsubseteq \varphi(f) \\ nil & \text{if } (\texttt{-i 0}) \not\sqsubseteq \varphi(f) \wedge (\texttt{i 0}) \not\sqsubseteq \varphi(f) \end{cases} \\ f_{L'_1|L'_2} & \text{otherwise; where } L'_1 = L_{1\downarrow\varphi^{\mathbb{II}}(f)}, \text{ and } L'_2 = L_{2\downarrow\varphi^{\mathbb{II}}(f)} \end{cases}$$

The following proposition shows that each evaluation step with negative annotations can be simulated by the transformed program. We define a *standard* E-strategy map φ as an E-strategy map where no index different to `0` appears at the right of any index `0` in the annotation sequence.

Proposition 9.13 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map such that $\mathcal{R}$ is φ-terminating. Let $\mathcal{R}^{\mathbb{II}} = (\mathcal{F}^{\mathbb{II}}, R^{\mathbb{II}})$ and $\varphi^{\mathbb{II}}$. For all $t, s \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$, and $q \in \mathcal{P}os_A(s)$, if $\langle t, \Lambda\rangle \overset{\sharp}{\to}_{\varphi,\mathcal{R}} \langle s, q\rangle$, then $\langle pos_\Lambda(t), \Lambda\rangle \overset{\sharp}{\to}^*_{\varphi^{\mathbb{II}},\mathcal{R}^{\mathbb{II}}} \langle s', q\rangle$ where either (1) $s' = \texttt{quote}_\tau(s_{\downarrow\varphi^{\mathbb{II}}})$, (2) $s' = pos_\Lambda(s)$, (3) $s' = neg_q(s)$, or (4) s and s' are $\overset{\sharp}{\to}$-normal forms with a defined symbol at root position.*

Proof. We consider the different cases for $\overset{\sharp}{\to}_{\varphi,\mathcal{R}}$.

1. The cases $t = f_{L|nil}(t_1, \ldots, t_k)$ or $t = \overline{f}_{L_1|L_2}(t_1, \ldots, t_k)$ are impossible.

2. Let $t = f_{L_1|i:L_2}(t_1, \ldots, t_k)$, $i > 0$, $q = i$, and $s = f_{L_1@i|L_2}(t_1, \ldots, t_k)$. If $f \notin \mathcal{F}^{\varphi}_{\mathcal{R}}$, then $root(pos_\Lambda(t)) = f_{L_1|i:L_2}$ and the conclusion follows. If $f \in \mathcal{F}^{\varphi}_{\mathcal{R}}$, then $root(pos_\Lambda(t)) = f_{L'_1|i:L'_2}$ such that L'_1 and L'_2 do not contain negative annotations. Thus, the conclusion follows and condition (2) is fulfilled.

3. Let $t = f_{L_1|-i:L_2}(t_1, \ldots, t_k)$, $i > 0$, $q = \Lambda$, and $s = f_{L_1@-j|L_2}(t_1, \ldots, t_k)$. In this case, it is clear that index $-i$ would not appear in $pos_\Lambda(t)$. Thus, since $neg_\Lambda(t') = t'$ for any term t', $\langle pos_\Lambda(t), \Lambda\rangle \xrightarrow{\sharp}{}^{=}_{\varphi^{\mathbb{I}\mathbb{I}},\mathcal{R}^{\mathbb{I}\mathbb{I}}} \langle neg_\Lambda(pos_\Lambda(t)), \Lambda\rangle$ and condition (2) is fulfilled.

4. Let $t = f_{L_1|0:L_2}(t_1, \ldots, t_k) = \sigma(l')$, $erase(l') = l$ for $l \to r \in \mathcal{R}$ and substitution σ, $s = \sigma(\varphi(r))$, and $q = \Lambda$. Note that since $\mathcal{R}$ is a CS and $f \in \mathcal{D}$, $f \notin \mathcal{F}^{\varphi}_{\mathcal{R}}$ and $root(pos_\Lambda(t)) = f_{L'_1|0:L'_2}$.

   If $l \to r \in \mathcal{R}^{\mathbb{I}\mathbb{I}}$, then the conclusion follows and condition (1) is fulfilled. Consider $l \to r \notin \mathcal{R}^{\mathbb{I}\mathbb{I}}$. Then, there is a set of rules $l_1 \to r_1, \ldots, l_n \to r_n \in \mathcal{R}^{\mathbb{I}\mathbb{I}}$ such that $l_1 = l[\overline{x}]_Q$ for a set of positions Q and $r_n = \mathtt{quote}_\tau(r)$. Moreover, when these rules are applied sequentially to term $pos_\Lambda(t)$, they produce term $s' = \mathtt{quote}_{\tau(root(s))}(\sigma(\varphi^{\mathbb{I}\mathbb{I}}(r)))$. Hence, the conclusion follows and condition (1) is fulfilled.

5. Let $t = f_{L_1|0:L_2}(t_1, \ldots, t_k)$, $erase(t)$ is not a redex w.r.t. $\mathcal{R}$, $OD_{\mathcal{R}}(t) = \varnothing$, $s = f_{L_1|L_2}(t_1, \ldots, t_k)$, and $q = \Lambda$. If $erase(t)$ is not a redex neither w.r.t. $\mathcal{R}^{\mathbb{I}\mathbb{I}}$, then the conclusion follows and condition (2) is fulfilled.

   First, note that it is impossible that $erase(t)$ is not a redex w.r.t. $\mathcal{R}^{\mathbb{I}\mathbb{I}}$ but $OD_{\mathcal{R}}(t) \neq \varnothing$, since there are no negative annotations in $\mathcal{R}^{\mathbb{I}\mathbb{I}}$.

   Consider $erase(t) = \sigma(l)$ for $l \to r \in \mathcal{R}^{\mathbb{I}\mathbb{I}}$. By definition, $\exists l' \to r' \in \mathcal{R}$ s.t. l differs from l' in a set of variables positions, $\exists Q \subseteq \mathcal{P}os(l')$ s.t. $l = l'[\overline{x}]_Q$. Then, there is a set of rules $l_1 \to r_1, \ldots, l_n \to r_n \in \mathcal{R}^{\mathbb{I}\mathbb{I}}$ such that $l_1 = l$ and $r_n = \mathtt{quote}_\tau(r')$. Moreover, when these rules are applied sequentially to term $pos_\Lambda(t)$, they produce a term t' which differs from t in a set of symbols $\{f^\Lambda, \ldots, f^{q'}\}$ for a position $q' \in \mathcal{P}os(t)$ such that $t' = t[f^\Lambda, \ldots, f^{q'}]_{\{\Lambda,\ldots,q'\}}$. Note that since $erase(t)$ is not a redex w.r.t. $\mathcal{R}$, the sequence of rules $l_1 \to r_1, \ldots, l_n \to r_n$ will not be able to evaluate t to r_n at it will stop at some intermediate step, which is just the expression t' with symbols f^{+i}. Then, since $\mathcal{R}$ is φ-terminating, the term t' can be produced without entering in an infinite evaluation sequence. Hence, condition (4) is fulfilled.

6. Let $t = f_{L_1|0:L_2}(t_1, \ldots, t_k)$, $erase(t)$ is not a redex w.r.t. $\mathcal{R}$, $OD_{\mathcal{R}}(t) = \{p'\}$, $s = mark(t, p')$ and $q = p'$.

   Again, there is a set of rules $l_1 \to r_1, \ldots, l_n \to r_n \in \mathcal{R}^{\mathbb{I}\mathbb{I}}$ such that $l_1 = l[\overline{x}]_Q$ for a set of positions Q and l_n differs from l in a set of symbols $\{f^\Lambda, \ldots, f^{p''}\}$ for a position $p'' \in \mathcal{P}os(l)$ s.t. $p''.i = p'$ for $i \in \mathbb{N}$ and $l_n = l[\overline{y}]_{Q'}[f^{\Lambda}, \ldots, f^{p''}]_{\{\Lambda,\ldots,p''\}}$ for another set of positions Q'. That is, these rules are auxiliary rules introduced to stepwise the pattern matching process until position p' (which is the demanded position) is reached and its evaluation started. Hence, condition (3) is fulfilled. $\Box$

We denote by $pos(t)$ the extension of $pos_p(t)$ to include symbols f_{root} w.r.t. all positive positions in t, i.e. $root(pos(t)|_p) = f_{root L'_1|L'_2}$ if $p \in \mathcal{P}os_P(t) \cap \mathcal{P}os_{\mathcal{F}^{\varphi}_{\mathcal{R}}}(t)$, where $t|_p = f_{L_1|L_2}$, $L'_1 = L_{1\downarrow\varphi^{\mathbb{I}\mathbb{I}}(f_{root})}$, and $L'_2 = L_{2\downarrow\varphi^{\mathbb{I}\mathbb{I}}(f_{root})}$; and $root(pos(t)|_p) = f_{L'_1|L'_2}$ otherwise, where $t|_p = f_{L_1|L_2}$, $L'_1 = L_{1\downarrow\varphi^{\mathbb{I}\mathbb{I}}(f)}$, and $L'_2 = L_{2\downarrow\varphi^{\mathbb{I}\mathbb{I}}(f)}$.

The following lemma ensures that normalization from a term $\mathtt{quote}_\tau(t)$ or from a term $pos(t)$ is equivalent.

Lemma 9.14 *Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS and φ be a standard strategy map. Let $\mathcal{R}^{\mathbb{I}\mathbb{I}} = (\mathcal{F}^{\mathbb{I}\mathbb{I}}, R^{\mathbb{I}\mathbb{I}})$ and $\varphi^{\mathbb{I}\mathbb{I}}$. Let $t, s \in \mathcal{T}(\mathcal{F}^{\mathbb{I}\mathbb{I}\sharp}_{\varphi^{\mathbb{I}\mathbb{I}}}, \mathcal{X}^{\sharp}_{\varphi^{\mathbb{I}\mathbb{I}}})$ s.t. $\tau = \tau(root(t))$. Then, $\langle \mathtt{quote}_\tau(t), \Lambda\rangle \overset{\sharp}{\to}{}^!_{\varphi^{\mathbb{I}\mathbb{I}},\mathcal{R}^{\mathbb{I}\mathbb{I}}} \langle s, \Lambda\rangle$ if and only if $\langle pos(t), \Lambda\rangle \overset{\sharp}{\to}{}^!_{\varphi^{\mathbb{I}\mathbb{I}},\mathcal{R}^{\mathbb{I}\mathbb{I}}} \langle s, \Lambda\rangle$.*

Proof. Straightforward because pos and $\mathtt{quote}_\tau$ modify the same symbols in $\mathcal{P}os_P(t) \cap \mathcal{P}os_{\mathcal{F}^{\varphi}_{\mathcal{R}}}(t)$ and, since φ is a standard strategy map, positive indices are always reduced before the root symbol. $\Box$

In the following theorem we prove completeness of the transformation, i.e. that the transformation preserves normal forms (while they can include extra symbols at some inner positions).

Definition 9.15 (Maximal constructor context) *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS and φ be an E-strategy map. The* maximal constructor context *$C_t[\ ]$ of a term $t \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$ is defined as: $C_t[\ ] = \Box$ if $root(erase(t)) \notin \mathcal{C}$; $C_t[\ ] = c(C_{t_1}[\ ], \ldots, C_{t_k}[\ ])$ if $root(erase(t)) = c \in \mathcal{C}$.*

Theorem 9.16 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map such that $\mathcal{R}$ is φ-terminating. Let $\mathcal{R}^{\mathbb{I}\mathbb{I}} = (\mathcal{F}^{\mathbb{I}\mathbb{I}}, R^{\mathbb{I}\mathbb{I}})$ and $\varphi^{\mathbb{I}\mathbb{I}}$. For all $t, s \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$, if $\langle t, \Lambda\rangle \overset{\sharp}{\to}{}^!_{\varphi,\mathcal{R}} \langle s, \Lambda\rangle$, then $\langle pos(t), \Lambda\rangle \overset{\sharp}{\to}{}^!_{\varphi^{\mathbb{I}\mathbb{I}},\mathcal{R}^{\mathbb{I}\mathbb{I}}} \langle s', \Lambda\rangle$ where $C_s = C_{s'}$ and $\forall p \in minimal_{\leq}(\mathcal{P}os_{\mathcal{F}^{\mathbb{I}\mathbb{I}}-\mathcal{F}}(s'))$, $s'|_p$ is a $\overset{\sharp}{\to}$-normal form.*

Proof. (Sketch) By induction on the length of the sequence $\langle t, \Lambda\rangle \overset{\sharp}{\to}{}^!_{\varphi,\mathcal{R}} \langle s, \Lambda\rangle$ and considering Proposition 9.13 and Lemma 9.14. $\Box$

Given a strategy map φ for $\mathcal{F}$, we say that a TRS $\mathcal{R} = (\mathcal{F}, R)$ is φ-terminating if, for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, there is no infinite $\overset{\sharp}{\to}_{\varphi}$-rewrite sequence starting from $\langle \varphi(t), \Lambda\rangle$. Note that Examples 9.2, 9.3, and 9.5 can be proved φ-terminating using the technique developed in Section 7.5 In the following theorem we prove completeness of the transformation, i.e. that the transformation preserves normal forms.

Theorem 9.17 (Completeness) *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map such that $\mathcal{R}$ is φ-terminating. Let $\mathcal{R}^{\mathbb{II}} = (\mathcal{F}^{\mathbb{II}}, R^{\mathbb{II}})$ and $\varphi^{\mathbb{II}}$. For all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $s \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, if $s \in eval^{\varphi}_{\mathcal{R}}(t)$, then $s \in eval^{\varphi^{\mathbb{II}}}_{\mathcal{R}^{\mathbb{II}}}(t)$.*

Proof. By Theorem 9.16 and Lemma 9.14. □

On the other hand, to prove correctness, we provide a translation of the labeling of terms in $\mathcal{T}(\mathcal{F}^{\mathbb{II}\sharp}_{\varphi^{\mathbb{II}}}, \mathcal{X}^{\sharp}_{\varphi^{\mathbb{II}}})$ back to the labeling of $\mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$. Given two strategy lists $L, L' \in \mathbb{L}$, we denote by $L_{\uparrow L'}$ the maximal sublist L'' such that $L \sqsubseteq L''$ and $L'' \sqsubseteq L'$.

Definition 9.18 *Let φ be a strategy map over signature $\mathcal{F}$. Let $\varphi^{\mathbb{II}}$ be a strategy map over signature $\mathcal{F}^{\mathbb{II}}$. Let $t \in \mathcal{T}(\mathcal{F}^{\mathbb{II}\sharp}_{\varphi^{\mathbb{II}}}, \mathcal{X}^{\sharp}_{\varphi^{\mathbb{II}}})$. We define the translation of t into terms $\mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$ as $rem(t) = s$ where $\mathcal{P}os(s) = \mathcal{P}os(t)$ and $\forall q \in \mathcal{P}os(t)$,*

$$root(s|_q) = \begin{cases} f_{L'_1|L'_2} \ \text{if } t|_q = f_{L_1|L_2};\ \text{where } L'_1 = L_{1\uparrow\varphi(f)}, L'_2 = L_{2\uparrow\varphi(f)}, \\ \qquad \text{and } L_1 ++ L_2 \sqsubseteq \varphi(f) \\ f_{L'_1|L'_2} \ \text{if } t|_q = f_{root\,L_1|L_2};\ \text{where } L'_1 = L_{1\uparrow\varphi(f)}, L'_2 = L_{2\uparrow\varphi(f)}, \\ \qquad \text{and } L_1 ++ L_2 \sqsubseteq \varphi(f) \\ f_{L'_1|L'_2} \ \text{if } t|_q = f_{+i\,L_1|L_2};\ \text{where } L'_1 = \begin{cases} \texttt{(-i)}_{\uparrow\varphi(f)} & \text{if } \texttt{(-i)} \sqsubseteq \varphi(f) \\ \texttt{(i)}_{\uparrow\varphi(f)} & \text{if } \texttt{(i)} \sqsubseteq \varphi(f) \end{cases} \\ \qquad L'_2 = L_{2\uparrow\varphi(f)},\ \text{and } L_1 ++ L_2 \sqsubseteq \varphi(f) \end{cases}$$

Proposition 9.19 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map. Let $\mathcal{R}^{\mathbb{II}} = (\mathcal{F}^{\mathbb{II}}, R^{\mathbb{II}})$ and $\varphi^{\mathbb{II}}$. For all $t, s \in \mathcal{T}(\mathcal{F}^{\mathbb{II}\sharp}_{\varphi^{\mathbb{II}}}, \mathcal{X}^{\sharp}_{\varphi^{\mathbb{II}}})$ without symbols $\texttt{quote}_\tau$, and $q \in \mathcal{P}os_A(s)$, if $\langle t, \Lambda\rangle \stackrel{\sharp}{\to}_{\varphi^{\mathbb{II}},\mathcal{R}^{\mathbb{II}}} \langle s, q\rangle$, then $\langle rem(t), \Lambda\rangle \stackrel{\sharp}{\to}^{=}_{\varphi,\mathcal{R}} \langle rem(s), q\rangle$.*

Proof. Straightforward since cases of $\stackrel{\sharp}{\to}$ for positive strategy annotations are only considered. □

Correctness of the transformation is also proved without any condition on termination of the TRS.

Theorem 9.20 *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map. Let $\mathcal{R}^{\mathbb{II}} = (\mathcal{F}^{\mathbb{II}}, R^{\mathbb{II}})$ and $\varphi^{\mathbb{II}}$. For all $t, s \in \mathcal{T}(\mathcal{F}^{\mathbb{II}\sharp}_{\varphi^{\mathbb{II}}}, \mathcal{X}^{\sharp}_{\varphi^{\mathbb{II}}})$ without symbols $\texttt{quote}_\tau$, and $q \in \mathcal{P}os_A(s)$, if $\langle t, \Lambda\rangle \stackrel{\sharp}{\to}^{!}_{\varphi^{\mathbb{II}},\mathcal{R}^{\mathbb{II}}} \langle s, \Lambda\rangle$, then $\langle rem(t), \Lambda\rangle \stackrel{\sharp}{\to}^{!}_{\varphi,\mathcal{R}} \langle s', q\rangle$ where $C_s = C_{s'}$ and $\forall p \in minimal_{\leq}(\mathcal{P}os_{\mathcal{F}^{\mathbb{II}}-\mathcal{F}}(s))$, $s'|_p$ is a $\stackrel{\sharp}{\to}$-normal form.*

Proof. Similar to Theorem 9.16 but using Proposition 9.19 and without any condition on termination. □

Theorem 9.21 (Correctness) *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map. Let $\mathcal{R}^{\mathbb{II}} = (\mathcal{F}^{\mathbb{II}}, R^{\mathbb{II}})$ and $\varphi^{\mathbb{II}}$. For all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $s \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, if $s \in eval^{\varphi^{\mathbb{II}}}_{\mathcal{R}^{\mathbb{II}}}(t)$, then $s \in eval^{\varphi}_{\mathcal{R}}(t)$.*

Proof. By Theorem 9.20 and Lemma 9.14. □

Finally, termination is preserved by the transformation.

Theorem 9.22 (Termination) *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map. $\mathcal{R}$ is φ-terminating iff $\mathcal{R}^{\mathbb{II}}$ is $\varphi^{\mathbb{II}}$-terminating.*

Proof. By Propositions 9.13 and 9.19. □

9.3 Experiments

A prototype implementation of the transformation proposed in this chapter has been integrated in the OnDemandOBJ prototype of Section 7.6. In this section, we reuse the experimental results of Section 7.6 to compare the improvement obtained by the program transformation.

Tables 9.1 and 9.2 show the runtimes[3] in milliseconds and the number of evaluation steps of the benchmark programs and the result of the program transformation for the different OBJ-family systems. Tables 9.1 and 9.2 expand the results of Tables 7.1 and 7.2, respectively. The OnDemandOBJ interpreter is the on-demand prototype interpreter of the on-demand evaluation of Chapter 7 CafeOBJ[4] (we use version 1.4.6) is developed in Lisp at the Japan Advanced Inst. of Science and Technology (JAIST); OBJ3[5] (we use version 2.0), also written in Lisp, is maintained by the University of California at San Diego; Maude[6] (we use version 1.0.5) is developed in C++ and maintained by the University of Illinois at Urbana-Champaign. OBJ3 and Maude provide only computations with positive annotations whereas CafeOBJ provides computations with negative annotations as well, using the on-demand evaluation of [Ogata and Futatsugi, 2000; Nakamura and Ogata, 2001]. OnDemandOBJ computes with negative annotations using the on-demand evaluation of Chapter 7. Note that CafeOBJ and OBJ3 implement sharing of variables whereas Maude and OnDemandOBJ do not; thus, the number of evaluation steps in Table 9.2 is pairwise equivalent: CafeOBJ and OBJ3 in one hand and Maude and OnDemandOBJ in the other hand. Also, since Maude is implemented in C++, typical execution times are nearly 0 milliseconds. Finally, the mark *overflow* in Table 9.2 indicates that execution raised a memory overflow and normal form was not achieved; whereas the mark *unavailable* in Tables 9.1 and 9.2 indicates that the program can not be executed in such OBJ implementation.

The benchmark `pi_noneg` consists of the application of the program transformation described in this chapter to the benchmark `pi` introduced in Appendix B-1. Table 9.1 compares the evaluation of expression `pi(square(square(3)))` using `pi` and

[3] The average of 10 executions measured in a Pentium III machine running redhat 7.2.
[4] Available at `http://www.ldl.jaist.ac.jp/Research/CafeOBJ/system.html`.
[5] Available at `http://www.kindsoftware.com/products/opensource/obj3/OBJ3/`.
[6] Available at `http://maude.cs.uiuc.edu/`.

ms./rewrites	`pi`	`pi_noneg`
OnDemandOBJ	25/364	215/35532
CafeOBJ	30/364	190/35532
OBJ3	*unavailable*	100/35532
Maude	*unavailable*	30/35532

Table 9.1: Execution of call `pi(square(square(3)))`

ms./rewrites	`msquare_eager`	`msquare_apt`	`msquare_neg`	`msquare_noneg`
OnDemandOBJ	33/ 715	62/ 1640	0/ 1	0/ 4
	40/ 914	78/ 1992	80/ 1992	750/126089
CafeOBJ	40/ 715	50/ 715	0/ 1	0/ 4
	50/ 914	60/ 914	60/ 914	630/126089
OBJ3	20/ 715	*overflow*	*unavailable*	0/ 4
	30/ 914	*overflow*	*unavailable*	*overflow*
Maude	0/ 715	0/ 1640	*unavailable*	0/ 4
	0/ 914	3/ 1992	*unavailable*	90/126089

Table 9.2: Execution of terms `minus(0,square(square(5)))` and `minus(square(square(5)),square(square(3)))`

`pi_noneg`. Note that the right input expression for `pi_noneg` is `quoteLRecip(pi(square(square(3))))`. It witnesses that negative annotations are extremely useful in practice and that the program transformation enables the execution of negatively annotated programs in all OBJ implementations. On the other hand, Table 9.1 also evidences that the implementation of the on-demand evaluation strategy in other systems is quite promising.

On the other hand, Table 9.2 illustrates the interest of using negative annotations to improve the behavior of programs. The benchmark `msquare_noneg` represents the application of the program transformation to `msquare_neg`. Note that the right input expressions for `msquare_noneg` are `quoteNat(minus(0,square(square(5))))` and `quoteNat(minus(square(square(5)),square(square(3))))`. Then, for instance, program `msquare_neg` runs in less time and requires a smaller number of rewrite steps than `msquare_eager` or `msquare_apt`, which do not include negative annotations. Note the difference in the number of rewrite steps of benchmarks `msquare_eager` and `msquare_apt` for the Maude and OnDemandOBJ systems, which is due to the absence of variable sharing. Moreover, note that the program transformation is also very useful since execution of the expression `minus(0,square(square(5)))` is improved.

9.4 Conclusions

This chapter presents a contribution to the extension of evaluation strategies for functional languages of the OBJ family with evaluation on demand, thereby introducing a flavour of laziness into such languages. Our proposal is based on a program transformation for OBJ programs which achieves correctness and works well in current OBJ interpreters. The main technical results of this work are as follows:

- The proposed transformation preserves the termination of the original program which uses (positive and) negative annotations. That is, if the original program terminates under the evaluation strategy of Chapter 7, then the transformed program terminates also.

- Correct and completeness of the transformation holds w.r.t. the semantics of strategy annotations given in Chapter 7. That is, the semantics of input expressions in the original program (under the on-demand E-strategy of Chapter 7) and in the transformed program (under the E-strategy) do coincide.

Moreover, our transformation is useful both for

1. making possible the use of arbitrary strategy annotations in languages that (syntactically) allow them but that still do not provide the necessary operational support (e.g., OBJ3).

2. providing a notion of negative strategy annotation (somewhat laziness) for languages that do not allow them (e.g., Maude).

and hence we think that our work contributes to foster the use of OBJ in programming. As future work, we plan to formally determine the overhead associated to the evaluation in the transformed program.

Part III

Analysis Techniques

Chapter 10

Redundant Arguments

In this chapter, we address the definition of program analysis techniques for detecting inefficiencies in the form of redundant arguments in the functions defined in a program. Redundancy of a parameter means that replacing it by any expression does not change the result. In Section 10.1, we briefly introduce the notion of redundancy of an argument and motivate the usefulness of detecting and eliminating redundant arguments. In Section 10.2, we consider different (reduction) semantics, including the standard normalization semantics (typical of pure rewriting) and the evaluation semantics (closer to functional programming). In Section 10.3, we provide a semantic characterization of redundancy which is parametric w.r.t. the observed semantics S. In Section 10.4, we derive a decidability result for the redundancy problem w.r.t. S using the "(W)SkS approach", which proves that redundancy is decidable for left-linear right-ground TRSs over finite signatures. In Section 10.5, we provide an alternative view of the redundancy of an argument and reformulate the redundancy problem in terms of the inductive theory of a TRS. Inefficiencies caused by the redundancy of arguments cannot be avoided by using standard rewriting strategies. Therefore, in Section 10.6 we formalize an elimination procedure which gets rid of the useless arguments and provide sufficient conditions for the preservation of the semantics. We conclude with an extensive comparison with the related literature in Section 10.7.

A short version of this chapter appeared in [Alpuente et al., 2000, 2002a,e]

10.1 Introduction

A number of researchers have noticed that certain processes of optimization, transformation, specialization and reuse of code often introduce anomalies in the generated code that programmers usually (or ideally) do not write [Aho *et al.*, 1986; Hughes, 1988; Leuschel and Sørensen, 1996; Liu and Stoller, 2002]. Examples are redundant arguments in the functions defined by the program, as well as useless program rules. The notion of redundant argument means that replacing it by whatever expression we like, the final result does not change; independently of actual computations. The following example motivates our ideas.

Example 10.1 *Consider the following program, which calculates the last element of a list and the concatenation of two lists of natural numbers, respectively:*

```
append nil    y = y                  last (x:nil)  = x
append (x:xs) y = x:(append xs y)    last (x:y:ys) = last (y:ys)
```

Assume that we specialize this program for the call `(applast ys z)` $\equiv$ `(last (append ys (z:nil)))`*, which appends an element* `z` *at the end of a given list* `ys` *and then returns the last element,* `z`*, of the resulting list (the example is borrowed from DPPD library of benchmarks [Leuschel, 1998] and was also considered in [Leuschel and Martens, 1995; Pettorossi and Proietti, 1996b] for logic program specialization). Commonly, the optimized program which can be obtained by using an automatic specializer of functional programs such as [Alpuente et al., 1997, 1998, 1999e] is:*

```
applast nil    z = z
applast (x:xs) z = lastnew x xs z
lastnew x nil    z = z
lastnew x (y:ys) z = lastnew y ys z
```

The first argument of the function `applast` *is redundant (as well as the first and second parameters of the auxiliary function* `lastnew`*) and would not typically be written by a programmer who writes this program by hand. This program is too far from* {`applast′ ys z = lastnew′ z`, `lastnew′ z = z`}*, a more feasible program with the same evaluation semantics, or even the "optimal" program –without redundant parameters–* {`applast″ z = z`} *which one would ideally expect (here the rule for the "local" function* `lastnew′` *is disregarded, since it is not useful when the definition of* `applast′` *is optimized). Note that standard (post-specialization) renaming/compression procedures cannot perform this optimization as they only improve programs where program calls contain dead functors or multiple occurrences of the same variable, or the functions are defined by rules whose rhs's are normalizable [Alpuente et al., 1997; Gallagher, 1993; Glück and Sørensen, 1994].*

It seems interesting to formalize program analysis techniques for detecting these kinds of redundancies as well as to formalize transformations for eliminating the dead code which appears in the form of redundant function arguments or useless rules and which, in some cases, can be safely erased without jeopardizing correctness. In this thesis, we investigate the problem of redundant arguments in term rewriting systems (TRSs), as a model for the programs that can be written in more sophisticated languages.

Strictness analysis[1] [Burn *et al.*, 1986; Burn, 1991; Jensen, 1991; Mycroft, 1980; Mycroft and Norman, 1992; Sekar *et al.*, 1990; Wadler and Hughes, 1987] determines whether an argument evaluation is "strictly" necessary for lazy evaluation and it is closely related to the notion of argument neededness (see Section 2.5). The counterpart of strictness analysis has also been studied in a number of different analysis techniques such as *dead code analysis* [Liu and Stoller, 2002], *unneededness analysis* [Hughes, 1988], *absence analysis* [Cousot and Cousot, 1994], *filtering analysis* [Leuschel and Sørensen, 1996], or *useless analysis* [Wand and Siveroni, 1999]. Indeed, unneeded terms w.r.t. a program are understood as *computationally irrelevant* terms, since their evaluation is not "needed" w.r.t. lazy evaluation and these terms can be safely erased in order to shrink the associated reduction space of the program.

Other similar notions have been widely used during last decades to detect and remove parts of a program which are computationally irrelevant. Examples are program specialization [Alpuente *et al.*, 1997, 1998, 1999e; Leuschel and Martens, 1995; Pettorossi and Proietti, 1994, 1996a], slicing [Gouranton, 1998; Schoenig and Ducasse, 1996; Reps and Turnidge, 1996; Szilagyi *et al.*, 2002; Tip, 1995; Weiser, 1984], and compile-time garbage collection [Jones and Métayer, 1989; Park and Goldberg, 1992; Knoop *et al.*, 1994].

However, traditionally, no method has considered redundancy, or *semantically irrelevant* terms in contrast to computationally irrelevant terms. This concept is clearly independent of the notion of dead code since a subterm can be needed (computationally relevant) though redundant (semantically irrelevant). For instance, known procedures for removing dead code such as [Berardi *et al.*, 2000; Kobayashi, 2000; Liu and Stoller, 2002] do not apply to Example 10.1.

Moreover, the reader should note that no reduction strategy can dodge the problem since the evaluation of a redundant subterm can be strictly necessary to achieve the canonical form of the input term w.r.t. the associated semantics; though such canonical form does not depend on the redundant subterm. In other words, *computationally irrelevant* terms (dead code) are always *semantically irrelevant* (redundant) whereas the opposite does not hold. The following example illustrates this fact.

[1] Let $D_1, \ldots, D_k, D$ be ordered sets with least elements $\bot_1, \ldots, \bot_k, \bot$ respectively, expressing undefinedness. A mapping $f : D_1 \times \cdots \times D_k \to D$ is said to be *strict* in its i-th argument if $f(d_1, \ldots, \bot_i, \ldots, d_k) = \bot$ for all $d_1 \in D_1, \ldots, d_k \in D_k$.

Example 10.2 *Consider the optimized program of Example 10.1 extended with:*

```
take 0     xs     = []
take (s n) (x:xs) = x:take n xs
```

The contraction of redex `(take 1 (1:2:[]))` *at position* 1 *in the term* $t =$`applast (take 1 (1:2:[])) 0` *(we use* 1*,* 2 *instead of* `S 0`*,* `S (S 0)`*) is* needed *to normalize* t *(in the sense of Definition 2.3 of Chapter 2.6). However, the first argument of* `applast` *is redundant for normalization, as we showed in Example 10.1, and the program could be improved by dropping this useless parameter.*

In the following, we first consider different semantics which could be associated to a TRS since our notion of redundancy of an argument is parametric w.r.t. the observed semantics.

10.2 Semantics

The redundancy of an argument of a function f in a TRS $\mathcal{R}$ depends on the semantic properties of $\mathcal{R}$ that we are interested in observing. Our notion of semantics is aimed to couch operational as well as denotational aspects.

A *term semantics* for a signature $\mathcal{F}$ is a mapping $\mathsf{S} : \mathcal{T}(\mathcal{F}) \to \mathcal{P}(\mathcal{T}(\mathcal{F}))$ [Lucas, 2001c] which associates a set of canonical expressions to a term. A *rewriting semantics* for a TRS $\mathcal{R} = (\mathcal{F}, R)$ is a term semantics S for $\mathcal{F}$ such that, for all $t \in \mathcal{T}(\mathcal{F})$ and $s \in \mathsf{S}(t)$, $t \to^*_{\mathcal{R}} s$, i.e. a term semantics where the set of canonical expressions associated to a term are determined only by the program.

The semantics which is most commonly considered in functional programming is the set of values (ground constructor terms) that $\mathcal{R}$ is able to produce in a finite number of rewriting steps ($\mathsf{eval}_{\mathcal{R}}(t) = \{s \in \mathcal{T}(\mathcal{C}) \mid t \to^*_{\mathcal{R}} s\}$). Other kinds of semantics often considered for $\mathcal{R}$ are, e.g., the set of all possible reducts of a term which are reached in a finite number of steps ($\mathsf{red}_{\mathcal{R}}(t) = \{s \in \mathcal{T}(\mathcal{F}) \mid t \to^*_{\mathcal{R}} s\}$), the set of such reducts that are ground head-normal forms ($\mathsf{hnf}_{\mathcal{R}}(t) = \mathsf{red}_{\mathcal{R}}(t) \cap \mathsf{HNF}_{\mathcal{R}}$), or ground normal forms ($\mathsf{nf}_{\mathcal{R}}(t) = \mathsf{hnf}_{\mathcal{R}}(t) \cap \mathsf{NF}_{\mathcal{R}}$). We also consider the (trivial) semantics empty which assigns an empty set to every term. We often omit $\mathcal{R}$ in the notations for rewriting semantics when it is clear from the context. Furthermore, a rewriting semantics S for a TRS $\mathcal{R}$ is called *($\mathcal{R}$-)normalized* if, for all $t \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(t) \subseteq \mathsf{NF}_{\mathcal{R}}$, i.e. the semantics associates only normal forms to a term. eval and nf are examples of normalized semantics whereas hnf and red are not normalized.

The ordering $\preceq$ between semantics [Lucas, 2001c] provides some interesting properties regarding the redundancy of arguments. Given term semantics S, S' for a signature $\mathcal{F}$, we write $\mathsf{S} \preceq \mathsf{S}'$ if there exists $T \subseteq \mathcal{T}(\mathcal{F})$ such that, for all $t \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(t) = \mathsf{S}'(t) \cap T$. Note that, then, we have $\mathsf{empty} \preceq \mathsf{eval}_{\mathcal{R}} \preceq \mathsf{nf}_{\mathcal{R}} \preceq \mathsf{hnf}_{\mathcal{R}} \preceq \mathsf{red}_{\mathcal{R}}$.

Given a rewriting semantics S, it is interesting to determine whether S provides non-trivial information for every input expression. Let $\mathcal{R}$ be a TRS and S be a rewriting semantics for $\mathcal{R}$, we say that $\mathcal{R}$ is S-*defined* if for all $t \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(t) \neq \varnothing$ [Lucas, 2001c]. S-definedness is monotone w.r.t. $\preceq$: if $\mathsf{S} \preceq \mathsf{S}'$ and $\mathcal{R}$ is S-defined, $\mathcal{R}$ is also S'-defined.

S-definedness has already been studied in the literature for different semantics [Lucas, 2001c]. In concrete, the eval-definedness is related to the standard notion of *completely defined* (CD) TRSs (see [Kapur *et al.*, 1987; Kounalis, 1985]). This notion was already introduced in Chapter 2. Roughly speaking, a defined function symbol is completely defined if it does not occur in any ground term in normal form, that is to say that functions are reducible on all ground terms (of appropriate sort). A TRS $\mathcal{R}$ is completely defined if each defined symbol of the signature is completely defined. In one-sorted theories, completely defined programs occur only rarely. However, they are common when using types, and each function is defined for all constructors of its argument types.

Let $\mathcal{R}$ be a normalizing (i.e., every term has a normal form) and completely defined TRS; then, $\mathcal{R}$ is $\mathsf{eval}_{\mathcal{R}}$-defined. Being completely defined is sensitive to extra constant symbols in the signature, and so is redundancy; we are not concerned with modularity in this chapter.

From now on, we formulate the notion of a redundant argument and study some important properties.

10.3 Redundant Arguments

Roughly speaking, a redundant argument of a function f is an argument t_i which we do not need to consider in order to compute the semantics of any call containing a subterm $f(t_1, \ldots, t_k)$.

Definition 10.3 (Redundancy of an argument) *Let* S *be a term semantics for a signature* $\mathcal{F}$*,* $f \in \mathcal{F}$*, and* $i \in \{1, \ldots, ar(f)\}$*. The* i*-th argument of* f *is* redundant w.r.t. S *if, for all contexts* $C[\,]$ *and for all* $t, s \in \mathcal{T}(\mathcal{F})$ *such that* $root(t) = f$*,* $\mathsf{S}(C[t]) = \mathsf{S}(C[t[s]_i])$*.*

We denote by $rarg_{\mathsf{S}}(f)$ the set of redundant arguments of a symbol $f \in \mathcal{F}$ w.r.t. a semantics S for $\mathcal{F}$. Note that every argument of every symbol is redundant w.r.t. empty. Redundancy is antimonotone with regard to the ordering $\preceq$ on semantics.

Theorem 10.4 *Let* S, S' *be term semantics for a signature* $\mathcal{F}$*. If* $\mathsf{S} \preceq \mathsf{S}'$*, then, for all* $f \in \mathcal{F}$*,* $rarg_{\mathsf{S}'}(f) \subseteq rarg_{\mathsf{S}}(f)$*.*

Proof. Given $i \in rarg_{\mathsf{S}'}(f)$, we have that, for all context $C[\,]$ and for all $t, s \in \mathcal{T}(\mathcal{F})$ such that $root(t) = f$, $\mathsf{S}'(C[t]) = \mathsf{S}'(C[t[s]_i])$. Now, since $\mathsf{S} \preceq \mathsf{S}'$, there exists $T \subseteq \mathcal{T}(\mathcal{F})$ such that $\mathsf{S}(C[t]) = \mathsf{S}'(C[t]) \cap T = \mathsf{S}'(C[t[s]_i]) \cap T = \mathsf{S}(C[t[s]_i])$. Hence, $i \in rarg_{\mathsf{S}}(f)$. □

The following result guarantees that constructor symbols have no redundant arguments, which agrees with the common understanding of constructor terms as completely meaningful pieces of information.

Proposition 10.5 *Let $\mathcal{R}$ be a TRS such that $\mathcal{T}(\mathcal{C}) \neq \varnothing$, and consider* S *such that* $\mathsf{eval}_{\mathcal{R}} \preceq \mathsf{S}$. *Then, for all $c \in \mathcal{C}$, $rarg_{\mathsf{S}}(c) = \varnothing$.*

Proof. Let $t \in \mathcal{T}(\mathcal{C})$ be such that $root(t) = c$. If $i \in rarg_{\mathsf{eval}_{\mathcal{R}}}(c)$, then $\mathsf{eval}_{\mathcal{R}}(t) = \{t\} = \{t[t]_i\} = \mathsf{eval}_{\mathcal{R}}(t[t]_i)$ thus contradicting that $t \neq t[t]_i$. □

In the following section, we consider several aspects about decidability of the redundancy of an argument.

10.4 Decidability Issues

In general, the redundancy of an argument is undecidable. However, we are able to provide a decidability result about redundancy w.r.t. all the semantics considered in this thesis. In the following, for a signature $\mathcal{F}$, term semantics S for $\mathcal{F}$, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$, by "the redundancy problem w.r.t. S", we mean the redundancy of the i-th argument of f w.r.t. S.

We follow the "(W)SkS approach" to decide a given property P, which is based on ascertaining the conditions for expressing P in a *decidable* logic, namely *the (weak) second-order monadic logic with k successors* (W)SkS (see [Thomas, 1990]). The following theorem by Rabin is the key element for our results in this section.

Theorem 10.6 [Rabin, 1969] *The (weak) monadic second-order theory of k successor functions* (W)SkS *is decidable.*

First, we recall some basic definitions about the WSkS logic (see e.g. [Thomas, 1990]). *Terms* of the WSkS logic are formed out of individual variables $x, y, z, \ldots$, the empty string Λ, and right concatenation with $1, \ldots, k$. Atomic formulas are equations between terms, inequations $w < w'$ between terms, or expressions $w \in X$ where w is a term and X is a (second-order) variable. Formulas are built from atomic formulas using the logical connectives $\wedge, \vee, \Rightarrow, \neg, \ldots$ and the quantifiers $\exists, \forall$ of both individual and second-order variables. Individual variables are interpreted as elements of $\{1, \ldots, k\}^*$ and second-order variables as finite subsets of $\{1, \ldots, k\}^*$. Equality is the string equality and inequality is the strict prefix ordering. Finite union and intersection, as well as inclusion and equality of sets, are definable in WSkS in an obvious way.

Let us relate TRSs and WSkS logic. Given a finite signature $\mathcal{F}$, let k be the maximal arity of a function symbol in $\mathcal{F}$ and n be the cardinality of $\mathcal{F}$. A term t is represented in WSkS using $n+1$ set variables X and $X_f, f \in \mathcal{F}$, which are denoted by $\vec{X}$ in the following. X will be the set of all positions of t, and X_f will be the set of positions that are labeled with the corresponding function symbol. The following WSkS formula expresses that $\vec{X}$ encodes a term in $\mathcal{T}(\mathcal{F})$ [Comon, 2000; Durand and Middeldorp, 1997]:

$$\begin{array}{rl} Term_{\mathcal{F}}(\vec{X}) \stackrel{def}{=} & X = \bigcup_{i=1}^{n} X_{f_i} \wedge \bigwedge_{i \neq j}(X_{f_i} \cap X_{f_j} = \varnothing) \wedge \ \forall x \in X \ \forall y < x \, (y \in X) \\ & \wedge \ \bigwedge_{f \in \mathcal{F}}(\forall x \in X_f : \bigwedge_{l=1}^{ar(f)}(x.l \in X) \wedge \bigwedge_{l=ar(f)+1}^{k}(x.l \notin X)) \end{array}$$

If $Term_{\mathcal{F}}(\vec{T})$ holds, then we let $t_{\vec{T}}$ define the term in $\mathcal{T}(\mathcal{F})$ which is uniquely determined by $\mathcal{P}os(t) = T$ and $root(t|_p) = f$ if $p \in T_f$ for all $p \in T$. A subset $L \subseteq \mathcal{T}(\mathcal{F})$ is called WSkS *definable* if there exists a WSkS formula Φ with free variables $\vec{T}$ such that $L = \{t_{\vec{T}} \mid Term_{\mathcal{F}}(\vec{T}) \wedge \Phi(\vec{T})\}$.

An arbitrary term semantics S can be encoded as a relation $\mathcal{S}$ between terms: $\mathcal{S} = \{(t, s) \mid t \in \mathcal{T}(\mathcal{F}) \wedge s \in \mathsf{S}(t)\}$. Hence, we say that semantics S is WSkS definable if there exists an WSkS formula Φ with free variables $\vec{T}$ and $\vec{S}$ such that $(t_{\vec{T}}, s_{\vec{S}}) \in \mathcal{S} \Leftrightarrow Term_{\mathcal{F}}(\vec{T}) \wedge Term_{\mathcal{F}}(\vec{S}) \wedge \Phi(\vec{T}, \vec{S})$.

Lemma 10.7 *For terms t, t', and position $p \in \mathcal{P}os(t) \cap \mathcal{P}os(t')$, the predicate $equiv(t, t', p) \equiv \exists s \in \mathcal{T}(\mathcal{F}).t' = t[s]_p$ is* WSkS *definable.*

Proof. Assume that the term t is represented by $\vec{T}$, and the term t' is represented by $\vec{T'}$. Then:

$$equiv(\vec{T}, \vec{T'}, p) \stackrel{def}{=} \forall q.(\neg(p \leq q) \Rightarrow \bigwedge_{f \in \mathcal{F}}(q \in T_f \Leftrightarrow q \in T'_f))$$

□

Now, we give an alternative characterization of redundancy of an argument in order to prove Theorem 10.9 below.

Proposition 10.8 *Let S be a term semantics for a signature $\mathcal{F}$, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$. The i-th argument of f is* redundant w.r.t. S *if, for all term $t \in \mathcal{T}(\mathcal{F})$, $\forall p \in \mathcal{P}os(t).root(t|_p) = f$ and $\forall t' \in \mathcal{T}(\mathcal{F})$ such that $equiv(t, t', p.i)$ is true, $\mathsf{S}(t) = \mathsf{S}(t')$.*

Proof. Immediate. □

Now, we can provide a decidability result using this alternative formulation of redundancy.

Theorem 10.9 (Decidability of redundancy) *Let* S *be a term semantics for a signature* $\mathcal{F}$*. If* S *is WSkS definable, then redundancy w.r.t.* S *is decidable.*

Proof. By using Proposition 10.8. Assuming that S is defined in WSkS by the formula $\Phi(\vec{X}, \vec{Y})$, redundancy is WSkS definable by using the following formula:

$$\begin{array}{l} \forall \vec{T}\ \forall \vec{S}\ \forall \vec{W}\ \forall p \in T.(Term(\vec{T}) \wedge Term(\vec{S}) \wedge Term(\vec{W}) \wedge \\ \quad p \in T_f \wedge equiv(\vec{T}, \vec{S}, p.i)\ \Rightarrow (\Phi(\vec{T}, \vec{W}) \Leftrightarrow \Phi(\vec{S}, \vec{W}))) \end{array}$$

Now, by Theorem 10.6, redundancy is decidable. □

Decidability of redundancy is *antimonotone* with regard to the ordering $\preceq$ on semantics.

Proposition 10.10 *Let* S, S' *be term semantics for a signature* $\mathcal{F}$*. If* $\mathsf{S} \preceq \mathsf{S}'$*,* S' *is WSkS definable, and there exists a window set of* S' *w.r.t.* S *which is WSkS definable, then* S *is WSkS definable.*

Proof. If $\mathsf{S} \preceq \mathsf{S}'$, then by properties of $\preceq$ there exists a set $T \subseteq \mathcal{T}(\mathcal{F})$ such that, for all $t \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(t) = \mathsf{S}'(t) \cap T$. Thus, assuming that semantics S' is defined by the WSkS formula $\Phi'(\vec{X}, \vec{Y})$, and that set T is defined by the WSkS formula $\Omega(\vec{X})$, we build the following formula defining S: $\Phi(\vec{X}, \vec{Y}) \stackrel{def}{=} \Phi'(\vec{X}, \vec{Y}) \wedge \Omega(\vec{Y})$ □

In Dauchet et al. [Dauchet *et al.*, 1990, 1987], ground (finite) tree transducers (GTT for short) were introduced to recognize the rewrite relation $\to^*_{\mathcal{R}}$ in (left-linear and right-)ground TRSs. Since GTT-recognizable relations are definable in WSkS [Comon, 2000], the semantics red is also WSkS definable, hence the redundancy w.r.t. red is decidable.

Theorem 10.11 *The set* $\mathsf{HNF}_{\mathcal{R}}$ *of a finite left-linear, right-ground TRS* $\mathcal{R}$ *is WSkS definable.*

Proof. Since the set $\mathsf{REDEX}_{\mathcal{R}}$ of all redexes of a TRS $\mathcal{R}$ is WSkS definable [Gallier and Book, 1985], and the set $(\to^*_{\mathcal{R}})[L] = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in L.t \to^*_{\mathcal{R}} s\}$ is WSkS definable[2] for any regular set of terms L. We can formulate the set $\mathsf{HNF}_{\mathcal{R}}$ as: $\Phi(\vec{X}) = \neg\Omega(\vec{X})$, where the set $(\to^*_{\mathcal{R}})[\mathsf{REDEX}_{\mathcal{R}}]$ is defined by the predicate $\Omega(\vec{X})$. □

The following theorem provides the first decidability result w.r.t. all the semantics considered in this thesis.

Theorem 10.12 (Decidability for semantics $\mathsf{red}_{\mathcal{R}}$, $\mathsf{hnf}_{\mathcal{R}}$, $\mathsf{nf}_{\mathcal{R}}$, and $\mathsf{eval}_{\mathcal{R}}$) *For a left-linear, right-ground TRS* $\mathcal{R} = (\mathcal{F}, R)$ *over a finite signature* $\mathcal{F}$*, the redundancy w.r.t. semantics* $\mathsf{red}_{\mathcal{R}}$*,* $\mathsf{hnf}_{\mathcal{R}}$*,* $\mathsf{nf}_{\mathcal{R}}$*, and* $\mathsf{eval}_{\mathcal{R}}$ *is decidable.*

[2] Actually, the set $(\to^*_{\mathcal{R}})[L]$ is recognizable for every recognizable tree language L [Comon, 2000]. Hence, by [Thatcher and Wright, 1968] it is WSkS definable.

Proof. Since the semantics $\mathsf{red}_{\mathcal{R}}$ is WSkS definable, and the sets $\mathsf{NF}_{\mathcal{R}}$ and $\mathcal{T}(\mathcal{C})$ are WSkS definable, by Proposition 10.10 we obtain that the semantics $\mathsf{nf}_{\mathcal{R}}$ and $\mathsf{eval}_{\mathcal{R}}$ are WSkS definable. Then, by considering also Theorem 10.11 we obtain that the semantics $\mathsf{hnf}_{\mathcal{R}}$ is WSkS definable. Finally, by Theorem 10.9, redundancy is decidable for semantics $\mathsf{red}_{\mathcal{R}}$, $\mathsf{hnf}_{\mathcal{R}}$, $\mathsf{nf}_{\mathcal{R}}$, and $\mathsf{eval}_{\mathcal{R}}$. □

This result recalls the decidability of other related properties of TRSs, such as confluence, joinability, and reachability problems (for left-linear, right-ground TRSs) [Dauchet *et al.*, 1987; Oyamaguchi, 1990]. For instance, the confluence problem was shown to be undecidable for right-ground TRSs, while it is decidable for ground TRSs and also for left-linear and right-ground TRSs [Dauchet *et al.*, 1987]. Note that we cannot weaken in our approach the requirement of right-groundness in Theorem 10.12 to the more general conditions of shallowness [Comon, 2000] or growingness [Jacquemard, 1996] as the induced rewrite relations are not expressible in the logic WSkS that we use to decide the property [Durand and Middeldorp, 1997].

In the following section, we address the redundancy problem from an alternative perspective. Rather than going more deeply in the decidability issues, we are interested in ascertaining conditions which (sufficiently) ensure that an argument is redundant in a given TRS. Thus, we reformulate the redundancy problem in terms of the inductive theory of a TRS.

10.5 Inductive Theorems expressing Redundancy of Arguments

An equation is a formula of the form $r = s$ (or $s = r$) where $r, s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. An *equational system* is a set of equations. If E is a set of equations between terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $\leftrightarrow_E^*$ is the smallest congruence on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $\sigma(s) \leftrightarrow_E^* \sigma(t)$ for all equations $s = t \in E$ and for all substitutions σ. Given a set of equations (or rewrite rules) E, $s = t$ is a *logical consequence* of E, denoted by $E \vdash s = t$, if $s \leftrightarrow_E^* t$. The *equational theory* of E is the set of equations that are logical consequences of E. The *minimal Herbrand model* (often called minimal model) $\mathcal{I}_E$ of a set of equations E is the quotient algebra $\mathcal{T}(\mathcal{F})/_{\leftrightarrow_E^*}$. We say that a first-order equation $s = t$ is an *inductive consequence* of a set of equations (or rewrite rules) E iff $\mathcal{I}_E \models s = t$, i.e. $\sigma(s) \leftrightarrow_E^* \sigma(t)$ for all grounding substitution σ for t and s; where a substitution θ is called grounding w.r.t. a term t if $\theta(t)$ is a ground term. The set of all inductive consequences of E is called the *inductive theory of* E. In the sequel, inductive consequences of E will also be called *inductive theorems*.

In order to formulate the redundancy of an argument in terms of the inductive theory of a program, let us just relax the notion of redundancy of an argument to the notion of "local redundancy" of an argument, which is simpler to test. Indeed, when analyzing a property of a function f in $\mathcal{R}$, it is useful to get rid of the contexts and perform easier, local analyses which allow us to center the attention on the syntactic structure of the rewriting rules. This motivates the following.

Definition 10.13 (Local redundancy of an argument) *Let* S *be a term semantics for a signature* $\mathcal{F}$, $f \in \mathcal{F}$, *and* $i \in \{1,\ldots,ar(f)\}$. *The* i*-th argument of* f *is* locally redundant w.r.t. S *if, for all* $t, s \in \mathcal{T}(\mathcal{F})$ *such that* $root(t) = f$, $\mathsf{S}(t) = \mathsf{S}(t[s]_i)$.

We denote by $lrarg_{\mathsf{S}}(f)$ the set of locally redundant arguments of a symbol f w.r.t. S. Note that all results in Sections 10.3 apply to *local* redundancy as well.

Theorem 10.14 *Let* S, S' *be two term semantics for a signature* $\mathcal{F}$. *If* $\mathsf{S} \preceq \mathsf{S}'$, *then, for all* $f \in \mathcal{F}$, $lrarg_{\mathsf{S}'}(f) \subseteq lrarg_{\mathsf{S}}(f)$.

Proof. Similar to Theorem 10.4. $\square$

Corollary 10.15 *Let* $\mathcal{R}$ *be a TRS such that* $\mathcal{T}(\mathcal{C}) \neq \varnothing$, *and consider* S *such that* $\mathsf{eval}_{\mathcal{R}} \preceq \mathsf{S}$. *For all* $c \in \mathcal{C}$, $lrarg_{\mathsf{S}}(c) = \varnothing$.

Redundancy of an argument w.r.t. a semantics S implies local redundancy w.r.t. S, i.e., $rarg_{\mathsf{S}}(f) \subseteq lrarg_{\mathsf{S}}(f)$. Unfortunately, the converse statement is not generally true, as the following example shows.

Example 10.16 *Consider the following TRS* $\mathcal{R}$:

$\texttt{from(n)} \to \texttt{n:from(s(n))} \qquad \texttt{head(x:xs)} \to \texttt{x}$

Assume that the constant `0` *belongs to the signature. According to Definition 10.13, the argument of* `from` *is locally redundant w.r.t.* $\mathsf{eval}_{\mathcal{R}}$, *since there is no derivation issuing from a term rooted by* `from` *which could produce a value in* $\mathcal{R}$. *However, the argument of the function* `from` *is not redundant w.r.t.* $\mathsf{eval}_{\mathcal{R}}$ *since, using the context* `head(□)`, *the values in* $\mathcal{R}$ *of the calls* `head(from(0))` *and* `head(from(s(0)))` *are different. Finally, note that* $\mathcal{R}$ *is not* $\mathsf{eval}_{\mathcal{R}}$*-defined.*

The above example suggests that S-definedness is a critical condition for redundancy to boil down to local redundancy. The following example reveals that the additional requirement that S is $\mathcal{R}$-normalized must also be satisfied.

Example 10.17 *Consider the following (confluent) TRS* $\mathcal{R}$:

$\texttt{f(g(0))} \to \texttt{f(0)} \qquad \texttt{g(x)} \to \texttt{0}$

and the semantics Λ-red *defined by:* $\Lambda\text{-}\mathsf{red}(t) = \{s \mid t \stackrel{\Lambda}{\to} s\}$ *if* t *is a redex, and* $\Lambda\text{-}\mathsf{red}(t) = \{t\}$, *otherwise. Note that* $\mathcal{R}$ *is trivially* Λ-red*-defined; however,* Λ-red *is not* $\mathcal{R}$*-normalized. Now, the argument of* `g` *is locally redundant w.r.t.* Λ-red, *whereas it is not redundant, since* $\texttt{f(0)} \in \Lambda\text{-}\mathsf{red}(\texttt{f(g(0))})$ *but* $\texttt{f(0)} \notin \Lambda\text{-}\mathsf{red}(\texttt{f(g(g(0)))})$.

Given a set of terms D, we write $C[D]$ instead of $\{C[\delta] \mid \delta \in D\}$ and we extend a semantic function S from terms to sets of terms in the obvious way. The following result establishes that the inclusion of a term in a context preserve its semantics if the TRS is confluent and defined, and the semantics is normalized and finitary.

Theorem 10.18 *Let $\mathcal{R}$ be a confluent TRS and S be an $\mathcal{R}$-normalized finitary semantics for $\mathcal{R}$. If $\mathcal{R}$ is S-defined, then, for all context $C[\ ]$ and term $t \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(C[t]) = \mathsf{S}(C[\mathsf{S}(t)])$.*

Proof. By S-definedness, $\mathsf{S}(t) \neq \varnothing$. Let $\delta_t \in \mathsf{S}(t)$ and $\delta \in \mathsf{S}(C[\delta_t])$. Since $t \to^* \delta_t$ we have that $C[t] \to^* C[\delta_t] \to^* \delta$. By confluence, δ is the only normal form of $C[t]$. Since S is $\mathcal{R}$-normalized and $\mathcal{R}$ is S-defined, it must be $\delta \in \mathsf{S}(C[t])$. On the other hand, let $\delta_t \in \mathsf{S}(t)$ and $\delta \in \mathsf{S}(C[t])$. Thus, $C[t] \to^* \delta$ and $C[t] \to^* C[\delta_t]$. By confluence and since S is $\mathcal{R}$-normalized, $C[\delta_t] \to^* \delta$. Thus, $\delta \in \mathsf{S}(C[\delta_t])$ and the conclusion follows. $\square$

We are able to ascertain the conditions to reduce the problem of redundancy of an argument to the simpler notion of local redundancy.

Theorem 10.19 *Let $\mathcal{R}$ be a confluent TRS and S be an $\mathcal{R}$-normalized semantics for $\mathcal{R}$. If $\mathcal{R}$ is S-defined, then $lrarg_{\mathsf{S}} = rarg_{\mathsf{S}}$.*

Proof. Since $rarg_{\mathsf{S}}(f) \subseteq lrarg_{\mathsf{S}}(f)$, we only need to prove that $lrarg_{\mathsf{S}}(f) \subseteq rarg_{\mathsf{S}}(f)$, i.e., for all context $C[\,]$ and $\forall t, s \in \mathcal{T}(\mathcal{F})$ such that $root(t) = f$, $\mathsf{S}(C[t]) = \mathsf{S}(C[t[s]_i])$. By using Theorem 10.18 and, since i is locally redundant, $\mathsf{S}(C[t]) = \mathsf{S}(C[\mathsf{S}(t)]) = \mathsf{S}(C[\mathsf{S}(t[s]_i)]) = \mathsf{S}(C[t[s]_i])$. $\square$

By Theorem 10.19, in a confluent, normalizing, and CD TRS $\mathcal{R}$, redundancy and local redundancy w.r.t. $\mathsf{eval}_{\mathcal{R}}$ coincide since $\mathcal{R}$ is $\mathsf{eval}_{\mathcal{R}}$-defined and $\mathsf{eval}_{\mathcal{R}}$ is $\mathcal{R}$-normalized.

Note that in Example 10.16 $\mathcal{R}$ is not $\mathsf{nf}_{\mathcal{R}}$-defined (hence, not $\mathsf{eval}_{\mathcal{R}}$-defined) since no term `from`(t) has a normal form w.r.t. $\mathcal{R}$. Also note that confluence cannot be dropped in Theorem 10.19.

Example 10.20 *Let $\mathcal{R}$ be the (non-confluent) terminating TRS*

$$\mathtt{zero(0)} \to \mathtt{0} \qquad \mathtt{zero(1)} \to \mathtt{0} \qquad \mathtt{zero(start(x))} \to \mathtt{0} \qquad \mathtt{start(zero(0))} \to \mathtt{0}$$

Since $\mathcal{R}$ is terminating, it is $\mathsf{nf}_{\mathcal{R}}$-defined. According to Definition 10.13, the argument of the function `zero` is locally redundant w.r.t. $\mathsf{nf}_{\mathcal{R}}$, since any derivation issuing from a term rooted by `zero` evaluates to `0`. However, the argument is not redundant when we consider arbitrary contexts since, e.g., the call `start(zero(1))` does not evaluate to `0`, whereas `start(zero(0))` does.

Now the question of how to single out locally redundant arguments arises. In order to tackle this problem, we formalize the redundancy problem in terms of the inductive theory of the program. Note that we focus on semantics eval since by Theorem 10.4, the more restrictive a semantics is, the more redundancies there are for the arguments of function symbols and according to our hierarchy of semantics (by $\preceq$), eval seems to be the most fruitful semantics for analyzing redundant arguments.

The following results formalize the relation between inductive theorems and redundancy of arguments in confluent and eval-defined TRSs. Namely, the redundancy of i-th argument of symbol $f \in \mathcal{D}$ w.r.t. semantics eval can be gained from the validity of the inductive theorem $f(x_1, \ldots, x_{ar(f)}) = f(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_{ar(f)})$ where $x_1, \ldots, x_{ar(f)}, y \in \mathcal{X}$.

Proposition 10.21 *Let $\mathcal{R}$ be a confluent TRS, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$. The i-th argument of f is* locally redundant *(w.r.t.* eval*) if the equation $t = t[y]_i$ is an inductive theorem of $\mathcal{R}$, where $t = f(x_1, \ldots, x_{ar(f)})$ and $x_1, \ldots, x_{ar(f)}, y$ are distinct variables.*

Proof. If $\mathcal{I}_\mathcal{R} \models t = t[y]_i$, then, by confluence of $\mathcal{R}$, we have that for all $s \in \mathcal{T}(\mathcal{F})$ and for all grounding $\sigma \in Subst(\mathcal{T}(\mathcal{F}))$ for t, let $\sigma' = \{y \mapsto s\}$, $\sigma(t) \to^* w_\sigma$ and $\sigma' \circ \sigma(t[y]_i) \to^* w_\sigma$, i.e. $\sigma(t) \to^* w_\sigma$ and $\sigma(t[s]_i) \to^* w_\sigma$. Thus, by confluence, $\mathsf{eval}_\mathcal{R}(\sigma(t)) = \mathsf{eval}_\mathcal{R}(w_\sigma) = \mathsf{eval}_\mathcal{R}(\sigma(t[s]_i))$ for all $s \in \mathcal{T}(\mathcal{F})$ and for all grounding $\sigma \in Subst(\mathcal{T}(\mathcal{F}))$ for t. Hence, the conclusion follows. □

Theorem 10.22 *Let $\mathcal{R}$ be a confluent and sufficiently complete TRS, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$. The i-th argument of f is* redundant *(w.r.t.* eval*) iff the equation $t = t[y]_i$ is an inductive theorem of $\mathcal{R}$; where $t = f(x_1, \ldots, x_{ar(f)})$ and $x_1, \ldots, x_{ar(f)}, y$ are distinct variables.*

Proof. The “if” part is clear by Proposition 10.21 and Theorem 10.19. For the “only if” part, we have that, since redundancy implies local redundancy, thus $\mathsf{eval}_\mathcal{R}(\sigma(t)) = \mathsf{eval}_\mathcal{R}(\sigma(t[y]_i))$ for all grounding $\sigma \in Subst(\mathcal{T}(\mathcal{F}))$ for t and $t[y]_i$. Now, since $\mathcal{R}$ is eval-defined and confluent, $\mathsf{eval}_\mathcal{R}(\sigma(t)) = \mathsf{eval}_\mathcal{R}(\sigma(t[y]_i)) \neq \varnothing$, i.e. $\sigma(t) \downarrow \sigma(t[y]_i)$ and therefore $\sigma(t) \leftrightarrow^* \sigma(t[y]_i)$. □

Hence, for confluent, eval-defined TRSs, the redundancy problem is reduced to the problem of checking validity of a particular class of inductive theorems. The problem of identifying the inductive theory of a TRS is in general undecidable, as shown by [Comon and Nieuwenhuis, 2000] even for a very restricted class of TRSs: finite, canonical, left- and right-linear, and right monadic (right hand sides have depth at most 1) CSs. Nevertheless, several methods for (semi)-automatically proving validity of inductive theorems have been developed, such as the *cover set* method [Zhang, 1988], *test set* method [Bouhoula and Rusinowitch, 1995], *rewriting induction* method [Reddy, 1990], and *inductionless induction* method [Comon, 2001; Comon and

Nieuwenhuis, 2000], which generalizes the former ones. Also, the *abstract rewriting* method of [Bert and Echahed, 1995] can be used for proving inductive theorems.

In Chapter 11 below, we provide different methods to successfully detect redundant arguments. For instance, we show how, both the inductionless induction as well as abstract rewriting methods can be applied for detecting redundancy of arguments.

In the following section, we address the complementary problem of removing redundant arguments from a TRS.

10.6 Erasing Redundant Arguments

The presence of redundant arguments within input expressions wastes memory space and can lead to time consuming explorations and transformations (by replacement) of their structure. Redundant arguments are not necessary to determine the result of a function call.

At first sight, one could expect that a suitable rewriting strategy which is able to avoid the exploration of redundant arguments of symbols could be defined. On the one hand, in Example 10.2, we showed that needed reduction is not able to avoid redundant arguments. Thus, the redundancy problem is independent of the neededness of a rewrite step shown in Part I of the thesis. On the other hand, context-sensitive rewriting (*csr*) [Lucas, 1998a], which can be used to forbid reductions on selected arguments of symbols, could also seem adequate for avoiding fruitless reductions at redundant arguments. Let $\mathcal{R}$ be the program `applast` of Example 10.1 extended with the rules for function `take` of Example 10.2. If we fix $\mu(\texttt{applast}) = \{2\}$ to (try to) avoid wasteful computations on the first argument of `applast`, using *csr* we are not able to reduce `applast (take 1 (1:2:[])) 0` to `0`. Thus, the redundancy problem is also independent of the restriction of rewriting in Part II of this thesis.

In this section, we formalize a procedure for *removing* redundant arguments from a TRS. The basic idea is simple: if an argument of f is redundant, it does not contribute to obtaining the value of any call to f and can be dropped from program $\mathcal{R}$. Hence, we remove redundant formal parameters and corresponding actual parameters for each function symbol in $\mathcal{R}$. We begin with the notion of syntactic erasure which is intended to pick up redundant arguments of function symbols.

Definition 10.23 (Syntactic erasure) *A* syntactic erasure *is a mapping $\rho : \mathcal{F} \to \mathcal{P}(\mathbb{N})$ such that for all $f \in \mathcal{F}$, $\rho(f) \subseteq \{1, \ldots, ar(f)\}$. We say that a syntactic erasure ρ is* sound *for a semantics* S *if, for all $f \in \mathcal{F}$, $\rho(f) \subseteq rarg_{\mathsf{S}}(f)$.*

Example 10.24 *Given the signature $\mathcal{F} = \{\texttt{0}, \texttt{nil}, \texttt{s}, \texttt{:}, \texttt{applast}, \texttt{lastnew}\}$ of the TRS $\mathcal{R}$ in Example 10.1, with $ar(\texttt{0}) = ar(\texttt{nil}) = 0$, $ar(\texttt{s}) = 1$, $ar(\texttt{:}) = ar(\texttt{applast}) = 2$, and $ar(\texttt{lastnew}) = 3$, the following mapping ρ is a* sound syntactic erasure *for* *the semantics* $\mathsf{eval}_{\mathcal{R}}$*:* $\rho(\mathtt{0}) = \rho(\mathtt{nil}) = \rho(\mathtt{s}) = \rho(\mathtt{:}) = \varnothing$*,* $\rho(\mathtt{applast}) = \{1\}$*, and* $\rho(\mathtt{lastnew}) = \{1, 2\}$*.*

Since we are interested in *removing* redundant arguments from function symbols, we transform the functions by reducing their arity according to the information provided by the redundancy analysis, thus building a new, *erased* signature.

Definition 10.25 (Erasure of a signature) *Given a signature* $\mathcal{F}$ *and a syntactic erasure* $\rho : \mathcal{F} \to \mathcal{P}(\mathbb{N})$*, the erasure of* $\mathcal{F}$ *is the signature* $\mathcal{F}_\rho$ *whose symbols* $f_\rho \in \mathcal{F}_\rho$ *are one to one with symbols* $f \in \mathcal{F}$ *and whose arities are related by* $ar(f_\rho) = ar(f) - |\rho(f)|$*.*

Example 10.26 *The erasure of the signature in Example 10.24 is* $\mathcal{F}_\rho = \{\mathtt{0}, \mathtt{nil}, \mathtt{s}, \mathtt{:}, \mathtt{applast}, \mathtt{lastnew}\}$*, with* $ar(\mathtt{0}) = ar(\mathtt{nil}) = 0$*,* $ar(\mathtt{s}) = ar(\mathtt{applast}) = ar(\mathtt{lastnew}) = 1$*, and* $ar(\mathtt{:}) = 2$*. Note that, by abuse, we use the same symbols for the functions of the erased signature.*

Now we extend the procedure to terms in the obvious way.

Definition 10.27 (Erasure of a term) *Given a syntactic erasure* $\rho : \mathcal{F} \to \mathcal{P}(\mathbb{N})$*, the function* $\tau_\rho : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{T}(\mathcal{F}_\rho, \mathcal{X})$ *on terms is:* $\tau_\rho(x) = x$ *if* $x \in \mathcal{X}$ *and* $\tau_\rho(f(t_1, \ldots, t_n)) = f_\rho(\tau_\rho(t_{i_1}), \ldots, \tau_\rho(t_{i_k}))$ *where* $\{1, \ldots, n\} - \rho(f) = \{i_1, \ldots, i_k\}$ *and* $i_m < i_{m+1}$ *for* $1 \leq m < k$*.*

The erasure procedure is extended to TRSs: we erase the lhs's and rhs's of each rule according to τ_ρ. In order to avoid extra variables in rhs's of rules (that arise from the elimination of redundant arguments of symbols in the corresponding lhs), we replace them by an arbitrary constant of $\mathcal{F}$ (which automatically belongs to $\mathcal{F}_\rho$).

Definition 10.28 (Erasure of a TRS) *Let* $\mathcal{R} = (\mathcal{F}, R)$ *be a TRS, such that* $\mathcal{F}$ *has a constant symbol* a*, and* ρ *be a syntactic erasure for* $\mathcal{F}$*. The erasure* $\mathcal{R}_\rho$ *of* $\mathcal{R}$ *is* $\mathcal{R}_\rho = (\mathcal{F}_\rho, \{\tau_\rho(l) \to \sigma_l(\tau_\rho(r)) \mid l \to r \in R\})$ *where the substitution* σ_l *for a lhs* l *is given by* $\sigma_l(x) = a$ *for all* $x \in \mathcal{V}ar(l) - \mathcal{V}ar(\tau_\rho(l))$ *and* $\sigma_l(y) = y$ *whenever* $y \in \mathcal{V}ar(\tau_\rho(l))$*.*

Example 10.29 *Let* $\mathcal{R}$ *be the TRS of Example 10.1 and* ρ *be the sound syntactic erasure of Example 10.24. The erasure* $\mathcal{R}_\rho$ *of* $\mathcal{R}$ *consists of the erased signature of Example 10.26 together with the following rules:*

$$
\begin{array}{ll}
\mathtt{applast(z)} \to \mathtt{z} & \mathtt{lastnew(z)} \to \mathtt{z} \\
\mathtt{applast(z)} \to \mathtt{lastnew(z)} & \mathtt{lastnew(z)} \to \mathtt{lastnew(z)}
\end{array}
$$

Below, we introduce a further improvement aimed to provide the final, "optimal" program.

Given a syntactic erasure ρ and a term $t \in \mathcal{T}(\mathcal{F})$, we define the *maximal non-redundant context* $MNRC^{\rho}(t)$ of t as $MNRC^{\rho}(t) = t[\Box]_{p_1.\rho(f_1)} \cdots [\Box]_{p_n.\rho(f_n)}$, where $p_1, \ldots, p_n$ are the positions of all outermost subterms rooted by symbols $f_1, \ldots, f_n$ such that $\rho(f_i) \neq \varnothing$ for $1 \leq i \leq n$.

Proposition 10.30 *Let ρ be a syntactic erasure for a signature Σ, $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $\sigma \in Subst(\mathcal{T}(\mathcal{F}, \mathcal{X}))$. Let $\sigma_\rho \in Subst(\mathcal{F}_\rho, \mathcal{X})$ be such that $\sigma_\rho(x) = \tau_\rho(\sigma(x))$ for all $x \in \mathcal{X}$. Then, $\tau_\rho(\sigma(t)) = \sigma_\rho(\tau_\rho(t))$.*

Proof. By structural induction. If $t = x \in \mathcal{X}$, then the result is immediate, since $\tau_\rho(x) = x$. For the induction step, we take $t = f(t_1, \ldots, t_n)$ for $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Then, $\tau_\rho(\sigma(t)) = \tau_\rho(\sigma(f(t_1, \ldots, t_n))) = \tau_\rho(f(\sigma(t_1), \ldots, \sigma(t_n))) = f_\rho(\tau_\rho(\sigma(t_1)), \ldots, \tau_\rho(\sigma(t_k)))$, where $k = n - |\rho(f)|$, and for all t_i, $i \notin \rho(f)$, we have that $j = |\{1, \ldots, i\} - \rho(f)|$ is the argument position of $\tau_\rho(t_i)$ in $f_\rho(\tau_\rho(\sigma(t_1)), \ldots, \tau_\rho(\sigma(t_k)))$. By induction hypothesis, $f_\rho(\tau_\rho(\sigma(t_1)), \ldots, \tau_\rho(\sigma(t_k))) = f_\rho(\sigma_\rho(\tau_\rho(t_1)), \ldots, \sigma_\rho(\tau_\rho(t_k)))$. And finally, $f_\rho(\sigma_\rho(\tau_\rho(t_1)), \ldots, \sigma_\rho(\tau_\rho(t_k))) = \sigma_\rho(f_\rho(\tau_\rho(t_1), \ldots, \tau_\rho(t_k))) = \sigma_\rho(\tau_\rho(t))$. $\Box$

The following result establishes that, when considering a set of redundant arguments, the semantics of a term only depends on its non-redundant arguments.

Lemma 10.31 *Let S be a term semantics for a signature Σ. Let $f \in \mathcal{F}$, and $I \subseteq rarg_{\mathsf{S}}(f)$. Then, for all context $C[\,]$ and for all $t, s_1, \ldots, s_k \in \mathcal{T}(\mathcal{F})$ such that $root(t) = f$, $\mathsf{S}(C[t]) = \mathsf{S}(C[t[\overline{s_k}]_I])$.*

Proof. By induction on $k = |I|$. If $k = 0$, it is immediate. If $k > 0$, let $i \in I$ and $\overline{s'_{k-1}} = s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k$. By the induction hypothesis, $\mathsf{S}(C[t]) = \mathsf{S}(C[t[\overline{s'_{k-1}}]_{I-\{i\}}])$. Since i is redundant w.r.t. S, $\mathsf{S}(C[t[\overline{s'_{k-1}}]_{I-\{i\}}]) = \mathsf{S}(C[t[\overline{s_k}]_I])$, i.e., $\mathsf{S}(C[t]) = \mathsf{S}(C[t[\overline{s_k}]_I])$. $\Box$

The following result basically states that the semantics of function calls is preserved when we simultaneously disregard the redundant arguments from different symbols which occur at disjoint positions.

Lemma 10.32 *Let S be a term semantics for a signature Σ. Let $f, g \in \mathcal{F}$, and $i \in rarg_{\mathsf{S}}(f)$, $j \in rarg_{\mathsf{S}}(g)$. Then, for all context $C[\,]$ and for all $t, t', s, s' \in \mathcal{T}(\mathcal{F})$ such that $root(t) = f$ and $root(t') = g$, $\mathsf{S}(C[t, t']) = \mathsf{S}(C[t[s]_i, t'[s']_j])$.*

Proof. Let $C'[\,]$ be the context $C'[\,] = C[\Box, t']$. By redundancy of i, we have $\mathsf{S}(C[t, t']) = \mathsf{S}(C'[t]) = \mathsf{S}(C'[t[s]_i])$. By redundancy of j, $\mathsf{S}(C'[t[s]_i]) = \mathsf{S}(C[t[s]_i, t']) = \mathsf{S}(C[t[s]_i, t'[s']_j])$, i.e., $\mathsf{S}(C[t, t']) = \mathsf{S}(C[t[s]_i, t'[s']_j])$. $\Box$

The mapping τ_ρ induces an equivalence $\equiv_{\tau_\rho}$ on terms given by: $t \equiv_{\tau_\rho} s$ iff $\tau_\rho(t) = \tau_\rho(s)$. It is easy to see that $t \equiv_{\tau_\rho} s$ implies that $MNRC^{\rho}(t) = MNRC^{\rho}(s)$. We also have the following property of sound erasures of terms.

Proposition 10.33 *If the syntactic erasure $\rho : \mathcal{F} \to \mathcal{P}(\mathbb{N})$ is sound with respect to the semantics S, then for all $t, s \in \mathcal{T}(\mathcal{F})$, $t \equiv_{\tau_\rho} s$ implies that $\mathsf{S}(t) = \mathsf{S}(s)$.*

Proof. By induction on the structure of $MNRC^{\rho}(t)$ and using Lemmata 10.31 and 10.32. □

The following theorem establishes the completeness of the erasure procedure for semantics S.

Theorem 10.34 (Completeness) *Let $\mathcal{R}$ be a left-linear TRS, S be a rewriting semantics for $\mathcal{R}$ such that $\mathsf{S} \preceq \mathsf{red}_{\mathcal{R}}$, ρ be a sound syntactic erasure for S, and $t, \delta \in \mathcal{T}(\Sigma_\rho)$. If $t \to^*_{\mathcal{R}_\rho} \delta$, then $\forall t', \delta' \in \mathcal{T}(\mathcal{F})$ such that $\tau_\rho(t') = t$ and $\tau_\rho(\delta') = \delta$, $\mathsf{S}(\delta') \subseteq \mathsf{S}(t')$.*

Proof. By induction on the length m of the derivation $t \to^*_{\mathcal{R}_\rho} \delta$. If $m = 0$, then $t = \delta$, and for all $t', \delta' \in \mathcal{T}(\mathcal{F})$ such that $\tau_\rho(t') = t$ and $\tau_\rho(\delta') = \delta$, by Proposition 10.33, $\mathsf{S}(t') = \mathsf{S}(\delta')$.

If $m > 0$, then $t \xrightarrow{p}_{\mathcal{R}_\rho} s \to^*_{\mathcal{R}_\rho} \delta$. Consider $t'', \delta'' \in \mathcal{T}(\mathcal{F})$ such that $\tau_\rho(t'') = t$ and $\tau_\rho(\delta'') = \delta$. First we prove, by induction on p, that there exist t', s' such that $\tau_\rho(t') = t$, $\tau_\rho(s') = s$, and $t' \xrightarrow{p'}_{\mathcal{R}} s'$.

1. If $p = \Lambda$, then $t = \sigma(l)$ for some $l \to r$ in $\mathcal{R}_\rho$. Then, by Definition 10.28, there exists $l' \to r' \in \mathcal{R}$ such that $\tau_\rho(l') = l$ and $\sigma_{l'}(\tau_\rho(r')) = r$. Now, there exist a term $t' \in \mathcal{T}(\mathcal{F})$ and a substitution σ' such that $\tau_\rho(t') = t$ and $t' = \sigma'(l')$. Then, by left-linearity, for all x in $\mathcal{V}ar(l')$, if $x \in \mathcal{V}ar(l)$, we have $\sigma(x) = \tau_\rho(\sigma'(x))$. Otherwise, let $l'|_q = x$, $\sigma(x) = t'|_q$. Hence, $t' \xrightarrow{\Lambda}_{\mathcal{R}} \sigma'(r')$.

2. If $p = i.q$, then we consider the terms $t = f_\rho(t_1, \ldots, t_k)$, $s = f_\rho(s_1, \ldots, s_k)$, $t' = f(t'_1, \ldots, t'_n)$ and $s' = f(s'_1, \ldots, s'_n)$, such that $k = n - |\rho(f)|$. Then, $p' = i'.q'$, where $i = |\{1 \leq i'\} - \rho(f)|$, and for all j, j' s.t. $1 \leq j \leq k$ and $1 \leq j' \leq n$, $j = |\{1 \leq j'\} - \rho(f)|$, $t_j = \tau_\rho(t'_{j'})$, $s_j = \tau_\rho(s'_{j'})$. By the induction hypothesis, the conclusion follows.

Now, by induction hypothesis, for all $w, w' \in \mathcal{T}(\mathcal{F})$ such that $\tau_\rho(w) = s$ and $\tau_\rho(w') = \delta$, we have that $\mathsf{S}(w') \subseteq \mathsf{S}(w)$. Thus, since $w \equiv_{\tau_\rho} s'$ and $w' \equiv_{\tau_\rho} \delta'$, by Proposition 10.33, we have that $\mathsf{S}(\delta') \subseteq \mathsf{S}(s')$. By definition of red, $\mathsf{red}_{\mathcal{R}}(s') \subseteq \mathsf{red}_{\mathcal{R}}(t')$. Then, let T be the window set such that $\mathsf{S} \preceq \mathsf{red}_{\mathcal{R}}$, $\mathsf{red}_{\mathcal{R}}(s') \cap T \subseteq \mathsf{red}_{\mathcal{R}}(t') \cap T$, and thus, $\mathsf{S}(s') \subseteq \mathsf{S}(t')$. Hence, we obtain that $\mathsf{S}(\delta') \subseteq \mathsf{S}(t')$. But, by Proposition 10.33, $\mathsf{S}(t'') = \mathsf{S}(t')$ and $\mathsf{S}(\delta'') = \mathsf{S}(\delta')$; thus, the conclusion follows. □

The following theorem establishes the correctness of the erasure procedure for a semantics S.

Theorem 10.35 (Correctness) *Let $\mathcal{R} = (\Sigma, R)$ be a left-linear TRS, S be a rewriting semantics for $\mathcal{R}$, ρ be a sound syntactic erasure for S, and $t \in \mathcal{T}(\mathcal{F})$. If $\delta \in \mathsf{S}(t)$, then $\tau_\rho(t) \to^*_{\mathcal{R}_\rho} \tau_\rho(\delta)$.*

Proof. Let $t' = C[c, \ldots, c]$, where $C[\Box, \ldots, \Box] = MNRC^\rho(t)$ and $c \in \mathcal{F}$ is the constant used in $\mathcal{R}_\rho$. Since $t \equiv_\rho t'$, by Proposition 10.33, $t \to^*_{\mathcal{R}} \delta$ if and only if $t' \to^*_{\mathcal{R}} \delta$. Now we prove, by induction on the length m of derivation $t' \to^*_{\mathcal{R}} \delta$ that $\tau_\rho(t') \to^*_{\mathcal{R}_\rho} \tau_\rho(\delta)$. If $m = 0$, then $t' = \delta$ and the result is immediate. If $m > 0$, we let $t' \stackrel{p}{\to}_{\mathcal{R}} s \to^*_{\mathcal{R}} \delta$. By induction on p, we prove that either $\tau_\rho(t') \to_{\mathcal{R}_\rho} \tau_\rho(s)$ or $\tau_\rho(t') = \tau_\rho(s)$.

1. If $p = \Lambda$, then there exists $l \to r$ in $\mathcal{R}$ such that $t' = \sigma(l)$ and $s = \sigma(r)$. By Proposition 10.30, $\tau_\rho(\sigma(l)) = \sigma_\rho(\tau_\rho(l))$ and $\tau_\rho(\sigma(r)) = \sigma_\rho(\tau_\rho(r))$ where $\sigma_\rho(x) = \tau_\rho(\sigma(x))$ for all $x \in \mathcal{X}$. Left-linearity of $\mathcal{R}$ ensures that, every variable x that occurs within an erasable subterm of l (i.e., a subterm $l|_p$ such that there exists $q.i < p$ such that $i \in \rho(root(l|_q)))$ does not occur in $\tau_\rho(l)$. Thus, when considering $x \in \mathcal{V}ar(l) - \mathcal{V}ar(\tau_\rho(l))$, by definition of t', it must be $\sigma(x) = c$. Hence, $\sigma_\rho(\tau_\rho(r)) = \sigma_\rho(\sigma_l(\tau_\rho(r)))$ where σ_l is fixed as in Definition 10.40. Thus, by definition of $\mathcal{R}_\rho$, $\tau_\rho(t') \to_{\mathcal{R}_\rho} \tau_\rho(s)$.

2. If $p = i.q$, then we let $t' = f(t'_1, \ldots, t'_k)$ and $s = f(s_1, \ldots, s_k)$ and consider two cases:

   (a) If $i \in \rho(f)$, then $\tau_\rho(t') = \tau_\rho(s)$ since t only differs from s in the i-th argument t_i of f in t' (which is removed by τ_ρ).

   (b) If $i \notin \rho(f)$, then, the i-th argument of f in t becomes the (transformed) j-th argument $\tau_\rho(t_i)$ of f in $\tau_\rho(t')$, where $j = |\{1 \leq i\} - \rho(f)|$. By the induction hypothesis, either $\tau_\rho(t'_i) \to_{\mathcal{R}_\rho} \tau_\rho(s_i)$ or $\tau_\rho(t'_i) = \tau_\rho(s_i)$. In both cases, the conclusion follows.

Therefore, we have that either $\tau_\rho(t') \to \tau_\rho(s)$ or $\tau_\rho(t') = \tau_\rho(s)$. By the induction hypothesis, $\tau_\rho(s) \to^*_{\mathcal{R}_\rho} \tau_\rho(\delta)$. Thus, $\tau_\rho(t') \to^*_{\mathcal{R}_\rho} \tau_\rho(\delta)$. Since $\tau_\rho(t) = \tau_\rho(t')$, the conclusion follows. □

The following theorem establishes the correctness and completeness of the erasure procedure for the semantics eval.

Theorem 10.36 (Correctness and Completeness) *Let $\mathcal{R} = (\mathcal{F}, \mathcal{R})$ be a left-linear TRS, ρ be a sound syntactic erasure for $\mathsf{eval}_{\mathcal{R}}$, $t \in \mathcal{T}(\mathcal{F})$, and $\delta \in \mathcal{T}(\mathcal{C})$. Then, $\tau_\rho(t) \to^*_{\mathcal{R}_\rho} \delta$ iff $\delta \in \mathsf{eval}_{\mathcal{R}}(t)$.*

Proof. Immediate from Theorem 10.34 and Theorem 10.35. □

In the following, we are able to ascertain the conditions for the preservation of some computational properties of TRSs after erasure.

Theorem 10.37 *Let $\mathcal{R}$ be a left-linear TRS. Let ρ be a sound syntactic erasure for* $\mathsf{eval}_{\mathcal{R}}$*. If $\mathcal{R}$ is* $\mathsf{eval}_{\mathcal{R}}$*-defined and confluent, then the erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ is confluent.*

Proof. Given $t \in \mathcal{T}(\Sigma_\rho)$, if $t_1 \;{}_{\mathcal{R}_\rho}^{*}\!\!\leftarrow t \rightarrow_{\mathcal{R}_\rho}^{*} t_2$ with $t_1 \neq t_2$, by Theorem 10.34, there exist $s, s_1, s_2 \in \mathcal{T}(\mathcal{F})$ such that $\tau_\rho(s) = t$, $\tau_\rho(s_1) = t_1$, $\tau_\rho(s_2) = t_2$, $\mathsf{eval}_{\mathcal{R}}(s_1) \subseteq \mathsf{eval}_{\mathcal{R}}(s)$, and $\mathsf{eval}_{\mathcal{R}}(s_2) \subseteq \mathsf{eval}_{\mathcal{R}}(s)$.

Since $\mathsf{eval}_{\mathcal{R}}$ is $\mathcal{R}$-normalized, and $\mathcal{R}$ is confluent and $\mathsf{eval}_{\mathcal{R}}$-defined, $\mathsf{eval}_{\mathcal{R}}(s)$ is a singleton consisting of the normal form t'. Moreover, $\mathsf{eval}_{\mathcal{R}}(s_1) = \mathsf{eval}_{\mathcal{R}}(s_2) = \mathsf{eval}_{\mathcal{R}}(s)$, and by Theorem 10.35, $t_1 \rightarrow_{\mathcal{R}_\rho}^{*} \tau_\rho(t') \;{}_{\mathcal{R}_\rho}^{*}\!\!\leftarrow t_2$. $\square$

Theorem 10.38 *Let $\mathcal{R}$ be a left-linear and CD TRS, and ρ be a sound syntactic erasure for* $\mathsf{eval}_{\mathcal{R}}$*. If $\mathcal{R}$ is normalizing, then the erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ is normalizing.*

Proof. Since $\mathcal{R}$ is normalizing and CD, $\forall t \in \mathcal{T}(\mathcal{F})$, $\exists \delta \in \mathcal{T}(\mathcal{C}) \in \mathsf{eval}(t)$. Then, by Theorem 10.35, $\tau_\rho(t) \rightarrow_{\mathcal{R}_\rho}^{*} \tau_\rho(\delta)$, and, by Proposition 10.5, $\tau_\rho(\delta) = \delta$. Hence, the conclusion follows. $\square$

In the theorem above, we cannot strengthen normalization to termination. A simple counterexample showing that termination may get lost is the following.

Example 10.39 *Consider the left-linear, (confluent, CD, and) terminating TRS $\mathcal{R}$*

```
f(a,y) → a
f(c(x),y) → f(x,c(y))
```

The first argument of `f` *is redundant w.r.t.* $\mathsf{eval}_{\mathcal{R}}$*. However, after erasing the argument, we get the TRS*

```
f(y) → a
f(y) → f(c(y))
```

which is not terminating.

In the example above, note that the resulting TRS is not orthogonal, whereas the original program is. Hence, this example also shows that orthogonality is not preserved under erasure.

The following post-processing transformation can improve the optimization achieved.

Definition 10.40 (Reduced erasure of a TRS) *Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS and ρ be a syntactic erasure for $\mathcal{F}$. The reduced erasure $\mathcal{R}'_\rho$ of $\mathcal{R}$ is obtained from the erasure $\mathcal{R}_\rho$ of $\mathcal{R}$ by a* compression transformation *defined as removing any trivial rule $t \to t$ of $\mathcal{R}_\rho$ and then normalizing the rhs's of the rules w.r.t. the non-trivial rules of $\mathcal{R}_\rho$.*

Reduced erasures are well-defined whenever $\mathcal{R}_\rho$ is confluent and normalizing since, for such systems, every term has a unique normal form.

Example 10.41 *Let $\mathcal{R}_\rho$ be the erasure of Example 10.29. The reduced erasure consists of the rules* {`applast(z)` $\rightarrow$ `z`, `lastnew(z)` $\rightarrow$ `z`}.

Since right-normalization preserves confluence, termination and the equational theory (as well as confluence, normalization and the equational theory, in almost orthogonal and normalizing TRSs) [Gramlich, 2001], and the removal of trivial rules does not change the evaluation semantics of the TRS $\mathcal{R}$ either, we have the following.

Corollary 10.42 *Let $\mathcal{R}$ be a left-linear TRS, ρ be a sound syntactic erasure for $\mathsf{eval}_\mathcal{R}$, $t \in \mathcal{T}(\mathcal{F})$, and $\delta \in \mathcal{T}(\mathcal{C})$. If (the TRS which results from removing trivial rules from) $\mathcal{R}_\rho$ is confluent and terminating (alternatively, if it is almost orthogonal and normalizing), then, $\tau_\rho(t) \rightarrow^*_{\mathcal{R}'_\rho} \delta$ if and only if $\delta \in \mathsf{eval}_\mathcal{R}(t)$, where $\mathcal{R}'_\rho$ is the reduced erasure of $\mathcal{R}$.*

Erasures and reduced erasures of a TRS preserve left-linearity. For a TRS $\mathcal{R}$ satisfying the conditions in Corollary 10.42, by using [Gramlich, 2001], it is immediate that the reduced erasure $\mathcal{R}'_\rho$ is confluent and normalizing. Also, $\mathcal{R}'_\rho$ is CD if $\mathcal{R}$ is.

Hence, let us note that these results allow us to perform the optimization of program `applast` while guaranteeing that the intended semantics is preserved.

10.7 Conclusion

This chapter provides the first results concerning the detection and removal of useless arguments in program functions. We have given a semantic definition of redundancy which takes the semantics S as a parameter. We have provided some decidability results about redundancy of an argument and an alternative definition of redundancy of an argument in terms of the inductive theory of a program.

Actually, inefficiencies caused by the redundancy of arguments cannot be avoided by using standard reduction strategies such as the work defined in Parts I and II of this thesis. Therefore, we have developed a transformation for eliminating dead code which appears in the form of useless function calls and we have proven that the transformation preserves the semantics (and the operational properties) of the original program under ascertained conditions. The optimized program that we produce cannot be created as the result of applying standard transformations of functional programming to the original program (such as partial evaluation, supercompilation, and deforestation, see e.g. [Pettorossi and Proietti, 1996a]). We believe that the semantic grounds for redundancy analyses and elimination laid in this chapter may foster further insights and developments in the functional programming community and neighbouring fields.

10.7.1 Related Work

Some notions have appeared in the literature of what it means for a term in a TRS $\mathcal{R}$ to be "computationally irrelevant". Our analysis is different from all the related methods in many respects and, in general, incomparable to them.

Contrarily to our notion of redundancy, the meaninglessness of [Kuper, 1994; Kennaway *et al.*, 1996] is a property of the terms themselves (they may have meaning in $\mathcal{R}$ or may not), whereas our notion refers to arguments (positions) of function symbols. In [Kuper, 1994], Section 7.1, a term t is called *meaningless* if, for each context $C[\ ]$ s.t. $C[t]$ has a normal form, we have that $C[t']$ has the same normal form for all terms t'. This can be seen as a kind of superfluity (w.r.t. normal forms) of a fixed expression in any context, whereas our notion of redundancy refers to the possibility of getting rid of some arguments of a given function symbol with regard to some observed semantics. The meaninglessness of [Kuper, 1994] is not helpful for the purposes of optimizing programs by removing useless arguments of function symbols which we pursue. On the other hand, terms with a normal form are proven meaningful (i.e., not meaningless) in [Kuper, 1994; Kennaway *et al.*, 1996], whereas we might have redundant arguments which are normal forms.

Among the vast literature on analysis (and removal) of unnecessary data structures, the analyses of *unneededness* (or *absence*) of functional programming [Cousot and Cousot, 1994; Hughes, 1988], and the *filtering* of useless arguments and unnecessary variables of logic programming [Leuschel and Sørensen, 1996; Pettorossi and Proietti, 1994] are the closest to our work. In [Hughes, 1988], a notion of *needed/unneeded* parameter for list-manipulation programs is introduced which is closely related to the redundancy of ours in that it is capable of identifying whether the value of a subexpression is ignored. The method is formulated in terms of a fixed, finite set of projection functions which introduces some limitations on the class of neededness patterns that can be identified. Since our method gives the information that a parameter is definitely not necessary, our redundancy notion implies Hughes's unneededness, but not vice versa. For instance, constructor symbols cannot have redundant arguments in our framework (Proposition 10.5), whereas Hughes' notion of unneededness can be applied to the elements of a list: Hughes' analysis is able to determine that in the `length` function (defined as usual), the spine of the argument list is needed but the elements of the list are not needed; this is used to perform some optimizations for the compiler. However, this information cannot be used for the purposes of our work, that is, to remove these elements when the entire list cannot be eliminated.

On the other hand, Hughes's notion of *neededness/unneededness* should not be confused with the standard notion of needed (positions of) redexes of [Huet and Lévy, 1992]: Example 10.2 shows that Huet and Levy's neededness does not imply the non-redundancy of the corresponding argument or position (nor vice versa).

The notion of redundancy of an argument in a term rewriting system can be seen as a kind of *comportment property* as defined in [Cousot and Cousot, 1994]. Cousot's comportment analysis generalizes not only the unneededness analyses but also strictness, termination and other standard analyses of functional programming. In [Cousot and Cousot, 1994], comportment is mainly investigated within a denotational framework, whereas our approximation is independent from the semantic formalism.

Proietti and Pettorossi's *elimination procedure* for the removal of unnecessary variables is a powerful unfold/fold-based transformation procedure for logic programs; therefore, it does not compare directly with our method, which would be seen as a post-processing phase for program transformers optimization. Regarding the kind of *unnecessary variables* that the elimination procedure can remove, only variables that occur more than once in the body of the program rule and which do not occur in the head of the rule can be dropped. This is not to say that the transformation is powerless; on the contrary, the effect can be very striking as these kinds of variables often determine multiple traversals of intermediate data structures which are then removed from the program. Our procedure for removing redundant arguments is also related to the Leuschel and Sørensen RAF and FAR algorithms [Leuschel and Sørensen, 1996], which apply to removing unnecessary arguments in the context of (conjunctive) partial evaluation of logic programs. However, a comparison is not easy either as we have not yet considered the semantics of computed answers for our programs.

People in the functional programming community have also studied the problem of useless variable elimination (UVE). Apparently, they were unaware of the works of the logic programming community, and they started studying the topic from scratch, mainly following a flow-based approach [Wand and Siveroni, 1999] or a type-based approach [Berardi *et al.*, 2000; Kobayashi, 2000] (see [Berardi *et al.*, 2000] for a discussion of this line of research). All these works address the problem of safe elimination of dead *variables* but heavily handle data structures. A notable exception is [Liu and Stoller, 2002], where Liu and Stoller discuss how to safely eliminate dead code in the presence of recursive data structures by applying a methodology based on regular tree grammars. Unfortunately, the method in [Liu and Stoller, 2002] does not apply to achieve the optimization pursued in our running example `applast`.

Obviously, there exist examples (inspired) in the previously discussed works which cannot be directly handled with our results, consider the following TRS:

```
length([]) → 0
length(x:xs) → s(length(xs))
f(x) → length(x:[])
```

Our methods do not capture the redundancy of the argument of `f`. In [Liu and Stoller, 2002] it is shown that, in order to evaluate `length(xs)`, we do not need to evaluate the elements of the argument list `xs`. In Liu et al.'s methodology, this means that we could replace the rule for `f` above by `f(_)` $\rightarrow$ `length(_:[])` where `_` is a new constant. However, as discussed in Section 10.6, this could still lead to wasteful computations if, e.g., an eager strategy is used for evaluating the expressions: in that case, a term t within a call `f(`t`)` would be wastefully reduced. Nevertheless, the new TRS can be used now to recognize the first argument of `f` as redundant. That is, we are allowed to use the following rule:

`f` $\rightarrow$ `length(_:[])`

which completely avoids wasteful computations on redundant arguments. Hence, the different methods are complementary and an enhanced test might be developed by properly combining them.

Chapter 11

Characterizations of Redundancy

In this chapter, we demonstrate how the problem of detecting redundant arguments can be addressed by different techniques. In Section 11.1, we provide an approximation technique based on decidability issues (following Section 10.4) which can detect redundant arguments in many practical examples. On the other hand, since the redundancy problem can be expressed in terms of validity of inductive theorems in confluent, eval-defined TRSs, we show in Section 11.2 how the "inductionless induction" technique [Comon, 2001; Comon and Nieuwenhuis, 2000] can be applied to detect redundant arguments. Similarly, in Section 11.3, we show how the "abstract rewriting" technique [Bert and Echahed, 1995] can be also applied to detect redundant arguments. In order to provide more practical methods to recognize redundancy, we ascertain in Section 11.4 suitable conditions which allow us to simplify the general redundancy problem to the analysis of redundant positions within rhs's of the program rules, which is a simpler technique.

A short version of this chapter appeared in [Alpuente et al., 2002a,e]

11.1 Approximations of Redundancy

Whenever a property is undecidable or costly to decide, we use approximations. A notion of approximation (for TRSs) that has been proven useful for approximating interesting properties in term rewriting (namely neededness of redexes for normalization) is the following [Durand and Middeldorp, 1997; Jacquemard, 1996]: Given TRSs $\mathcal{R}$ and $\mathcal{R}'$ (possibly with extra variables) over the same signature, $\mathcal{R}'$ *approximates* $\mathcal{R}$ if $\to^*_{\mathcal{R}} \subseteq \to^*_{\mathcal{R}'}$ and $\mathsf{NF}_{\mathcal{R}} = \mathsf{NF}_{\mathcal{R}'}$. An approximation of TRSs is a mapping α from TRSs to TRSs with the property that TRS $\alpha(\mathcal{R})$ approximates TRS $\mathcal{R}$ [Durand and Middeldorp, 1997]. We write $\mathcal{R}_\alpha$ instead of $\alpha(\mathcal{R})$ to denote the approximation of $\mathcal{R}$ according to α. *Strong*, *nv* [Durand and Middeldorp, 1997], *shallow* [Comon, 2000], and *growing* [Jacquemard, 1996] are examples of such approximations of TRSs. In all these approximations, the rhs's of the rules are modified in different ways. For instance, given a TRS $\mathcal{R}$, $\mathcal{R}_{nv}$ is obtained by replacing *all* variables in the rhs by new, different variables that do *not* occur in the lhs; this is possible since the framework deals with extra variables.

In order to approximate redundancy, we need to use a new symbol Ω to represent arbitrary terms (in particular, terms occurring at the argument position which is tested for redundancy). Inspired by [Durand and Middeldorp, 1997; Oyamaguchi, 1986], we define our notion of approximation as follows. Let $\mathcal{R}$ be a TRS over a signature Σ and $\mathcal{R}'$ be a TRS over the signature $\Sigma \cup \{\Omega\}$, where Ω is a new constant symbol defined by the rules $\{\Omega \to f(\overline{\Omega}) \mid f \in \Sigma\}$. We extend the approximation notion of [Durand and Middeldorp, 1997; Jacquemard, 1996] naturally to TRSs over signatures $\mathcal{F}$ and $\mathcal{F} \cup \{\Omega\}$, where Ω is a special symbol that potentially expresses any term. Note that, in this part of the thesis, we consider the normalization semantics only for ground terms, i.e. $\to_{\mathcal{R}} \in \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$ instead of $\mathcal{T}(\mathcal{F},\mathcal{X}) \times \mathcal{T}(\mathcal{F},\mathcal{X})$. Thus, we say that $\mathcal{R}'$ approximates $\mathcal{R}$ (but notice that, now, $\mathcal{R}'$ is a TRS on $\mathcal{F} \cup \{\Omega\}$) if $\to^*_{\mathcal{R}} \cap (\mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})) \subseteq \to^*_{\mathcal{R}'} \cap (\mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F}))$ and $\mathsf{NF}_{\mathcal{R}} = \mathsf{NF}_{\mathcal{R}'}$. Note that, $\to^*_{\mathcal{R}} \subseteq (\mathcal{T}(\mathcal{F},\mathcal{X}) \times \mathcal{T}(\mathcal{F},\mathcal{X}))$ whereas $\to^*_{\mathcal{R}'} \subseteq (\mathcal{T}_\Omega(\mathcal{F},\mathcal{X}) \times \mathcal{T}_\Omega(\mathcal{F},\mathcal{X}))$; however, by definition of $\mathcal{R}'$, $\mathsf{NF}_{\mathcal{R}'} \subseteq \mathcal{T}(\mathcal{F})$.

The following notation is auxiliary. We say that the semantics S is *determined* w.r.t. a symbol f and an argument $i \in \{1, \ldots, ar(f)\}$ if for all context $C[\,]$ and $t \in \mathcal{T}(\mathcal{F})$ such that $root(t) = f$, $|\mathsf{S}(C[t[\Omega]_i])| = 1$, where $|A|$ stands for the cardinality of the set A.

The following theorem provides a sufficient condition for redundancy which is the basis of our decidable approximations to redundancy.

Theorem 11.1 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS, $\mathcal{R}'$ be an approximation of $\mathcal{R}$, $f \in \Sigma$, $i \in \{1, \ldots, ar(f)\}$, and $\mathsf{S} \in \{\mathsf{eval}, \mathsf{nf}\}$. If $\mathcal{R}$ is $\mathsf{S}_{\mathcal{R}}$-defined and $\mathsf{S}_{\mathcal{R}'}$ is determined w.r.t. f and i, then $i \in rarg_{\mathsf{S}_{\mathcal{R}}}(f)$.*

Proof. We prove the result for eval; the proof for nf is analogous. Note that, since $\mathcal{R}$ is $\mathsf{eval}_{\mathcal{R}}$-defined, $|\mathsf{eval}_{\mathcal{R}}(C[t])| \geq 1$ and $|\mathsf{eval}_{\mathcal{R}}(C[t[s]_i])| \geq 1$. Moreover, since $|\mathsf{eval}_{\mathcal{R}'}(C[t[\Omega]_i])| = 1$, then $|\mathsf{eval}_{\mathcal{R}}(C[t])| = 1$ and $|\mathsf{eval}_{\mathcal{R}}(C[t[s]_i])| = 1$. Otherwise, since $C[t[\Omega]_i] \to^*_{\mathcal{R}'} C[t]$ and $C[t[\Omega]_i] \to^*_{\mathcal{R}'} C[t[s]_i]$, $\to^*_{\mathcal{R}} \subseteq \to^*_{\mathcal{R}'}$, and constructor symbols of $\mathcal{R}$ and $\mathcal{R}'$ are identic (since $\mathsf{NF}_{\mathcal{R}} = \mathsf{NF}_{\mathcal{R}'}$ and Ω is a defined symbol) we would also have $|\mathsf{eval}_{\mathcal{R}'}(C[t[\Omega]_i])| > 1$. Assume that $i \notin rarg_{\mathsf{eval}_{\mathcal{R}}}(f)$. Then, there exists $C[\,]$, $t \in \mathcal{T}(\mathcal{F})$ such that $root(t) = f$, and $s \in \mathcal{T}(\mathcal{F})$ such that $\mathsf{eval}_{\mathcal{R}}(C[t]) \neq \mathsf{eval}_{\mathcal{R}}(C[t[s]_i])$. Then, since $|\mathsf{eval}_{\mathcal{R}}(C[t])| = 1$ and $|\mathsf{eval}_{\mathcal{R}}(C[t[s]_i])| = 1$, it follows that $\delta \in \mathsf{eval}_{\mathcal{R}}(C[t])$ and $\delta' \in \mathsf{eval}_{\mathcal{R}}(C[t[s]_i])$ verify $\delta \neq \delta'$. By reasoning as above, this would mean that $|\mathsf{eval}_{\mathcal{R}'}(C[t[\Omega]_i])| > 1$ thus leading to a contradiction. □

It is an open problem whether redundancy is decidable for terminating TRSs. Nevertheless, Theorem 11.1 ensures that redundancy w.r.t. nf is approximable in terminating TRSs (since any terminating TRS $\mathcal{R}$ is $\mathsf{nf}_{\mathcal{R}}$-defined). The following theorem ensures that WSkS definability of a semantics entails the possibility of guaranteeing decidability of a given approximation.

Theorem 11.2 *Let S be a term semantics for a signature $\mathcal{F} \cup \{\Omega\}$. If S is WSkS definable, then it is decidable whether $\forall C[\,], t \in \mathcal{T}(\mathcal{F}), root(t) = f, i \in \{1, \ldots, ar(f)\}$, $|\mathsf{S}(C[t[\Omega]_i])| = 1$.*

Proof. Assuming that S is defined in WSkS by the formula $\Phi(\vec{X}, \vec{Y})$, the property is WSkS definable by using the following formula:

$$\forall \vec{T}\, \forall p\; Term(\vec{T}) \wedge p \in T_f \wedge p.i \in T_\Omega \;\Rightarrow$$
$$(\forall q \in T_\Omega, q = p.i) \wedge \exists \vec{S}\; Term(\vec{S}) \wedge \Phi(\vec{T}, \vec{S}) \wedge \forall \vec{W}\; (Term(\vec{W}) \wedge \Phi(\vec{T}, \vec{W}) \Rightarrow \vec{W} = \vec{S})$$

Now, by Theorem 10.6, the conclusion follows. □

According to the discussion in Section 10.4, semantics eval and nf are WSkS definable for left-linear, right ground TRSs over finite signatures. This suggests us to use the following approximation of left-linear TRSs.

We use the following approximation of left-linear TRSs. Given $\mathcal{R} = (\mathcal{F}, R)$, we define $\mathcal{R}_{rg} = (\mathcal{F} \cup \{\Omega\}, R_{rg})$ as follows:

$$R_{rg} = \{l \to r_\Omega \mid l \to r \in R\} \cup \{\Omega \to f(\overline{\Omega}) \mid f \in \Sigma\}$$

where t_Ω is the term t with all variables replaced by Ω. It is straightforward to see that rg is an approximation of TRSs. The following theorem ensures that the property of semantics S being determined w.r.t. a symbol f and an argument i is decidable for an approximation $\mathcal{R}_{rg}$ of a TRS $\mathcal{R}$ and semantics $\mathsf{S} \in \{\mathsf{nf}_{\mathcal{R}_{rg}}, \mathsf{eval}_{\mathcal{R}_{rg}}\}$.

Theorem 11.3 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS, $\mathcal{R}_{rg}$ be the approximation rg of $\mathcal{R}$, $f \in \Sigma$, $i \in \{1, \ldots, ar(f)\}$, and $\mathsf{S} \in \{\mathsf{eval}_{\mathcal{R}_{rg}}, \mathsf{nf}_{\mathcal{R}_{rg}}\}$. It is decidable whether S is determined w.r.t. f and i.*

Proof. Since $\mathcal{R}_{rg}$ is a left-linear and right ground TRS, the semantics $\mathsf{red}_{\mathcal{R}_{rg}}$ is WSkS definable. The sets $\mathsf{NF}_{\mathcal{R}}$ and $\mathcal{T}(\mathcal{C})$ are WSkS definable and, by Proposition 10.10, we obtain that the semantics $\mathsf{nf}_{\mathcal{R}_{rg}}$ and $\mathsf{eval}_{\mathcal{R}_{rg}}$ are WSkS definable. Then, by Theorem 11.2, we obtain that the property $\forall C[\,], t \in \mathcal{T}(\mathcal{F}), root(t) = f, i \in \{1, \ldots, ar(f)\}$, $|\mathsf{S}(C[t[\Omega]_i])| = 1$ is decidable for $\mathsf{S} \in \{\mathsf{eval}_{\mathcal{R}_{rg}}, \mathsf{nf}_{\mathcal{R}_{rg}}\}$. □

By Theorems 11.1 and 11.3, redundancy of an argument w.r.t. $\mathsf{nf}_{\mathcal{R}}$ (and $\mathsf{eval}_{\mathcal{R}}$) is *effectively* approximable by using rg.

Example 11.4 *Consider the TRS $\mathcal{R}$*

$\mathtt{f(a,y)} \to \mathtt{a}$
$\mathtt{f(c(x),y)} \to \mathtt{g(x,y)} \quad \mathtt{g(x,y)} \to \mathtt{f(x,c(y))}$

Note that $\mathcal{R}$ is terminating, hence $\mathsf{nf}_{\mathcal{R}}$-defined. Approximation $\mathcal{R}_{rg}$ is:

$\mathtt{f(a,y)} \to \mathtt{a}$
$\mathtt{f(c(x),y)} \to \mathtt{g}(\Omega,\Omega) \quad \mathtt{g(x,y)} \to \mathtt{f}(\Omega,\mathtt{c}(\Omega))$
$\Omega \to \mathtt{f}(\Omega,\Omega) \quad \Omega \to \mathtt{g}(\Omega,\Omega) \quad \Omega \to \mathtt{c}(\Omega) \quad \Omega \to \mathtt{a}$

It is not difficult to see that $\mathsf{nf}_{\mathcal{R}_{rg}}$ is determined w.r.t. f *and arguments* 1 *and* 2*. It is possible to construct an automaton which tests that condition (see e.g., [Thatcher and Wright, 1968] for more details) thus making it automatically provable. By Theorem 11.1, this means that $\{1,2\} \subseteq rarg_{\mathsf{nf}_{\mathcal{R}}}(\mathtt{f})$.*

The approximation rg is similar to nv of [Durand and Middeldorp, 1997]. However, including the new symbol Ω in the approximated program is essential for our development: we use Ω to distinguish the argument which is tested for redundancy, and the rules $\{\Omega \to f(\overline{\Omega}) \mid f \in \Sigma\}$ express that this argument is allowed to take 'any' value. This effect could not be achieved in a rewriting setting by the approximation nv.

From now on, we consider two additional methods for effectively detecting redundant arguments which are based on reformulating the redundancy problem in terms of the inductive theory of a TRS presented in Section 10.5. For instance, we show how the inductionless induction as well as abstract rewriting methods for validity of inductive theorems can be applied for detecting redundancy of arguments.

11.2 Inductionless Induction

We briefly recall the inductionless induction method for proving validity of inductive theorems (see [Comon, 2001; Comon and Nieuwenhuis, 2000] for details). Inductionless induction tackles how to (semi)-automatically prove a set of equations $\mathcal{C}$ in the minimal Herbrand model of a set of equations E without making use of induction schemes (induction rules). It uses a (first-order) axiomatization $\mathcal{A}$ of the minimal model of E, $\mathcal{I}_E$, such that $\mathcal{C} \cup \mathcal{A} \cup E$ is consistent if and only if $\mathcal{C}$ is valid in $\mathcal{I}_E$, i.e. $\mathcal{I}_E \models \mathcal{C}$. A *normal axiomatization* $\mathcal{A}$ of $\mathcal{I}_E$ is a finite recursive set of purely universal formulas such that $\mathcal{I}_E \models \mathcal{A}$, $\mathcal{I}_E$ is the only Herbrand model of $E \cup \mathcal{A}$ up to isomorphism, and for all ground terms s, t representative of its congruence class of $\mathcal{I}_E$, $s \not\equiv t \Rightarrow \mathcal{A} \models s \neq t$. The method relies on *saturation techniques* [Bachmair and Ganzinger, 1994, 2001] for performing the proof by consistency of $\mathcal{C} \cup \mathcal{A} \cup E$, thus any saturation-based general-purpose first-order theorem prover can be used for proving inductive validity. The (in)consistency proofs are performed in two stages: first deductions on $\mathcal{C} \cup E$ are computed by saturation, yielding new consequences; then, these new consequences are checked for inconsistency w.r.t. $\mathcal{A}$.

Deductions are performed by *superposition*, which can be defined by the following inference rules. We assume below that $\succeq$ is a reduction ordering [Dershowitz and Plaisted, 2001] which is total on ground terms, i.e. a relation $\succ$ which is irreflexive, transitive, well-founded, total, monotonic, and stable under substitutions. A well-known reduction ordering is the *recursive path ordering*, based on a total ordering $\succ_\Sigma$ (called *precedence*) on Σ.

$$\text{Superposition} \quad \frac{l = r \quad c[s]}{\sigma(c[r])} \quad \text{if } \begin{array}{l} \sigma = mgu(l, s), s \text{ is not a variable}, \\ \sigma(r) \not\succeq \sigma(l), l = r \in E, c[s] \in \mathcal{C}. \end{array}$$

$$\text{Equality resolution} \quad \frac{C \vee s \neq t}{\sigma(C)} \quad \text{if } \sigma = mgu(s, t), C \vee s \neq t \in \mathcal{C}.$$

Given a ground equation c, $\mathcal{C}^{\prec}$ is the set of ground instances of equations in $\mathcal{C}$ that are strictly smaller than c in this ordering. A ground equation (conjecture) c is *entailed* by a set of equations (conjectures) $\mathcal{C}$ if $E \cup \mathcal{A} \cup \mathcal{C}^{\prec} \vdash c$. A non-ground equation is entailed if all its ground instances are. An inference is redundant if one of its premises or its conclusion are entailed by $\mathcal{C}$. A set of equations is *saturated* if all inferences are redundant. A *derivation sequence* is a sequence $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_n, \ldots$ such that each $\mathcal{C}_{i+1}$ is obtained from $\mathcal{C}_i$ by adding some logical consequences or by removing some entailed equations. A derivation sequence is *fair* if every equation which can be persistently derived is eventually derived.

The application of inductionless induction to the redundancy problem is illustrated in the following.

Example 11.5 *Consider the following program borrowed from [Bert and Echahed, 1995], which can be used for adding and subtracting natural numbers in Peano's notation:*

```
minus(x,0) → x                          p(0) → 0
minus(0,s(y)) → 0                       p(s(0)) → 0
minus(s(x),s(y)) → p(s(minus(x,y)))     p(s(s(x))) → s(x)
plus(0,y) → y
```

$\texttt{plus(s(x),y)} \rightarrow \texttt{s(plus(x,y))}$

If we specialize this program for the call $\texttt{minusplus(x,y)}\equiv\texttt{minus(plus(y,x),y)}$*, which adds* x *to* y *and then removes* y *from the sum, thus returning the original* x*, the optimized program which can be obtained by using an automatic specializer of functional programs such as the one described in [Alpuente et al., 1998] is:*

$\texttt{minusplus(x,0)} \rightarrow \texttt{x}$ $\qquad$ $\texttt{p(0)} \rightarrow \texttt{0}$
$\texttt{minusplus(x,s(y))} \rightarrow \texttt{p(minusplus2(x,y))}$ $\qquad$ $\texttt{p(s(x))} \rightarrow \texttt{s(x)}$

$\texttt{minusplus2(x,0)} \rightarrow \texttt{x}$
$\texttt{minusplus2(x,s(y))} \rightarrow \texttt{p(minusplus2(x,y))}$

Note that the second argument of the function minusplus *is redundant for the semantics of computed values. This program is saturated and can be oriented using the recursive path ordering with the precedence* $\texttt{minusplus} > \texttt{minusplus2} > \texttt{p} > \texttt{s} > \texttt{0}$*, and the axiomatization* $\{\forall \texttt{x},\texttt{y}, \texttt{s(x)} \neq \texttt{0} \wedge \texttt{s(x)} = \texttt{s(y)} \Rightarrow \texttt{x} = \texttt{y}\}$*.*

We can prove the redundancy of the second argument of minusplus *using Theorem 10.22 by proving the validity of the equation:*

$$(c_1)\ \texttt{minusplus(x,y)} = \texttt{minusplus(x,w)}$$

We have two possible inferences by saturation, the other ones are renamings:

$$\begin{array}{lrcl} (c_{1,1}) & \texttt{x} & = & \texttt{minusplus(x,w)} \\ (c_{1,2}) & \texttt{p(minusplus2(x,y))} & = & \texttt{minusplus(x,w)} \end{array}$$

Here, no equation is entailed. After superposing again, we obtain (up to renaming):

$$\begin{array}{lrcl} (c_{1,3}) & \texttt{x} & = & \texttt{x} \\ (c_{1,4}) & \texttt{x} & = & \texttt{p(minusplus2(x,w))} \\ (c_{1,5}) & \texttt{p(minusplus2(x,y))} & = & \texttt{x} \\ (c_{1,6}) & \texttt{p(minusplus2(x,y))} & = & \texttt{p(minusplus2(x,w))} \end{array}$$

Here, equation $c_{1,3}$ *is trivially entailed,* $c_{1,4} \vdash c_{1,5}$*, and* $c_{1,4} \vdash c_{1,6}$*. If we superpose one more time, we obtain (up to renaming):*

$$\begin{array}{lrcl} (c_{1,7}) & \texttt{x} & = & \texttt{p(x)} \\ (c_{1,8}) & \texttt{x} & = & \texttt{p(p(minusplus2(x,w)))} \end{array}$$

Here, equation $c_{1,8}$ *is entailed:* $c_{1,4} \cup c_{1,7} \vdash c_{1,8}$*. And, finally, after superposing, we obtain:*

$$\begin{array}{lrcl} (c_{1,9}) & \texttt{0} & = & \texttt{0} \\ (c_{1,10}) & \texttt{s(x)} & = & \texttt{s(x)} \end{array}$$

And here, the two equations are trivially entailed.

Then, the set $\mathcal{R} \cup \{c_1, c_{1,1}, c_{1,2}, c_{1,4}, c_{1,7}\}$ *is saturated and it is immediate to check its consistency w.r.t. the axiomatization. Therefore, the theorem is proved and hence* *the second argument of* `minusplus` *is redundant. We have checked this automatically by using the theorem prover* Spike *[Bouhoula and Rusinowitch, 1995] which implements a particular implicit induction technique, namely the one which is based on* test sets *(see the Appendix D-4 for the practical execution in* Spike*).*

Unfortunately, in many interesting cases, existing methods for inductive validity may run forever without proving validity of inductive theorems, as in the following example.

Example 11.6 *Let $\mathcal{R}$ be the following saturated TRS oriented using the recursive path ordering with the precedence* $\mathtt{f} > \mathtt{id} > \mathtt{s} > \mathtt{0}$ *and status left to right for* `f`*:*

$$\begin{array}{ll}
\mathtt{f(0,0)} \rightarrow \mathtt{0} & \mathtt{id(0)} \rightarrow \mathtt{0} \\
\mathtt{f(0,s(y))} \rightarrow \mathtt{f(0,y)} & \mathtt{id(s(x))} \rightarrow \mathtt{s(id(x))} \\
\mathtt{f(s(x),0)} \rightarrow \mathtt{f(x,0)} & \\
\mathtt{f(s(x),s(y))} \rightarrow \mathtt{f(x,s(id(y)))} &
\end{array}$$

Consider the axiomatization of Example 11.5. In order to prove the redundancy of the first argument of `f`*, we consider the conjecture:* (c_1) $\mathtt{f(x,y)} = \mathtt{f(z,y)}$*. Superposing* c_1 *with* $\mathcal{R}$*, we get the following (the other equations are renamings):*

$$\begin{array}{lrcl}
(c_{1,1}) & \mathtt{0} & = & \mathtt{f(z,0)} \\
(c_{1,2}) & \mathtt{f(0,y)} & = & \mathtt{f(z,s(y))} \\
(c_{1,3}) & \mathtt{f(x,0)} & = & \mathtt{f(z,0)} \\
(c_{1,4}) & \mathtt{f(x,s(id(y)))} & = & \mathtt{f(z,s(y))}
\end{array}$$

Here, equation $c_{1,3}$ *is entailed. Superposing* $c_{1,1}$*,* $c_{1,2}$ *and* $c_{1,4}$ *with* $\mathcal{R}$*, we obtain (up to renaming):*

$$\begin{array}{lrcl}
(c_{1,4}) & \mathtt{0} & = & \mathtt{0} \\
(c_{1,5}) & \mathtt{0} & = & \mathtt{f(z,0)} \\
(c_{1,6}) & \mathtt{f(0,y)} & = & \mathtt{f(0,y)} \\
(c_{1,7}) & \mathtt{f(0,y)} & = & \mathtt{f(z,s(id(y)))} \\
(c_{1,8}) & \mathtt{0} & = & \mathtt{f(z,s(0))} \\
(c_{1,9}) & \mathtt{f(0,y)} & = & \mathtt{f(z,s(s(y)))} \\
(c_{1,10}) & \mathtt{f(x,s(id(y)))} & = & \mathtt{f(z,s(id(y)))} \\
(c_{1,11}) & \mathtt{f(x,s(0))} & = & \mathtt{f(z,s(0))} \\
(c_{1,12}) & \mathtt{f(x,s(s(id(y))))} & = & \mathtt{f(z,s(s(y)))}
\end{array}$$

Here, all equations except $c_{1,12}$ *are entailed. Thus, we can obtain infinitely many equations* $\mathtt{f(x,s^{\mathit{n}}(id(y)))} = \mathtt{f(z,s^{\mathit{n}}(y))}$ *and the process may run forever unless an extra lemma* $\mathtt{id(x)} = \mathtt{x}$ *is manually provided.*

Several techniques to improve termination of the inductive validity process have been developed such as deduction of lemmata, which might help to prove an inductive theorem [Walsh, 1996; Kapur and Subramaniam, 1996; Ireland and Bundy, 1996]. On the other hand, different criteria can be used to stop the saturation process, such as the homeomorphic embedding (which is commonly used in program transformation for avoiding infinite sequences [Alpuente *et al.*, 1998]). Unfortunately, important properties are lost, such as refutationally completeness or finite saturation under common conditions.

Therefore, it is interesting to consider decidable classes of TRSs where the inductionless induction method terminates, and thus, redundancy of arguments can be decided. In the next section we present a result for the decidability of redundancy based on inductionless induction, which is complementary to the decidability result of Section 11.1. We postpone to Section 11.3 the use of finite approximations based on abstract interpretation, such as the abstract rewriting of [Bert and Echahed, 1995], to formalize static analyses of redundancy.

11.2.1 Standard Theories

In this section we consider the standard theories of [Nieuwenhuis, 1996], a class of TRSs where the saturation process is finite, thus the validity of an inductive theorem is decidable.

Standard theories are particular sets of equations which are finitely closed by superposition. We need the following: the *depth* d of a subterm $s = t|_p$ is the length of the position p: $d = |p|$; and a variable is *shallow* in a term if it occurs only at depth 0 or 1 in the term.

Definition 11.7 *[Nieuwenhuis, 1996] A* standard *signature Σ is a signature where every function symbol f in Σ has an associated set of shallow positions $sh(f)$ and a set of linear positions $lin(f)$, such that $lin(f) \cap sh(f) = \varnothing$ and $sh(f) \cup lin(f) = \{1, \ldots, ar(f)\}$.*

Definition 11.8 *[Nieuwenhuis, 1996] A term s is a* standard *term iff it is a variable or a term of the form $f(s_1, \ldots, s_n)$ where if $i \in sh(f)$ then s_i is a variable or a ground term and if $i \in lin(f)$ then all variables in s_i are linear in s.*

Note that, according to the previous definition, all ground terms are standard, whereas not every linear term is, because no term with variables occurring at depth ≥ 1 is allowed at a shallow position. Furthermore, the only non-linear variables of a standard term are shallow variables occurring at shallow positions.

Definition 11.9 *[Nieuwenhuis, 1996] An equation $s = t$ is* standard *iff*

1. *s is linear and t is ground or*
2. *s is a standard term $f(\ldots, g(t), \ldots)$ and t is a variable or*

3. *s and t are standard terms sharing only shallow variables and no variable x is both a shallow position argument and a linear position argument in $s = t$.*

A standard presentation *is a set of* standard equations *and a* standard theory *is a theory axiomatizable by a standard presentation.*

Theorem 11.10 [Nieuwenhuis, 1996] *Every standard presentation E can be finitely closed under superposition.*

Hence, for standard theories, the saturation process is finite, and then the inductionless induction method (hence, the redundancy of arguments) is decidable. Note that we naturally specialize the notion of standard presentations (as originally defined in [Nieuwenhuis, 1996]) to "standard TRSs".

Theorem 11.11 [Comon, 2001] *Let E be a finite saturated set of equations, $\mathcal{C}_0$ be a finite set of equations to prove, $\mathcal{A}$ be a normal axiomatization, and $\mathcal{C}_0, \mathcal{C}_1, \ldots$ be a fair derivation. Then, $\mathcal{I}_E \models \mathcal{C}_0$ iff $\mathcal{A} \cup \{c\}$ is consistent for all equation $c \in \bigcup_i \mathcal{C}_i$.*

Theorem 11.12 *Let $\mathcal{R}$ be a standard confluent TRS. Let the equation $t = s$ be standard (within $\mathcal{R}$). It is decidable whether $t = s$ is an inductive theorem of $\mathcal{R}$.*

Proof. By Theorem 11.10, $\mathcal{R}$ can be finitely saturated. Let $\mathcal{R}'$ be the saturated TRS of $\mathcal{R}$. Again, by Theorem 11.10, $\mathcal{R}' \cup \{t = s\}$ is finitely saturated. Then, by Theorem 11.11, validity of $t = s$ can be recovered since derivation sequences are finite. □

Corollary 11.13 *Let $\mathcal{R}$ be a standard, confluent, and sufficiently complete TRS, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$. Let the equation $t = t[y]_i$ be standard (within $\mathcal{R}$) such that $t = f(x_1, \ldots, x_{ar(f)})$ and $x_1, \ldots, x_{ar(f)}, y$ are distinct variables. It is decidable whether the i-th argument of f is* redundant *(w.r.t.* eval*).*

Proof. It follows from Theorems 11.12 and 10.22. □

Even if the class of standard theories is somehow restrictive, it still allows to detect redundancy of arguments in significant examples.

Example 11.14 *Consider the following TRS, where extra variables are allowed in right hand sides, as it happens in functional logic programs.*

```
f(0,y) → y                g(0,y) → y
f(s(x),y) → g(u,y)        g(s(x),y) → f(u,y)
```

Here, we can automatically prove that the first argument of `f` *(and* `g`*) is redundant.*

We consider the problem of identifying new decidable classes of TRSs (w.r.t. the particular class of inductive theorems which express redundancy), as well as developing new decision algorithms for these programs, as an interesting line of work which we plan to pursue as future work.

Rather than focusing in deeper decidability matters, in the following section we investigate finite approximations for the validity problem, which lead us to formulate more practical static redundancy analyses. As an application of the analysis, we are able to correctly analyze Example 11.6 by applying the new methodology based on abstract rewriting, whereas it cannot be checked by (unoptimized) inductionless induction.

11.3 Abstract Rewriting

Abstract interpretation is a theory to extract relevant information from programs without considering all details given by the standard semantics [Cousot and Cousot, 1977]. In [Bert and Echahed, 1995], Bert and Echahed proposed a framework called *abstract rewriting* which is based on an abstract interpretation of (conditional) term rewriting systems for approximating the normal form $t{\downarrow_{\mathcal{R}}}$ of a term t in a canonical CS $\mathcal{R}$. They make use of the notion of an abstract domain of terms. In this section, we use the technique of abstract rewriting in order to prove inductive theorems, and thus to detect redundant arguments, in the setting of canonical and sufficiently complete CS's. We first recall the abstract rewriting methodology of [Bert and Echahed, 1995].

Definition 11.15 [Bert and Echahed, 1995] *Let $\mathcal{F}$ be a signature. The abstract specification of $\mathcal{F}$ is $\mathcal{A}(\Sigma) = \mathcal{F} \cup \{\top, \bot, \sqcup, \sqcap\}$.*

Intuitively, an abstract term t approximates the set of its ground instances, where the symbols $\bot$ and $\top$ stand for the empty set and the set of all constructor terms, respectively. Similarly, the symbols $\sqcup$ and $\sqcap$ correspond to set union and set intersection operators, respectively.

Let $\_^a : \mathcal{F} \to \mathcal{A}(\Sigma)$ be the obvious identity signature morphism between the concrete and the abstract signatures. The signature morphism $\_^a$ is extended to a translation function on terms $\_^a : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{T}(\mathcal{A}(\Sigma))$ such that $x^a = \top$ $\forall x \in \mathcal{X}$ and $(f(t_1, \ldots, t_n))^a = f^a(t_1^a, \ldots, t_n^a)$ $\forall f \in \mathcal{F}$. Besides, given an abstract term t, the concrete set $\gamma(t)$ is the largest set of ground terms such that $\gamma(\top) = \mathcal{T}(\mathcal{F})$, $\gamma(\bot) = \varnothing$, $\gamma(f^a(t_1, \ldots, t_{ar(f)})) = \{f(s_1, \ldots, s_{ar(f)}) \mid \forall 1 \leq i \leq ar(f), s_i \in \gamma(t_i)\}$, $\gamma(t_1 \sqcup t_2) = \gamma(t_1) \cup \gamma(t_2)$, and $\gamma(t_1 \sqcap t_2) = \gamma(t_1) \cap \gamma(t_2)$.

In order to approximate normal forms of terms, it is defined a partial order $\leq$ on abstract terms such that $t \leq t'$ iff $\gamma(t) \subseteq \gamma(t')$. Given a term $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,

$t \in \mathcal{T}(\mathcal{A}(\Sigma))$ is an approximation of s iff $\forall s' \in \mathcal{T}(\mathcal{F})$ such that $(s')^a \leq s^a$, $(s'{\downarrow_{\mathcal{R}}})^a \leq t$, or equivalently $\{\sigma(s){\downarrow_{\mathcal{R}}} \mid \sigma(s) \in \mathcal{T}(\mathcal{F})\} \subseteq \gamma(t)$.

The set of abstract terms is larger than the set of concrete terms. In order to compute approximations of normal forms of concrete terms, finite subsets of the set of abstract terms are introduced by the so-called finite upper closures $up : \mathcal{T}(\mathcal{A}(\Sigma)) \rightarrow \mathcal{T}(\mathcal{A}(\Sigma))$ such that up is *monotonic* ($\forall t, t' \in \mathcal{T}(\mathcal{A}(\Sigma)), t \leq t' \Rightarrow up(t) \leq up(t')$), *extensive* ($\forall t \in \mathcal{T}(\mathcal{A}(\Sigma)), t \leq up(t)$), and *idempotent* ($up \circ up = up$). The main objective of a finite upper closure is to restrict $\mathcal{T}(\mathcal{A}(\mathcal{C}))$ to a finite set $\mathcal{T}^{up}(\mathcal{A}(\mathcal{C}))$.

The notions of *abstract rewriting system* and *abstract rewriting calculus* are defined. An abstract rewriting system is associated to a TRS $\mathcal{R}$ and a finite upper closure up in order to approximate the normal forms of ground instances of concrete terms with variables. In concrete, a "computed" abstract TRS efficiently determines an approximation for any concrete term. Due to the properties of abstract terms and the ordering $\leq$, the classical definition of rewriting is extended to the abstract rewriting calculus (see [Bert and Echahed, 1995] for details).

Now, we can exploit abstract rewriting for proving inductive theorems, which demonstrates redundancy of arguments. Let us first introduce some preparatory results.

Definition 11.16 *We define the set of up-minimal substitutions as:* $Subst_{up}(\mathcal{C}, \mathcal{X}) = \{\sigma \in Subst(\mathcal{C}, \mathcal{X}) \mid \forall \sigma' \in Subst(\mathcal{C}, \mathcal{X}) \wedge \forall x \in \mathcal{X}, \sigma'(x)^a \leq \sigma(x)^a \Rightarrow up(\sigma'(x)^a) \equiv \sigma(x)^a\}$

Theorem 11.17 [Bert and Echahed, 1995] *Let $\mathcal{R}$ be a left linear, canonical, sufficiently complete CS, and* up *be a finite upper closure. Let $\mathcal{R}_c^{up}$ be the "computed" abstract TRS associated to $\mathcal{R}$ and up. Then,* (1) $\forall t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\forall \sigma \in Subst(\mathcal{T}(\mathcal{F}))$, $(\sigma(t){\downarrow_{\mathcal{R}}})^a \leq up(t^a){\downarrow_{\mathcal{R}_c^{up}}}$. *And,* (2) $\forall t_1, t_2 \in \mathcal{T}(\mathcal{A}(\Sigma))$, $t_1 \leq t_2 \Rightarrow up(t_1){\downarrow_{\mathcal{R}^{up}}} \leq up(t_2){\downarrow_{\mathcal{R}^{up}}}$.

Theorem 11.18 *Let $\mathcal{R}$ be a left linear, canonical, sufficiently complete CS, and* up *be a finite upper closure. Let $\mathcal{R}_c^{up}$ be the "computed" abstract TRS associated to $\mathcal{R}$ and up. The equation $s = t$ is an inductive theorem of $\mathcal{R}$ if for all $\sigma \in Subst_{up}(\mathcal{C}, \mathcal{X})$, $up(\sigma(s)^a){\downarrow_{\mathcal{R}_c^{up}}} \equiv up(\sigma(t)^a){\downarrow_{\mathcal{R}_c^{up}}} \equiv \delta$ such that $\delta \in \mathcal{T}(\mathcal{A}(\mathcal{C}) - \{\top, \bot, \sqcup, \sqcap\})$.*

Proof. Let $\sigma \in Subst(\mathcal{T}(\mathcal{F}))$ such that $\sigma(s), \sigma(t) \in \mathcal{T}(\mathcal{F})$. We must prove that $\sigma(s){\downarrow_{\mathcal{R}}} \equiv \sigma(t){\downarrow_{\mathcal{R}}}$. By Definition 11.16, there exists $\theta \in Subst_{up}(\mathcal{C}, \mathcal{X})$ such that $\forall x \in \mathcal{V}ar(s) \cup \mathcal{V}ar(t)$, $\sigma(x)^a \leq \theta(x)^a$ and $up(\sigma(x)^a) \equiv \theta(x)^a$. Then, by hypothesis, $up(\theta(s)^a){\downarrow_{\mathcal{R}_c^{up}}} \equiv up(\theta(t)^a){\downarrow_{\mathcal{R}_c^{up}}} \equiv \delta^a$ such that $\delta^a \in \mathcal{T}(\mathcal{A}(\mathcal{C}) - \{\top, \bot, \sqcup, \sqcap\})$. By property (2) of Theorem 11.17, $\theta(s)^a{\downarrow_{\mathcal{R}_c^{up}}} \leq \delta^a$ and $\theta(t)^a{\downarrow_{\mathcal{R}_c^{up}}} \leq \delta^a$. Thus, $up(\sigma(s)^a){\downarrow_{\mathcal{R}_c^{up}}} \leq \delta^a$ and $up(\sigma(t)^a){\downarrow_{\mathcal{R}_c^{up}}} \leq \delta^a$. By property (1) of Theorem 11.17, $(\sigma(s){\downarrow_{\mathcal{R}}})^a \leq \delta^a$ and $(\sigma(t){\downarrow_{\mathcal{R}}})^a \leq \delta^a$. And finally, by approximation, $\gamma(\delta^a) = \{\delta\}$ and then, $\sigma(s){\downarrow_{\mathcal{R}}} \equiv \sigma(t){\downarrow_{\mathcal{R}}} \equiv \delta$. □

The following result is the key for the detection of redundant arguments.

Corollary 11.19 *Let $\mathcal{R}$ be a left linear, canonical, sufficiently complete CS, and* up *be a finite upper closure. Let $\mathcal{R}_c^{up}$ be the "computed" abstract TRS associated to $\mathcal{R}$ and up. Let $f \in \mathcal{F}$ and $i \in \{1, \ldots, ar(f)\}$. The i-th argument of f is* redundant *(w.r.t.* eval*) iff for all $\sigma \in Subst_{up}(\mathcal{C}, \mathcal{X})$, $up(\sigma(t)^a)\downarrow_{\mathcal{R}_c^{up}} \equiv up(\sigma(t[y]_i)^a)\downarrow_{\mathcal{R}_c^{up}} \equiv \delta$ such that $\delta \in \mathcal{T}(\mathcal{A}(\mathcal{C}) - \{\top, \bot, \sqcup, \sqcap\})$, where $t = f(x_1, \ldots, x_{ar(f)})$ and $x_1, \ldots, x_{ar(f)}, y$ are distinct variables.*

Proof. By Theorems 11.18 and 10.22. □

Clearly, the accuracy of the analysis depends on the chosen upper closure *up*. A standard upper closure is defined by taking the maximum depth of all constructor terms in the left-hand sides of the TRS.

Example 11.20 *Consider the TRS of Example 11.6 and the finite upper closure head defined by $head(\mathtt{0}^a) = \mathtt{0}^a$, $head(\mathtt{s}(t)) = \mathtt{s}(\top)$ if $t \in \mathcal{T}(\mathcal{C})$, and $head(f(t_1, \ldots, t_n)) = f(head(t_1), \ldots, head(t_n))$ otherwise. We can prove by abstract rewriting that both arguments of* `f` *are redundant. The "computed" abstract TRS for $\mathcal{R}$ and head is:*

$$\begin{array}{ll}
\mathtt{f}^a(\mathtt{0}^a,\mathtt{0}^a) \rightarrow \mathtt{0}^a & \mathtt{id}^a(\mathtt{0}^a) \rightarrow \mathtt{0}^a \\
\mathtt{f}^a(\mathtt{0}^a,\mathtt{s}^a(\top)) \rightarrow \mathtt{0}^a & \mathtt{id}^a(\mathtt{s}^a(\top)) \rightarrow \mathtt{s}^a(\top) \\
\mathtt{f}^a(\mathtt{s}^a(\top),\mathtt{0}^a) \rightarrow \mathtt{0}^a & \\
\mathtt{f}^a(\mathtt{s}^a(\top),\mathtt{s}^a(\top)) \rightarrow \mathtt{0}^a &
\end{array}$$

Then, the first and second arguments of `f` *are redundant since for the equations $\mathtt{f}(x, y) = \mathtt{f}(x', y)$ and $\mathtt{f}(x, y) = \mathtt{f}(x, y')$, and every substitution $\sigma \in Subst_{head}(\mathcal{C}, \mathcal{X})$, $\sigma(\mathtt{f}(x, y))^a\downarrow_{\mathcal{R}_c^{up}} \equiv \sigma(\mathtt{f}(x', y))^a\downarrow_{\mathcal{R}_c^{up}} \equiv \mathtt{0}^a$ and $\sigma(\mathtt{f}(x, y))^a\downarrow_{\mathcal{R}_c^{up}} \equiv \sigma(\mathtt{f}(x, y'))^a\downarrow_{\mathcal{R}_c^{up}} \equiv \mathtt{0}^a$. Note that this example can not be handled by the inductionless induction method.*

In the following sections, we address redundancy analyses from a different perspective. Rather than going more deeply in approximation techniques, we are interested in ascertaining effective, sufficient conditions which ensure that an argument is redundant in a given TRS.

11.4 Characterizing Redundancy through Redundancy of Positions

When considering a particular (possibly non-ground) function call, we can observe a more general notion of redundancy which allows us to consider arbitrary (deeper) positions within the call. We say that two terms $t, s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ are *p-equal*, with $p \in \mathcal{P}os(t) \cap \mathcal{P}os(s)$ if, for all occurrences w with $w < p$, $t|_w$ and $s|_w$ have the same symbol at the root.

Definition 11.21 (Redundant position) *Let* S *be a term semantics for a signature* $\mathcal{F}$ *and* $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. *The position* $p \in \mathcal{P}os(t)$ *is* redundant *in* t *w.r.t.* S *if, for all* $t', s \in \mathcal{T}(\mathcal{F})$ *such that* t *and* t' *are* p*-equal,* $\mathsf{S}(t') = \mathsf{S}(t'[s]_p)$.

We denote by $rpos_{\mathsf{S}}(t)$ the set of redundant positions of a term t w.r.t. a semantics S. Note that the previous definition cannot be simplified by getting rid of t' and simply requiring that for all $s \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(t) = \mathsf{S}(t[s]_p)$ mimicking the definition of redundancy of an argument, since redundant positions cannot be analyzed independently if the final goal is to remove the useless arguments, i.e. our notion of redundancy of positions is not more compositional, as the following example shows.

Example 11.22 *Let us consider the TRS* $\mathcal{R}$:

$$\mathtt{f(a,a)} \to \mathtt{a} \qquad \mathtt{f(a,b)} \to \mathtt{a} \qquad \mathtt{f(b,a)} \to \mathtt{a} \qquad \mathtt{f(b,b)} \to \mathtt{b}$$

Given the term $\mathtt{f(a,a)}$, *for all term* $s \in \mathcal{T}(\mathcal{F})$, $\mathsf{eval}_{\mathcal{R}}(t[s]_1) = \mathsf{eval}_{\mathcal{R}}(t)$ *and* $\mathsf{eval}_{\mathcal{R}}(t[s]_2) = \mathsf{eval}_{\mathcal{R}}(t)$. *However,* $\mathsf{eval}_{\mathcal{R}}(t[\mathtt{b}]_1[\mathtt{b}]_2) \neq \mathsf{eval}_{\mathcal{R}}(t)$.

The following result states that the positions of a term which are below the indices addressing the redundant arguments of any function symbol occurring in t are redundant.

Proposition 11.23 *Let* S *be a term semantics for a signature* $\mathcal{F} = \mathcal{F} \uplus \mathcal{C}$, $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $p \in \mathcal{P}os(t)$, $f \in \mathcal{F}$. *For all positions* q, p' *and* $i \in rarg_{\mathsf{S}}(f)$ *such that* $p = q.i.p'$ *and* $root(t|_q) = f$, $p \in rpos_{\mathsf{S}}(t)$ *holds.*

Proof. Let $t' = C[f(\overline{t})]_q$ such that t and t' are equal down to p. Since $i \in rarg_{\mathsf{S}}(f)$, for all term $s \in \mathcal{T}(\mathcal{F})$, $\mathsf{S}(C[f(\overline{t})]_q) = \mathsf{S}(C[f(\overline{t})[s]_i]_q)$. In particular, if $s' = t'[s]_{q.i}|_p$, then $\mathsf{S}(t') = \mathsf{S}(t'[s']_p)$ and the conclusion follows. □

In the following section, we provide some general criteria for ensuring redundancy of arguments on the basis of the (redundancy of some) positions in the rhs's of program rules, specifically the positions of the rhs's where the arguments of the functions defined in the lhs's 'propagate' to.

11.4.1 Using Redundant Positions for Characterizing Redundancy

In the following, we focus on the problem of characterizing the redundant arguments w.r.t. eval by studying the redundancy w.r.t. eval of some positions in the rhs's of program rules.

We say that $p \in \mathcal{P}os(t)$ is a sub-constructor position of t if for all $q < p$, $root(t|_q) \in \mathcal{C}$. In particular, Λ is a sub-constructor position.

Definition 11.24 **((f,i)-redundant variable)** *Let* S *be a term semantics for a signature* $\mathcal{F}$*,* $f \in \mathcal{F}$*,* $i \in \{1,\ldots,ar(f)\}$*, and* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$*. The variable* $x \in \mathcal{X}$ *is* (f,i)*-redundant in* t *if* it occurs only at positions $p \in \mathcal{P}os_x(t)$ which are redundant in t, *in symbols* $p \in rpos_{\mathsf{S}}(t)$*, or* it appears in sub-constructor positions of the i-th parameter of f-rooted subterms of t, *in symbols* $\exists q,q'$ *such that* $p = q.i.q'$*,* $root(t|_q) = f$ *and* q' *is a sub-constructor position of subterm* $t|_{q.i}$*.*

Note that variables which do not occur in a term t are trivially (f,i)-redundant in t for any $f \in \mathcal{F}$ and $i \in \{1,\ldots,ar(f)\}$. Given a TRS $\mathcal{R} = (\mathcal{F},R)$, we write $\mathcal{R}_f$ to denote the TRS $\mathcal{R}_f = (\mathcal{F},\{l \to r \in R \mid root(l) = f\})$ which contains the set of rules defining $f \in \mathcal{F}$.

Now, we are able to provide a first effective method to determine redundant arguments based on the (f,i)-redundant variables occurring in rhs's. In order to prove Theorem 11.28 below, we introduce some auxiliary definitions and lemmata.

Lemma 11.25 *Let* $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ *be a TRS such that* $|\mathcal{T}(\mathcal{C})| > 1$*, and* $t \in \mathcal{T}(\mathcal{C},\mathcal{X})$*. Then,* $rpos_{\mathsf{eval}_\mathcal{R}}(t) = \varnothing$*.*

Proof. By contradiction. Let $p \in rpos_{\mathsf{eval}_\mathcal{R}}(t)$, we have that, for all $t' \in \mathcal{T}(\mathcal{F})$ such that t and t' are p-equal, and for all $s \in \mathcal{T}(\mathcal{F})$, $\mathsf{eval}_\mathcal{R}(t') = \mathsf{eval}_\mathcal{R}(t'[s]_p)$. Consider $\sigma \in Subst(\mathcal{T}(\mathcal{C}))$ such that $\sigma(t) \in \mathcal{T}(\mathcal{C})$. Clearly, $\sigma(t)$ and t are p-equal. If $p > \Lambda$, then we consider $s = \sigma(t)$. Thus, $\mathsf{eval}_\mathcal{R}(\sigma(t)[\sigma(t)]_p) = \{\sigma(t)[\sigma(t)]_p\} = \mathsf{eval}_\mathcal{R}(\sigma(t))$, thus contradicting $\sigma(t)[\sigma(t)]_p \neq \sigma(t)$. If $p = \Lambda$, then since $|\mathcal{T}(\mathcal{C})| > 1$, there exists $s \in \mathcal{T}(\mathcal{C})$ such that $s \neq \sigma(t)$. Thus, $\mathsf{eval}_\mathcal{R}(\sigma(t)[s]_p) = \{s\} = \mathsf{eval}_\mathcal{R}(\sigma(t))$, contradicting $s \neq \sigma(t)$. □

Definition 11.26 *Let* $f \in \Sigma$*,* $i \in \{1,\ldots,ar(f)\}$*, and* $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$*. We define* $Pos_{i,f}(t) = \{p \in \mathcal{P}os(t) \mid p = q.i.q', root(t|_q) = f$*, and* q' *is a sub-constructor position in subterm* $t|_{q.i}\}$*.*

In the following, we introduce some helpful notation. We denote the restriction of a set of positions P w.r.t. a position p as $P|_p = \{p'' \mid \exists p' \in P \wedge p' = p.p''\}$. Let $P = \{p_1,\ldots,p_n\}$ be a sequence of positions, $\overline{t} = \{t_1,\ldots,t_n\}$ be a sequence of terms associated to P, and $P' = \{p'_1,\ldots,p'_m\} \subseteq P$; we denote $\overline{t}|_{P'} = \{t'_1,\ldots,t'_m\}$ such that, for $i \in \{1,\ldots,m\}$, there exists $j \in \{1,\ldots,n\}$ such that $p'_i = p_j$ and $t'_i = t_j$.

Proposition 11.27 *Let* $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ *be a left-linear CS,* $f \in \mathcal{F}$*, and* $i \in \{1,\ldots,ar(f)\}$*. Let* $t \in \mathcal{T}(\mathcal{F})$*, and* $P \subseteq Pos_{i,f}(t) \cup rpos_{\mathsf{eval}_\mathcal{R}}(t)$ *be a set of disjoint positions. If, for all* $l \to r \in \mathcal{R}_f$*,* $l|_i$ *is a variable which is* (f,i)*-redundant in* r*, then for all* $\overline{s} \in \mathcal{T}(\mathcal{F})$*,* $\mathsf{eval}_\mathcal{R}(t) = \mathsf{eval}_\mathcal{R}(t[\overline{s}]_P)$*.*

Proof. Consider that there exists $\delta \in \mathsf{eval}_{\mathcal{R}}(t)$ such that $t \to^n \delta$. Now we prove by induction on n that, for all $\overline{s} \in \mathcal{T}(\mathcal{F})$, $t[\overline{s}]_P \to^* \delta$. We consider that $|\mathcal{T}(\mathcal{C})| > 1$ since it is trivial otherwise.

1. If $n = 0$, then $t = \delta$, $Pos_{i,f}(t) = \varnothing$, and by Lemma 11.25, $rpos_{\mathsf{eval}_{\mathcal{R}}}(t) = \varnothing$. Hence, $P = \varnothing$ and $t[\overline{s}]_P = t = \delta$.

2. If $n > 0$, then $t \xrightarrow{q}_{l\to r} t' \to^{n-1} \delta$, $t|_q = \theta(l)$, and $t' = t[\theta(r)]_q$. Let $P_i = P \cap Pos_{i,f}(t)$ and $P_{rpos} = (P - P_i) \cap rpos_{\mathsf{eval}_{\mathcal{R}}}(t)$. Now, we prove that $t[\overline{s}|_{P_i}]_{P_i} \to^* \delta$.

   (a) If $\forall p \in P_i$, $p||q$ or $\exists p \in P_i.p \leq q$, then since P is a disjoint set, $P_i = P_i\backslash\langle q, l\to r\rangle$, and by induction hypothesis, $t'[\overline{s}|_{P_i}]_{P_i} \to^* \delta$. Thus, $t[\overline{s}|_{P_i}]_{P_i} = t'[\overline{s}|_{P_i}]_{P_i}$ (when there is $p \in P_i.p \leq q$) or $t[\overline{s}|_{P_i}]_{P_i} \xrightarrow{q}_{l\to r} t'[\overline{s}|_{P_i}]_{P_i}$ (when for all $p \in P_i$, $p||q$), and then, $t[\overline{s}|_{P_i}]_{P_i} \to^* \delta$.

   (b) If there is $p \in P_i.q < p$, then by definition of $Pos_{i,f}(t)$, for all $p \in P_i.q < p$, there exists $p'.(q \leq p' < p \wedge root(t|_{p'}) = f)$. Consider the set $P' = \{p \in P_i \mid q.i \not\leq p\}$ (when $root(l) = f$) or $P' = P_i$ (when $root(l) \neq f$). We have that for all $p \in P'.(p = q.p_1)$, $\exists p_2.(q < p_2 < p_1 \wedge root(t|_{q.p_2}) = f)$, and since l is a pattern, there exists $p_3.(q < p_3 \leq p_2 \wedge l|_{p''} \in \mathcal{X})$. Thus, by left-linearity, there exists $\overline{s'} \in \mathcal{T}(\mathcal{F})$ such that $t[\overline{s}|_{P'}]_{P'} \xrightarrow{q}_{l\to r} t'[\overline{s'}]_{P'\backslash\langle q,l\to r\rangle}$. Also, $P'\backslash\langle q, l\to r\rangle \subseteq Pos_{i,f}(t')$, and by induction hypothesis, $t'[\overline{s'}]_{P'\backslash\langle q,l\to r\rangle} \to^* \delta$. Thus, $t[\overline{s}|_{P'}]_{P'} \to^* \delta$.

   Now, if $root(l) = f$, $P_i - P' = q.i.(P_i|_{q.i})$, and since $l|_i \in \mathcal{X}$, by left-linearity of $\mathcal{R}$, there exists $\overline{s''} \in \mathcal{T}(\mathcal{F})$ such that $t[\overline{s}|_{P_i}]_{P_i} \xrightarrow{q}_{l\to r} t'[\overline{s''}]_{P_i\backslash\langle q,l\to r\rangle}$. By (f,i)-redundancy of variables, $P_i\backslash\langle q, l\to r\rangle \subseteq Pos_{i,f}(t') \cup rpos_{\mathsf{eval}_{\mathcal{R}}}(t')$. Then by induction hypothesis, $t'[\overline{s''}]_{P_i\backslash\langle q,l\to r\rangle} \to^* \delta$. And, $t[\overline{s}|_{P_i}]_{P_i} \to^* \delta$.

   Finally, $(t[\overline{s}|_{P_i}]_{P_i})[\overline{s}|_{P_{rpos}}]_{P_{rpos}} \to^* \delta$, i.e. $t[\overline{s}]_P \to^* \delta$.

Now, for proving the opposite direction it suffices to consider that there exists $\delta \in \mathcal{T}(\mathcal{C})$ such that $t[\overline{s}]_P \to^* \delta$, then $t \to^* \delta$. □

Now, we are able to provide the first effective method to detect redundancy.

Theorem 11.28 *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be a left-linear CS. Let $f \in \mathcal{F}$ and $i \in \{1, \ldots, ar(f)\}$. If, for all $l \to r \in \mathcal{R}_f$, $l|_i$ is a variable which is (f,i)-redundant in r, then $i \in rarg_{\mathsf{eval}_{\mathcal{R}}}(f)$.*

Proof. Let $C[\,]$ be a context such that $C|_p = \square$, and $t, s \in \mathcal{T}(\mathcal{F})$ be terms such that $root(t) = f$, by Proposition 11.27, $\mathsf{eval}_{\mathcal{R}}(C[t]) = \mathsf{eval}_{\mathcal{R}}(C[t[s]_i])$. □

Example 11.29 *A standard example in the literature on* useless variable elimination *(UVE) -a popular technique for removing dead variables, see [Wand and Siveroni, 1999; Kobayashi, 2000]- is the following program*[1]*:*

```
loop(a,bogus,0) → loop(f(a,0),s(bogus),s(0))
loop(a,bogus,s(j)) → a
```

Here it is clear that the second argument does not contribute to the value of the computation. By Theorem 11.28, the second argument of `loop` *is redundant w.r.t.* $\mathsf{eval}_{\mathcal{R}}$*.*

The following example demonstrates that the restriction to constructor systems in Theorem 11.28 above is necessary.

Example 11.30 *Consider the following TRS* $\mathcal{R}$*:*

```
f(x) → g(f(x))
g(f(x)) → x
```

Then, the argument 1 *of* `f(x)` *in the lhs of the first rule is a variable which, in the corresponding rhs of the rule, occurs within the argument* 1 *of a subterm rooted by* f*, namely* `f(x)`*. Hence, by Theorem 11.28 we would have that* $1 \in rarg_{\mathsf{eval}_{\mathcal{R}}}(\mathtt{f})$*. However,* $\mathsf{eval}_{\mathcal{R}}(\mathtt{f(a)}) = \{\mathtt{a}\} \neq \{\mathtt{b}\} = \mathsf{eval}_{\mathcal{R}}(\mathtt{f(b)})$ *(considering* $\mathtt{a}, \mathtt{b} \in \mathcal{C}$*), which contradicts* $1 \in rarg_{\mathsf{eval}_{\mathcal{R}}}(\mathtt{f})$*.*

Example 11.31 *Let us revisit the following CS* $\mathcal{R}$ *from Example 10.1:*

```
applast(nil,z) → z
applast(x:xs,z) → lastnew(x,xs,z)
lastnew(x,nil,z) → z
lastnew(x,y:ys,z) → lastnew(y,ys,z)
```

Using Theorem 11.28, we are also able to conclude that the first argument of function `lastnew` *is redundant w.r.t.* $\mathsf{eval}_{\mathcal{R}}$*.*

Unfortunately, Theorem 11.28 does not suffice to prove that the *second* argument of `lastnew` is redundant w.r.t. $\mathsf{eval}_{\mathcal{R}}$. In the following, we provide a different sufficient criterion for redundancy which is less demanding regarding the shape of the left hand sides, although it requires orthogonality and eval-definedness, in return. The following definitions are auxiliary.

Definition 11.32 *Let* $\mathcal{F}$ *be a signature,* $t = f(t_1, \ldots, t_k)$*,* $s = f(s_1, \ldots, s_k)$ *be terms and* $i \in \{1, \ldots, k\}$*. We say that* t *and* s unify up to i-th argument with *mgu* $\mathcal{F}$ *if* $\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_k\rangle$ *and* $\langle s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k\rangle$ *unify mith mgu* σ*.*

[1] The original example uses natural `100` as stopping criteria for the third argument, while we simplify here to natural `1` in order to code it as `0`/`s` terms.

Definition 11.33 (**(f, i)-triple**) *Let $\mathcal{R} = (\mathcal{F}, R)$ be a CS, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$. Given two different (possibly renamed) rules $l \to r$, $l' \to r'$ in $\mathcal{R}_f$ such that $\mathcal{V}ar(l) \cap \mathcal{V}ar(l') = \varnothing$, we say that $\langle l \to r, l' \to r', \sigma\rangle$ is an (f, i)-triple of $\mathcal{R}$ if l and l' unify up to i-th argument with* mgu σ.

Example 11.34 *Consider the CS $\mathcal{R}$ from Example 11.31. This program has a single* $(\texttt{lastnew}, 2)$*-triple:*
$\langle$`lastnew(x,nil,z)`$\to$`z`, `lastnew(x,y:ys,z)`$\to$`lastnew(y,ys,z)`, $id\rangle$

Definition 11.35 (**Joinable (f, i)-triple**) *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be a CS, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$. An (f, i)-triple $\langle l \to r, l' \to r', \sigma\rangle$ of $\mathcal{R}$ is joinable if $\sigma_{\mathcal{C}}(r)$ and $\sigma_{\mathcal{C}}(r')$ are joinable (i.e., they have a common reduct). Here, substitution $\sigma_{\mathcal{C}}$ is given by:*

$$\sigma_{\mathcal{C}}(x) = \begin{cases} \sigma(x) \text{ if } x \notin \mathcal{V}ar(l|_i) \cup \mathcal{V}ar(l'|_i) \\ a \quad \text{otherwise, where } a \in \mathcal{C} \text{ is an arbitrary constant of appropriate sort.} \end{cases}$$

Example 11.36 *Consider again the CS $\mathcal{R}$ and the single* $(\texttt{lastnew}, 2)$*-triple given in Example 11.34. With ϑ given by $\vartheta = \{\texttt{y} \mapsto \texttt{0},\ \texttt{ys} \mapsto \texttt{nil}\}$, the corresponding rhs's instantiated by ϑ, namely* `z` *and* `lastnew(0,nil,z)`*, are joinable (*`z` *is the common reduct). Hence, the considered* $(\texttt{lastnew}, 2)$*-triple is joinable.*

Roughly speaking, the result below formalizes a method to determine redundancy w.r.t. eval which is based on finding a common reduct of (some particular instances of) the right-hand sides of rules.

Definition 11.37 (**(f, i)-joinable TRS**) *Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS, S be a rewriting semantics for $\mathcal{R}$, $f \in \mathcal{F}$, and $i \in \{1, \ldots, ar(f)\}$. $\mathcal{R}$ is (f, i)-joinable if, for all $l \to r \in \mathcal{R}_f$ and $x \in \mathcal{V}ar(l|_i)$, x is (f, i)-redundant in r and all (f, i)-triples of $\mathcal{R}$ are joinable.*

In order to prove Theorem 11.40 below, we introduce the following auxiliary results. In the following, we will use notation $\overline{t}$ either for a k-tuple of terms $t_1, \ldots, t_k$ or for a sequence of a unique term $t, \ldots, t$; the distinction will be clarified by the context.

Lemma 11.38 *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be an orthogonal and $\mathsf{eval}_{\mathcal{R}}$-defined CS. Let $f \in \mathcal{F}$ and $i \in \{1, \ldots, ar(f)\}$. Let $c \in \mathcal{C}$ be an arbitrary constant. Let $t \xrightarrow{q}_{l \to r} t' \to^* \delta$, $l \to r \in \mathcal{R}_f$, $t|_q = \sigma(l)$, $t' = t[\sigma(r)]_q$, and $\delta \in \mathcal{T}(\mathcal{C})$. Let $P \subseteq \{p \in \mathcal{P}os(t) \mid q.i \leq p\}$ be a disjoint set. If $\mathcal{R}$ is (f, i)-joinable, and for all $P' \subseteq Pos_{i,f}(t') \cup rpos_{\mathsf{eval}_{\mathcal{R}}}(t')$, $t'[\overline{c}]_{P'} \to^* \delta$, then $t[\overline{c}]_P \to^* \delta$.*

Proof. Let $P_c = \{q.p \in P \mid root(l|_p) \neq c\}$. Since l is a pattern, $P - P_c = \{q.p \in P \mid \exists p' \leq p, l|_{p'} \in \mathcal{X}\} \cup \{q.p \in P \mid l|_p = c\}$. By left-linearity, there exists

$\sigma' \in Subst(\mathcal{T}(\mathcal{F}))$ such that $t[\overline{c}]_{P-P_c}|_q = \sigma'(l)$ and $t[\overline{c}]_{P-P_c} \xrightarrow{q}_{l\to r} t'[\overline{c}]_{(P-P_c)\backslash\langle q,l\to r\rangle}$. By (f,i)-redundancy of variables of $l|_i$, $(P-P_c)\backslash\langle q,l\to r\rangle \subseteq Pos_{i,f}(t') \cup rpos_{\mathsf{eval}_\mathcal{R}}(t')$, and by hypothesis, $t'[\overline{c}]_{(P-P_c)\backslash\langle q,l\to r\rangle} \to^* \delta$. Thus, $t[\overline{c}]_{P-P_c} \to^* \delta$.

Then, by $\mathsf{eval}_\mathcal{R}$-definedness, there exists $l' \to r' \in \mathcal{R}_f$ and $\sigma'' \in Subst(\mathcal{T}(\mathcal{F}))$ such that $t[\overline{c}]_P|_q \to^* \sigma''(l')$. By confluence, for all $q.p \in P_c$, $l'|_p = c$. Then, l and l' unify up to the i-th argument with mgu θ. Thus, $\langle l \to r, l' \to r', \theta\rangle$ is a joinable (f,i)-triple of $\mathcal{R}$.

Moreover, we have that there exists a substitution $\tilde{\sigma}' \in Subst(\mathcal{T}(\mathcal{F}))$ such that $t[\overline{c}]_{P-P_c}|_q \to^* \tilde{\sigma}'(l)$ and $\tilde{\sigma}' = \varphi \circ \theta$. Also, $\sigma'' = \varphi' \circ \theta$. Note that by left-linearity, $\varphi(x) = \varphi'(x)$ if $x \notin \bigcup_{p\in P_c} \mathcal{V}ar(\theta(l)|_p) \cup \mathcal{V}ar(\theta(l')|_p)$. By joinability of (f,i)-triples, there exists $w \in \mathcal{T}(\mathcal{F},\mathcal{X})$ such that $\theta_\mathcal{C}(r) \to^* w$ and $\theta_\mathcal{C}(r') \to^* w$. By stability of $\to^*$, $\varphi\circ\theta_\mathcal{C}(r) \to^* \varphi(w)$ and $\varphi'\circ\theta_\mathcal{C}(r') \to^* \varphi'(w)$. Finally, by definition of $\theta_\mathcal{C}$, $\varphi(w) = \varphi'(w)$.

Now, we need to prove that $t[\overline{c}]_{P-P_c}[\varphi\circ\theta_\mathcal{C}(r)]_q \to^* \delta$, i.e. $t[\varphi\circ\theta_\mathcal{C}(r)]_q \to^* \delta$. Consider the set $P_{\theta_\mathcal{C}} = \{q.p' \in \mathcal{P}os(t') \mid \exists q.p \in P_c, \exists x \in \mathcal{V}ar(l|_p) \wedge r|_{p'} = x\}$. By (f,i)-redundancy of variables of $l|_i$, $P_{\theta_\mathcal{C}} \subseteq Pos_{i,f}(t') \cup rpos_{\mathsf{eval}_\mathcal{R}}(t')$. Let $P'_{\theta_\mathcal{C}} = (P - P_c)\backslash\langle q, l \to r\rangle) \cup P_{\theta_\mathcal{C}}$, $P'_{\theta_\mathcal{C}} \subseteq Pos_{i,f}(t') \cup rpos_{\mathsf{eval}_\mathcal{R}}(t')$, and by hypothesis, $t'[\overline{c}]_{P'_{\theta_\mathcal{C}}} \to^* \delta$. But, by definition of $\theta_\mathcal{C}$, $t'[\overline{c}]_{P'_{\theta_\mathcal{C}}}|_q \to^* \varphi\circ\theta_\mathcal{C}(r)$. Thus, by confluence, $t[\varphi\circ\theta_\mathcal{C}(r)]_q \to^* \delta$. That is, $t[\varphi(w)]_q \to^* \delta$ and $t[\varphi'\circ\theta_\mathcal{C}(r')]_q \to^* \delta$.

Finally, let us prove that $t[\varphi'\circ\theta(r')]_q \to^* \delta$. For all $q.p \in P_c$, $l'|_p = c$, and by definition of $\theta_\mathcal{C}$, $\varphi'\circ\theta_\mathcal{C}(r') = \varphi'\circ\theta(r')$. Then, $t[\varphi'\circ\theta(r')]_q \to^* \delta$. Hence, $t[\overline{c}]_P \to^* \delta$. □

Proposition 11.39 *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be an orthogonal and $\mathsf{eval}_\mathcal{R}$-defined CS. Let $f \in \mathcal{F}$ and $i \in \{1,\ldots,ar(f)\}$. Let $c \in \mathcal{C}$ be an arbitrary constant. Let $t \in \mathcal{T}(\mathcal{F})$ and $P \subseteq Pos_{i,f}(t) \cup rpos_{\mathsf{eval}_\mathcal{R}}(t)$ be a disjoint set. If $\mathcal{R}$ is (f,i)-joinable, then $\mathsf{eval}_\mathcal{R}(t) = \mathsf{eval}_\mathcal{R}(t[\overline{c}]_P)$.*

Proof. By $\mathsf{eval}_\mathcal{R}$-definedness and confluence of $\mathcal{R}$, $\mathsf{eval}_\mathcal{R}(t) = \{\delta\}$ with $\delta \in \mathcal{T}(\mathcal{C})$. We consider $t \to^n \delta$. Now, we prove by induction on n that $t[\overline{c}]_P \to^* \delta$. We consider that $|\mathcal{T}(\mathcal{C})| > 1$ since it is trivial otherwise.

1. If $n = 0$, then $t = \delta$, $Pos_{i,f}(t) = \varnothing$, and by Lemma 11.25, $rpos_{\mathsf{eval}_\mathcal{R}}(t) = \varnothing$. Hence, $P = \varnothing$ and $t[\overline{s}]_P = t = \delta$.

2. If $n > 0$, then $t \xrightarrow{q}_{l\to r} t' \to^{n-1} \delta$, $t|_q = \theta(l)$, and $t' = t[\sigma(r)]_q$. Let $P_i = P \cap Pos_{i,f}(t)$ and $P_{rpos} = (P - P_i) \cap rpos_{\mathsf{eval}_\mathcal{R}}(t)$. Now, we prove that $t[\overline{c}]_{P_i} \to^* \delta$.

   (a) If for all $p \in P_i$, $p||q$ or there exists $p \in P_i.p \leq q$, then since P is a disjoint set, $P_i = P_i\backslash\langle q, l\to r\rangle$, and by induction hypothesis, $t'[\overline{c}]_{P_i} \to^* \delta$. Thus, $t[\overline{c}]_{P_i} = t'[\overline{c}]_{P_i}$ (when there exists $p \in P_i.p \leq q$) or $t[\overline{c}]_{P_i} \xrightarrow{q}_{l\to r} t'[\overline{c}]_{P_i}$ (when for all $p \in P_i$, $p||q$), and then, $t[\overline{c}]_{P_i} \to^* \delta$.

(b) If there exists $p \in P_i.q < p$, then by (f,i)-redundancy of variables of $l|_i$, for all $p \in P_i.q < p$, there exists $p'.(q \leq p' < p \wedge root(t|_{p'}) = f)$. Consider the set $P' = \{p \in P_i \mid q.i \not\leq p\}$ (if $root(l) = f$) or $P' = P_i$ (when $root(l) \neq f$). In fact, we have that for all $p \in P'.(p = q.p_1)$, there exists $p_2.(q < p_2 < p_1 \wedge root(t|_{q.p_2}) = f)$, and since l is a pattern, there exists $p_3.(q < p_3 \leq p_2 \wedge l|_{p''} \in \mathcal{X})$. Thus by left-linearity, $t[\overline{c}]_{P'} \xrightarrow{q}_{l\to r} t'[\overline{c}]_{P'\setminus\langle q,l\to r\rangle}$. Also, $P'\setminus\langle q,l\to r\rangle \subseteq Pos_{i,f}(t')$, and by induction hypothesis, $t'[\overline{c}]_{P'\setminus\langle q,l\to r\rangle} \to^* \delta$. Thus, $t[\overline{c}]_{P'} \to^* \delta$.
Now, if $root(l) = f$, then $P_i - P' = q.i.(P_i|_{q.i})$, and by Lemma 11.38, $(t[\overline{c}]_{P'})[\overline{c}]_{q.i.(P_i|_{q.i})} \to^* \delta$, i.e. $t[\overline{c}]_{P_i} \to^* \delta$.

Finally, $(t[\overline{c}]_{P_i})[\overline{c}]_{P_{rpos}} \to^* \delta$, i.e. $t[\overline{c}]_P \to^* \delta$. □

Theorem 11.40 *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{F}, R)$ be an orthogonal and $\mathsf{eval}_{\mathcal{R}}$-defined CS. Let $f \in \mathcal{F}$ and $i \in \{1,\ldots,ar(f)\}$. If $\mathcal{R}$ is (f,i)-joinable, then $i \in rarg_{\mathsf{eval}_{\mathcal{R}}}(f)$.*

Proof. Let $c \in \mathcal{C}$ be an arbitrary constant. Let $C[\ ]$ be a context such that $C|_p = \Box$, and $t, s \in \mathcal{T}(\mathcal{F})$ be terms such that $root(t) = f$. By Proposition 11.39, we have that $\mathsf{eval}_{\mathcal{R}}(C[t]) = \mathsf{eval}_{\mathcal{R}}(C[t[c]_i])$ and $\mathsf{eval}_{\mathcal{R}}(C[t[s]_i]) = \mathsf{eval}_{\mathcal{R}}(C[t[c]_i])$. Thus, $\mathsf{eval}_{\mathcal{R}}(C[t]) = \mathsf{eval}_{\mathcal{R}}(C[t[s]_i])$. □

Joinability is decidable for terminating, confluent TRSs as well as for other classes of TRSs such as right-ground TRSs (see e.g., [Oyamaguchi, 1990]). Hence, Theorem 11.40 gives us an effective method to recognize redundancy in CD, left-linear, and (semi-)complete TRSs, as illustrated in the following.

Example 11.41 *Consider again the CS $\mathcal{R}$ of Example 11.34. This program is orthogonal, terminating and CD (considering sorts), hence is* eval*-defined. Now, we have the following. The first argument of* `lastnew` *is redundant w.r.t.* $\mathsf{eval}_{\mathcal{R}}$ *(Theorem 11.28). The second argument of* `lastnew` *is redundant w.r.t.* $\mathsf{eval}_{\mathcal{R}}$ *(Theorem 11.40). As a consequence, the positions of variables* `x` *and* `xs` *in the rhs of the first rule of* `applast` *have been proven redundant. Then, since both* `lastnew(0,nil,z)` *and* `z` *rewrite to* `z`*, $\mathcal{R}_{\mathtt{applast}}$ is $(\mathtt{applast}, 1)$-joinable. By Theorem 11.40, we conclude that the first argument of* `applast` *is redundant.*

11.4.2 Experiments

The practicality of our ideas is witnessed by the implementation of a prototype system which delivers encouraging good results for the techniques deployed in this section, together with the erasure procedure of Section 10.6. The prototype has been implemented in PAKCS, the current distribution of the multi-paradigm declarative language Curry [Hanus *et al.*, 2003], and is publicly available at http://www.dsic.upv.es/users/elp/redargs

We have used the prototype to perform some preliminary experiments which show that our methodology does detect and remove redundant arguments of some common transformation benchmarks, such as `bogus`, `lastappend`, `allzeros`, `doubleflip`, etc (see [Leuschel, 1998] and references therein). Tables 11.1 and 11.2 summarize the experiments. Benchmarks code as well as the program obtained by the erasure procedure are included in Appendix E.

Table 11.1 shows the execution runtimes of the original and transformed programs in Curry[2]. Important optimizations are obtained for most examples. In the case of program `bogus`, no appreciable optimization is achieved by removing redundant arguments, since Curry is a lazy language and the redundant argument in `bogus` is a useless variable. Runtimes have been measured in a Pentium III class machine running linux and using version 1.3.2 of the PAKCS Compiler under Sicstus Prolog 3.8.4. Natural numbers are given by numbers 0, 1, 2, etc instead of the notation `Z`/`S` x. For benchmarking purposes, goals make use of the auxiliary factorial function, defined in a usual way. The number of elements of a list (when used) is indicated by a subindex. In the case of binary trees, the subindex indicates the depth of the tree.

In order to overcome the difference in execution times associated to a lazy language. Table 11.2 shows the execution runtimes of the benchmarks in the CiME 2.0 system[3], which uses an innermost rewriting strategy. Note that, in this case, significant optimizations are also measured for programs `bogus`, `applast` and `plus_minus`.

11.5 Conclusion

We have shown how the problem of detecting redundant arguments can be addressed by different techniques. First, we have provided an approximation technique based on decidability issues which can detect redundant arguments in many practical examples. Since the redundancy problem can be expressed in terms of validity of inductive theorems in confluent, eval-defined TRSs, we have shown how "inductionless induction" as well as "abstract rewriting" techniques can be applied to detect redundant arguments. Moreover, since the set of inductive theorems is not generally recursively enumerable, we identify a class of rewrite systems in which detection of redundant arguments is decidable. This result is incomparable to the decidability of Section 10.4. On the other hand, in order to provide more practical methods to recognize redundancy, we have ascertained suitable conditions which allow us to simplify the general redundancy problem to the analysis of redundant positions within rhs's of

[2] See http://www.informatik.uni-kiel.de/~curry/examples/

[3] See http://cime.lri.fr/.

Name	Call in original/erased program	Time (ms)	Improvement
bogus	`loop (fact 8) (fact 8) (fact 8)`	260	
	`loop' (fact 8) (fact 8)`	260	0%
applast	`applast [(fact 8)]`$_{1000}$ `(fact 8)`	340	
	`applast' (fact 8)`	260	20%
plus_minus	`minus_pe (fact 8) (fact 8)`	480	
	`minus_pe' (fact 8)`	270	40%
plus_leq	`leq_pe (fact 8) (fact 8)`	210	
	`leq_pe'`	∼0	100%
double_even	`even_pe (fact 8)`	210	
	`even_pe'`	∼0	100%
minimal_append	`minimal_pe [(fact 8)]`$_{10000}$	5	
	`minimal_pe'`	∼0	100%
sum_allzeros	`sum_pe [(fact 8)]`$_{10000}$	70	
	`sum_pe'`	∼0	100%
Mutual recursion 1	`f (fact 8) (fact 8)`	320	
	`f'`	∼0	100%
Mutual recursion 2	`f (fact 8)`	380	
	`f'`	∼0	100%

Table 11.1: Execution of the original and transformed programs in Curry.

the program rules. These conditions are quite demanding (requiring $\mathcal{R}$ to be orthogonal and $\mathsf{eval}_{\mathcal{R}}$-defined) but also the optimization which they enable is strong, and powerful.

It is worth noting that the four different methods are incomparable. For instance, redundant arguments in Example 11.4 of approximations of redundancy are not identified by the characterization of Section 11.4, the inductionless induction method of Section 11.2, or the abstract rewriting method of Section 11.3. On the other hand, redundancies in Example 11.5 of inductionless induction of Section 11.2 are not identified by the approximation of redundancy of Section 11.1, the characterization of Section 11.4, or the abstract rewriting method of Section 11.3. Moreover, redundancies in Example 11.20 dealt by abstract rewriting of Section 11.3 are not identified by the approximation of redundancy of Section 11.1, the characterization of Section 11.4, or the inductionless induction of Section 11.2. Finally, the characterization of Section 11.4 is also incomparable to the others, as shown by the following example.

Example 11.42 *Consider the following TRS $\mathcal{R}$ which is a slight modification of Example 3.26 of [Arts and Giesl, 2001]:*

```
h(0,0) → s(0)
h(0,s(y)) → s(0)
```

Name	Call in original/erased program	Time (ms)	Improvement
bogus	`loop (fact 6) (fact 6) (fact 6)`	2820	
	`loop' (fact 6) (fact 6)`	1880	33%
applast	`applast [(fact 6)]`$_{10}$ `(fact 6)`	2570	
	`applast' (fact 6)`	940	64%
plus_minus	`minus_pe (fact 6) (fact 6)`	7930	
	`minus_pe' (fact 6)`	960	87%
plus_leq	`leq_pe (fact 6) (fact 6)`	7970	
	`leq_pe'`	∼0	100%
double_even	`even_pe (fact 6)`	1930	
	`even_pe'`	∼0	100%
minimal_append	`minimal_pe [(fact 6)]`$_{10}$	1070	
	`minimal_pe'`	∼0	100%
sum_allzeros	`sum_pe [0]`$_{1000}$	1330	
	`sum_pe'`	∼0	100%
Mutual recursion 1	`f (fact 6) (fact 6)`	9580	
	`f'`	∼0	100%
Mutual recursion 2	`f (fact 6)`	2020	
	`f'`	∼0	100%

Table 11.2: Execution of the original and transformed programs in CiME 2.0

$$\mathtt{h(s(0),y)} \to \mathtt{s(s(0))}$$
$$\mathtt{h(s(s(x)),y)} \to \mathtt{s(h(h(x,y),y))}$$

The approximation of Section 11.1 is not able to detect redundancy of the two arguments. The inductionless induction method can not automatically prove that the second argument of `h` *is redundant since there is no automatizable reduction ordering for the TRS which could orient the equations (see [Arts and Giesl, 2001]). On the other hand, abstract rewriting cannot prove the redundancy of the second argument of* `h` *since, for any finite upper closure, there exists a value for the second argument which returns a term containing* $\top$*, e.g. using the upper closure head of Example 11.20,* $\mathtt{h}(\mathtt{s}(\top),0)\downarrow_{\mathcal{R}_c^{up}} \equiv \mathtt{s}(\top)$*. Nevertheless, the method in Section 11.4 succeeds in proving that the second argument of* `h` *is redundant since all variables of the second argument appear in positions of redundant arguments of the rhs of the corresponding rule, and* $\mathtt{s(0)} \downarrow \mathtt{s(0)}$*.*

Appendix D Execution Excerpt

In this Appendix, we present the practical analysis and detection of redundant arguments in the program of Example 11.5.

First, in Subsection D-1, we express this program in the syntax of the partial evaluator[4] Indy [Alpuente *et al.*, 1998]. Then, we show in Section D-2 the partially evaluated program, which corresponds to the optimized program of Example 11.5. Next, in order to illustrate the use of the *inductionless induction* method for detecting redundant arguments, the optimized program is translated to the syntax of the theorem prover[5] Spike [Bouhoula and Rusinowitch, 1995] in Subsection D-3. Finally, we transcript the inductive proof generated by Spike in Subsection D-4.

D-1 Original program in Indy

```
plus(:0,Y) -> Y;
plus(:s(X),Y) -> :s(plus(X,Y));

minus(X,:0) -> X;
minus(:0,:s(Y)) -> :0;
minus(:s(X),:s(Y)) -> p(:s(minus(X,Y)));

p(:0) -> :0;
p(:s(:0)) -> :0;
p(:s(:s(X))) -> :s(X);

minusplus(X,Y) -> minus(plus(Y,X),Y);
```

D-2 Specialized program in Indy

```
p1_1(:0) -> :0;
p1_1(:s(A)) -> :s(A);

minusplus2_2(A,:0) -> A;
minusplus2_2(A,:s(B)) -> p1_1(minusplus2_2(A,B));

minusplus2_1(A,:0) -> A;
minusplus2_1(A,:s(B)) -> p1_1(minusplus2_2(A,B));
```

[4] Available at http://www.dsic.upv.es/users/elp/indy/.
[5] Available at http://www.loria.fr/equipes/cassis/softwares/spike/.

D-3 Specialized program translated to Spike

```
specification : minusplus

sorts nat;

constructors :

0  : -> nat;
s_ : nat -> nat;

defined functions :

p_ : nat -> nat;
minusplus__ : nat nat -> nat;
minusplus2__ : nat nat -> nat;

axioms:

p(0) = 0;
p(s(x)) = s(x);

minusplus2(x,0) = x;
minusplus2(x,s(y)) = p(minusplus2(x,y));

minusplus(x,0) = x;
minusplus(x,s(y)) = p(minusplus2(x,y));
```

D-4 Inductive proof in Spike

Below, we include the transcription of a proving session with Spike. The theorem which proves the redundancy of the second argument of `minusplus` is `minusplus(x1,x2)=minusplus(x1,x3)`. This theorem appears as the initial set `E0` of theorems to prove. Some additional notes are included in the transcription to help the reader.

```
All the rules are oriented !

test set of R :

 -> nat = {0 ; s(x1)}
```

```
induction positions of functions:

 -> p : [[1]]
 -> minusplus : [[2]]
 -> minusplus2 : [[2]]

E0 = {minusplus(x1,x2) = minusplus(x1,x3)}           | E0 is the
                                                     | initial set
Application of generate on:                          | of theorems
    minusplus(x1,x2) = minusplus(x1,x3)              | to prove
 with cover substitutions:
    x2 -> {0; s(x1)}

 1) x1 = minusplus(x1,x3) ;
 2) p(minusplus2(x1,x2)) = minusplus(x1,x3)


E1 = {x1 = minusplus(x1,x3) ;                        | After
      p(minusplus2(x1,x2)) = minusplus(x1,x3)}       | superposition

H1 = {minusplus(x1,x2) = minusplus(x1,x3)}           | H1 is the
                                                     | initial set
Application of generate on:                          | of inductive
    x1 = minusplus(x1,x3)                            | hypothesis
 with cover substitutions:
    x3 -> {0; s(x1)}

 1) x1 = x1 ;
 2) x1 = p(minusplus2(x1,x2))



Delete  x1 = x1                                      | Equation is
                                                     | entailed and
E2 = {p(minusplus2(x1,x2)) = minusplus(x1,x3) ;      | removed
      x1 = p(minusplus2(x1,x2))}

H2 = {x1 = minusplus(x1,x3) ;
```

```
      minusplus(x1,x2) = minusplus(x1,x3)}

Application of generate on:
    p(minusplus2(x1,x2)) = minusplus(x1,x3)
 with cover substitutions:
    x2 -> {0; s(x1)}

 1) p(x1) = minusplus(x1,x3) ;
 2) p(p(minusplus2(x1,x2))) = minusplus(x1,x3)


E3 = {x1 = p(minusplus2(x1,x2)) ;
      p(x1) = minusplus(x1,x3) ;
      p(p(minusplus2(x1,x2))) = minusplus(x1,x3)}

H3 = {p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
      x1 = minusplus(x1,x3) ;
      minusplus(x1,x2) = minusplus(x1,x3)}

Simplification of:
    p(p(minusplus2(x1,x2))) = minusplus(x1,x3) by H3 U E3[R]:
    p(x1) = minusplus(x1,x3)

E4 = {x1 = p(minusplus2(x1,x2)) ;                    | Simplification
      p(x1) = minusplus(x1,x3) ;                     | detects possible
      p(x1) = minusplus(x1,x3)}                      | entailed
H4 = {p(minusplus2(x1,x2)) = minusplus(x1,x3) ;     | equations
      x1 = minusplus(x1,x3) ;
      minusplus(x1,x2) = minusplus(x1,x3)}


Delete  p(x1) = minusplus(x1,x3)                     | Subsumption
 it is subsumed by:p(x1) = minusplus(x1,x3) of E4   | also detects
                                                     | entailed
E5 = {x1 = p(minusplus2(x1,x2)) ;                    | equations
      p(x1) = minusplus(x1,x3)}

H5 = {p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
      x1 = minusplus(x1,x3) ;
```

```
      minusplus(x1,x2) = minusplus(x1,x3)}

Application of generate on:
    x1 = p(minusplus2(x1,x2))
 with cover substitutions:
    x2 -> {0; s(x1)}

 1) x1 = p(x1) ;
 2) x1 = p(p(minusplus2(x1,x2)))


E6 = {p(x1) = minusplus(x1,x3) ;
      x1 = p(x1) ;
      x1 = p(p(minusplus2(x1,x2)))}

H6 = {x1 = p(minusplus2(x1,x2)) ;
      p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
      x1 = minusplus(x1,x3) ;
      minusplus(x1,x2) = minusplus(x1,x3)}

Simplification of:
    x1 = p(p(minusplus2(x1,x2))) by H6 U E6[R]:
    x1 = p(minusplus2(x1,x2))

E7 = {p(x1) = minusplus(x1,x3) ;
      x1 = p(x1) ;
      x1 = p(minusplus2(x1,x2))}

H7 = {x1 = p(minusplus2(x1,x2)) ;
      p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
      x1 = minusplus(x1,x3) ;
      minusplus(x1,x2) = minusplus(x1,x3)}


Delete  x1 = p(minusplus2(x1,x2))
 it is subsumed by:x1 = p(minusplus2(x1,x2)) of H7

E8 = {p(x1) = minusplus(x1,x3) ;
      x1 = p(x1)}
```

```
H8 = {x1 = p(minusplus2(x1,x2)) ;
      p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
      x1 = minusplus(x1,x3) ;
      minusplus(x1,x2) = minusplus(x1,x3)}

Application of generate on:
    p(x1) = minusplus(x1,x3)
 with cover substitutions:
    x3 -> {0; s(x1)}

 1) p(x1) = x1 ;
 2) p(x1) = p(minusplus2(x1,x2))


E9 = {x1 = p(x1) ;
      p(x1) = x1 ;
      p(x1) = p(minusplus2(x1,x2))}

H9 = {p(x1) = minusplus(x1,x3) ;
      x1 = p(minusplus2(x1,x2)) ;
      p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
      x1 = minusplus(x1,x3) ;
      minusplus(x1,x2) = minusplus(x1,x3)}


Delete  x1 = p(x1)
 it is subsumed by:p(x1) = x1 of E9

E10 = {p(x1) = x1 ;
       p(x1) = p(minusplus2(x1,x2))}

H10 = {p(x1) = minusplus(x1,x3) ;
       x1 = p(minusplus2(x1,x2)) ;
       p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
       x1 = minusplus(x1,x3) ;
       minusplus(x1,x2) = minusplus(x1,x3)}

Application of generate on:
```

```
    p(x1) = x1
 with cover substitutions:
    x1 -> {0; s(x1)}

 1) 0 = 0 ;
 2) s(x1) = s(x1)



Delete  0 = 0


Delete  s(x1) = s(x1)

E11 = {p(x1) = p(minusplus2(x1,x2))}

H11 = {p(x1) = x1 ;
       p(x1) = minusplus(x1,x3) ;
       x1 = p(minusplus2(x1,x2)) ;
       p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
       x1 = minusplus(x1,x3) ;
       minusplus(x1,x2) = minusplus(x1,x3)}

Application of generate on:
    p(x1) = p(minusplus2(x1,x2))
 with cover substitutions:
    x2 -> {0; s(x1)}

 1) p(x1) = p(x1) ;
 2) p(x1) = p(p(minusplus2(x1,x2)))



Delete  p(x1) = p(x1)

E12 = {p(x1) = p(p(minusplus2(x1,x2)))}

H12 = {p(x1) = p(minusplus2(x1,x2)) ;
       p(x1) = x1 ;
```

```
       p(x1) = minusplus(x1,x3) ;
       x1 = p(minusplus2(x1,x2)) ;
       p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
       x1 = minusplus(x1,x3) ;
       minusplus(x1,x2) = minusplus(x1,x3)}

Simplification of:
    p(x1) = p(p(minusplus2(x1,x2))) by H12 U E12[R]:
    p(x1) = p(x1)

E13 = {p(x1) = p(x1)}

H13 = {p(x1) = p(minusplus2(x1,x2)) ;
       p(x1) = x1 ;
       p(x1) = minusplus(x1,x3) ;
       x1 = p(minusplus2(x1,x2)) ;
       p(minusplus2(x1,x2)) = minusplus(x1,x3) ;
       x1 = minusplus(x1,x3) ;
       minusplus(x1,x2) = minusplus(x1,x3)}


Delete  p(x1) = p(x1)                                   | E13 becomes
                                                        | empty and the
~                                                       | saturation
The initial conjectures are inductive theorems of R.| process ends
                                                        | without proving
#~                                                      | inconsistency
~
```

Appendix E Benchmarks Code

We give some example programs which contain redundant arguments, borrowed from the literature and/or obtained by applying common transformation processes. For each example, we show the final program which results from optimizing the program by using our automatic redundant argument removal tool.

E-1 Program bogus

The following program `bogus` is borrowed from [Kobayashi, 2000; Wand and Siveroni, 1999], where it is introduced for useless variable elimination (UVE), a popular technique for removing dead variables.

```
data nat = Z | S nat
loop :: nat -> nat -> nat -> nat
loop a bogus Z     = loop (S a) (S bogus) (S Z)
loop a bogus (S x) = a
```

The second argument of `loop` is signaled as redundant and then removed.

```
loop' :: nat -> nat -> nat
loop' a Z     = loop' (S a) (S Z)
loop' a (S x) = a
```

E-2 Mutual Recursion 1

This program is taken from Example 2.23 of [Arts and Giesl, 2001].

```
data nat = Z | S nat
f :: nat -> nat -> nat
f Z     y = Z
f (S x) y = f (f x y) y
```

Both arguments of `f` are identified as redundant and removed.

```
f' :: nat
f' = Z
```

E-3 Mutual Recursion 2

This program is taken from Example 2.24 of [Arts and Giesl, 2001].

```
data nat = Z | S nat
f :: nat -> nat
f Z         = S Z
f (S Z)     = S Z
f (S (S x)) = f (f (S x))
```

The argument of `f` is identified as redundant and removed.

```
f' :: nat
f' = S Z
```

E-4 Program `applast`

The following program is borrowed from [Leuschel, 1998] and is obtained by program specialization [Alpuente *et al.*, 1998].

```
data Nat = 0 | S Nat
```

```
append::[Nat] -> [Nat] -> [Nat]
append nil    y = y
append (x:xs) y = x:(append xs y)
last::[Nat] -> Nat
last (x:nil)  = x
last (x:y:ys) = last (y:ys)
```

The specialization of the program `applast` for goal `last (append xs (x:nil))` yields:

```
applast::[Nat] -> Nat -> Nat
applast nil    z = z
applast (x:xs) z = lastnew x xs z
lastnew::Nat -> [Nat] -> Nat -> Nat
lastnew x nil    z = z
lastnew x (y:ys) z = lastnew y ys z
```

The first argument of `applast` and the first and second arguments of `lastnew` are identified as redundant and removed.

```
applast' :: nat -> nat
applast' z = z
lastnew' :: Nat -> Nat
lastnew' z = z
```

E-5 Program `plus_minus`

This example is borrowed from [Leuschel, 1998] and is obtained by program specialization [Alpuente *et al.*, 1998].

```
data nat = Z | S nat
plus :: nat -> nat -> nat
plus Z     x = x
plus (S x) y = S (plus x y)
minus :: nat -> nat -> nat
minus x     Z     = x
minus (S x) (S y) = minus(X,Y)
```

The specialization for goal `minus (plus x y) x` yields:

```
minus_pe :: nat -> nat -> nat
minus_pe Z     y = y
minus_pe (S x) y = minus_pe x y
```

The first argument of `minus_pe` is identified as redundant and removed.

```
minus_pe' :: nat -> nat
```

```
minus_pe' y = y
```

E-6 Program `plus_leq`

This example is borrowed from [Leuschel, 1998] and is obtained by program specialization [Alpuente *et al.*, 1998].

```
data nat = Z | S nat
plus :: nat -> nat -> nat
plus Z     x = x
plus (S x) y = S (plus x y)
leq :: nat -> nat -> Bool
leq Z     x     = True
leq (S x) Z     = False
leq (S x) (S y) = leq x y
```

The specialization for goal `leq x (plus x y)` yields:

```
leq_pe :: nat -> nat -> Bool
leq_pe Z     x = True
leq_pe (S x) y = leq_pe x y
```

Both arguments of `leq_pe` are identified as redundant and removed.

```
leq_pe' :: Bool
leq_pe' = True
```

E-7 Program `double_even`

This example is borrowed from [Leuschel, 1998] and is obtained by program specialization [Alpuente *et al.*, 1998].

```
data nat = Z | S nat
double :: nat -> nat
double Z     = Z
double (S x) = S (S (double x))
even :: nat -> Bool
even Z         = True
even (S Z)     = False
even (S (S x)) = even x
```

The specialization for goal `even (double x)` yields:

```
even_pe :: nat -> Bool
even_pe Z = True
even_pe (S x) = even_pe x
```

The argument of `even_pe` is identified as redundant and removed.

```
even_pe' :: Bool
even_pe' = True
```

E-8 Program `minimal_append`

This example is borrowed from [Leuschel, 1998] and is obtained by program specialization [Alpuente *et al.*, 1998].

```
data nat = Z | S nat
min :: nat -> nat -> nat
min Z     x     = Z
min (S x) Z     = Z
min (S x) (S y) = S (min x y)
minimal :: [nat] -> nat
minimal (x:nil)  = x
minimal (x:y:xs) = min x (minimal (y:xs))
append :: [Nat] -> [Nat] -> [Nat]
append nil    y = y
append (x:xs) y = x:(append xs y)
```

The specialization for goal `minimal (append (Z:nil) xs)` yields:

```
minimal_pe :: nat -> nat
minimal_pe nil = Z
minimal_pe (x:y) = Z
```

The argument of `minimal_pe` is identified as redundant and removed.

```
minimal_pe' :: nat
minimal_pe' = Z
```

E-9 Program `sum_allzeros`

This example is borrowed from [Leuschel, 1998] and is obtained by program specialization [Alpuente *et al.*, 1998].

Consider the following program:

```
data nat = Z | S nat
plus :: nat -> nat -> nat
plus Z     x = x
plus (S x) y = S (plus x y)
sum :: [nat] -> nat
sum nil    = Z
```

```
sum (x:xs) = plus x (sum xs)
allzeros :: [nat] -> [nat]
allzeros nil    = nil
allzeros (x:xs) = Z:(allzeros xs)
```

The specialization for goal `sum (allzeros x)` yields:

```
sum_pe :: [nat] -> nat
sum_pe nil    = Z
sum_pe (x:xs) = sum_pe xs
```

The argument of `sum_pe` is identified as redundant and removed.

```
sum_pe' :: nat
sum_pe' = Z
```

Bibliography

Aho A., Sethi R., and Ullman J., 1986. *Compilers, Principles Techniques and Tools.* Addison-Wesley, Reading, MA.

Alpuente M., Echahed R., Escobar S., and Lucas S., 2002a. Redundancy of Arguments Reduced to Induction. In M. Comini and M. Falaschi, editors, *Proc. of the 11th Int'l Workshop on Functional and (Constraint) Logic Programming WFLP'02*, volume 76 of *Electronic Notes in Theoretical Computer Science*, pages 100–200. Elsevier Sciences Publisher.

Alpuente M., Escobar S., Gramlich B., and Lucas S., 2002b. Improving On-demand Strategy Annotations. In M. Baaz and A. Voronkov, editors, *Proc. 9th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'02)*, volume 2514 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, Berlin, Tbilisi, Georgia.

Alpuente M., Escobar S., and Lucas S., 1999a. An Incremental Curry Interpreter. In R. Echahed, editor, *Proc. of the 8th Int'l Workshop on Functional and Logic Programming WFLP'99*, pages 160–170.

Alpuente M., Escobar S., and Lucas S., 1999b. An Incremental Needed Narrowing Calculus for Curry. In M. Meo and M. Vilares, editors, *Proc of the 1999 Joint Conference on Declarative Programming, AGP'99, L'Acquila (Italy)*, pages 75–87.

Alpuente M., Escobar S., and Lucas S., 1999c. Incremental Needed Narrowing. In P. Tarau and K. Sagonas, editors, *Proc of the Int'l Workshop on Implementation of Declarative Languages, IDL'99*, pages 1–18.

Alpuente M., Escobar S., and Lucas S., 1999d. 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., Escobar S., and Lucas S., 2000. Redundant Arguments in Term Rewriting. In M. Alpuente, editor, *Proc. of the 9th Int'l Workshop on Functional and Logic Programming WFLP'00*, SPUPV 2000.2039, pages 309–323.

Alpuente M., Escobar S., and Lucas S., 2002c. Correct and complete (positive) strategy annotations for OBJ. In F. Gadducci and U. Montanari, editors, *Proc. of the 4th International Workshop on Rewriting Logic and its Applications, WRLA 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

Alpuente M., Escobar S., and Lucas S., 2002d. A program transformation for implementing on-demand strategy annotations. In R. Nieuwenhuis, editor, *14th International Workshop on the Implementation of Functional Languages, IFL'02*. Universidad Complutense de Madrid, Madrid, Spain.

Alpuente M., Escobar S., and Lucas S., 2002e. Removing Redundant Arguments of Functions. In H. Kirchner and C. Ringeissen, editors, *9th International Conference on Algebraic Methodology And Software Technology, AMAST 2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 117–131. Springer-Verlag, Berlin.

Alpuente M., Escobar S., and Lucas S., 2002f. A transformation for implementing on-demand strategy annotations. In R. Nieuwenhuis, editor, *3rd International Workshop on the Implementation of Logics*. Universidad Politècnica de Catalunya, Tbilisi, Georgia.

Alpuente M., Escobar S., and Lucas S., 2003a. On-demand evaluation by program transformation. In J. Giavitto and P. Moreau, editors, *Proc. of the 4th International Workshop on Rule-Based Programming, RULE 2003*, volume 86.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

Alpuente M., Escobar S., and Lucas S., 2003b. OnDemandOBJ: a laboratory for strategy annotations. In J. Giavitto and P. Moreau, editors, *Proc. of the 4th International Workshop on Rule-Based Programming, RULE 2003*, volume 86.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

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, number 12 of *ACM Sigplan Notices*, pages 151–162. ACM Press, New York.

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*, volume 1058 of *Lecture Notes in Computer Science*, pages 45–61. Springer-Verlag, Berlin.

Alpuente M., Falaschi M., and Vidal G., 1998. 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., 1999e. Specialization of Inductively Sequential Functional Logic Programs. In P. Lee, editor, *Proc. of the ACM SIGPLAN Conf. on Functional Programming, ICFP'99*, volume 34, number 9 of *ACM Sigplan Notices*, pages 273–283. ACM Press, New York.

Antoy S., 1992. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming ALP'92*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, Berlin.

Antoy S., 1997. Optimal Non-Deterministic Functional Logic Computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, volume 1298 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, Berlin.

Antoy S., 2002. Presentation of [Antoy and Lucas, 2002] at the *11th International Workshop on Functional and (Constraint) Logic Programming (WFLP'02).*

Antoy S., Echahed R., and Hanus M., 1994. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279.

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)*, volume 1794 of *Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, Berlin.

Antoy S. and Lucas S., 2002. Demandness in Rewriting and Narrowing. In M. Comini and M. Falaschi, editors, *Proc. of the 11th Int'l Workshop on Functional and (Constraint) Logic Programming WFLP'02*, volume 76 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

Antoy S. and Middeldorp A., 1996. A Sequential Reduction Strategy. *Theoretical Computer Science*, 165:75–95.

Apt K., 1997. *From Logic Programming to Prolog.* The MIT Press, Cambridge, MA.

Arts T. and Giesl J., 2001. A Collection of Examples for Termination of Term Rewriting Using Dependency Pairs. Technical Report AIB-2001-09, RWTH Aachen, Germany.

Baader F. and Nipkow T., 1998. *Term Rewriting and All That.* Cambridge University Press.

Bachmair L. and Dershowitz N., 1994. Equational Inference, canonical proofs and proof orderings. *Journal of the ACM*, 41(2):236–276.

Bachmair L. and Ganzinger H., 1994. Rewrite-based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247.

Bachmair L. and Ganzinger H., 2001. Resolution Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 2, pages 19–99. Elsevier Science.

Barendregt H., van Eekelen M., Glauert J., Kennaway J., Plasmeijer M., and Sleep M., 1987. Term Graph Rewriting. In *Proc. Parallel Architectures and Languages Europe, PARLE'87*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer-Verlag, Berlin.

Beck K., 1999. *Extreme Programming Explained: Embrace Changes.* Addison-Wesley, Reading, MA.

Berardi S., Coppo M., Damiani F., and Giannini P., 2000. Type-Based Useless-Code Elimination for Functional Programs. In *Proceedings of SAIG 2000*, volume 1924 of *Lecture Notes in Computer Science*, pages 172–189. Springer-Verlag, Berlin.

Bert D. and Echahed R., 1995. Abstraction of Conditional Term Rewriting Systems. In J. Lloyd, editor, *International Symposium on Logic Programming, ILPS-95, Portland, Oregon*, pages 162–176. MIT Press.

Bicarregui J., 1999. Exploiting formality in software engineering. In J. Pavelka, G. Tel, and M. Bartosek, editors, *26th Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 99*, volume 1725 of *Lecture Notes in Computer Science*, pages 163–180. Springer-Verlag, Berlin.

Bidoit M., Kreowski H.J., Lescanne P., Orejas F., and Sannella D., editors, 1993. *Algebraic System Specification and Development. A Survey and Annotated Bibliography*, volume 501 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

Bird R., 1998. *Introduction to Functional Programming Using Haskell.* Prentice-Hall International.

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.

Boehm B., 1981. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ.

Borovanský P., Kirchner C., Kirchner H., Moreau P., and Ringeissen C., 1998. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *2nd International Workshop on Rewriting Logic and its Applications, WRLA 98*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 1–16. Elsevier Sciences Publisher.

Borralleras C., Lucas S., and Rubio A., 2002. Recursive Path Orderings can be Context-Sensitive. In A. Voronkov, editor, *18th International Conference on Automated Deduction, CADE'02*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 314–331. Springer-Verlag, Berlin.

Bosco P., Cecci C., Giovannetti E., Moiso C., and Palamidessi C., 1989. Using resolution for a sound and efficient integration of logic and functional programming. In J. deBakker, editor, *Languages for parallel architectures*. Wiley (UK).

Bouhoula A. and Rusinowitch M., 1995. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14:189–235.

Bowen J.P. and Hinchey M.G., 1995. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41.

Brooks F., 1986. No silver bullet–Essence and accident in software engineering. *Journal of Information Processing*.

Brooks F., 1995. The Mythical Man-Month: Essays on Software Engineering. *Twentieth Anniversary Edition, Reading, MA: Addison-Wesley, 322 pp.*

Brooks F., 2003. Three Great Challenges for Half-Century-Old Computer Science. *Journal of the ACM*, 50(1):25–26.

Broy M., 1999. Software technology formal methods and scientific foundations. *Information and Software Technology*, 41:947–950.

Burn G., 1991. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Pitman, London.

Burn G.L., Hankin C.L., and Abramsky S., 1986. The Theory of Strictness Analysis for Higher Order Functions. In *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 42–62. Springer-Verlag, Berlin.

Clack C. and Peyton-Jones S., 1985. Strictness analysis - a practical approach. In J.P. Jouannaud, editor, *Proc of International Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 35–39. Springer-Verlag, Berlin.

Clavel M., Eker S., Lincoln P., and Meseguer J., 1996. Principles of Maude. In J. Meseguer, editor, *Proc. of the 1st International Workshop on Rewriting Logic and its Applications, RWLW 96*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier Sciences Publisher.

Comon H., 1995. Sequentiality, Second Order Monadic Logic and Tree Automata. In *10th Annual IEEE Symposium on Logic in Computer Science, LICS'95*, pages 508–517. IEEE Computer Society Press.

Comon H., 2000. Sequentiality, Second Order Monadic Logic, and Tree Automata. *Information and Computation*, 157:25–51.

Comon H., 2001. Inductionless induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 14, pages 913–962. Elsevier Science.

Comon H. and Nieuwenhuis R., 2000. Induction = I-Axiomatization + First-Order Consistency. *Information and Computation*, 159(1/2):151–186.

Cousot P., 2002. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 1-2:47–103.

Cousot P. and Cousot R., 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252.

Cousot P. and Cousot R., 1979. Systematic Design of Program Analysis Frameworks. In *Proc. of Sixth ACM Symp. on Principles of Programming Languages*, pages 269–282.

Cousot P. and Cousot R., 1994. Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages, ICCL'94*, pages 95–112. IEEE Computer Society Press, Los Alamitos, California, Toulouse, France.

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, volume 355 of *Lecture Notes in Computer Science*, pages 92–108. Springer-Verlag, Berlin.

Dauchet M., Heuillard T., Lescanne P., and Tison S., 1987. Decidability of the Confluence of Finite Ground Term Rewrite Systems and of Other Related Term Rewrite Systems. *Proc. of Second IEEE Symp. on Logic In Computer Science*, pages 353–359.

Dauchet M., Heuillard T., Lescanne P., and Tison S., 1990. Decidability of the Confluence of Ground Term Rewriting Systems. In *Information and Computation*, 88, pages 187–201. Academic Press, New York.

Dershowitz N. and Jouannaud J.P., 1990. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 244–320. Elsevier Science.

Dershowitz N. and Plaisted D., 2001. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 9, pages 535–610. Elsevier Science.

van Deursen A., Heering J., and Klint P., editors, 1996. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore.

Doets K., 1994. *From Logic to Logic Programming*. The MIT Press, Cambridge.

Dolstra E., 2001. Functional stratego. In E. Visser, editor, *Second Stratego Users Day*, pages 10–17.

Durand I., 1994. Bounded, Strongly Sequential and Forward Branching Term Rewriting Systems. *Journal of Symbolic Computation*, 18:319–352.

Durand I. and Middeldorp A., 1997. Decidable Call by Need Computations in Term Rewriting. In W. McCune, editor, *Proc. of CADE'97*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 4–18. Springer-Verlag, Berlin.

Durand I. and Salinier B., 1993. Constructor Equivalent Term Rewriting Systems. *Information Processing Letters*, 47:131–137.

Echahed R., 1990. On completeness of narrowing strategies. *Theoretical Computer Science*, 72(2-3):133–146.

van Eekelen M., Nöcker E., Plasmeijer M., and Smetsers J., 1991. Concurrent Clean. Language Manual - Version 0.8. Technical report, University of Nijmegen.

Ehrig H., 2001. On the Role of Formal Specification Techniques: From TAPSOFT 1985 to ETAPS 2000. *Current Trends in Theoretical Computer Science*, pages 131–133.

Ehrig H. and Mahr B., 1985. Fundamentals of Algebraic Specification. In *Information Processing Letters*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

Ehrig H. and Mahr B., 2001a. Algebraic Techniques in Software Development: A Review of Progress up to the Mid Nineties. *Current Trends in Theoretical Computer Science*, pages 134–152.

Ehrig H. and Mahr B., 2001b. Theory and Practice of Software Development: A Review of Driving Forces and Expectations of TAPSOFT from 1985 to 1997. *Current Trends in Theoretical Computer Science*, pages 118–130.

Eker S., 2000. Term rewriting with operator evaluation strategies. In C. Kirchner and H. Kirchner, editors, *Proc. of the 2nd International Workshop on Rewriting Logic and its Applications, WRLA 98*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

Escobar S., 2003a. Improving (Weakly) Outermost-Needed Narrowing: Natural Narrowing. In G. Vidal, editor, *Proc. of the 12th Int'l Workshop on Functional and (Constraint) Logic Programming WFLP'03*, pages 47–60. Universidad Politécnica de Valencia, Valencia.

Escobar S., 2003b. Refining Weakly Outermost-Needed Rewriting and Narrowing. In D. Miller, editor, *Proc. of 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'03*, pages 113–123. ACM Press, New York.

Escobar S., Alpuente M., and Lucas S., 1998. UPV-Curry User's Manual. Technical Report DSIC-II/38/98, DSIC, Universidad Politécnica de Valencia. Available at `http://www.dsic.upv.es/users/elp/papers.html`.

Fairley R., 1985. *Software Engineering Concepts*. McGraw Hill, New York, NY, USA.

Fay M., 1979. First Order Unification in an Equational Theory. In *Proc of 4th Int'l Conf. on Automated Deduction*, pages 161–167.

Field A. and Harrison P., 1988. *Functional Programming*. Addison-Wesley, Reading, MA.

Fissore O., Gnaedig I., and Kirchner H., 2001. Induction for termination with local strategies. In M. Bonacina and B. Gramlich, editors, *Proc. of the 4th International*

Workshop on Strategies in Automated Deduction, STRATEGIES 2001, volume 58.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

Floyd R.W., 1967. Assigning meanings to programs. *Proc. Amer. Soc. Symp. Appl. Math*, 19:19–31.

Fokkink W., Kamperman J., and Walters P., 2000. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86.

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.

Friedman D. and Wise D., 1976. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257–284. Edinburgh University.

Futatsugi K., Goguen J., Jouannaud J.P., and Meseguer J., 1985. Principles of OBJ2. In *Proc. of 12th Annual ACM Symp. on Principles of Programming Languages (POPL'85)*, pages 52–66. ACM Press, New York.

Futatsugi K. and Nakagawa A., 1997. An Overview of CAFE Specification Environment – An algebraic approach for creating, verifying, and maintaining formal specification over networks –. In *1st International Conference on Formal Engineering Methods*.

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.

Gallier J. and Book R., 1985. Reductions on tree replacement systems. *Theoretical Computer Science*, 37(2):123–150.

Giesl J. and Middeldorp A., 2003. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*. To appear.

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. and Sørensen M., 1994. Partial Deduction and Driving are Equivalent. In *Proc. of PLILP'94*, volume 844 of *Lecture Notes in Computer Science*, pages 165–181. Springer-Verlag, Berlin.

Goguen J., Thatcher J., and Wagner E., 1978. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In R. Yeh, editor, *Current Trends in Programming Methodology, vol. IV: data structuring*, pages 80–149. Prentice Hall, Englewood Cliffs, NJ.

Goguen J., Winkler T., Meseguer J., Futatsugi K., and Jouannaud J.P., 2000. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Gouranton V., 1998. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proc. of the 5th International Static Analysis Symposium, SAS'98*, volume 1503 of *Lecture Notes in Computer Science*, pages 115–133. Springer-Verlag, Berlin.

Gramlich B., 2001. On Interreduction of Semi-Complete Term Rewriting Systems. *Theoretical Computer Science*, 258(1-2):435–451.

Gramlich B. and Lucas S., 2002. Modular termination of context-sensitive rewriting. In C. Kirchner, editor, *Proc. of 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'02*. ACM Press, New York.

Guttag J. and Horning J., 1978. The Algebraic Specification of Abstract Data Types. Technical report, Acta Informatica.

Hall A., 1990. Seven myths of formal methods. *IEEE Software*, 7(5):11–19.

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. Efficient Translation of Lazy Functional Logic Programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, volume 1048 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, Berlin.

Hanus M., 1995b. Functional Logic Languages: Combine Search and Efficient Evaluation. In *Proc. of the 1995 International Symposium on Logic Programming, ILPS'95*. The MIT Press, Cambridge, MA.

Hanus M., 1997. Lazy Narrowing with Simplification. *Computer Languages*, 23(2–4):61–85.

Hanus M., Antoy S., Kuchen H., López-Fraguas F., Lux W., Moreno Navarro J., and Steiner F., 2003. Curry: An Integrated Functional Logic Language (Version 0.8). Available at: `http://www.informatik.uni-kiel.de/~curry/report.html`.

Hanus M., Antoy S., Kuchen H., López-Fraguas F., Moreno Navarro J., and Steiner F., 1998a. Curry: An Integrated Functional Logic Language (Draft of November 23, 1998). Available at: `http://www.informatik.uni-kiel.de/~curry/report.html`.

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., 1998b. 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 Schwab A., 1991. The Implementation of the Functional-Logic Language ALF. FB Informatik, Univ. Dortmund.

Henderson P. and Morris J., 1976. A Lazy Evaluator. In S. Michaelson and R. Milner, editors, *Proc. of 3rd Annual ACM Symp. on Principles of Programming Languages (POPL'76)*, pages 95–103. ACM Press, New York.

Hill P. and Lloyd J., 1994. *The Gödel Programming Language*. The MIT Press, Cambridge, MA.

Hoare T., 2003. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM*, 50(1):63–69.

Hoffmann C. and O'Donnell M., 1982. Programming with Equations. *ACM Transactions on Programming Languages and Systems*, 4(1):83–112.

Hölldobler S., 1989. *Foundations of Equational Logic Programming*. Springer LNAI 353.

Hudak P., 2000. *The Haskell School of Expression: learning functional programming through multimedia*. Cambridge University Press, Cambridge, UK.

Hudak P. and Fasel J., 1992. A Gentle Introduction to Haskell. *ACM Sigplan Notices*, 27(5):1–53.

Hudak P. and Jones M., 1993. Haskell vs. C, vs. Ada, vs. Awk, vs.... `http://haskell.org`.

Hudak P., Peterson J., and Fasel J., 1999. A Gentle Introduction to Haskell 98. Technical Report YALEU, Yale University, Department of Computer Science.

Hudak P., Peyton–Jones S., and Wadler P., 1992. Report on the Functional Programming Language Haskell: a non–strict, purely functional language. *Sigplan Notices*, 27(5).

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.J., 1979. Call by need computations in nonambiguous linear term rewriting systems. Technical report, IRIA Laboria, LeChesnay, France.

Huet G. and Lévy J.J., 1992. Computations in Orthogonal Term Rewriting Systems, Part I + II. In *Computational logic: Essays in honour of J. Alan Robinson*, pages 395–414 and 415–443. The MIT Press, Cambridge, MA.

Hughes J., 1988. Backwards Analysis of Functional Programs. In D. Bjørner, A. Ershov, and N. Jones, editors, *IFIP Workshop on Partial Evaluation and Mixed Computation*, pages 187–208.

Hullot J., 1980. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag, Berlin.

Ida T. and Nakahara K., 1997. Leftmost outside-in narrowing calculi. *Journal of Functional Programming*, 7(2):129–161.

Ireland A. and Bundy A., 1996. Using failure to guide inductive proofs. *Automated Reasoning*, 16:38–35.

Jacquemard F., 1996. Decidable approximations to Term Rewriting Systems. In H. Ganzinger, editor, *Proc. of 7th International Conference on Rewriting Techniques and Applications, RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, Berlin.

Jensen T.P., 1991. Strictness Analysis in Logical Form. In R.J.M. Hughes, editor, *Proc of International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 352–366. Springer-Verlag, Berlin.

Jones M., 1992. *Introduction to Gofer 2.20.* Oxford Programming Research Group.

Jones S.B. and Métayer D.L., 1989. Compile-time garbage collection by sharing analysis. In *Proc of International Conference on Functional Programming Languages and Computer Architecture*, pages 54–74. ACM Press, New York.

Jouannaud J., 1994. Rewrite Proofs and Computations. In H. Schwichtenberg, editor, *NATO Series F: Computer and Sciences*, volume 139 of *Lecture Notes in Computer Science*, pages 173–218. Springer-Verlag, Berlin.

Kapur D., Narendran P., and Zhang Z., 1987. On sufficient-completeness and related properties of term rewriting systems. *Acta Informatica*, 24:395–416.

Kapur D. and Subramaniam M., 1996. Lemma discovery in automating induction. In *Proc. of CADE'96*, volume 914 of *Lecture Notes in Computer Science*, pages 403–407. Springer-Verlag, Berlin.

Kennaway R., 1989. Sequential Evaluation Strategies for ParallelOr and Related Reduction Systems. *Theoretical Computer Science*, 43:31–56.

Kennaway R., van Oostrom V., and de Vries F., 1996. Meaningless Terms in Rewriting. In *Proceedings of the ALP'96*, volume 1139 of *Lecture Notes in Artificial Intelligence*, pages 254–268. Springer-Verlag.

Klop J., 1992. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–116. Oxford University Press, Oxford.

Klop J. and Middeldorp A., 1991. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, pages 161–195.

Knoop J., Rüthing O., and Steffen B., 1994. Partial Dead Code Elimination. *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'94)*, 29(6):147–158.

Kobayashi N., 2000. Type-based useless variable elimination. In *Proceedings of PEPM-00*, pages 84–93. ACM Press, New York.

Kounalis E., 1985. Completeness in data type specifications. In B. Caviness, editor, *Proc. of European Conference on Computer Algebra, EUROCAL'85*, volume 204 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, Berlin.

Kuchen H. and Anastasiadis J., 1996. Higher Order Babel: Language and Implementation. In *editors Proc. 5th International Workshop on Extensions of Logic Programming ELP'96*, pages 193–207. Springer LNAI 1050.

Kuchen H., Loogen R., Moreno-Navarro J., and Rodríguez-Artalejo M., 1990. Lazy Narrowing in a Graph Machine. In *Proc. of the Int'l Conf. on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 298–317.

Kuper J., 1994. *Partiality in Logic and Computation. Aspects of Undefinedness.* Ph.D. thesis, Universiteit Twente.

Leuschel M., 1998. On the Power of Homeomorphic Embedding for Online Termination. In G. Levi, editor, *Proc. of the 5th International Static Analysis Symposium, SAS'98*, volume 1503 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, Berlin.

Leuschel M. and Martens B., 1995. Partial Deduction of the Ground Representation and Its Application to Integrity Checking. Technical Report CW 210DSIC-II/8/02, K.U. Leuven.

Leuschel M. and Sørensen M.H., 1996. Redundant Argument Filtering of Logic Programs. In J. Gallager, editor, *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, volume 1207 of *Lecture Notes in Computer Science*, pages 83–103. Springer-Verlag, Berlin, Stockholm, Sweden.

Liu Y.A. and Stoller S.D., 2002. Eliminating dead code on recursive data. *Science of Computer Programming*.

Lloyd J., 1999. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, 1999(3):1–49.

Loogen R., López-Fraguas F., and Rodríguez-Artalejo M., 1993. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93*, volume 714 of *Lecture Notes in Computer Science*, pages 184–200. Springer-Verlag, Berlin.

López-Fraguas F. and Sánchez-Hernández J., 1999. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, volume 1631 of *Lecture Notes in Computer Science*, pages 244–247. Springer-Verlag, Berlin.

Lucas S., 1997. Transformations for Efficient Evaluations in Functional Programming. In H. Glaser and P. Hartel, editors, *Proc. of 9th International Symposium on Programming Language Implementation and Logic Programming, PLILP'97*, volume 1292 of *Lecture Notes in Computer Science*, pages 127–141. Springer-Verlag, Berlin.

Lucas S., 1998a. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1–61.

Lucas S., 1998b. *Rewriting with replacement restrictions.* Ph.D. thesis, DSIC, Universidad Politécnica de Valencia. In spanish.

Lucas S., 1998c. Root-neededness and approximations of neededness. *Information Processing Letters*, 67(5):245–254.

Lucas S., 2001a. Termination of on-demand rewriting and termination of OBJ programs. In *Proc. of 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82–93. ACM Press, New York.

Lucas S., 2001b. Termination of Rewriting With Strategy Annotations. In R. Nieuwenhuis and A. Voronkov, editors, *Proc. of 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'01*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 669–684. Springer-Verlag, Berlin.

Lucas S., 2001c. Transfinite Rewriting Semantics for Term Rewriting Systems. In A. Middeldorp, editor, *Proc. of 12th International Conference on Rewriting Techniques and Applications, RTA'01*, volume 2051 of *Lecture Notes in Computer Science*, pages 216–230. Springer-Verlag, Berlin.

Lucas S., 2002a. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343.

Lucas S., 2002b. Lazy rewriting and context-sensitive rewriting. In M. Hanus, editor, *Proc. of the 10th Int'l Workshop on Functional and (Constraint) Logic Programming WFLP'01*, volume 64 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

Lucas S., 2002c. Termination of (Canonical) Context-Sensitive Rewriting. In S. Tison, editor, *Proc. of 13th International Conference on Rewriting Techniques and Applications, RTA'02*, volume 2378 of *Lecture Notes in Computer Science*, pages 296–310. Springer-Verlag, Berlin.

Maiden N.A. and Sutcliffe A.G., 1992. Exploiting reusable specifications through analogy. *Communications of the ACM*, 35(4):55–64.

McCarthy J., 1960. Recursive Functions of Symbolic Expressions and their Computations by Machine, Part I. *Communications of the ACM*, 3(4):184–195.

McCarthy J., 1963. Towards a mathematical theory of computation. In *IFIP World Congress Proceedings*. North-Holland, Amsterdam.

McCarthy J., 1978. History of Lisp. In R. Wexelblat, editor, *History of Programming Languages*. Academic Press, New York.

McCarthy J., 2003. Problems and Projections in CS for the Next 49 Years. *Journal of the ACM*, 50(1):73–79.

Métayer D.L., Nicolas V.A., and Ridoux O., 1998. Exploring the Software Development Trilogy. *IEEE Software*, 6:75–81.

Middeldorp A., 1997. Call by Need Computations to Root-Stable Form. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 94–105. ACM Press, New York.

Middeldorp A. and Okui S., 1998. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757.

Milner R., Tofte M., and Harper R., 1990. *The Definition of ML*. The MIT Press.

Milner R., Tofte M., Harper R., and MacQueen D., 1997. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA.

Moreno-Navarro 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*, volume 463 of *Lecture Notes in Computer Science*, pages 298–317. Springer-Verlag, Berlin.

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.

Mycroft A., 1980. The theory and practice of transforming callbyneed into callbyvalue. In B. Robinet, editor, *4th International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, page 269281. Springer-Verlag, Berlin.

Mycroft A. and Norman A., 1992. Optimising Compilation. Part II: lazy functional languages. In *XIX Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM'92*, Lecture Notes in Computer Science. Springer-Verlag, Berlin.

Nadathur G. and Miller D., 1988. An Overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of Fifth Int'l Conf. on Logic Programming*, pages 810–827. The MIT Press, Cambridge, MA.

Nagaya T., 1999. *Reduction Strategies for Term Rewriting Systems*. Ph.D. thesis, School of Information Science, Japan Advanced Institute of Science and Technology.

Nakamura M. and Futatsugi K., 2001. Completeness and strictness analysis for the evaluation strategy. In Y. Toyama, editor, *Proc. of 1th International Workshop on Rewriting y Proof and Computation, RPC'01*, pages 80–89. RIEC, Tohoku University.

Nakamura M. and Ogata K., 2001. The evaluation strategy for head normal form with and without on-demand flags. In K. Futatsugi, editor, *Proc. of the 3rd International Workshop on Rewriting Logic and its Applications, WRLA 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

Nieuwenhuis R., 1996. Basic Paramodulation and Decidable Theories. In *Proceedings of the Eleventh Annual IEEE Symposium On Logic In Computer Science (LICS'96)*, pages 473–483. IEEE Computer Society Press, New York, USA.

Nöcker E., Smetsers J., van Eekelen M., and Plasmeijer M., 1991. Concurrent Clean. In *Proceedings of PARLE'91*, volume 506 of *Lecture Notes in Computer Science*, pages 202–219. Springer-Verlag, Berlin.

O'Donnell M.J., 1977. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

O'Donnell M.J., 1985. *Equational Logic as a Programming Language*. The MIT Press, Cambridge, MA.

O'Donnell M.J., 1987. Term Rewriting Implementation of Equational Logic Programming. In P. Lescanne, editor, *Proc. of 2nd International Conference on Rewriting Techniques and Applications, RTA'87*, volume 256 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, Berlin.

O'Donnell M.J., 1995a. Equational Logic Programming. In C. Hogger, D. Gabbay, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4. Oxford University Press, Oxford.

O'Donnell M.J., 1995b. Introduction: Logic and Logic Programming Languages. In C. Hogger, D. Gabbay, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, chapter 1. Oxford Science Publications.

O'Donnell M.J., 1998. Equational Logic Programming. In D. Gabbay, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 2, pages 69–161. Oxford Science Publications.

Ogata K. and Futatsugi K., 1997. Implementation of Term Rewritings with the Evaluation Strategy. In H. Glaser and P. Hartel, editors, *Proc. of 9th International Symposium on Programming Language Implementation and Logic Programming, PLILP'97*, volume 1292 of *Lecture Notes in Computer Science*, pages 225–239.

Ogata K. and Futatsugi K., 2000. Operational semantics of rewriting with the on-demand evaluation strategy. In *Proc. of 2000 International Symposium on Applied Computing, SAC'00*, pages 756–763. ACM Press, New York.

Ohlebusch E., 2002. *Advanced Topics in Term Rewriting*. Springer-Verlag, Berlin.

Oyamaguchi M., 1986. The Reachability Problems for Quasi-Ground for Term Rewriting Systems. *Journal of Information Processing*, 9(4):232–236.

Oyamaguchi M., 1990. The reachability and joinability problems for right-ground term rewriting systems. *Journal of Information Processing*, 13(3):347–354.

Oyamaguchi M., 1993. NV-Sequentiality: a Decidable Condition for Call-by-Need Computations in Term-Rewriting Systems. *SIAM Journal of Computation*, 22(1):114–135.

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, *Automata, Languages and Programming, 17th International Colloquium (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 386–399. Springer-Verlag, Berlin.

Park Y.G. and Goldberg B., 1992. Escape Analysis on Lists. *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'92)*, 27(7):116–127.

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*, volume 1110 of *Lecture Notes in Computer Science*, pages 355–385. Springer-Verlag, Berlin.

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., 1987. *The Implementation of Functional Programming Languages.* Prentice Hall International, London.

Plaisted D., 1994. Equational Reasoning and Term Rewriting Systems. In D. Gabbay and J. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1. Oxford University Press, Oxford.

Plasmeijer R. and Eekelen M.v., 1993. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley.

Plotkin G., 1972. Building-in equational theories. *Machine Intelligence*, 7:73–90.

van de Pol J., 2001. Just-in-time: on Strategy Annotations. In B. Gramlich and S. Lucas, editors, *Proc. of the 1st International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2001*, volume 57 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

Rabhi F. and Lapalme G., 1999. *Algorithms: A Functional Programming Approach.* Addison-Wesley, Harlow, England.

Rabin M.O., 1969. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, 141:1–35.

Reade C., 1993. *Elements of Functional Programming.* Addison-Wesley, Reading, MA.

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.

Reddy U., 1990. Term rewriting induction. In M.E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 162–177. Springer-Verlag.

Reddy U., 1995. Bridging the gap between logic and functional programming. In J. Lloyd, editor, *Proc. of the 1995 International Symposium on Logic Programming, ILPS'95*, pages 627–628. The MIT Press, Cambridge, MA.

Reps T. and Turnidge T., 1996. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429. Springer-Verlag, Berlin.

Rodriguez-Artalejo M., 1995. The Challenge of Declarative Programming. In J. Lloyd, editor, *Proc. of the 1995 International Symposium on Logic Programming, ILPS'95*, pages 629–630. The MIT Press, Cambridge, MA.

Rusinowitch M., 1987. *Demonstration automatique par des techniques de r'e'ecriture.* Ph.D. thesis, Nancy.

Schoenig S. and Ducasse M., 1996. A Backward Slicing Algorithm for Prolog. In *Proc. of the 3rd International Static Analysis Symposium, SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, Berlin.

Sekar R., Pawagi S., and Ramakrishnan I., 1990. Small Domains Spell Fast Strictness Analysis. In *16th Annual ACM Symposium on Principles of Programming Languages, POPL'89*, page 169183. ACM Press, New York.

Sekar R. and Ramakrishnan I., 1993. Programming in Equational Logic: Beyond Strong Sequentiality. *Information and Computation*, 104(1):78–109.

Slagle J., 1974. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642.

Strandh R., 1989. Classes of Equational Programs that Compile into Efficient Machine Code. In N. Dershowitz, editor, *Proc. of RTA'89*, volume 355 of *Lecture Notes in Computer Science*, pages 449–461. Springer-Verlag, Berlin.

Szilagyi G., Gyimothy T., and Maluszynski J., 2002. Static and Dynamic Slicing of Constraint Logic Programs. *Journal of Automated Software Engineering*, 9(1):41–65.

TeReSe, editor, 2003. *Term Rewriting Systems*. Cambridge University Press, Cambridge.

Thatcher J.W. and Wright J.B., 1968. Generalized finite automata with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2:57–82.

Thatte S., 1987. A Refinement of Strong Sequentiality for Term Rewriting with Constructors. *Information and Computation*, 72:46–65.

Thiel J., 1984. Stop Losing Sleep over Incomplete Data Type Specifications. In *In 11th ACM Symposium on Principles of Programming Languages*, pages 76–82.

Thomas W., 1990. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 133–191. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass.

Thompson S., 1999. *Haskell: The Craft of Functional Programming*. Addison-Wesley, Harlow, England.

Tip F., 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189.

Toyama Y., Smetsers S., van Eekelen M., and Plasmeijer R., 1993. The Functional Strategy and Transitive Term Rewriting Systems. In M. Sleep, M. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting Theory and Practice*, pages 117–129. John Wiley & sons.

Turing A.M., 1949. Checking a large routine. Report on a Conference on High Speed Automatic Calculating Machines. Cambridge Univ. Math. Lab. 67-69.

Visser E., 1999. Strategic Pattern Matching. In P. Narendran and M. Rusinowitch, editors, *Proc. of 10th International Conference on Rewriting Techniques and Applications, RTA'99*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, Berlin.

Visser E., 2001. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Proc. of the 1st International Workshop* *on Reduction Strategies in Rewriting and Programming, WRS 2001*, volume 57 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences Publisher.

Wadler P. and Hughes R., 1987. Projections for Strictness Analysis. In *Proc of International Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer-Verlag, Berlin.

Walsh T., 1996. A Divergence Critic for Inductive Proofs. *Artificial Intelligence Research*, 4:209–235.

Wand M. and Siveroni I., 1999. Constraint systems for useless variable elimination. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'99)*, pages 291–302. ACM Press, New York.

Weiser M., 1984. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.

Wordsworth J., 1999. Getting the best from formal methods. *Information and Software Technology*, 41:1027–1032.

You J.H., 1991. Unification Modulo an Equality Theory for Equational Logic Programming. *Journal of Computer and System Sciences*, 42:54–75.

Zantema H., 1997. Termination of context-sensitive rewriting. In *Proc. of 8th International Conference on Rewriting Techniques and Applications, RTA'97*, volume 1232 of *Lecture Notes in Computer Science*, pages 172–186. Springer-Verlag, Berlin.

Zhang H., 1988. *Reduction, Superposition, and Induction: Automated Reasoning in an Equation Logic*. Ph.D. thesis, Renselaer Polytechnic Institute.