Dedicated to my parents

La stessa forza in una idea
Ci sospinge e poi, come un'energia
Ci accompagnerà sempre
Il destino è un grande mare
Dove navigare è sempre scegliere

– A. Minghi

't Was op een warme lenteavond, terwijl de novemberzon haar laatste lichtstralen over het kristalheldere water van de Indische Oceaan strooide, dat m'n promotor me vroeg of de tijd niet gekomen was om de knoop door te hakken en te beginnen schrijven aan een thesis. Starend naar de zwarte palmbomen in het licht van een gouden zonsondergang moet ik dit zonder twijfel beaamd hebben. Eens terug in het noordelijk halfrond ben ik dan beginnen schrijven in de donkere maar mooie dagen voor Kerstmis en nu, een paar maanden later, is het resultaat er. Deze thesis is met veel plezier geschreven, soms ook met een beetje pijn, maar altijd met de vreugde van het schrijven en de wens om iets moois te creëren.

Ik zou mijn beide promotoren, Maurice Bruynooghe en Danny De Schreye, willen bedanken; niet alleen voor de ondersteuning die ze geboden hebben tijdens mijn onderzoek, maar ook om mij de mogelijkheid te geven te reizen, interessante mensen te ontmoeten, boeiende plaatsen te ontdekken en bovenal in een stimulerende en leergierige omgeving te kunnen werken. Mijn oprechte dank gaat naar de leden van mijn jury, Professor Karel De Vlaminck, Professor Marc Gobin, Professor María Alpuente en Professor Michael Leuschel voor hun interesse in dit werk, en voor hun gedetailleerde en waardevolle opmerkingen en commentaren.

Een speciaal woord van dank gaat naar Bern Martens, die me in het onderzoeksdomein van partiële deductie introduceerde en zodoende een leraar, medewerker maar bovenal een zeer geapprecieerde vriend werd. Dit werk zou zeker en vast niet geworden zijn wat het is zonder zijn invloed, hulp en steun.

Ook zou ik uitdrukkelijk Zoltan Somogyi willen bedanken om mij de mogelijkheid

Preface

Dancing bears, painted wings
Things I almost remember
And a song, someone sings
once upon a December...

– from "Anastasia"

It was on a warm spring evening while the November sun was shedding its light over the crystal clear water of the Indian Ocean, that my supervisor inquired whether the time hadn't come to bite the bullet, and start writing a thesis. Staring at the black palm trees in front of a golden sunset, I must undoubtedly have confirmed. Once back in the northern hemisphere, I started writing the text in the dark but cheerful days around Christmas and now, a few months later, the result is right here. It has been written with a lot of pleasure, sometimes with a bit of pain, but never without the joy of writing and the desire to create.

I would like to thank both my supervisors, Maurice Bruynooghe and Danny De Schreye, not only for the support they gave me during my research, but also for providing me with the opportunity to travel, to meet interesting people, to visit interesting places but most of all for providing the ability to work in a stimulating environment eager of discovery and continuous learning. I sincerely thank the members of my jury, Professor Karel De Vlaminck, Professor Marc Gobin, Professor María Alpuente and Professor Michael Leuschel for their interest in this work and for providing many detailed and valuable comments.

A special word of gratitude goes to Bern Martens, who introduced me to the field of partial deduction and became a teacher, co-worker and most of all a much appreciated friend. Definitely, this work wouldn't have become what it is without his influence, help and support.

Special thanks go to Zoltan Somogyi for providing me with the opportunity of working together with the Mercury team, and to Tyson Dowd, Fergus Henderson,

te geven samen te werken met het Mercury team, alsook Tyson Dowd, Fergus Henderson, David Jeffrey, David Overton, Peter Ross, Mark Brown en Anthony Senyard; niet alleen om keer op keer de implementatiedetails van de Melbourne Mercury compiler uit de doeken te doen, maar bovenal om een omgeving te creëren die mijn bezoek aan Australië tot een ongeëvenaarde en onvergetelijke gebeurtenis gemaakt heeft.

Op een meer persoonlijk vlak zou ik mijn ouders willen bedanken voor hun alom aanwezige steun, geloof en geduld. Ook zou ik mijn familie willen bedanken voor hun interesse en steun, in het bijzonder Kenny en Gitta die dapper en strijdvaardig genoeg waren om zich door de tekst te worstelen en zo een aantal ontbrekende verwijzingen en vele mogelijkheden tot verbetering aan te geven.

Misschien klinkt het als een cliché, maar ik zou dit alles nooit bereikt hebben zonder de steun van verschillende vrienden die ik hier zou willen bedanken: Nico en Siska, omwille van hun vriendschap en hun ongeëvenaarde gastvrijheid en steun, Gunther om vaak te luisteren naar mijn frequent gezeur (en – er waarschijnlijk genoeg van hebbend – "constructief" mee te werken aan een van de bewijzen in deze thesis), en Raf, Bart, Veerle (en Bandy), Dieter en Stefan voor hun vriendschap en voortdurende bron van inspiratie. Met de altijd aanwezige angst om sommigen te vergeten wil ik de volgende mensen aan het lijstje toevoegen: Mike, Conny, Patrick, Gerd, Maarten, Natalie, Marc, Nadia, Stefaan, Kristel, Thierry, Ilse, Danny en Annette. Ik bedank hen allemaal van harte om, elk op z'n eigen manier, de best mogelijke vrienden te zijn die iemand zich kan indenken.

Ik zou ook de mensen willen bedanken met wie ik het genoegen heb gehad een bureau te mogen delen, in het bijzonder Yvan en Kris om me gedurende mijn eerste dagen op het departement op te vangen en me er snel thuis te laten voelen, alsook voor de vele plezierige en vaak leerzame gesprekken die we gehad hebben – en nog steeds hebben. Het is haast onmogelijk om individueel alle mensen te bedanken die het departement computerwetenschappen van de K.U.Leuven tot een zeer aangename werkplek weten te maken. Daarom: een grote "dankjewel" aan hen allemaal, in het bijzonder aan Denise, Karin, Margot, Esther en Lieve om me te helpen met de onoverkomelijke administratieve en praktische probleempjes, en de mensen van de systeemgroep om "het allemaal te doen werken".

En last but not least zou ik Nancy en Sofie willen bedanken; Nancy om me te vergezellen op een fantastische trip naar Australië en zo een werkbezoek om te vormen tot een prachtige mix van hard werken en vakantie, maar hen beide om een fantastische sfeer op bureau te creëren – inclusief de vele aangename koffiepauzes – maar bovenal om gewoonweg de beste bureaugenootjes ooit te zijn. Bedankt.

Wim Vanhoof, juni 2001.

David Jeffrey, David Overton, Peter Ross, Mark Brown and Anthony Senyard for explaining over and over again several implementation issues of the Melbourne Mercury compiler, but most of all for providing an environment which made my visit to Australia an unmatched and unforgettable experience.

On a more personal basis, I would like to thank my parents for their never fading support, belief and patience. In addition, I would like to thank my family for their interest and support and in particular Kenny and Gitta for being bold and brave enough to struggle through the text, indicating on their way several missing links and opportunities for improvement.

It may sound like a cliché, but I would never have made it through without the support of several of my friends whom I would like to thank: Nico and Siska for their friendship and their unequaled hospitality and support, Gunther for often listening to my frequent nagging (and, tired of doing so, "coauthoring" one of the proofs in this thesis), and Raf, Bart, Veerle (and Bandy), Dieter and Stefan for their friendship and continuous source of inspiration. With the ever present fear of forgetting people, I must add Mike, Conny, Patrick, Gerd, Maarten, Natalie, Marc, Nadia, Stefaan, Kristel, Thierry, Ilse, Danny and Annette. I thank all of them sincerely for being, each in their own way, the best friends one could imagine.

I would also like to thank those with whom I had the pleasure of sharing an office, in particular Yvan and Kris for rapidly making me feel comfortable at the office, and for the many enjoyable and often instructive discussions we've had – and still have. It is simply impossible to thank individually all those people that make K.U.Leuven's department of computer science a very nice place to work. Therefore: a big "thank you" to all of them, in particular to Denise, Karin, Margot, Esther and Lieve for helping me out with many practical and administrative issues, and to the technical staff for "making it all work".

And last but certainly not least, I would like to thank Nancy and Sofie; Nancy for accompanying me on a wonderful trip to Australia, as such transforming a work visit into a wonderful blend of hard work and holidays, but both of them for creating a fantastic atmosphere at the office – including the many pleasant coffee-breaks – and simply for being the very best office mates ever. Thanks.

Wim Vanhoof, June 2001.

Structure of the Thesis

This text describes research that was performed between October 1996 and December 2000. The text is structured as follows. The first two chapters are introductory chapters of which the first one aims at providing a gentle, informal and non technical introduction to the concept of program specialisation. It gives an overview of the current state-of-the-art in techniques for automatic program specialisation, and describes some applications. The second chapter provides a more technical introduction to logic programming, and program specialisation in the latter paradigm, better known as partial deduction.

The main body of the thesis is divided in three parts. In a first part, we investigate an off-line approach towards specialisation of logic programs. Off-line specialisation of logic programs has only been considered occasionally before, and in this work, we concentrate on the most important part in an off-line setting, constituted by the so-called binding-time analysis. In chapters 3 and 4, we develop a binding-time analysis for Mercury. Although being a logic programming language, Mercury is strongly moded and evaluation of Mercury programs resembles much the evaluation of functional programs. As such, Mercury is an excellent candidate to develop a binding-time analysis for. Indeed, its functional nature enables to build on the research basis that is established on binding-time analysis in the field of functional programming. However, its more involved data- and control flow features – inherent to a logic programming language – add a new dimension to the developed analyses. In Chapter 3, we develop a binding-time analysis for a first-order subset of Mercury. We define a denotational semantics for the language, which we approximate by an abstract semantics over a sophisticated domain of binding-times that – based on Mercury's type structure – allows to represent partially static structures. The chapter contains a correctness result showing that specialisation of the program according to the annotations computed by the binding-time analysis preserves the semantics of the program. Motivated by the necessity to scale the binding-time analysis towards multi-module programs, we redevelop the analysis in Chapter 4 by constraint normalisation. We extend the analysis towards handling the higher-order features of Mercury and discuss the relation between a purely modular approach and the incorporation of higher-order control flow.

This thesis' first part is concluded by Chapter 5 in which we consider binding-time analysis for a plain, unmoded logic programming language, essentially a pure subset of Prolog. The absence of modes to fix the data flow makes an approach as the one we employed in previous chapters for Mercury unfeasibly. Hence, we develop a binding-time analysis starting from first principles, and concentrate on an important aspect of binding-time analysis: guaranteeing that reduction according to the annotations computed by the binding-time analysis terminates. Inspired by the well-researched field of termination analysis for logic programming, we develop a technique that basically iterates a termination analysis with annotating a

program until enough calls in the program text are annotated to residualise, such that reducing the program according to the annotations terminates.

Before jumping to an alternative approach of specialisation for logic programs, the second part of this text – constituting a single chapter – combines the ideas underlying the domain of binding-times presented for Mercury in Part I with a general dependency analysis by means of positive boolean functions. We show how such a *Pos*-based analysis can be refined in the presence of type information. The resulting analysis no longer computes dependencies at the level of a program variable, but rather at the level of a particular set of the variable's subvalues. Hence, it allows *Pos*-based analyses of more involved properties of the program variables. We demonstrate the approach by reformulating a *Pos*-based groundness analysis using our more fine-grained domain – which we baptised $Pos(\mathcal{T})$ – resulting in a more general instantiatedness analysis, in which the notion of a variable being ground is generalised into a more detailed characterisation of the variable's instantiatedness.

The final part of the thesis investigates an on-line approach in partial deduction of logic programs. Chapter 7 investigates the effectiveness of a state-of-the-art control strategy on the specialisation of the well-known vanilla meta interpreter. While removing the overhead introduced by a meta interpreter has always been one of the main motivations to develop partial deduction (or partial evaluation in general), this chapter shows that achieving this – even in the case of the simple vanilla meta interpreter – with an automatic and completely general control strategy is far from trivial. We investigate why this is the case, and discover the reason to be the fluctuations in the structured data that is handled by the meta interpreter. One way of dealing with this phenomenon in a satisfactory – and automatable – way, further explored in Chapter 7, is the construction of a more aggressive unfolding strategy that incorporates knowledge about the consequences of halting the unfolding at a particular point on the general continuation of the partial deduction process. The resulting control strategy, although inspired by the problem of partially deducing the vanilla meta interpreter, is a general and automatic strategy that further merges the two traditionally known control levels in the partial deduction process. It succeeds in removing the parsing overhead introduced by the vanilla interpreter and we investigate its performance on an extension of the meta interpreter, comparing the achieved results with an ad hoc specialisation technique, specifically designed to deal with the particular interpreter. We show that our general control strategy achieves results that equal and sometimes even surpass the results obtainable by the ad hoc specialiser.

Although the control strategy developed in Chapter 7 suffices to deal with the fluctuating structures that occur during partial deduction of the vanilla meta interpreter and simple variants of the latter, it is too weak to deal with more involved meta interpreters. In Chapter 8, we further scrutinise the relation between the data flow in a logic program and the process of partial deduction. We

develop a novel partial deduction technique, which we call bottom-up partial deduction, that partially deduces a program by propagating information the other way round: information residing in a program's unit clauses is propagated upwards, and a suitable abstraction operator is defined in order to compute a finite bottom-up partial deduction. We prove the transformation correct with respect to the S-semantics. Since the bottom-up partial deduction process concentrates on the bottom-up data flow in a program, it can obtain information propagation that is hard to achieve with a top-down system. We recognise that the vanilla meta interpreter, exposing a pronounced combination of bottom-up and top-down data flow, is a good candidate for a combined process of bottom-up and the classical top-down partial deduction and demonstrate that such a combined approach, each component concentrating on a particular direction in the data flow, achieves satisfying results when partially deducing the vanilla meta interpreter. The resulting blend of bottom-up and top-down control is conceptually cleaner and requires less sophisticated control to achieve the same results as the top-down approach of Chapter 7.

The thesis is concluded in Chapter 9, in which we give an overview of some of the achievements and indicate some directions for further research.

Publications and Co-authors

Most of the work presented in this thesis was published at different occasions. The basic ideas of the binding-time analysis for Mercury by abstract interpretation as described in Chapter 3 (joint work with Maurice Bruynooghe) were first presented at LOPSTR'99 and appeared in the form of an extended abstract in the conference's preproceedings (Vanhoof and Bruynooghe 1999a). A more detailed version of the paper appeared in the ICLP'99 conference proceedings (Vanhoof and Bruynooghe 1999b), while the first steps towards reformulating the analysis in a modular fashion were published in the WOID'99 workshop proceedings (Vanhoof and Bruynooghe 1999c). A refinement of the modular approach, also incorporating the higher-order aspects of the language (Chapter 4) was first presented in the form of an extended abstract at LOPSTR'2000 (Vanhoof and Bruynooghe 2000) – again co-authored by Maurice Bruynooghe – followed by a more detailed paper published at LPAR2000 (Vanhoof 2000).

The work on specialisation of the vanilla meta interpreter using a general, automatic top-down partial deduction system, joint work with Bern Martens and described in Chapter 7, was published in the LOPSTR'97 conference proceedings. It was the main motivation for the research concerning the novel bottom-up partial deduction technique developed in Chapter 8. Some motivational work on the topic was published in the ILPS'97 post conference workshop on specialisation of declarative languages (Vanhoof 1997). A concrete control strategy for computing bottom-up partial deductions was published in the JICSLP'98 conference

proceedings, being joint work with Bern Martens, Danny De Schreye and Karel De Vlaminck (Vanhoof, Martens, Schreye, and Vlaminck 1998). This work was generalised into a framework of bottom-up partial deduction, published in the LOPSTR'98 proceedings in the form of a small abstract (Vanhoof, Schreye, and Martens 1998a), and in more detail in the PLILP'98 conference proceedings (Vanhoof, Schreye, and Martens 1998b). Both papers were co-authored by Danny De Schreye and Bern Martens. An elaborated version of the PLILP'98 paper was published in the Journal of Functional and Logic Programming (Vanhoof, Schreye, and Martens 1999), while an abstract appeared in the NAIC'98 conference proceedings (Vanhoof, Schreye, and Martens 1998c).

An extended abstract of the material presented in Chapter 5 (joint work with Maurice Bruynooghe) on binding-time analysis by termination analysis was presented at the *Fifth International Workshop on Termination* (Vanhoof and Bruynooghe 2001); a report of some experiments with the analysis in the context of the cogen approach using Michael Leuschel's LOGEN system is to be found in a paper jointly authored by Michael Leuschel, Jesper Jørgensen and Maurice Bruynooghe (Leuschel, Jørgensen, Vanhoof, and Bruynooghe 2001) submitted for publication. The work described in Chapter 6 on the $Pos(\mathcal{T})$ domain for abstract interpretation (joint work with Maurice Bruynooghe and Michael Codish) will be presented at the *Andrei Ershov Fourth International Conference "Perspectives of System Informatics"* (Bruynooghe, Vanhoof, and Codish 2001).

Contents

Chapter 1

Introduction

A journey of a thousand miles begins with a single step.

— *Chinese proverb.*

This chapter aims at providing a gentle introduction to the concept of program specialisation (or partial evaluation). It describes some applications of the technique, and gives an overview of the current state-of-the-art on the topic in different programming paradigms.

1.1 Program Specialisation

Program specialisation is a technique that transforms a program into another program, by precomputing some of its operations. Suppose we have a program P of which the input can be divided in two parts, say s and d. If one of the input parts, say s, is known at some point in the computation, we can *specialise* P with respect to the available input s. This specialisation process comprises performing the computations of P that depend only on s, and recording their *results* in a new program, together with the *code* for those computations that could not be performed (because they rely on the input part d – unknown at this point in the computation). The result of the specialisation is a new program, P_s that computes, when provided with the remaining input part d, the *same* result as P does when provided with the complete input $s + d$. Figure 1.1 represents this process schematically. In this figure, the rounded arrow between P and P_s represents the specialisation process. The part s of the program's input that is used for specialisation is often called the *static* input part, whereas the remaining

Figure 1.1: Program specialisation.

input is often referred to as the *dynamic* input part. The result of the specialisation process is a new program that – being a residue from the original program's code – is often called the *residual* program. Program specialisation is usually considered a source-to-source transformation: the residual program is written in the same language as the original program. The process of program specialisation comprises a mixture of program evaluation and code generation. Hence, it is often referred to by the names *partial evaluation* or *mixed computation*. Consider, for example, the following program written in some informal functional syntax, to compute the n-th power of a given value x, where both $x, n \in \mathbb{N}$.

Example 1.1

$$\begin{array}{lcl} power(x, n) & = & \textit{if } (n = 1) \\ & & \textit{then } x \\ & & \textit{else } (x * power(x, n - 1)) \end{array}$$

Now, suppose we specialise the above program for the situation where we want to compute the fifth power, that is $n = 5$. Looking at the definition of the *power* function, we notice that the following statements depend only on the value of n:

- the test of conditional statement,
- the expression $n - 1$ in the recursive call
- the recursive call, since the recursion is completely determined by the value of n.

Performing these statements, and residualising the others, the result of specialising the call *power(x,5)* is the residual program:

$$power(x, 5) \quad = \quad x * x * x * x * x$$

If the specialiser is correct, the residual program computes the same function as the original program, but naturally only for inputs of which the static part equals the

values with respect to which the program was specialised. In the above example, the residual program $power(x, 5)$ still implements the *power* function, but only the *fifth*-power function. It can be used to compute the fifth power of any value, but can no longer compute the n-th power of a value.

The effect of using a program specialisation transformation is that the computation of a program is performed in *stages*. In a first stage, the program is (partially) evaluated with respect to the *static* part of the input only. Afterwards, in a second stage, the residual program is evaluated with respect to the *dynamic* part of the input. The first such stage is often referred to by the notion of *specialisation-time*, as opposed to the notion of *runtime*, referring to the second stage. If we adhere to the notation of (Jones, Gomard, and Sestoft 1993), and represent a program specialiser by `mix` and denote with $[\![\ ]\!]$ the *program meaning function*, we can denote the staged computation process for a program P with respect to static input s and dynamic input d as follows:

$$\begin{array}{lcl} res & = & [\![\texttt{mix}]\!][P, s] \\ out & = & [\![res]\!]d \end{array}$$

where res denotes the residual program, i.e. the result of running `mix` with the program P and the static data s as input, and out denotes the result of running the residual program res with the dynamic input d. Note that we use a list notation to combine multiple inputs to the program meaning function, like $[P, s]$, which we drop in case of single input d. The equivalent computation in one stage (running the program P with both the static and dynamic input) is denoted by the following:

$$out = [\![P]\!][s, d]$$

Since the output of both processes is the same, we can combine the above equations, resulting in the *partial evaluation equation* (Jones, Gomard, and Sestoft 1993):

$$[\![P]\!][s, d] = [\![\,[\![\texttt{mix}]\!][P, s]]\!]d \tag{1.1}$$

Staging the computations of a program can be useful when different parts of a program's input become known at different times. Suppose that part of a program's input is already known at compile-time. In that case, the process of program specialisation can be seen as *moving* some of the computations from *runtime* (the moment the program is run) to *compile-time* (or better "specialisation-time" – the moment the program is specialised). The main practical reason for doing so is *efficiency*: since the residual program in general has less computations to perform, it seems natural to expect that it will run faster than the original program. The best benefit of program specialisation (with respect to efficiency) can be obtained when a single program is run a number of times while a part of its input remains constant over the different runs. In this case, the program can first be specialised with respect to the constant part of the input, while afterwards the

residual program can be run a number of times, once for each of the remaining (different) input parts. In such a staged approach, the computations that depend only on the constant input part are performed only once – during specialisation. In the non-staged approach, *all* computations – including those depending on the constant part – are performed over and over again in each run of the program. If we represent the time needed to evaluate a semantic function by T, and assume that a program P is run for inputs $[s, d_1], \ldots, [s, d_n]$ (with s being constant in each input), we can formally express the condition under which efficiency is gained by program specialisation by the following equation:

$$\sum_{i=1}^{n} T([\![P]\!][s, d_i]) > T([\![\texttt{mix}]\!][P, s]) + \sum_{i=1}^{n} T([\![P_s]\!]d_i)$$

where P_s denotes the residual program, that is $P_s = [\![\texttt{mix}P]\!][P, s]$.

In order to maximise the efficiency gain, a program specialiser should aim at performing as much computations as possible during specialisation; ultimately, every operation that does not depend on dynamic input should be performed by the specialiser. As can be seen from the above equation, also the efficiency of the specialisation process itself might be worth considering. Indeed, a practical system may be of no use when the execution time of the specialisation process exceeds a certain limit – even when the specialisation it performs is in some sense "optimal". Therefore, a practical specialisation system must often consider a tradeoff between the optimality of the residual program and the time it takes to perform the specialisation: it should perform a substantial amount of specialisation while guaranteeing a limited execution time.

Even in case none of the program's input is available for specialisation, it may be worthwhile to perform specialisation. Indeed, some parts of the program may have some static input which is available within the program itself. Consider for example the following program that computes the second power of a value.

Example 1.2

$$\mathit{quadratic}(x) \quad = \quad \mathit{power}(x, 2).$$

Even if the function $\mathit{quadratic}(x)$ itself can not be specialised in case the value of x is not known, the function call $\mathit{power}(x, 2)$ in the definition of $\mathit{quadratic}(x)$ *can* be specialised as in Example 1.1, leading to the following residual code:

$$\mathit{quadratic}(x) \quad = \quad x * x$$

The above example shows the usability of program specialisation as an optimisation technique that could be applied prior to (or even during) compilation. In the example, the program is optimised in the sense that some of its computations (those that do not depend at all on the program's input) are moved from run-time to specialisation-time.

We have described program specialisation as a two-stage process. More general approaches towards program specialisation exist, in which the computation of a program is divided in n stages (with $n \geq 2$), suitable for those applications in which input becomes known at n different times. In the remainder of this thesis, we will occasionally refer to such a *multi-level specialisation* approach (Glück and Jørgensen 1997) but focus on a traditional 2-stage process, dividing a program's computation over *specialisation-time* versus *run-time*.

1.2 Applications of Program Specialisation

We have seen that the basic motivation for doing program specialisation is efficiency: specialising a program (or, more general, a program *part*) with respect to some part of its input, may result in a program that has less statements to perform, and hence should run faster than the original program. Therefore, it is tempting to classify program specialisation as a program optimisation technique, and compare it with other existing techniques.

1.2.1 Program Specialisation and Compiler Optimisation

Most program optimisation techniques are described at the level of the compilation process: an optimising compiler performs several transformations on the program it is compiling in order to make execution of the compiled program as efficient as possible. A wide range of such techniques exist, and are described in the literature. We give a short overview of some of the existing compiler optimisation techniques and their relation with specialisation. A detailed comparison between existing program specialisation and compiler optimisation techniques is beyond the scope of this thesis.

Most compiler optimisation techniques are developed and described for imperative languages, most commonly C or Fortran, enabling the techniques to be applicable for a wide range of commonly used languages. But also other, so-called "higher-level" languages can profit from these optimisations, since these languages are often compiled to an imperative language (for example C) before execution, the imperative language being used as a high-level machine-independent assembly language. Examples are Mercury (Somogyi, Henderson, and Conway 1996; Henderson, Conway, and Somogyi 1995), Haskell (Thompson 1996), Turbo Erlang (Hausman 1993), wamcc (Codognet and Diaz 1995), and Janus (Saraswat, Kahn, and Levy 1990). In this way, the different optimisations become available also to these higher-level languages, often improving their performance. Let us first discuss some of the most commonly used techniques in optimising compilers. See (Aho, Sethi, and Ullman 1986; Bacon, Graham, and Sharp 1994) for a comprehensive survey.

Expression simplification Some techniques try to reduce the evaluation time of (mathematical) expressions. Known techniques in this area include *constant propagation* (propagating known values through a block of code) (Callahan, Cooper, Kennedy, and Torczon 1986; Kildall 1973; Wegman and Zadeck 1991), a transformation that is usually accompanied by *constant folding* (evaluating an expression to a constant value at compile-time), *expression simplification* (replacing an expression by an equivalent but more efficient one), and *common subexpression evaluation* (changing a block of code in such a way that a subexpression which is common to a number of expressions is computed only once) (Aho, Sethi, and Ullman 1986; Cocke 1970). More sophisticated techniques include *value range optimisations*, where not the value of an expression is propagated, but rather a limited range of values the expression can evaluate to (Patterson 1995).

Control flow optimisations Other techniques alter the control flow in a program, or propagate control flow information. Examples are *function inlining* (replacing a function call by an appropriate instance of the function's body) (Allen and Cocke 1972; Ball 1979; Scheifler 1977), *branch elimination* (removing a branch of an if-then-else statement), *if optimisation* (incorporating the knowledge that the conditional expression in an if-then-else statement evaluates to true in the then-branch and to false in the else-branch), and *dead code elimination* (removing code which is never reached in the program) (Aho, Sethi, and Ullman 1986; Allen and Cocke 1972; Bacon, Graham, and Sharp 1994).

Loop optimisations Another category of techniques also alter the control flow of a program, but concentrate on the handling of loops. These include *loop collapsing* (changing nested loops into a single loop) (Allen and Cocke 1972), *loop fusion* (fusing adjacent loops into a single loop) (Buda, Granovsky, and Ershov 1975) and *loop fission* (splitting a loop into many) (Kuck, Kuhn, Padua, Leasure, and Wolfe 1981), *loop unrolling* (reducing the number of iterations of a loop by replicating the loop's body) (Dongarra and Hinds 1979), *induction variable elimination* (reducing the number of induction variables that control a loop) (Aho, Sethi, and Ullman 1986; Allen 1969), and *tail-recursion elimination* (transforming a recursive function into a loop) (Aho, Sethi, and Ullman 1986). Other techniques move code from the body of the loop outside the loop when this code is invariable with respect to a particular iteration. Examples include *hoisting* (expressions that are invariant with respect to a particular iteration are moved outside the loop's body) (Aho, Sethi, and Ullman 1986; Cocke and Schwartz 1970), and *unswitching* (transforming a loop that contains an if-then-else statement of which the condition is independent of a particular iteration into an if-then-else statement that contains two instances of the loop) (Allen and Cocke 1972).

Optimising integer computation Other techniques try to optimise a single mathematical expression such that it is computed in a more efficient way. They include techniques to combine several instructions into a single one (*instruction combining*), and to transform some instances of integer multiplication, division and modulo operations with faster instructions like additions and bit-shift instructions (*strength reduction*) (Bernstein 1986). Sometimes expressions can be simplified by exploiting knowledge about the range of an integer variable that is limited, for example by previous bit-shifts on the variable.

Optimising memory usage A last category of techniques we discuss are techniques that alter the storage properties of a program, reducing its memory bandwidth requirements. Important techniques in this area are *register allocation* (allocating the available registers for optimal use) (Chaitin 1982; Chaitin, Auslander, Chandra, Cocke, Hopkins, and Markstein 1981; Chow and Hennessy 1990), *array padding* (inserting unused data locations between the columns of an array) (Burnett and Coffman, Jr. 1970; Bacon, Chow, ching R. Ju, Muthukumar, and Sarkar 1994), and *code collocation* (placing related code in close proximity) (Ferrari 1976; Hwu and Chang 1989).

Specific optimisations For completeness' sake, we note that there does exist a number of very specific optimisations that are sometimes found in optimising compilers. Examples are the compilation of the I/O format string into library routines (e.g. printf in C), and optimisations that are specifically targeted towards optimising some of the standard SPEC benchmarks.

From the above enumeration, it can be seen that there is a vast amount of optimisation techniques available to optimising compilers. Most existing optimising compilers only employ a limited number of these whereas some compilers use a number of optimisation techniques that are targeted specifically towards the application domain of the compiler. For example, most of the more aggressive loop optimisations are only applied by compilers which are targeted specifically towards use for scientific applications – typically containing a lot of nested loops.

The set of optimisation techniques described above can roughly be divided in two parts: a set of *lower level* optimisations (those dealing with memory usage and the optimisation of integer computations, for example) and a set of *higher level* optimisations. The lower level optimisations are often dependent of the specific computer architecture that is targeted by the compiler. This kind of optimisations is usually not achievable by program specialisation, by definition being an architecture-independent source-to-source transformation. On the other hand, some of the higher level optimisations are also achievable by program specialisation. In fact, program specialisation, being a general technique, embodies several

of the described higher level optimisations. Constant folding, constant propagation, common subexpression elimination, expression simplification, function inlining and loop unrolling can be seen as side effects of the program specialisation process. Also branch elimination, induction variable elimination and dead code elimination are obtainable by a general (but powerful) program specialiser.

The net effect of a particular optimisation is often hard to predict or to measure, since it may depend on the combination with other optimisation techniques. Also, optimisations do not always benefit from the results of earlier optimisations. Array padding, for example, can reduce the benefits of loop collapsing, due to the holes that are introduced in the array and the accompanying more involved index calculations (Bacon, Graham, and Sharp 1994). Even worse, the effect of a particular higher level optimisation may depend (regardless of further lower level optimisations) on the particular hardware on which the program is run. For example function inlining increases the locality of the code, but when multiple instances of a function are inlined in a single code block, it may also increase the number of cache misses (Bacon, Graham, and Sharp 1994).

From the above observations we conclude that, although every optimisation – including the more general technique of program specialisation – can be shown to be beneficial with respect to some property (for example reducing the number of iterations of a loop), it is hard to predict its effect on the final efficiency of the "optimised" program, since this effect may depend on other optimisations used during the compilation process, and on the particular hardware on which the program is run.

1.2.2 Program Specialisation in the Software Development Process

Program specialisation is a general technique that can achieve the same results as several techniques resulting from the research on optimising compilers. For different reasons (an important one being the efficiency of the compiler), the techniques used in optimising compilers are often *local* techniques that work on a limited scope of the program at hand: often the optimisation takes only a single basic block into account, like the body of a loop for example. This approach differs from the one taken by program specialisation, which is a *global* technique, partially evaluating expressions and propagating their results throughout the complete program. Thanks to the generality of the technique, a program specialiser is capable to incorporate input for the program into its optimisations, something compiler optimisation techniques are unable to do. This illustrates that program specialisation is *more* than an optimisation tool: it specialises code towards use in a specific context; the obtained optimisations can be seen as a "side effect" of this specialisation.

In the software-engineering community, it is widely recognized that software

should be built from general components, in order to be easily maintainable and adaptable to changing needs. A classical example of such a general component is an abstract data type. An abstract data type enables to represent values of a given type but does not unveil the representation of these values. Instead, the programmer is forced to deal with these kind of values through a number of procedures and functions only. A traditional example of an abstract data type is a *stack*, defined through the functions *new*, *push* and *pop*. Whereas the stack may be implemented as a list, the user of the *stack* data type is only entitled to use the *new*, *push* and *pop* functions to deal with values of the stack type. He has no knowledge whatsoever about their implementation manipulating a list.

A major drawback of using abstraction and general software components is the diminished performance compared with software written without them (for example software that could deal with the above stacks as if they were lists – addressing their values through their internal list representation). Generality introduces a runtime overhead due to an extra layer of *interpretation* that is added to the program. This interpretation overhead should be seen in a broad sense: it varies from passing ("interpreting") an extra parameter, over the traversal of intermediate data structures, to parsing and interpreting a whole program. Even worse, the extra layer of interpretation often forms an unbreakable barrier for traditional compiler optimisation techniques.

Program specialisation can be used as a software development tool, that is precisely capable of removing (part of) this interpretation overhead. Indeed, program specialisation can *specialise* a general software component with respect to a specific context the component is used in. Using the stack example, the program can be specialised with respect to a particular concrete representation for the abstract data type (transforming the calls to the *new*, *push* and *pop* functions into primitive operations dealing with, for example, the list representation). Such a scheme results in the best of both worlds: the program is developed using general components – with all the advantages regarding software development and maintenance – but before the program is compiled, program specialisation is used to remove as much abstraction and generality as possible from the program, specialising the general components with respect to their context. The resulting program can be optimised by a traditional optimising compiler. This scheme is depicted in Fig. 1.2.

To summarise, program specialisation can be considered at two levels in the software development process:

- As a tool to created specialised versions of "components", and
- as a generalised framework incorporating some of the higher level optimisations of an optimising compiler.

Figure 1.2: Program specialisation in the software development process.

1.2.3 Practical Applications of Program Specialisation

An early recognised application of partial evaluation is the conversion of *structural* information in a program into *procedural* information. Some programs like lexical scanners and parsers use such structural information (often in the form of tables) to drive their execution. Structural information is easy to read and to adapt to changing needs, but rather slow when used to control the execution of a program. An example where the ideas behind program specialisation are used to speed up table-driven execution is the compilation of LR parse tables into machine code (Penello 1986). More results on the specialisation of parsers are presented in (Pagan 1990).

Other early applications of program specialisation include the specialisation of neural network simulators with respect to a fixed network topology (reporting speedups from 25% to 50% (Jacobsen, Gomard, and Sestoft 1993)) and the specialisation of simulation and numerical applications (Berlin 1990; Berlin and Surati 1994; Glück, Nakashige, and Zöchling 1995).

An area which is, by nature, a fairly good candidate for program specialisation applications is *computer graphics*. Goad (Goad 1982) reports on the specialisation of a renderer in a flight simulator application: specialising the sorting of polygons with respect to a given landscape can result in rather extreme speedups ((Goad 1982) reports on a residual decision tree of depth 27, whereas the full sort requires more than 10000 comparisons). Another, almost classical, example is the specialisation of a ray-tracer with respect a to given scene. Reported speedups range from more than 6 for a simple ray-tracer (Mogensen 1986) to 1.5 or 3 for a more realistic ray-tracer (Andersen 1995).
More recent applications include:

Specialisation of hardware (Singh and McKay 1999) reports on a technique to specialise a FPGA chip with respect to some given input data. An FPGA (Field Programmable Gate Array) is a micro chip that can be programmed to assume a given logic function. Some of these chips can dynamically be reprogrammed by software, even while the chip is running. (Singh and McKay 1999) describes a prototype program specialiser that performs run-time constant propagation on a specific FPGA. Applications include, for example, on the fly specialisation of a decryption circuit with respect to a given decryption key. A circuit optimised for a given key results in a shorter critical path, allowing the data to be decrypted faster. Since the specialisation occurs at run-time, the chip can be re-specialised for a different key when the key changes.

Specialisation of planning problems The planning operations that need to be performed by any major airline form a complex task (Andersson, Housos, Kohl, and Wedelin 1997). A timetable needs to be constructed based on marketing expectations, airplanes need to be assigned to the individual legs of the timetable

and crew members need to be assigned a number of legs. Most major European airlines use a commercial system that generates solutions to some of the planning problems, and uses a dedicated language (Carmen (Bohlin 1990)) to express the rules that define the legality and optimality of different solutions. However, the system usually requires different runs (checking some of the rules) to obtain "good" solutions. Often these reruns need to be done with respect to a given subset of the rules only. Using a program specialiser for this problem results in a reduction of the computation time ranging from 35% to 70% on the Lufthansa rule set (Augustsson 1997).

Specialisation of fast Fourier transformations Fast Fourier Transformations (FFT) are expensive operations that are widely used in a lot of applications, among them audio and image processing (Press, Teukolsky, Vetterling, and Flannery 1992). Their performance can be improved by specialising an FFT with respect to a given function, since in that case some loops can be unfolded and some calls to goniometric library functions can be eliminated from the residual program (Lawall 1999). Optimising an FFT is often done by hand, but (Lawall 1999) explores the possibilities of specialising FFT's using the program specialiser Tempo for the C language and reports speedups of up to 9 times.

To conclude, (Mogensen 1999) reports on experiments where a program specialiser was used as a traditional optimiser in the compilation process and a program is "specialised" without incorporating a part of its input. In the experiments, a legal move generator for the Othello game is specialised, resulting in speedups of about 6 to 7, due to the more aggressive loop unrolling than obtainable with traditional compiler optimisations.

1.2.4 Theoretical Applications of Program Specialisation

Being a program transformation technique, program specialisation is related with two other important program manipulation techniques: *interpretation* and *compilation*. In what follows, we adhere to the notation of (Jones, Gomard, and Sestoft 1993) and denote with `int` and `compiler` respectively an interpreter and a compiler. For any program P with input i, we have the following:

$$[\![\texttt{int}]\!][P, i] = [\![P]\!]i. \tag{1.2}$$

An interpreter takes as input the program to interpret, together with that program's input. Equation 1.2 shows symbolically that running the interpreter on P and i has the same effect as running P on i.

Using a compiler, a program is run in two phases: first, the program is translated to a target program (usually in another language), which is then run on its

input. Symbolically:

$$\begin{array}{lcl} target & = & [\![\texttt{compiler}]\!]P \\ [\![target]\!]i & = & [\![P]\!]i. \end{array} \tag{1.3}$$

The relation between an interpreter, compiler and program specialiser are made explicit by the so called Futamura projections.

The first Futamura projection The first Futamura projection shows that *compilation* can be achieved by applying a program specialiser to an interpreter, symbolically:

$$[\![\texttt{mix}]\!][\texttt{int}, P] = target.$$

The above equation can be verified as follows ((Jones, Gomard, and Sestoft 1993))

$$\begin{array}{lcll} [\![target]\!]i & = & [\![P]\!]i & \text{By (1.3)} \\ & = & [\![\texttt{int}]\!][P, i] & \text{From (1.2)} \\ & = & [\![[\![\texttt{mix}]\!][\texttt{int}, P]]\!]i & \text{From (1.1)} \end{array}$$

The second Futamura projection A program specialiser also can generate a stand-alone compiler. This is called the second Futamura projection:

$$[\![\texttt{mix}]\!][\texttt{mix}, \texttt{int}] = \texttt{compiler}$$

Again, this equation can easily be verified as follows ((Jones, Gomard, and Sestoft 1993)).

$$\begin{array}{lcll} [\![\texttt{compiler}]\!]P & = & [\![\texttt{mix}]\!][\texttt{int}, P] & \text{First Futamura projection} \\ & = & [\![[\![\texttt{mix}]\!][\texttt{mix}, \texttt{int}]]\!]P & \text{From (1.1)} \end{array}$$

In order to generate a compiler using the second Futamura projection, the program specialiser `mix` needs to be written in its own input language, since it needs to be able to specialise itself.

The third Futamura projection The third Futamura projection shows how to obtain a *compiler generator*. A compiler generator is a program that transforms an interpreter into a compiler. As in (Jones, Gomard, and Sestoft 1993), we will denote a compiler generator with `cogen`, symbolically:

$$[\![\texttt{cogen}]\!]\texttt{int} = \texttt{compiler} \tag{1.4}$$

The third Futamura projection then is as follows:

$$[\![\texttt{mix}]\!][\texttt{mix}, \texttt{mix}] = \texttt{cogen}$$

Again, verification is straightforward:

$$\begin{array}{rcll} [\![[\![\texttt{cogen}]\!]\texttt{int}]\!]P & = & [\![[\![\texttt{mix}]\!][\texttt{mix}, \texttt{int}]]\!]P & \text{Second Futamura projection} \\ & = & [\![[\![[\![\texttt{mix}]\!][\texttt{mix}, \texttt{mix}]]\!]\texttt{int}]\!]P & \text{From (1.1)} \end{array}$$

The first and second Futamura projections are due to Futamura (Futamura 1971). The third Futamura projection was discovered independently by Beckman et al. (Beckman et al. 1976) and Turchin et al. From the beginning, the Futamura projections have been the driving force behind much of the research on program specialisation. In particular in the field of functional program specialisation, much attention has been paid to developing program specialisers capable of *self-application* – that is being able of effectively specialising itself – and using them to generate compilers and even compiler-generators. Apart from the theoretical interest in the subject, the so-called "cogen approach" to program specialisation – writing a compiler generator instead of a specialiser – has been used with considerable success, leading to simple and efficient specialisers.

1.3 State-of-the-Art in Program Specialisation

Until now, we have described *what* program specialisation does, and why it may be of use for the computer science community. In this section, we describe – at a high and informal level – *how* program specialisation can be achieved, and what the state-of-the-art is on the topic in different programming language paradigms.

1.3.1 On-line versus Off-line Specialisation

The act of program specialisation, also dubbed *partial evaluation*, can be described as a mixture of *evaluation* and *code generation*. The basic problem a program specialiser faces, is to decide *which* of a program's statements it should evaluate. The driving force behind this decision is twofold:

- The decisions made about what statements to evaluate should ensure that the specialisation process itself terminates.
- The obtained degree of specialisation should be "as good as possible".

Blindly evaluating program statements clearly is not a good idea, as it may result in a non terminating specialisation process. Reconsider the definition of the *power(x,n)* function in Example 1.1. When specialising a call, say *power(5,n)* in which the parameter n is unknown, one should not evaluate the subsequent recursive calls *power(5,n-1)*, *power(5,n-2)*,... as they all have an unknown second argument, and it is precisely this argument that controls the recursion process.

The other extreme, not evaluating any of the program's statements clearly is a safe (i.e. terminating) specialisation strategy. It is, however, not a satisfying

solution as in this case the residual program equals the original one, and hence no specialisation at all is obtained. As the decisions about what statements to evaluate control the behaviour of the specialiser, they are often referred to by the notion of *control decisions*. In the literature, one usually distinguishes between two main options when controlling the specialisation process.

On-line Control

A program specialiser that controls the process in an *on-line* fashion (also called an *on-line specialiser*) starts evaluating the program with respect to the partially available input, under the supervision of a control system. This control system continuously monitors the specialisation process, and may decide at any point to stop the evaluation, generate some code, and resume the evaluation at a different point.

Example 1.3 *Reconsider the definition of the* power(x,n) *function of Example 1.1. An on-line specialiser starts evaluating the call* power(x,5)*, building an execution trace as depicted in Fig. 1.3. At each point during the computation, the specialiser must decide whether or not to unfold the recursive call. A clever specialiser may notice that the recursion must stop eventually, since the parameter that controls the recursion is known and decreases with each call, hence the base case must be reached eventually. When the base case is reached, the remaining expression* $x * x * x * x * x$ *can not be evaluated, since the value of* x *is unknown. Hence, this expression is residualised.*

Figure 1.3: (Partial) evaluation of *power(x,5)*

On-line specialisation can be represented schematically as depicted in Fig. 1.4: the process takes the program P and some *concrete* values as input, and produces the residual program.

On-line specialisers usually keep track of the complete evaluation history (for example the sequence of recursive calls that were made) and use this information

Figure 1.4: On-line program specialisation

when making control decisions. This, together with the fact that an on-line specialiser has a statement's concrete (although partial) input available (the concrete value 5 for n in Example 1.3) makes the on-line specialiser a powerful tool. On-line specialisers tend to achieve a higher degree of specialisation than their *off-line* counterparts, precisely due to the information available for their decision making. The price to pay is efficiency of the specialiser itself: continuously monitoring the specialisation process and deciding, for each statement occurring during the process, whether or not to evaluate it, makes the specialiser inherently slower than competing *off-line* specialisers.

Off-line Control

Contrary to its on-line counterpart, an off-line specialiser works in two stages. First, the program to be specialised is analysed by a so called *binding-time analysis*. Such an analysis does not consider the concrete partial input of a program, but rather a *description* of this input, stating what parts of the input will be known at specialisation-time and what parts will only be known at runtime. In its most rudimentary form, binding-time analysis uses the descriptions *static* and *dynamic* to describe an argument that is known respectively at specialisation-time and at run-time. This description (*static* or *dynamic*) is often referred to by the *binding-time* of the argument. The notion of a binding-time refers to the stage in the computation where the argument is bound to a concrete value: *static* denotes an early binding (the argument is known at specialisation-time), *dynamic* denotes a late binding (the argument becomes bound to a concrete value only at run-time). Given such a *binding-time approximation* of the program's input arguments, binding-time analysis then computes

- a binding-time of each *variable* in the program, denoting whether the variable will definitely become bound to a value during specialisation, or possibly only at runtime.
- for each of the program's statements, whether it should be evaluated during specialisation or be residualised.

Basically, binding-time analysis performs a data flow analysis to propagate the binding-times from the program's input arguments throughout the complete program, computing binding-times for the program's variables.

The second task of the binding-time analysis is to derive, for every program statement, a control instruction stating whether the statement should be evaluated during specialisation or not. Note that this control decision is taken based *solely* on the *binding-times* of the involved variables; *not* on the statement's concrete input, nor on any specialisation history (which is simply not available during analysis). The output of binding-time analysis is often represented as an *annotated program*, which is a version of the original (source) program, where every statement is annotated with an instruction either to *reduce* the statement during specialisation or to *residualise* the statement.

Example 1.4 *Reconsider the* power(x,n) *function from Example 1.1. If we want to specialise this function for the value* $n = 5$ *using an off-line system, the binding-times of the function's arguments are* x*:*dynamic *and* n*:*static*. The annotated program resulting from binding-time analysis is depicted in Fig. 1.5, where we use the following (standard) notation: underlined variables denote* dynamic *variables and underlined operations denote operations that are to be residualised. The an-*

$$\begin{array}{lcl} power(\underline{x}, n) & = & \textit{if } n = 1 \\ & & \textit{then } \underline{x} \\ & & \textit{else } \underline{x * } power(\underline{x}, n-1) \end{array}$$

Figure 1.5: Annotated version of the *power(x,n)* function

notations in Fig. 1.5 denote that the variable x *is dynamic,* n *static and that the multiplication operation must be residualised. All other expressions and operations are annotated reducible (the test* $n = 1$*, the expression* $n - 1$*, the recursive call and the if-then-else).*

The actual specialisation can be performed once the results from binding-time analysis are available. The input to an off-line specialiser thus consists of an annotated source program, together with concrete values for those program arguments that were assumed *static* during binding-time analysis. An off-line specialiser performs its specialisation by following the annotations on the source program: if a program statement is annotated to *reduce*, the statement is evaluated by the specialiser, otherwise the statement is recorded in the residual program. Following the annotations in Example 1.4 for the input $n = 5$ results in the same execution trace and residual program as in Example 1.3. Recall that the difference between the two approaches is the moment at which the control decisions that build the execution trace are taken. In the on-line approach, these decisions are made on

the fly, while in the off-line approach the decisions are taken beforehand by the binding-time analysis.

Since all control decisions are taken beforehand by the binding-time analysis, the off-line specialiser is inherently simpler, and hence more efficient than an on-line specialiser. On the other hand, since all control decisions are based on *approximations* of the available input, off-line specialisation can in general not achieve as high a degree of specialisation as an on-line specialiser can. The process of off-line specialisation is depicted in Fig. 1.6. It is noteworthy that most

Figure 1.6: Off-line program specialisation

off-line specialisers take *some* control decisions during the specialisation process, and hence are in a sense *hybrid* approaches.

1.3.2 Compile-time versus Run-time Specialisation

In the previous sections, we have described program specialisation as a part in a staged computation process: instead of evaluating a program with respect to its complete input, a program is first (partially) evaluated with respect to a part of its input, after which the remaining computations are performed with respect to the remaining input. The specialisation we have considered is so called *compile-time specialisation*: the specialisation phase is performed prior to or during the compilation stage in the software development process. This is the standard view on program specialisation, as it is also embodied by most of the practical systems mentioned above.

For some applications, however, compile-time specialisation might be too limited as an optimisation technique. Indeed, some applications contain code which is well suited for specialisation, the only problem being that the values with respect to which the code could be specialised are only known at run-time. An example taken from (Consel et al. 1996) is a program that implements session-oriented transactions. When a session is opened, some information about the session parameters becomes available. This information is used by all subsequent procedure calls that perform the actual transactions comprising the session. Hence, these

procedures could be specialised with respect to the information from the particular session in order to speed up their execution. After the session is closed, the specialised procedures are no longer valid and should be discarded.

Different forms of so called *run-time specialisation* have been explored, relying on the concept of *run-time code generation* (Keppel, Eggers, and Henry 1991; Keppel, Eggers, and Henry 1993; Leone and Lee 1994; Fujinami 1997; Marlet, Consel, and Boinot 1999) (also called *dynamic compilation* (Auslander, Philipose, Chambers, Eggers, and Bershad 1996)). The general idea is to postpone the actual code generation for those code blocks that should be specialised with respect to some value until run-time, when the particular value is known and the code can be specialised. In general, run-time specialisation only pays off when the time needed to perform the (specialised) code generation does not exceed the time that is gained by the specialisation. Therefore, most techniques for run-time specialisation focus on ways to generate the specialised code as fast as possible. In (Keppel, Eggers, and Henry 1993; Keppel, Eggers, and Henry 1991), a system is developed in which the programmer can manually construct templates (or expression trees) that can be specialised and compiled into machine code at run-time. The system 'C ("tick" C)(Engler, Hsieh, and Kaashoek 1996) extends the C language with provisions for writing and manipulating such templates. The system in (Auslander, Philipose, Chambers, Eggers, and Bershad 1996) combines a static and a dynamic compiler and uses programmer-inserted annotations on the original source code to identify those code parts that should be compiled dynamically. The *static* compiler compiles these so called *dynamic regions* into machine code templates in which the instructions can contain holes to be filled in with a value that is provided at run-time. Also during static compilation, a number of directives is generated, instructing the dynamic compiler how to produce executable code from the machine code templates. The dynamic compiler (which is called the *stitcher*) then simply follows the directives to glue the templates together, performs some static computations and fills in the holes with the appropriate values. Note that the system is off-line in nature: the directives specify what and how to glue the templates together (implementing, for example, the unrolling of a loop by concatenating the template of the loop body a number of times). Also the approach of (Leone and Lee 1994; Lee and Leone 1996) requires user-provided annotations to guide a run-time code generator for ML.

In (Consel et al. 1996), a uniform approach is presented for compile-time and run-time specialisation, implemented in Tempo, a program specialiser for the C language. The approach is off-line. First, a binding-time analysis is used to distinguish the static and dynamic computations in a program. Next, an *action analysis* is used to assign specialisation actions to each program construct (Consel and Danvy 1990). These actions can not only be used to perform compile-time specialisation, but also to generate and compile source templates during static compilation. The compiled templates can than be glued together and filled with

run-time values by a dynamic compiler at run-time (Consel and Noël 1996; Consel et al. 1996).

1.3.3 State-of-the-Art

Program specialisation has been investigated in different paradigms. Breaking work on program specialisation of imperative languages include C-mix by Andersen (Andersen 1993) and more recently Tempo (Consel et al. 1996; Hornof and Noyé 1997) by Consel and his group. Both being off-line specialisers for the C language, the latter includes a compile-time as well as a run-time specialiser.

With the advent of object-oriented languages, some attention has been paid to specialisation for these languages, as the unfolding of (at least some) method invocations seems crucial in boosting performance of such applications. Dean and others (Dean, Chambers, and Grove 1994; Dean, Chambers, and Grove 1995) develop a technique in which the inlining of method invocations is based on the information that was previously recorded in a database regarding the inlining behaviour of the same method at call sites with similar static information. To that end, a *type group analysis* is developed (Dean, Chambers, and Grove 1994) that computes how much of the static information available at a call site was profitably used during inlining. Another approach is followed in (Volanschi, Consel, and Cowan 1997), where an extension to the Java language is described, providing annotations that enable the programmer to specify how generic programs should be specialised for a particular usage.

A lot of attention has been paid to specialisation of functional programs, where on-line as well as off-line approaches have been followed. Research in the on-line field include Fuse (Weise, Conybeare, Ruf, and Seligman 1991) and Turchin's original work on supercompilation (Turchin 1986), later on revisited by Sørensen and others (Sørensen 1994; Sørensen and Glück 1995; Sørensen, Glück, and Jones 1996). However, in the functional setting most work focuses on binding-time analysis and off-line specialisation, originally motivated to achieve better self-application (Futamura 1971; Jones, Sestoft, and Søndergaard 1985). Whereas initial analysis dealt with first-order languages (Jones, Sestoft, and Søndergaard 1985), more recently developed analyses deal with higher-order aspects (Gomard and Jones 1991; Bondorf 1991), polymorphism (Mogensen 1989; Henglein and Mossin 1994) and partially static data structures (Launchbury 1990).

Already from the beginning, the logic programming community has paid most attention to on-line specialisation (Komorowski 1992; Gallagher 1993). Numerous techniques have been proposed for the on-line control of the specialisation process, most of them relying on either well-founded orderings (Bruynooghe, De Schreye, and Martens 1992; Martens and De Schreye 1996) or well-quasi orderings (Leuschel, Martens, and De Schreye 1998) of the constructed derivations – an approach also explored in the context of supercompilation (Sørensen and Glück 1995). The relation between on-line specialisation of logic programs and supercompilation

has been established in (Glück and Sørensen 1994). Generalising the frameworks of (Lloyd and Shepherdson 1991) and (Martens and Gallagher 1995) for partial deduction of logic programs, a *narrowing*-driven, on-line specialisation technique for the language Curry (Hanus 1997) has been developed (Alpuente, Falaschi, and Vidal 1998). Narrowing is the basic execution mechanism for functional-logic programs, integrating the functional- and logic programming paradigms (Hanus 1994).

The work in this thesis comprises work in an off-line as well as on-line setting for the logic programming paradigm. More detailed descriptions and comparisons with work in the logic-, functional- and functional-logic programming communities is given throughout the thesis.

Chapter 2

Technical Background

In the beginning was the word.
But by the time the second word was added to it,
there was trouble. For with it came syntax...

— *John Simon*

In this chapter, we give some technical background on logic programming, the semantics of logic programs and concepts and techniques of program specialisation in a logic programming setting. The presentation of the material is partly inspired by (Martens 1994; Leuschel and Bruynooghe 2001).

2.1 Logic Programming Basics

In this section, we give an overview of the basics of logic programming. For a more detailed overview, we refer to (Lloyd 1987b; Apt 1990). Assume that a first-order language is given, containing variables, function symbols – or functors, and predicate symbols. For some fixed language, we denote with $\mathcal{V}$, Σ and Π the sets of, respectively, variables, function symbols and predicate symbols. Function and predicate symbols have an arity associated, which is a natural number identifying the number of arguments the function or predicate symbol has. Function symbols of arity 0 are often denoted "constants". Throughout this work, we use uppercase letters from the Latin alphabet to denote elements from $\mathcal{V}$, and lowercase letters – usually f, g, and h – possibly adorned by a subscript or accent, to denote elements from Σ while the lowercase letters p and q are used to denote predicate symbols

from Π. In case the arity of a function or predicate symbol is important, we identify the symbols as, for example, f/n or p/n where $n \in \mathbb{N}$ denotes the arity of the symbol.

A term constructed from $\mathcal{V}$ and Σ is either a variable (from $\mathcal{V}$) or a function symbol $f/n \in \Sigma$ applied to a sequence of n terms. The set of all such terms is denoted by $\mathcal{T}(\mathcal{V}, \Sigma)$. To name a term we use again lowercase letters, but usually from the end of the alphabet. Given a term t, we denote with $\mathcal{V}(t)$ the set of variables occurring in t. An atom is a predicate symbol $p/n \in \Pi$ applied to a sequence of n terms. A literal is an atom, possibly preceded by $\neg$ denoting the negation of the atom. Literals of the latter kind are called negative; otherwise they are positive. A clause is a formula of the form

$$H \leftarrow B_1, \ldots, B_n, n \geq 0$$

where H is an atom and $B_1, \ldots, B_n$ are literals. The formula is a rule, consisting of a condition part $B_1, \ldots, B_n$ being a conjunction of literals and a consequent H being an atom. The atom H is called the clause's head, the conjunction $B_1, \ldots, B_n$ its body. The body may be empty, in which case the clause is called a fact. All variables occurring in a clause's body are supposed to be universally quantified, with the scope of the quantifiers ranging over the entire clause. A set of clauses is called a program. A clause with an empty head and a non-empty body is called a query, or sometimes an initial goal. A query is hence of the form

$$\leftarrow B_1, \ldots, B_n, n \geq 1.$$

If the body of a clause or a query contains only positive literals, it is called a definite clause. If all the clauses of a program are definite, the program is called a definite program. We will use "expression" to denote any object that is a term, an atom, a literal, a clause or a query. Terms and atoms are also called simple expressions. Expressions that do not contain any variables are said to be ground.

Regarding notation: for any set S, we denote with $\wp(S)$ its powerset, and with S^* the set of finite sequences over S. When we deal with a sequence of elements $a_1, \ldots, a_n$ where the sequence itself is an object of interest, we denote it explicitly with $\langle a_1, \ldots, a_n \rangle$, the empty sequence being $\langle\rangle$. If B denotes a syntactic object of some sort, we use $\overline{B}$ to denote a particular but unspecified sequence over such objects.

A substitution is defined as a finite mapping $\mathcal{V} \mapsto \mathcal{T}(\Sigma, \mathcal{V})$ from distinct variables to terms. For any function f, we denote with $dom(f)$ the domain of f and for any $a \in dom(f)$, $f(a)$ denotes the object associated to a in f. As usual, we denote substitutions as follows

$$\theta = \{X_1/t_1, \ldots, X_n/t_n\}$$

where each $X_i \neq t_i$. A ground substitution is a substitution that maps variables to ground terms. If E is an expression and θ a substitution, than $E\theta$ denotes

the result of applying θ to E and is defined as the expression obtained from E by simultaneously replacing the variables from the domain of θ that occur in E by their corresponding term. We call $E\theta$ an instance of E. If E is an expression and F is an instance of E, then E is said to be more general than F, denoted $E \leq F$, or simply a generalisation of F. If E and F are expressions such that E is an instance of F and F is an instance of E, then E and F are called variants, denoted by $E \approx F$. A substitution θ such that $E\theta \approx E$ is called a renaming of E. If $E \leq F$ and $E \not\approx F$, we say that E is strictly more general than F, denoted with $E < F$. Substitutions can be composed in the following way (from (Leuschel 1997)):

Definition 2.1 *Let $\theta = \{X_1/s_1, \ldots, X_n/s_n\}$ and $\sigma = \{Y_1/t_1, \ldots, Y_k/t_k\}$ be substitutions. Then the* composition *of θ and σ, denoted by $\theta\sigma$, is defined to be the substitution $\{X_i/s_i\sigma \mid 1 \leq i \leq n \wedge s_i\sigma \neq X_i\} \cup \{Y_i/t_i \mid 1 \leq i \leq k \wedge Y_i \notin \{X_1, \ldots, X_n\}\}$.*

An important operation on terms and atoms is *unification*.

Definition 2.2 *Let S be a set of simple expressions. A substitution θ is called a* unifier *for S if $S\theta$ is a singleton. A unifier θ for S is called a* most general unifier *for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$*

The most general unifiers of a set S are unique modulo variable renaming; hence we often refer to *the* most general unifier of a set S of simple expressions. The notion of most general unifier can easily be extended to deal with other syntactic constructs, for example conjunctions of atoms. Simplifying notation, we write $\mathrm{mgu}(E_1, E_2)$ instead of $\mathrm{mgu}(\{E_1, E_2\})$ when a set of two expressions is involved, as will often be the case. Algorithms to compute the most general unifier can be found, for example in (Lloyd 1987b; Apt 1990). We will sometimes refer to the most general unifier simply by "mgu" as is common practice.

In what follows, we restrict our attention to definite logic programs and we discuss two useful semantics for definite logic programs. We start by defining the well-known procedural semantics, and later on turn our attention towards a more declarative semantics, the S-semantics. The procedural semantics models the most commonly used execution mechanism for logic programs: SLD-resolution. The S-semantics, which can be shown to be "equivalent" with the procedural semantics, abstracts a particular execution mechanism and – being a declarative semantics – makes it easier to reason about correctness of several program transformations. We start by giving the basic definitions concerning SLD-resolution. They can be found in (Lloyd 1987b; Apt 1990).

Definition 2.3 *Let Q be the query $\leftarrow A_1, \ldots, A_k, \ldots, A_n$ and C be the clause $A \leftarrow B_1, \ldots, B_q$. Then Q' is* derived *from Q and C using the most general unifier θ if the following conditions hold:*

1. *A_k is an atom, called the* selected *atom in Q*
2. *θ is a most general unifier of A_k and A*
3. *Q' is the query $\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_q, A_{k+1}, \ldots, A_n)\theta$.*

Definition 2.4 *Let P be a definite program and Q_0 a definite query. An SLD-derivation of $P \cup \{Q_0\}$ consists of a possibly infinite sequence $Q_0, Q_1, Q_2 \ldots$ of queries, a sequence of renamed apart variants of program clauses $C_1, C_2, \ldots$ of P and a sequence $\theta_1, \theta_2, \ldots$ of most general unifiers such that each Q_{i+1} is derived from Q_i and C_{i+1} using θ_{i+1}.*

An SLD-derivation can be finite or infinite. A finite SLD-derivation that ends in an empty query (i.e. a query without any atom) is called a successful derivation, or an SLD-refutation. A finite derivation that ends in a query of which the selected atom does not unify with the head of any program clause is called a failed SLD-derivation.

The basic execution mechanism of logic programs consists in constructing SLD-derivations for a query and a program. If a succeeding SLD-derivation is constructed, one is interested in what is actually "computed" by the derivation. This is formally defined by the concept of a computed answer (substitution).

Definition 2.5 *Let P be a definite program and Q_0 a definite query. A* computed answer (substitution) *θ for $P \cup \{Q_0\}$ is the substitution obtained by restricting the composition $\theta_1 \ldots \theta_n$ – being the sequence of most general unifiers used in an SLD-refutation of $P \cup \{Q_0\}$ – to the variables of Q_0.*

In what follows, we sometimes use $Q \overset{\theta}{\rightsquigarrow}_P B_1, \ldots, B_n$ to denote an SLD-derivation from Q to $B_1, \ldots, B_n$ in the program P such that θ is the composition of the most general unifiers computed in the derivation. Likewise, $Q \overset{\theta}{\longrightarrow}_P \square$ denotes the SLD-refutation of Q in P with computed answer substitution θ. Computing answers for a given program and query requires a form of reasoning. Indeed, when continuing the construction of an SLD-derivation $Q \overset{\theta}{\rightsquigarrow}_P B_1, \ldots, B_n$, one has to select a particular atom in $B_1, \ldots, B_n$ and, if the selected atom unifies with more than one clause in P, to select one of the unifying clauses. A commonly used representation of this reasoning process employs the notion of an SLD-tree.

Definition 2.6 *Let P be a definite program and Q a definite goal. An SLD-tree for $P \cup \{Q\}$ is a tree in which each node of the tree is a possibly empty definite query, the root node is the query Q and for each node $\leftarrow A_1, \ldots, A_k, \ldots, A_n$ ($n \geq 1$) we have the following: if A_k is the selected atom, then for each variant of a clause $A \leftarrow B_1, \ldots, B_m$ in P such that A_k and A are unifiable with most general unifier θ, the node has a child of the form*

$$\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n)\theta.$$

Note that each branch in an SLD-tree is an SLD-derivation. If one or more of the branches are infinite derivations, the tree is called infinite, otherwise it is finite. If all branches of a finite SLD-tree are failing, the tree is called a finitely failing SLD-tree. Given a definite program P and query Q, an SLD-tree for $P \cup \{Q\}$ is completely determined by a computation rule that determines, in each derivation step, which atom is selected in the query. Various such computation rules exist, varying in complexity.

Example 2.1 *Consider the following definite program P, borrowed from (Lloyd 1987b):*

$$p(X, X) \leftarrow$$
$$p(X, Y) \leftarrow q(X, Z), p(Z, Y)$$
$$q(a, b) \leftarrow$$

Using the computation rule "always choose the leftmost atom", the SLD-tree for $P \cup \{\leftarrow p(X, b)\}$ is depicted in Fig. 2.1. Selected atoms are printed in italic fashion and branches are annotated with the necessary substitutions to allow the reconstruction of computed answer substitutions.

Figure 2.1: An SLD-tree.

SLD-trees are interesting, as they literally contain the SLD-derivations and computed answers for a query Q in a program P (under a particular computation rule). A somewhat more abstract representation of the SLD-resolution process is a so-called *proof tree* that registers, in finer detail, the relations that exist between the selected atoms in the derivation. A proof tree is an AND-OR tree in which the nodes are labelled with an atom (and possibly a substitution). An OR-node represents a branching in the derivation: each child of the OR-node represents a possibly further derivation from the atom onwards. An AND-node, on the other hand, represents that each of its children must be followed in order to find a derivation for the query; the order in which these children are to be considered (possibly incorporating the substitutions computed by the previously considered child) is determined by the selection rule.

Example 2.2 *Reconsider the program P from Example 2.1. The proof tree constructed for $P \cup \{\leftarrow p(X, b)\}$ under the "always choose the leftmost atom" computation rule is depicted in Fig. 2.2.* AND*-nodes are identified by a dot and the branches originating from them are joined by an arc to stress the fact that they should be considered together. As the "left-to-right" computation rule is used, the leftmost child of an* AND*-node is considered first, and if a succeeding derivation can be built for it, the computed answer substitution is applied to the* AND*-node's second leftmost atom for which the derivation is continued, and so on.*

Figure 2.2: A proof tree.

In the above discussion, we have adopted the traditional way of defining the "behaviour" of a definite logic program by defining the notions of an SLD-derivation and an SLD-tree as a means of computation, and a computed answer substitution being the "observable" property of the computation. In the S-semantics approach to logic programming (Falaschi, Levi, Martelli, and Palamidessi 1989; Denis and Delahaye 1991; Bossi, Gabbrielli, Levi, and Meo 1994; Bossi, Gabbrielli, Levi, and Martelli 1994), one defines a bottom-up fixed point semantics that is equivalent with the operational semantics, in the sense that it characterises precisely the set of computed answer substitutions. Hence, the fixed point semantics of a program can be seen as modeling the program's operational behaviour in a declarative way (Falaschi, Levi, Martelli, and Palamidessi 1989). This is useful when considering the "correctness" of an abstraction of the semantics for goal independent abstract interpretation, as is the case in the bottom-up specialisation framework we define in Chapter 8. The operational semantics of a definite program P with respect to the computed answer substitution as the observable property can be defined as follows (Falaschi, Levi, Martelli, and Palamidessi 1989):

Definition 2.7 *Let P be a definite program. Then,*

$$\mathcal{O}(P) = \left\{ \; p(\overline{X})\theta \;\middle|\; p(\overline{X}) \xrightarrow{\theta}_P \Box \; \right\}$$

The denotation of a program, $\mathcal{O}(P)$ is a set of non ground atoms. In order to model $\mathcal{O}(P)$, the usual Herbrand base (Lloyd 1987b; Apt 1990) is extended to the set of all the possibly non-ground atoms modulo variance. Let $\mathcal{H}_V$ denote the extended, or non-ground Herbrand Base; that is the set of all atoms modulo variance for a fixed language comprising Π, Σ and $\mathcal{V}$. A π-interpretation is then defined as a subset of $\mathcal{H}_V$. The set of π-interpretations is organised in a lattice $(\Im, \subseteq)$. These notions are due (Falaschi, Levi, Martelli, and Palamidessi 1989; Bossi, Gabbrielli, Levi, and Martelli 1994; Bossi, Gabbrielli, Levi, and Meo 1994).

The denotation of a program P, $\mathcal{O}(P)$, is a π-interpretation (thus a set of possibly non ground atoms) that is observationally equivalent with respect to any goal (Falaschi, Levi, Martelli, and Palamidessi 1989). Formally, this can be stated as follows:

Theorem 2.1 *(From (Falaschi, Levi, Martelli, and Palamidessi 1989; Bossi, Gabbrielli, Levi, and Martelli 1994)) Let P be a definite program and $Q = \leftarrow B_1, \ldots, B_n$ a definite query. Then $Q \xrightarrow{\theta}_P \Box$ if and only if there exist (renamed apart) atoms $A_1, \ldots, A_n \in \mathcal{O}(P)$ and a renaming ρ such that $\theta = \gamma\rho$ restricted to the variables of Q where $\gamma = mgu((A_1, \ldots, A_n), (B_1, \ldots, B_n))$.*

Let us now introduce the immediate consequence operator T_P^π from (Falaschi, Levi, Martelli, and Palamidessi 1989) whose least fixed point can be shown to be equivalent to the computed answer substitution semantics $\mathcal{O}(P)$.

Definition 2.8 *(From (Falaschi, Levi, Martelli, and Palamidessi 1989; Bossi, Gabbrielli, Levi, and Martelli 1994)) Let P be a definite program and I a π-interpretation.*

$$T_P^\pi(I) = \left\{ \; H\theta \in \mathcal{H}_V \;\middle|\; \begin{array}{l} \exists H \leftarrow B_1, \ldots, B_n \in P, \\ \exists A_1, \ldots, A_n \textit{ renamed apart variants of atoms in } I, \\ \exists \theta = mgu((A_1, \ldots, A_n), (B_1, \ldots, B_n)) \end{array} \right\}$$

The immediate consequence operator T_P^π differs from the standard T_P operator (Lloyd 1987b; Apt 1990) in that it derives (possibly non ground) instances of clause heads by unifying the body atoms with atoms in the π-interpretation whereas T_P considers all possible ground instances. Since the set of all π-interpretations, $(\Im, \subseteq)$ is a complete lattice, we have the following:

Theorem 2.2 *(From (Falaschi, Levi, Martelli, and Palamidessi 1989; Bossi, Gabbrielli, Levi, and Martelli 1994)) The T_P^π operator is continuous on $(\Im, \subseteq)$. Then there exists the least fixed point $T_P^\pi \uparrow \omega$ of T_P^π.*

Definition 2.9 *(From (Falaschi, Levi, Martelli, and Palamidessi 1989; Bossi, Gabbrielli, Levi, and Martelli 1994)) The fixed point semantics of a definite program P is defined as $\mathcal{F}(P) = T_P^\pi \uparrow \omega$.*

In (Falaschi, Levi, Martelli, and Palamidessi 1989), the equivalence between the computed answer substitution semantics and the fixed point semantics is established:

Theorem 2.3 *(From (Falaschi, Levi, Martelli, and Palamidessi 1989; Bossi, Gabbrielli, Levi, and Martelli 1994)) Let P be a definite program. Then $\mathcal{F}(P) = \mathcal{O}(P)$.*

We will use the fixed point semantics in Chapter 8, where we introduce the concept of a bottom-up partial deduction, for which we state a correctness result with respect to the fixed point semantics, and hence (by Theorem 2.3) the operational semantics.

2.2 From Plain Evaluation to Partial Evaluation in Logic Programming

In the literature on program specialisation for logic programming, one often finds different terminology. In a seminal paper on the subject (Komorowski 1992), Komorowski introduced the terminology "partial deduction" in the context of pure logic programs – programs without side effects and cuts – for which the notion of "deduction" rather than evaluation seems appropriate. Some subsequent authors reintroduced the terminology "partial evaluation" when logic programs *with* side effects and cuts were the subject (Sahlin 1993). Other authors (Leuschel and Bruynooghe 2001) stick to the terminology of partial deduction, in order to distinguish the technique from its counterpart in functional languages. In a functional setting, one can partially evaluate only those expressions that depend solely on static input, whereas in a logic programming setting, one can often reduce expressions that do depend on some dynamic input (Leuschel and Bruynooghe 2001), which relates the technique more to supercompilation (Turchin 1986; Sørensen and Glück 1995; Sørensen, Glück, and Jones 1996; Glück and Sørensen 1996; Sørensen, Glück, and Jones 1994) in functional languages and unfold/fold transformation techniques (Burstall and Darlington 1977; Pettorossi and Proietti 1994). In the context of this thesis, we consider only pure logic programs and hence the "partial deduction" terminology seems appropriate. However, we will sometimes also refer to the process by partial evaluation particularly in the context of our work on Mercury in which both the language and the line of work is related to the field of functional programming and the notion of partial evaluation therein.

2.2.1 Basic Notions of Partial Deduction

In a logic programming setting, "running" a program P consists of evaluating a query Q with respect to P. Input to the program is represented by the atom(s) and terms in Q. *Partial* input to the program is represented by a less instantiated query Q' – that is, $Q' \leq Q$. In what follows, we refer to Q' by the notion of a *partial deduction query* being a generalisation of the run-time queries we are interested in. By the nature of logic programming, we can build an SLD-tree for the program P with respect to such a more general query Q' as usual. While at first sight, the process of "partial" deduction seems to boil down to plain deduction, the resulting SLD-tree is likely to be infinite, and the computation nonterminating. In order to approximate such a possibly infinite computation by a finite (and hence practically applicable) computation, the basic task of partial deduction is to construct a finite number of *partial* SLD-trees that together "cover" the complete and possibly infinite SLD-tree for $P \cup \{Q'\}$.

The basic framework for partial deduction was developed by Lloyd and Shepherdson (Lloyd and Shepherdson 1991). In what follows, we recall some of the basic notions of (Lloyd and Shepherdson 1991). First, we extend the notion of an SLD-tree into that of a *partial* SLD-tree.

Definition 2.10 *Let P be a definite program and Q a definite goal. A* partial *SLD-tree for $P \cup \{Q\}$ is a tree in which each node of the tree is a possibly empty definite query, the root note is the query Q and for each node $\leftarrow A_1, \ldots, A_k, \ldots, A_n$ ($n \geq 1$), either the node is a leaf node or we have the following: if A_k is the selected atom, then for each variant of a clause $A \leftarrow B_1, \ldots, B_m$ in P such that A_k and A are unifiable with a most general unifier θ, the node has a child of the form*

$$\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n)\theta.$$

The difference between a partial SLD-tree and an SLD-tree is that in the former the leafs may be arbitrary queries in which no atom is selected. Such leafs are sometimes called *dangling*. A partial SLD-tree is called *trivial* if no atom is selected in the root node, and *nontrivial* otherwise. Selecting an atom in a leaf of a partial SLD-tree and adding all the resolvents as children is called *unfolding*. Hence, new partial SLD-trees can be obtained from a trivial SLD-tree by performing a sequence of unfolding steps.

Definition 2.11 *Let P be a definite program, A an atom and $A \overset{\theta}{\rightsquigarrow}_P B_1, \ldots, B_n$ an SLD-derivation for $P \cup \{\leftarrow A\}$. The* resultant *of the derivation is the clause*

$$A\theta \leftarrow B_1, \ldots, B_n.$$

The notion of a resultant can be extended to an SLD-tree:

Definition 2.12 *Let P be a definite program, A an atom and τ a finite partial SLD-tree for $P \cup \{\leftarrow A\}$. Let leaves$(\tau) = \{Q_1, \ldots, Q_r\}$ be the (non-root) leaves of the non failing branches of τ and resultants$(\tau) = \{R_1, \ldots, R_n\}$ the resultants corresponding to the derivations $A \overset{\theta}{\rightsquigarrow}_P Q_i$. The set resultants$(\tau)$ is called a* partial deduction *for A in P.*

If $\mathcal{A} = \{A_1, \ldots, A_n\}$ is a finite set of atoms, then a *partial deduction for $\mathcal{A}$* in P is defined as the union of the partial deductions for $A_1, \ldots, A_n$ in P. In order for a partial deduction to be "correct", it must satisfy the *closedness* and *independence* conditions (Lloyd and Shepherdson 1991):

Definition 2.13 *Let S be a set of first-order formulas and $\mathcal{A}$ a finite set of atoms. Then S is* $\mathcal{A}$-closed *if and only if each atom in S is an instance of an atom in $\mathcal{A}$. Furthermore we say that $\mathcal{A}$ is* independent *if and only if no pair of atoms in $\mathcal{A}$ have a common instance.*

The following theorem defines a notion of correctness and expresses under what conditions a partial deduction of $\mathcal{A}$ is "correct" in P.

Theorem 2.4 *Let P be a definite program and $\mathcal{A}$ a finite, independent set of atoms, Q a definite query and P' a partial deduction of $\mathcal{A}$ in P such that $P' \cup \{Q\}$ is $\mathcal{A}$-closed. Then, the following hold:*

- *$Q \overset{\theta}{\longrightarrow}_{P'} \square$ if and only if $Q \overset{\theta}{\longrightarrow}_P \square$.*
- *The SLD-tree for $P' \cup \{Q\}$ is finitely failing if and only if the SLD-tree for $P \cup \{Q\}$ is.*

In other words, under the conditions stated in the above theorem, computation with a partial deduction of a program is sound and complete with respect to computation with the original program. Note that Theorem 2.4 does not guarantee the preservation of termination characteristics of P in P', nor does it guarantee that the order in which solutions are found in P' equals the order they are found in P. Due to the closedness condition, every atom A in the residual program P' and query Q is an instance of an atom B in $\mathcal{A}$, that is $A = B\theta$ for some substitution θ. Hence, when creating the residual predicate associated to the atom B, one could filter out all the structure from B, keeping only the variables as arguments of the residual predicate. The call to the residual predicate is renamed accordingly, keeping only the appropriate terms from θ as arguments. Formally, we define the notion of a *filtered partial deduction* as follows:

Definition 2.14 *Let P be a definite program, $\mathcal{A}$ a finite independent set of atoms, Q a definite query and P' a partial deduction of $\mathcal{A}$ in P such that $P' \cup \{Q\}$ is $\mathcal{A}$-closed. We define the $\mathcal{A}$-filtering of $P' \cup \{Q\}$ as $P'_f \cup \{Q_f\}$ where*

$$P'_f = \left\{ A'_0\theta \leftarrow A'_1\theta_1, \ldots, A'_n\theta_n \;\middle|\; \begin{array}{l} B_0\theta \leftarrow B_1, \ldots, B_n \in P' \\ B_i = A_i\theta_i \text{ with } A_i \in \mathcal{A} \\ A'_i = p_{A_i}(\overline{X}) \text{ with } \overline{X} = \mathcal{V}(A_i) \end{array} \right\}$$

and

$$Q_f = \leftarrow A'_1\theta_1, \ldots, A'_k\theta_k$$

if $Q = \leftarrow B_1, \ldots, B_k$ *and* $B_i = A_i\theta_i$ *with* $A_i \in \mathcal{A}$ *and* $A'_i = p_{A_i}(\overline{X})$ *with* $\overline{X} = \mathcal{V}(A_i)$. *In this definition, for any atom* A, p_A *denotes a fresh predicate symbol uniquely determined by* A.

Structure filtering is considered in detail in (Gallagher and Bruynooghe 1990; Gallagher and Bruynooghe 1991; Benkerimi and Hill 1993; Proietti and Pettorossi 1993; Leuschel and Sørensen 1996) and has been applied in a lot of practical approaches, for example (Leuschel and De Schreye 1995; Leuschel and De Schreye 1998; Leuschel, Martens, and De Schreye 1998). Adapted correctness results can be found in (Benkerimi and Hill 1993). We return to the issue of structure filtering in the context of specialisation of meta interpreters in Chapter 7.

In (Lloyd and Shepherdson 1991), the above definitions and correctness results are stated in the broader setting of *normal* logic programs – programs that contain negation. The only part of this thesis in which we consider programs with negation is the work on Mercury in which the semantics of negation differ somewhat from a normal logic program setting (the basic difference being that a negated goal in Mercury may not bind variables that are used outside the negation). Hence, we thought it appropriate to keep the basic definitions as simple as possible and restrict our attention to definite programs.

The above presentation follows (Leuschel and Bruynooghe 2001), deviating slightly from the original presentation in (Lloyd and Shepherdson 1991), where a partial deduction is defined as the program obtained by "replacing the set of clauses in P whose head contains one of the predicate symbols occurring in $\mathcal{A}$" with a partial deduction of $\mathcal{A}$ in P. In other words, the original predicate definitions are kept of those predicates that do not occur in $\mathcal{A}$. (Leuschel and Bruynooghe 2001) notes that keeping the original definitions for those predicates prevents a practical system from achieving dead code elimination, and makes the framework more burdensome than necessary. We refer to (Leuschel and Bruynooghe 2001) for more details. In recent work, (Leuschel, De Schreye, and de Waal 1996; De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999), the framework of (Lloyd and Shepherdson 1991) has been extended to so-called *conjunctive partial deduction*. Instead of partially deducing a set of *atoms*, the partial deduction of a set of *conjunctions* is defined. Extending the definitions from the framework of (Lloyd and Shepherdson 1991) to deal with conjunctions allows to achieve tupling and deforestation like optimisations – unachievable by standard partial deduction – but raises a number of issues in converting the constructed resultants (their head possibly being a conjunction of atoms) back to Horn clauses. The formal framework of conjunctive partial deduction is presented in (Leuschel, De Schreye, and de Waal 1996; De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999). Issues regarding the practical aspects of this new partial deduction framework are discussed in (Glück, Jørgensen, Martens, and Sørensen 1996).

The framework of Lloyd and Shepherdson (1991) is generalised by Alpuente, Falaschi, and Vidal (1998) in the context of functional logic programs. Functional logic languages (see (Hanus 1994; Hanus 1997) for an overview) combine the expressivity from logic programming languages – for example the availability of logic variables, partial data structures and search – with lazy and efficient functional computations. The operational semantics of such languages is based on narrowing. In (Alpuente, Falaschi, and Vidal 1998), the basic notions from Lloyd and Shepherdson (1991) are generalised to deal with functional computations and a generic algorithm is given for the specialisation of functional logic programs. The algorithm is parametric with respect to the narrowing strategy that is used for the automatic construction of finite narrowing trees. The propagation of partial information is treated in a natural way by the logic dimension of narrowing – logic variables and unification – while the functional dimension allows for efficient evaluation strategies (Alpuente, Falaschi, and Vidal 1998).

2.2.2 Controlling Partial Deduction in a Nutshell

The above definitions define what a partial deduction is and under what conditions a partial deduction is correct, in the sense that the transformation is sound and complete. The framework does *not* define how to construct a suitable partial deduction. Building a "good" partial deduction, either in a completely automatic way or not, has been the subject of a vast amount of research. References will be given later throughout the text.

In what follows, we discuss a basic algorithm to build a non trivial partial deduction and define the notion of local and global control, by which the algorithm is parametrised. A detailed discussion or comparison of existing concrete mechanisms for controlling the partial deduction process is beyond the scope of this thesis. We will, however, discuss some of the main characteristics and refer to the wealth of existing literature on the subject for more details.

The following control issues rise when building a partial deduction of a set of atoms $\mathcal{A}$ with respect to a definite program P:

- Given an atom $A \in \mathcal{A}$, one needs to construct a *finite* non trivial partial SLD-tree for $P \cup \{\leftarrow A\}$. The shape of the constructed tree determines the structure of a residual predicate since the latter's clauses are constructed from the tree's resultants.

- In order to guarantee that the closedness condition holds on $\mathcal{A}$, one must be careful to choose the elements of $\mathcal{A}$; that is, one must construct a *finite* set $\mathcal{A}$ such that it is closed with respect to the initial query Q.

The former control issue, building a finite partial SLD-tree for an atom is often referred to by the notion *local control*, whereas the latter control issue, choosing

the set $\mathcal{A}$, by the notion *global control*. This terminology is due to (Gallagher 1993; Martens and Gallagher 1995).

Local Control

In what follows we will often refer to the process of building a finite partial SLD-tree for an atom by "specialisation" of the atom, reserving the notion of "unfolding" for the operation in which a selected atom is replaced by its resolvents during construction of an SLD-tree. In order to be consistent with the majority of the partial deduction literature, we do however employ the notion of an *unfolding rule* to denote a particular strategy by which an atom is specialised.

Definition 2.15 *An* unfolding rule *U is a function which, given a program P and atom A returns a finite partial SLD-tree for $P \cup \{\leftarrow A\}$.*

The basic requirements for an acceptable unfolding rule are: specialising an atom should terminate and it should avoid search space explosion as well as work duplication. The fact that an unfolding rule constructs a *finite* SLD-tree for any atom, is often referred to by the notion of *local termination*. The most simple approaches towards local control have been based on depth-bounds (terminating the unfolding process after a fixed number of derivation steps) (Venken 1984; Prestwich 1993) and subsumption checking (Takeuchi and Furukawa 1986; Fuller and Abramsky 1988; Levi and Sardu 1988; Benkerimi and Lloyd 1990; van Harmelen 1989; Leuschel 1995b; Leuschel and De Schreye 1998). Other unfolding rules are based on determinacy (Gallagher and Bruynooghe 1991; Gallagher 1993): only (except once) select atoms that match a single clause head. The strategy can be refined with a so-called "look-ahead" to detect failure at a deeper level. Determinate unfolding has been proposed as a way to ensure that partial deduction will never duplicate computations in the residual program. Methods solely based on this heuristic tend not to worsen a program, but are often somewhat conservative. Subsumption checking and determinate unfolding are in itself not enough to guarantee termination (Bruynooghe, De Schreye, and Martens 1992), hence they are often combined with a depth-bound to ensure termination.

More systematic solutions to the local termination problem are based upon well-founded orders or well-quasi orders.

Definition 2.16 *A strict partial order relation $>_S$ on a set S is an anti-reflexive, anti-symmetric and transitive binary relation on $S \times S$. We call $>_S$ a* well-founded order *if and only if there is no infinite sequence of elements $e_1, e_2, \ldots$ in S such that $e_i > e_{i+1}$ for all $i \geq 1$.*

Well-founded orders have been used successfully to ensure local termination of partial deduction (Bruynooghe, De Schreye, and Martens 1992; Martens, De Schreye, and Horváth 1994; Martens and De Schreye 1996), inspired by their usefulness in

the context of static termination analysis (Dershowitz and Manna 1979; De Schreye and Decorte 1994). The general idea is to construct a *size* function, that maps a node in an SLD-tree to an element of a set S on which there exists a well-founded order relation $>_S$. An often used example of such a well-founded order relation is $>$ on $\mathbb{N}$. When constructing an SLD-tree, one guarantees for each branch in the tree that the *size* of a node is strictly smaller (according to the $>_S$ relation) than the size of its immediate ancestor (and consequently smaller than the sizes of all its ancestors). Since no such infinite sequence of strictly smaller sizes exists and any SLD-tree is finitely branching, the resulting SLD-tree (sometimes called a *well-founded* SLD-tree) is finite.

Different refinements of this basic scheme exist. One such refinement, introduced in (Bruynooghe, De Schreye, and Martens 1992), is to restrict the sequence of nodes on a branch that must be kept well-founded to the so-called *covering ancestors*, in which case the size of an atom is only compared with the size of the closest ancestor from which it descends via resolution. Another refinement (Bruynooghe, De Schreye, and Martens 1992; Martens, De Schreye, and Horváth 1994; Martens and De Schreye 1996; Martens 1994) is to construct the SLD-tree using a size function that is *dynamically* refined during the process: if at some point during the construction of the SLD-tree the tree no longer is well-founded, one switches to another (more refined) size function under which the constructed tree still is well-founded, and continues the construction using the latter size function; a process that can be repeated until no more refined size function is found that keeps the tree well-founded. Constructing a (in itself terminating) sequence of such refined size functions (automatically) is a nontrivial task. See (Bruynooghe, De Schreye, and Martens 1992; Martens, De Schreye, and Horváth 1994; Martens and De Schreye 1996; Martens 1994; Leuschel and Bruynooghe 2001) for a more detailed discussion.

A drawback of using well-founded orders is that they require a strict decrease between the subsequent compared atoms rendering the technique practically unusable for the partial deduction of datalog programs (Leuschel and Bruynooghe 2001). Another approach is based on *well-quasi* orders.

Definition 2.17 *A quasi order relation $\leq_S$ on a set S is a reflexive and transitive binary relation on $S \times S$. We call $\leq_S$ a* well-quasi order *if and only if for any infinite sequence of elements $e_1, e_2, \ldots$ in S there are $i < j$ such that $e_i \leq_S e_j$.*

Again, the well-quasi order relation can be used to compare the (covering) ancestors in a branch of an SLD-tree under construction. Local termination is ensured if we do not allow a branch to be constructed in which $A \leq_S B$ for A and B nodes on the branch and A a (covering) ancestor of B. An often used well-quasi order relation is the *homeomorphic embedding*. The following definition is due (Sørensen and Glück 1995) where it was used to control the process of supercompilation in a functional programming setting, and was refined in (Leuschel and Martens 1996) to allow a more refined treatment of variables in a logic programming setting.

Definition 2.18 *Let $X, Y \in \mathcal{V}$, $f \in \Sigma$, $p \in \Pi$. We define $\trianglelefteq$ on terms and atoms as follows:*

$$
\begin{array}{lcl}
X \trianglelefteq Y & & \\
s \trianglelefteq f(t_1, \ldots, t_n) & \Leftarrow & s \trianglelefteq t_i \text{ for some } i \\
f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n) & \Leftarrow & s_i \trianglelefteq t_i \text{ for all } i \\
p(s_1, \ldots, s_n) \trianglelefteq p(t_1, \ldots, t_n) & \Leftarrow & s_i \trianglelefteq t_i \text{ for all } i \\
 & & \text{and } p(t_1, \ldots, t_n) \not\lessapprox p(s_1, \ldots, s_n)
\end{array}
$$

We say that $A \triangleleft B$ if and only if $A \trianglelefteq B$ and $A \not\approx B$.

The intuition behind $A \trianglelefteq B$ is that A can be obtained from B by striking out some of B's structure. For example, we have that $p(a) \trianglelefteq p(f(a))$. Due to the "strictly more general" constraint between atoms (the refinement of (Leuschel and Martens 1996)), we have that $p(X, X) \not\trianglelefteq p(X, Y)$ whereas $p(X, Y) \trianglelefteq p(X, X)$.

Well-quasi orders have been extensively used in one form or another to ensure local termination of partial deduction (Bol 1993; Sahlin 1993; Sørensen and Glück 1995; Leuschel 1999; Glück, Jørgensen, Martens, and Sørensen 1996; Alpuente, Falschi, Julián, and Vidal 1997; Lafave and Gallagher 1998; Albert, Alpuente, Falaschi, and Julian 1998; De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999). A disadvantage of using well-quasi orders over well-founded orders is that one has to compare a node with all its (covering) ancestors, whereas in the well-founded approach it suffices to compare a node with its immediate (covering) ancestor only. On the other hand, the well-quasi approach does not require a strict decrease on subsequent atoms on a branch of the SLD-tree, it only requires a "lack of growth" between them. Thanks to it flexibility, it can permit the full unfolding of most terminating Datalog programs, and even the quicksort and mergesort programs when the list to be sorted is known (Leuschel and Bruynooghe 2001; Leuschel 1999). We return to the use of the homeomorphic embedding relation as a means to solve the local control problem in the context of specialisation of the Vanilla meta interpreter in Chapter 7.

A detailed comparison between the different approaches towards solving the local control problem is beyond the scope of this thesis. Some comparison between the widely used well-founded and well-quasi orders to guarantee local termination can be found in (Leuschel 1998a). All the approaches we mentioned implement a so-called *on-line* control strategy, since the decisions whether or not to unfold a particular atom are taken during the specialisation process, that is during the construction of an SLD-tree. The other option, *off-line* control, in which the control decisions are taken beforehand by a separate (binding-time) analysis and communicated to the partial deducer by means of annotations on the source program, has received far less attention in the logic programming community, notable exceptions being (Mogensen and Bondorf 1993; Gurr 1994a; Gurr 1994b; Bruynooghe, Leuschel, and Sagonas 1998). We return in detail to the issue of solving the local control problem in an off-line setting in Chapters 3 and 4 where we develop a

binding-time analysis for the logic programming language Mercury, and in Chapter 5 where we reconsider binding-time analysis for pure, definite logic programs.

Global Control

We now return to the second control issue: choosing the set of atoms $\mathcal{A}$ that are specialised. Recall that the principal aim of partial deduction is to specialise a program with respect to some partial input, represented by the partial deduction query Q', being a generalisation of the runtime queries we are interested in. The closedness condition requires that every atom occurring in such a runtime query Q and in the bodies of the residual program's clauses is an instance of an atom occurring in $\mathcal{A}$. This can be obtained as follows: starting from an initial set $\mathcal{A}$ containing the atoms of the partial deduction query Q', one builds the partial SLD-trees for these atoms. Since the atoms in the leafs of these trees become atoms in the residual clauses, closedness is guaranteed if these atoms are added to the set $\mathcal{A}$, at least those atoms that are not instances of atoms that are already present in $\mathcal{A}$ (Benkerimi and Hill 1993). New SLD-trees are constructed for these atoms, and again the atoms occurring in their leafs must be added to the set. The task of the global control component is to guarantee that the constructed set $\mathcal{A}$ is a finite set. In other words, it must guarantee that the process of adding new atoms to $\mathcal{A}$, building an SLD-tree for them and in turn adding the atoms of the constructed tree's leaves to $\mathcal{A}$ terminates. To ensure finiteness of the resulting set, one generally abstracts the set of atoms into more general atoms such that more atoms become covered by the set.

Definition 2.19 *Let $\mathcal{A}$ and $\mathcal{A}'$ be sets of atoms. We say that $\mathcal{A}'$ is an* abstraction *of $\mathcal{A}$ if and only if every atom in $\mathcal{A}$ is an instance of an atom in $\mathcal{A}'$. An* abstraction operator *is an operator which maps every finite set of atoms to a finite abstraction of it.*

Proper use of an abstraction operator can ensure global termination, but comes at the cost of possibly loosing specialisation. Indeed, if an atom $A \in \mathcal{A}$ is generalised into a more general atom A' ($A' \leq A$), the information that is "lost" by the generalisation (in the form of the substitution θ if $A'\theta = A$) can no longer be exploited when building an SLD-tree for the more general atom A'. A lot of the most commonly used abstraction operators are based on the concept of the *most specific generalisation*.

Definition 2.20 *The* most specific generalisation *of a finite set of expressions S, denoted by $msg(S)$, is an expression M such that*

- *$\forall E \in S : M \leq E$, and*
- *if M' is an expression such that also $M' \leq E (\forall E \in S)$, then $M' \leq M$.*

Algorithms to compute the msg can be found in (Lassez, Maher, and Marriott 1988; Sørensen and Glück 1995). In order to guarantee global termination using the most specific generalisation, one still has to be careful how (or better: on what atoms) to apply the msg operation. An early approach (Benkerimi and Lloyd 1990) generalised only those atoms in $\mathcal{A}$ that have a common instance, but this technique was shown (Martens 1994) not to ensure termination. Another, rather straightforward way of ensuring termination consists in imposing an artificial depth-bound on the number of atoms in the set $\mathcal{A}$, and use the msg operation to ensure that the number of atoms in $\mathcal{A}$ remains below the depth-bound. One could, for example, keep only a single atom per predicate symbol in the set $\mathcal{A}$ (Martens 1994), as such restricting the residual program to have only a single specialised version of each predicate, resulting in a so-called *monovariant* specialisation. In general, however, better results can be obtained by a *polyvariant* specialisation, allowing the creation of several specialised versions of a single predicate. Since such a specialised predicate is derived from a constructed partial SLD-tree, polyvariant specialisation involves keeping a number of atoms in $\mathcal{A}$ having the same predicate symbol. Finding the right amount of polyvariance (i.e. the "right" number of specialised versions of a single predicate) is sometimes referred to as the *control of polyvariance* problem (Leuschel and Martens 1996).

A systematic approach to the control of polyvariance problem was first presented in (Martens and Gallagher 1995). The idea is to structure the atoms (from $\mathcal{A}$) in a tree structure, the so-called *global tree*. The tree structure allows to relate the atoms with each other: if an atom A descends from an atom B in this tree, the atom A was encountered (and specialised) as a result of the specialisation of the atom B. The tree structure allows to apply the msg operation in a more structured way, avoiding the generalisation of atoms that are "unrelated", in general leading to a better control of polyvariance (Martens and Gallagher 1995). Let us present a generic partial deduction algorithm found in (Leuschel and Bruynooghe 2001) that is based upon (Martens and Gallagher 1995; Leuschel 1999). The algorithm builds a global tree γ, in which the nodes are labelled with an atom. Note that although the tree structure provides a means to explicitly state a relation between the atoms in the tree, it does not define which atoms must be generalised in order to keep the tree finite. To keep the algorithm as general as possible, it is parametrised with respect to the following operations: an unfolding rule U, a predicate $covered(L, \gamma)$, a whistle function $whistle(L, \gamma)$ and an abstraction function $abstract(L, W, \gamma)$ which we will explain in more detail shortly. Termination of the algorithm is guaranteed if a suitable instantiation of these operations is provided.

Algorithm 2.1 is parametrised with respect to an unfolding rule U, that constructs for a program P and an atom A a finite, nontrivial partial SLD-tree $U(P, A)$. The task of the predicate $covered(L, \gamma)$ is to check whether the presence of the node L is required in γ in order to satisfy the closedness condition. If

Algorithm 2.1
Input: *A definite program P and a partial deduction query Q*
Output: *a set of atoms $\mathcal{A}$ and a global tree γ*
Initialise: $\gamma :=$ *a global tree, consisting of an unlabelled root and an unmarked node labelled A, for every atom A in Q*
  repeat
    pick *an unmarked leaf node L in γ*
    if *covered*(L, γ) **then** *mark L as processed*
    else
      $W =$ *whistle*(L, γ)
      if $W \neq fail$ **then** $label(L) :=$ *abstract*(L, W, γ)
      else
        mark L as processed
        for all *atoms* $A \in$ *leaves*$(U(P, label(L)))$ **do**
          add a new unmarked child node C of L to γ
          $label(C) := A$
  until *all nodes are processed*
output: $\mathcal{A} := \{label(A) \mid A \in \gamma\}$ *and* γ

this is not the case, L itself must not be processed any further and no SLD-tree must be built for the atom labelling L. The task of the *whistle* function is to detect whether further processing the node L might eventually lead to an infinite branch; the terminology of a "whistle" function is due to (Sørensen and Glück 1995). The function is defined such that if the node L has an ancestor W in γ and L is not admissible as its descendant, then *whistle*(L, γ) returns that atom W, otherwise it returns $fail$. If L has such an ancestor, the algorithm replaces the label of L by the abstraction of W and L as computed by *abstract*(L, W, γ). The abstraction can be W itself or a more general atom and the processing of L has to start over. If L has no such ancestor, a partial SLD-tree is created for the atom labelling L, and the atoms in the leaves of the freshly constructed tree are added to γ as children of the node L. The implementations of the *whistle* and *abstract* functions constitute effectively the global control component of a partial deduction system, their task being to keep the global tree finite, and hence the partial deduction process terminating. This kind of termination is often referred to by *global termination*.

In order to derive a practical algorithm for partial deduction, the above mentioned functions and operations need to be instantiated with concrete operations. The traditional implementation of the *covered*(L, γ) predicate is an "instance of" test: it checks whether the global tree γ contains a node M such that M was processed or abstracted and $label(M)\theta = label(L)$ for some substitution θ. In other words, *covered*(L, γ) checks whether there exists a generalisation of the atom in

the tree that is already processed or abstracted. If that is the case, the closedness condition is satisfied and the node L is not required in γ. The "instance of" test can be replaced by a stronger test on variance, such that *covered*(L, γ) returns true only when there is a variant of L in γ. The implementation of the *whistle* function must be such that no infinite branch can be constructed without the *whistle* blowing on at least one of the nodes in the branch. The same approaches as in the case of the local control component can be exploited towards implementing the *whistle* function. Two well-known such approaches are either to keep the global tree *well-founded* (Martens and Gallagher 1995), or to keep its branches *well-quasi* ordered (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998; Leuschel 1998a). When comparing the atoms in the tree, sophisticated techniques (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998; Leuschel and De Schreye 1998) not only compare the syntactic structure of the atoms (using, for example, the homeomorphic embedding relation), but also compare the "computational behaviour" of the atoms. To that extent, they employ the notion of a *characteristic tree*. Informally, a characteristic tree is an abstraction of an SLD-tree that registers which atoms have been selected for unfolding and which clauses were used for resolution when building the SLD-tree. Characteristic trees were initially presented in (Gallagher and Bruynooghe 1991). In (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998), a well-quasi order relation is defined that takes a combination of an atom and its associated characteristic tree into account and shows that this combined approach leads to more precision than either approach alone.

The other essential component of a global control system is the *abstract* function. Most instantiations of *abstract* are based on the *msg* operator. However, a number of refinements exists in order to obtain a more precise abstraction or, put otherwise, to loose less information by the abstraction operation. Examples are *ecological partial deduction* (Leuschel 1995a; Leuschel, Martens, and De Schreye 1998) and *constrained partial deduction* (Leuschel and De Schreye 1998; Lafave and Gallagher 1998). The intuitive idea behind ecological partial deduction is not only to abstract the syntactic structure of the atoms, but also their associated characteristic trees. The resulting characteristic tree – representing the computational behaviour that is common to the abstracted atoms – is then *imposed* on the abstraction in the sense that when this atom is specialised, the branches that are not represented by the imposed characteristic tree are simply pruned away. The net result is in some cases a residual predicate that is still – to some extent – specialised with respect to the calls that occur to it in the sense that its implementation does not cover all syntactically possible calls that would be covered in the original program. The basic idea of constrained partial deduction is to produce a partial deduction for a set of *constrained atoms* which are atoms together with constraints that limit the atom's set of possible instances and which can be used to prune the computation (Leuschel and De Schreye 1998). See (Leuschel 1995a;

Leuschel, Martens, and De Schreye 1998; Leuschel 1997; Leuschel and De Schreye 1998) for details and examples.

Beyond Local and Global Control

The generic partial deduction algorithm sketched in Algorithm 2.1 ensures that the set of partially deduced atoms $\mathcal{A}$ satisfies the closedness condition. In order to be correct (according to Theorem 2.4), the set $\mathcal{A}$ must also be *independent*, meaning that no two atoms from the set should have a common instance. One of the originally proposed solutions to the global control problem (Benkerimi and Lloyd 1990) ensures independence by abstracting precisely those atoms that have a common instance. The drawback of ensuring independence in this manner is of course the precision loss due to the abstraction. Independence of the set $\mathcal{A}$ can be guaranteed without precision loss by renaming the atoms in $\mathcal{A}$ into a set $\mathcal{A}'$ in which every atom has a unique predicate symbol. Next, the renaming transformation has to map the atoms occurring in the residual program P' and query Q to the correct versions in $\mathcal{A}'$. Renaming can often be combined with structure filtering (see Section 2.2.1).

In the above discussion, a clear distinction is made between control at the local and the global level. This is the approach taken in most of the recent work on partial deduction since the distinction was first made in (Gallagher 1993; Martens and Gallagher 1995). Some approaches, however, incorporate information from the global level into the local control mechanism. An example of such an interaction is presented in (Glück, Jørgensen, Martens, and Sørensen 1996; De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999) in which a number of concrete strategies to control *conjunctive* partial deduction is presented. One of the strategies involves stopping the unfolding of a conjunction (local level) in case a variant of the conjunction is already present at the global level. We return to the interaction between local and global level in Chapter 7.

Although a strong relation exists between partial deduction of logic programs and supercompilation of functional programs (Glück and Sørensen 1994), recent work on supercompilation (Sørensen and Glück 1995; Sørensen, Glück, and Jones 1996) does not distinguish between the two control levels but rather constructs a single global tree. Leafs of the global tree are labelled with expressions that are folded back to other nodes, higher up in the tree. Together with the root node of the tree, the destination nodes of such a folding operation become the heads of the residual functions, much in the same way as the atoms in $\mathcal{A}$ – being the roots of the local trees – become the heads of the residual predicates. A difference, however, is that in partial deduction the residual predicates are specialised versions of the original predicates, which is not necessarily the case in supercompilation. The distinction between local and global control is again made in the framework for narrowing-based partial evaluation of functional logic programs (Alpuente, Falaschi, and Vidal 1998). This framework generalises the partial de-

duction framework from (Martens and Gallagher 1995), but does not organises the global level into a tree-like structure. Strategies are given that guarantee local as well as global termination using well-quasi order relations. The framework is extended to deal with residuating functional logic programs in (Albert, Alpuente, Hanus, and Vidal 1999). A novel approach for the specialisation of functional logic languages is presented in (Albert, Hanus, and Vidal 2000). It considers a maximally simplified representation into which programs can be automatically translated. The framework is evaluated in practice for the functional logic language Curry (Albert, Hanus, and Vidal 2001). It combines a straightforward local control strategy that allows only to unfold a single function call – similar to the strategy of Sørensen and Glück (1995) in their supercompiler – with a well-founded based global control strategy. The resulting partial evaluator is shown to obtain good specialisation results (Albert, Hanus, and Vidal 2001).

To the best of our knowledge, no thorough comparison has been made between an approach distinguishing between a local and a global control level and an approach using a single (global) control level only. Also the relation that exists between the local control level and the global control level can be rather subtle and is largely unexplored. In some cases, a rather conservative abstraction strategy at the global level can spoil the result of a sophisticated unfolding strategy, whereas a conservative unfolding strategy can in turn be compensated for by a global control strategy that is very careful in its decision whether or not to generalise. A rather extreme example is conjunctive partial deduction (Leuschel, De Schreye, and de Waal 1996; Glück, Jørgensen, Martens, and Sørensen 1996; De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999). Since *conjunctions* are brought to the global level, rather than their individual atoms, the variable bindings that exist between the atoms of the conjunction are not lost, as is the case with standard partial deduction. Consequently, conjunctive partial deduction is capable of performing tupling- and deforestation-like optimisations (De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999), and also diminishes – to some extent – the need for more sophisticated unfolding rules (Leuschel, De Schreye, and de Waal 1996). Albert, Alpuente, Falaschi, and Julian (1998) present some control improvements in narrowing-based partial evaluation of functional logic programs that allow to obtain more accurate specialisation when complex expressions are involved. It defines an abstraction operator exploiting similar techniques as the partitioning techniques introduced in the context of conjunctive partial deduction (De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999).

Part I

Off-line Specialisation of Logic Programs

Chapter 3

Binding-time Analysis for Mercury

The story so far: In the beginning
the Universe was created. This
has made a lot of people very angry
and has been widely regarded as a bad move.

— *Douglas Adams.*

In this chapter, we present a binding-time analysis for a first-order subset of the logic programming language Mercury. We first discuss the origin, syntax and semantics of this language, and later on present the binding-time analysis as an application of abstract interpretation. The chapter concludes with a correctness proof, proving the analysis correct with respect to a particular specialiser.

3.1 Mercury

In October 1993, researchers at the university of Melbourne started the design and implementation of a new logic programming language, named Mercury. While logic programming languages had been around for quite some time, none seemed to fully realise the theoretical advantages such a language would have over more traditional, imperative languages. These advantages are widely known, and are summarised for example in (Somogyi, Henderson, and Conway 1995): a higher level of expressivity (enabling the programmer to concentrate on *what* has to be

done rather than on *how* to do it), the availability of a useful formal semantics (required for the – relatively – straightforward design of analysis and transformation tools), a semantics that is independent of any order of evaluation (useful for parallelising the code), and a potential for declarative debugging (Lloyd 1987a). While a language like Prolog does offer some of these advantages, others are destroyed by the impure features of the language.

The main objective of the Mercury designers was to create a logic programming language that would be *pure* and useful for the implementation of a large number of *real-world* applications. To achieve this goal, the main design objectives of Mercury can be summarised as follows (Somogyi, Henderson, and Conway 1995): *Support for the creation of reliable programs.* This involves a language that allows to detect some classes of bugs at compile-time. *Support for programming in teams.* Large software systems are usually build by a number of programmers. The language must provide good support for creating a single application from multiple parts that are build (sometimes in isolation) by different programmers. These two objectives form a major departure from Prolog which, at the time, had basically no support for programming in the large, and which does not allow a lot of type-, mode- and determinism errors to be caught at compile-time. Another important objective was *support for the creation of efficient programs.* The efficiency of the language implementation had to be at least comparative with (but preferably better than) comparable languages. The remaining design objectives mentioned in (Somogyi, Henderson, and Conway 1995) are *support for program maintenance* and *support for external databases.*

To meet these design objectives, Mercury was fitted with a strong system of type-, mode- and determinism declarations. Apart from providing excellent comments on how the data used in a predicate should look and how the code is supposed to be used, these declarations enable the compiler to perform a number of analyses and to spot a substantial number of bugs at compile time, rather than producing a program that shows some unexpected behaviour at run-time as is often the case with Prolog. Also, the availability of declarations provides the basis for an efficient execution mechanism of the language (Conway, Henderson, and Somogyi 1995; Somogyi, Henderson, and Conway 1994; Somogyi, Henderson, and Conway 1996). Mercury is equipped with a modern module system that enables to hide some data definitions and to encapsulate both data and code, and provides as such support for programming-in-the-large activities.

In the remainder of this section, we take a closer look at the Mercury declarations, as they provide a necessary basis for the concepts we develop in the following sections. We cover Mercury's module system in a following chapter, where we redevelop our analysis in such a way that enables to deal with modules.

3.1.1 Mercury's Type System

Mercury's type system is based on a polymorphic many-sorted logic, and corresponds to the Mycroft-O'Keefe type system (Mycroft and O'Keefe 1984). Basically, the types are discriminated union types and support parametric polymorphism: a type definition can be parametrised with some type variables, as the following example in Mercury syntax shows.

Example 3.1 `:- type list(T) ---> [] ; [T | list(T)].`

The above defines a polymorphic type `list(T)`: it defines values of this type to be terms that are either `[]` (the empty list) or of the form `[A|B]` where `A` is a value of type `T` and `B` is a value of type `list(T)`.

Formally, if we denote with $\Sigma_{\mathcal{T}}$ the set of type constructors and with $V_{\mathcal{T}}$ the set of type variables of a language $\mathcal{L}$, the set of *types* associated to $\mathcal{L}$ is represented by $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$; that is the set of terms that can be constructed from $\Sigma_{\mathcal{T}}$ and $V_{\mathcal{T}}$. A type containing variables is said to be *polymorphic*, otherwise it is a *monomorphic* type. A *type substitution* is a substitution from type variables to types. The application of a type substitution to a polymorphic type results in a new type, which is an *instance* of the original type.

Example 3.2 *If* `list/1` *and* `int/0` *are type constructors from* $\Sigma_{\mathcal{T}}$, `list(T)` *is a polymorphic type. If we apply the type substitution that maps the type variable* `T` *to the type* `int` *onto this type, we get the monomorphic type* `list(int)`.

The relation between a type and the values (terms) that constitute the type is made explicit by a *type definition* that consists of a number of *type rules*, one for every type constructor. Example 3.1 shows the type rule associated to the `list/1` type constructor. Formally, a type rule is defined as follows:

Definition 3.1 *The* type rule *associated to a type constructor* $h/n \in \Sigma_{\mathcal{T}}$ *is a definition of the form*

$$h(\overline{T}) \rightarrow f_1(\overline{t}_1)\ ;\ \ldots\ ;\ f_k(\overline{t}_k).$$

where $\overline{T}$ *is a sequence of* n *type variables from* $V_{\mathcal{T}}$ *and for* $1 \leq i \leq k$, $f_i/m \in \Sigma$ *with* $\overline{t}_i$ *a sequence of* m *types from* $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ *and all of the type variables occurring in the right hand side occur in the left hand side as well. The function symbols* $\{f_1, \ldots, f_k\}$ *are said to be associated with the type constructor* h. *A finite set of type rules is called a type definition.*

Given a type substitution, we define the notion of an instance of a type rule in a straightforward way. Note that types can be defined to be (mutually) recursive. We refer to a type t as *atomic* when it is not defined in terms of other types (each of function symbols associated to t's type constructor is of arity 0). In theory, every type can be defined by a type rule as above. In practice, however, it is useful to have some types builtin in the system. For Mercury, the types `int`, `float`, `char`,

`string` are builtin types that consist of, respectively, the set of integers, floating point numbers, characters and strings. Builtin types are atomic types.

Mercury requires that the types of the arguments of every predicate be declared by the programmer.

Example 3.3 *Consider a predicate* `append/3` *that is declared as follows:*

```
:- pred append(list(T), list(T), list(T)).
```

This declaration is an assertion that each argument of the `append/3` *predicate is of type* `list(T)`*.*

From these type declarations, the compiler can derive the types of every variable in the program, and infer whether the program is *type correct* (Mycroft and O'Keefe 1984; Pfenning 1992). Type correctness ensures for example that if a predicate is called, the types of the arguments of the call are instances of the declared types of the predicate. For example, if the predicate `append/3` is declared as above, every call to `append/3` in a type correct program has arguments of a type that is an instance of `list(T)`, examples being `list(int)`, `list(list(int))` but possibly also `list(T)` itself.

3.1.2 Mercury's Mode and Determinism System

In a Mercury program, a *mode* is associated to each predicate, and describes the change of instantiation of that predicate's arguments over execution of the predicate. More precisely, for each argument a mode maps an initial instantiatedness (describing that argument's state of instantiation at the time the predicate is called) onto a final instantiatedness (describing that argument's state of instantiation at the time the predicate exits). The most basic notions of instantiatedness are *input* (denoted by `in`), describing that the argument will be ground when the predicate is called and stay ground over the execution of the predicate, and *output* (denoted by `out`), describing that an argument be a free variable when the predicate is called, but bound to a ground term when the predicate exits. The Mercury language allows to declare more sophisticated modes, based on the structure of the involved types. These are required to define predicates that deal with *partially instantiated* data structures. We will return on the subject of these more involved modes later on, but for now we assume that the modes are restricted to `in` and `out`. As noted in (Somogyi, Henderson, and Conway 1995), these basic modes suffices for the vast majority of predicates in real-world programs. Also, at the time of writing, the support for using partially instantiated data structures is not complete in the available Mercury implementation.

Example 3.4 *A traditional mode declaration for the* `append/3` *predicate is as follows:*

```
:- mode append(in,in,out).
```

This mode declaration asserts that when `append/3` *is called, its first two arguments will be bound to a ground term, the third argument being a free variable that will be bound to a ground term when the predicate succeeds.*

A mode declaration in Mercury provides *design level* information as it states the intended use of a predicate (indicating what arguments are computed from what other arguments). If the predicate can be used in more than one way, it can be declared with more than one mode. In particular, for every mode in which an argument is produced (its associated mode being `out`), there is a mode for that predicate in which the argument is consumed (and hence has mode `in`). These modes are the so called *implied* modes of a predicate (Somogyi, Henderson, and Conway 1996). The *mode set* of a predicate (Somogyi, Henderson, and Conway 1996) consists of that predicate's defined and implied modes. Modes in Mercury are *prescriptive*: the compiler must guarantee that each predicate of the program is well-moded with respect to each mode declaration from its mode set. A predicate is well-moded with respect to a particular mode declaration if the following holds (Somogyi, Henderson, and Conway 1996):

- for every possible predicate call such that each argument is bound to a term that is approximated by the argument's initial instantiatedness from the mode declaration and all free variables of the call being distinct, if the predicate succeeds, then each argument will be bound to a term that is approximated by its final instantiatedness in the mode declaration (*mode declaration constraint*).
- for every predicate call, the arguments of the call are approximated by the initial instantiatedness of some mode declaration in the called predicate's mode set (*mode set constraint*).

The compiler guarantees well-modedness as follows. First, it makes a number of copies of each predicate, one for each mode declaration of its mode set, and tries to reorder each of the resulting predicate bodies such that it becomes well-moded with respect to the particular mode declaration under left to right evaluation. In fact, the compiler can handle several other evaluation orders as well, but in the remainder we assume left-to-right evaluation as it is the default and the most commonly employed evaluation order. The resulting predicates (each having a single mode declaration) are called *procedures* in Mercury terminology. The set of all such procedures is denoted by $Proc$. For a procedure $p \in Proc$, we use $\texttt{in}(p)$ and $\texttt{out}(p)$ to denote, respectively, the set of input, respectively output arguments of the procedure. Wherever appropriate, we interpret $\texttt{in}(p)$ and $\texttt{out}(p)$ as being *sequences* rather than sets.

Finally, a *determinism declaration* is associated with each single mode declaration. A determinism declaration states whether or not in this mode the predicate

can fail before producing its first solution, and if it succeeds, how many solutions it may produce. Each predicate is declared to belong to one of the following categories:

- A *deterministic* procedure (in Mercury `det`). All calls to the procedure succeed and produce precisely one solution.
- A *semi-deterministic* procedure (in Mercury `semidet`). A call to the procedure can either fail, or succeeds with precisely one solution.
- A *multisolution* procedure (in Mercury `multi`). All calls to the procedure succeed and produce at least one (but possibly more) solutions.
- A *nondeterministic* procedure (in Mercury `nondet`). A call to the procedure either fails or produces at least one (but possibly more) solutions.

Mercury allows two more determinism categories: `failure` and `erroneous`. The former indicates a procedure that cannot succeed but may fail, while the latter indicates a procedure that can neither fail nor succeed (hence it loops forever or aborts execution). These are rarely used (Somogyi, Henderson, and Conway 1996) and will not be considered further.

3.1.3 Mercury Programs for Analysis

In this chapter, we restrict attention to first-order Mercury programs on which no module structure is imposed. We will lift these restrictions in a next chapter. Even in such a restricted setting, Mercury offers a lot of expressivity, as programs can be composed of predicates, functions, using DCG notation, etc. However, if we consider only programs that are type correct and well-moded – which is natural, since the compiler should reject programs that are not (Somogyi, Henderson, and Conway 1996) – such a program can be translated into *superhomogeneous form* (Somogyi, Henderson, and Conway 1996). Translation to superhomogeneous form involves a number of analysis and transformation steps. These include translating an n-ary function definition into an $n+1$ ary predicate definition (Somogyi, Henderson, Conway, Bromage, Dowd, Jeffery, Ross, Schachte, and Taylor 1996), making the implicit arguments in DCG-predicate definitions and calls explicit, and copying and renaming predicate definitions and calls such that every predicate definition has a single mode declaration associated with it (Somogyi, Henderson, and Conway 1996). For our analysis purposes, we assume given a Mercury program in superhomogeneous form. This does not involve any loss of generality, as the transformation from a plain Mercury program into superhomogeneous form is completely defined and automated (Somogyi, Henderson, and Conway 1996). Formally, the syntax of Mercury programs in superhomogeneous form can be defined as follows. We overload the symbol Π to refer, in Mercury context, to the set of *procedure* symbols (rather than predicate symbols) underlying the language

associated to the program. As such, we consider two procedures that are derived from the same predicate as having different procedure symbols.

Definition 3.2

$$\begin{array}{lcl} \mathit{Proc} & ::= & p(\overline{X}) : -G. \\ \mathit{Goal} & := & \mathit{Atom} \mid \mathit{not}(G) \mid (G_1\ ,\ G_2) \mid (G_1\ ;\ G_2) \mid \mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3 \\ \mathit{Atom} & ::= & X := Y \mid X == Y \mid X \Rightarrow f(\overline{Y}) \mid X \Leftarrow f(\overline{Y}) \mid p(\overline{X}) \end{array}$$

where $p/n \in \Pi$ *and* $\overline{X}$ *is a sequence of* n *distinct variables of* $\mathcal{V}$*,* $f/m \in \Sigma$ *and* $\overline{Y}$ *a sequence of* m *distinct variables of* $\mathcal{V}$ *and* $G, G_1, G_2, G_3 \in \mathit{Goal}$*.*

The definition of a procedure p in superhomogeneous form consists of a single clause. The sequence of arguments in the head of the clause, denoted by $\mathcal{A}rgs(p)$, are distinct variables, explicit unifications are created for these variables in the body goal – denoted by $\mathcal{B}ody(p)$ – and complex unifications are broken down in several simpler ones. A goal is either an atom or a number of goals connected by *conjunction*, *disjunction*, *if then else* or *not*. An atom is either a unification or a procedure call. Note that, as an effect of mode analysis (Somogyi, Henderson, and Conway 1996), unifications are catagorised as follows:

- An *assignment* of the form $X := Y$. For such a unification, Y is input, whereas X is output.
- A *test* of the form $X == Y$. Both X and Y are input to the unification and of atomic type.
- A *deconstruction* of the form $X \Rightarrow f(\overline{Y})$. In this case, X is input of the unification whereas $\overline{Y}$ is a sequence of output variables.
- A *construction* of the form $X \Leftarrow f(\overline{Y})$. In this case X is output of the unification whereas $\overline{Y}$ is a sequence of input variables.

During the translation into superhomogeneous form, unifications between values of a complex data type may be transformed into a call to a newly generated procedure that (possibly recursively) performs the unification. For any goal G, we denote with $\texttt{in}(G)$ and $\texttt{out}(G)$ the set of its input, respectively output variables.

Example 3.5 *Consider the classical definition of the* `append/3` *predicate, both in normal syntax and in superhomogeneous form for the mode* `append(in,in,out)` *as depicted in Fig. 3.1.*

According to Definition 3.2, conjunctions and disjunctions are considered binary constructs. This differs from their representation inside the Melbourne compiler (Somogyi et al.), where conjunctions and disjunctions are represented in flattened form. Our syntactic definition however facilitates the conceptual handling of these constructs during analysis.

append/3	append/3 in superhomogeneous form
`append([],Y,Y).`	`append(X,Y,Z):-`
`append([E|Es],Y,[E|R]):-`	`  (X⇒[], Z:=Y ;`
`  append(Xs,Y,R).`	`   X⇒[E|Es], append(Es, Y, R), Z⇐[E|R]).`

Figure 3.1: The `append/3` predicate and `append(in,in,out)` in superhomogeneous form.

For analysis purposes, we assume that every subgoal of a procedure body is identified by a unique program point, the set of all such program points is denoted by $\mathcal{P}p$. If we are dealing with a particular procedure, we denote with η_0 the program point associated with the procedure's head atom, and with η_b the program point associated to its body goal. The set of program points identifying the subgoals of a goal G is denoted by $\mathcal{P}ps(G)$, this set including the program point identifying G itself. If the particular program point identifying a goal G in a procedure's body is important, we subscribe the goal with its program point, as in G_η or explicitly state that $\mathcal{P}p(G) = \eta$. An important use of program points is to identify those atoms in a procedure's body in which a particular variable becomes initialised or, said otherwise, those atoms of which the variable is an output variable. This information is computed by mode analysis, and we assume the availability of a function

$$\texttt{init} : \mathcal{V} \mapsto \wp(\mathcal{P}p)$$

with the intended meaning that, for a variable V used in some procedure, if $\texttt{init}(V) = \{\eta_1, \ldots, \eta_n\}$, the variable V is an output variable of the atoms identified by $\eta_1, \ldots, \eta_n$. Note that the function `init` is implicitly associated with a particular procedure, which we do not mention explicitly. When we use the function `init`, it will be clear from the context to what particular procedure it is associated.

Example 3.6 *Let us recall the definition of* `append/3` *in superhomogeneous form for the mode* `append(in,in,out)`*, with the atoms occurring in the procedure's definition explicitly identified by subscribing them with their respective program point:*

```
append(X,Y,Z)η0 :-
  (X⇒[]η1 , Z:=Yη2 ;
   X⇒[E|Es]η3 , append(Es, Y, R)η4 , Z⇐[E|R]η5 ).
```

From mode analysis, it follows that

$$\begin{array}{lll} \texttt{init}(X) = \{\eta_0\} & \texttt{init}(E) = \{\eta_3\} & \texttt{init}(R) = \{\eta_4\} \\ \texttt{init}(Y) = \{\eta_0\} & \texttt{init}(Es) = \{\eta_3\} & \texttt{init}(Z) = \{\eta_2, \eta_5\} \end{array}$$

Or, put otherwise, X and Y (being input arguments) are initialised in the procedure's head, E and Es are initialised in the deconstruction identified by η_3, R is

initialised in the recursive call whereas Z is initialised either by the assignment $Z := Y$ (η_2) or by the construction $Z \Leftarrow [E|R]$ (η_5).

Apart from identifying the atoms in which a variable becomes initialised, program points can be used to express the control flow in a procedure definition.

Definition 3.3 *Given a procedure $p(\overline{X}) \leftarrow G$. A* control flow path *in p is a sequence of program points $\langle \eta_0, \eta_1, \ldots, \eta_n \rangle$ such that η_0 is the program point associated to the head atom of p, and $\eta_1, \ldots, \eta_n$ identify a sequence of atoms in p such that for $0 < i \leq n$ it holds that there is possible control flow from the atom identified by η_{i-1} to the atom identified by η_i. We say that two program points η, η' identifying atoms in G* share a control flow path *if there exists a control flow path S in p such that $\eta \in S$ and $\eta' \in S$.*

Note that the definition of a control flow path comprises only the program points associated to *atoms* in the procedure's body and that the procedure's head atom is considered as the first atom of every control flow path.

Example 3.7 *Reconsider the definition of* `append/3` *from Example 3.6. Two control flow paths in* `append/3` *are $\langle \eta_0, \eta_1, \eta_2 \rangle$ and $\langle \eta_0, \eta_3, \eta_4, \eta_5 \rangle$. Examples of program points that share a control flow path are η_1 and η_2, η_3 and η_5 but not η_1 and η_3.*

In a well-moded procedure, a variable is initialised in a single atom on every control flow path only. However, several control flow paths can share program points. Hence, a variable that is input to a particular atom can have been initialised in several atoms, each of these sharing a different control flow path with the particular atom. For analysis purposes, we are often interested to know not only at what particular program points a variable is initialised, but rather which of these program points is "reachable" (through a control flow path) from another program point:

Definition 3.4 *The function $\texttt{reach} : \mathcal{V} \times \mathcal{P}p \mapsto \wp(\mathcal{P}p)$ is defined as*

$$\texttt{reach}(V, \eta) = \{\eta' \mid \eta' \in \texttt{init}(V) \text{ and } \eta, \eta' \text{ share a control flow path}\}$$

Example 3.8 *Reconsider the definition of* `append/3` *from Example 3.6. We have, for example, that $\texttt{reach}(E, \eta_2) = \emptyset$ since E is not initialised on any control flow path that includes η_2. Moreover, $\texttt{reach}(Z, \eta_2) = \{\eta_2\}$, $\texttt{reach}(Z, \eta_5) = \{\eta_5\}$ but $\texttt{reach}(Z, \eta_0) = \{\eta_2, \eta_5\}$ due to the fact that $\texttt{init}(Z) = \{\eta_2, \eta_5\}$ and η_0 shares a control flow path with both of them, whereas η_2, respectively η_5 only shares a control flow path with $\eta_2 \in \{\eta_2, \eta_5\}$, respectively $\eta_5 \in \{\eta_2, \eta_5\}$.*

Like `init`, `reach` is defined with respect to a particular procedure, which will be clear from the context.

3.1.4 A Semantic Function for Mercury

In this section, we formally define a semantics for Mercury. The semantics is given by a program meaning function, that takes a goal in superhomogeneous form, together with a substitution. As this substitution is assumed to map the goal's input variables to ground terms, we sometimes refer to it by *input substitution* or also by *environment*. The semantic function has the following signature:

$$\mathcal{S} : Goal \times Subst \mapsto \wp(Subst)$$

We define the meaning of a goal, denoted by $[\![G]\!]$, as a function from an (input) substitution to a set of substitutions. Hence, for a goal G and input substitution θ, $\mathcal{S}[\![G]\!]\theta$ either is the empty set $\emptyset$ denoting failure of G under θ, or a set of substitutions, where each such substitution is of the form $\theta\sigma$, that is, an update of the original substitution θ. Every such substitution denotes success of G under θ with the resulting substitution $\theta\sigma$. By representing multiple solutions of a goal by a set, the semantic function $\mathcal{S}$ abstracts the order in which solutions are found. This is consistent with the philosophy of Mercury: being a pure language, the programmer may not depend at all on the order in which solutions are found. In fact, the compiler itself already performs several transformations (e.g. reordering the branches in disjunctions) that affect the order in which solutions are computed (Somogyi, Henderson, and Conway 1996). The definition of $\mathcal{S}$ is depicted in Fig. 3.2. Unifications use and possibly update the environment. A

$$
\begin{array}{lcl}
\mathcal{S}[\![X \Rightarrow f(\overline{Y})]\!]\theta & = & \begin{cases} \{\theta\{\overline{Y}/\overline{t}\}\} & \text{if } \theta(X) = f(\overline{t}) \\ \{\} & \text{otherwise} \end{cases} \\
\mathcal{S}[\![X \Leftarrow f(\overline{Y})]\!]\theta & = & \{\theta\{X/f(t_1,\ldots,t_n)\}\} \text{ where } \forall i : t_i = \theta(Y_i) \\
\mathcal{S}[\![X == Y]\!]\theta & = & \begin{cases} \{\theta\} & \text{if } \theta(X) = \theta(Y) \\ \{\} & \text{otherwise} \end{cases} \\
\mathcal{S}[\![X := Y]\!]\theta & = & \{\theta\{X/t\}\} \text{ where } \theta(Y) = t \\
\mathcal{S}[\![p(\overline{X})]\!]\theta & = & \{\rho^o_B\theta' \mid \theta' \in \mathcal{S}[\![B\rho]\!]\rho_B\theta\} \text{ where } p(\overline{F}) \leftarrow B \in Proc \\
\mathcal{S}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\theta & = & \begin{cases} \mathcal{S}[\![G_3]\!]\theta & \text{if } \mathcal{S}[\![G_1]\!]\theta = \{\} \\ \bigcup\limits_{\theta' \in \mathcal{S}[\![G_1]\!]\theta} \mathcal{S}[\![G_2]\!]\theta' & \text{otherwise} \end{cases} \\
\mathcal{S}[\![(G_1\ ,\ G_2)]\!]\theta & = & \bigcup\limits_{\theta' \in \mathcal{S}[\![G_1]\!]\theta} \mathcal{S}[\![G_2]\!]\theta' \\
\mathcal{S}[\![(G_1\ ;\ G_2)]\!]\theta & = & \mathcal{S}[\![G_1]\!]\theta \cup \mathcal{S}[\![G_2]\!]\theta \\
\mathcal{S}[\![\mathit{not}(G)]\!]\theta & = & \begin{cases} \{\theta\} & \text{if } \mathcal{S}[\![G]\!]\theta = \{\} \\ \{\} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 3.2: Definition of $\mathcal{S}$.

deconstruction fails if the deconstructed variable is not bound to an outermost

functor f in the environment. Otherwise, it succeeds and binds the sequence of variables in the deconstruction to the terms that are to be found as arguments of f in the environment. We use $\{\overline{X}/\overline{t}\}$ as shorthand notation for $\{X_1/t_1, \ldots, X_n/t_n\}$ if $\overline{t} = \langle t_1, \ldots, t_n \rangle$. A construction unification always succeeds and creates the appropriate bindings. A test unification either succeeds or fails, but never creates any bindings, contrary to an assignment that never fails but always binds the assigned variable Y to the value of X that is found in the environment. In order to define the semantics of a call $p(\overline{X})$ to a procedure that is defined as $p(\overline{F}) \leftarrow B$, we consider the following renamings and substitutions. Let σ denote a renaming $\{\overline{F}/\overline{X}\}$ mapping the formal arguments of p to the corresponding actual arguments and let ρ be a renaming with $dom(\rho) = \mathcal{V}(B)$ mapping the variables of B to fresh variable names not seen before. Then we define the *renaming substitutions* ρ_B and ρ_B^o as follows:

$$\rho_B = \{\rho(F)/\sigma(F) \mid F \in \mathtt{in}(B)\} \text{ and } \rho_B^o = \{\sigma(F)/\rho(F) \mid F \in \mathtt{out}(B)\}.$$

The semantics of a predicate call is then defined as the semantics of the predicate's renamed body goal $B\rho$ under the substitution $\rho_B\theta$. The use of the renaming substitution guarantees that in the environment $\rho_B\theta$, the goal's renamed input variables are bound to the same terms as the call's input variables are bound to in θ. By combining the resulting substitutions with ρ_B^o, the call's original output variables become bound to the terms constructed for the renamed output arguments.

If the test-goal of an if-then-else fails, the meaning of the if-then-else is defined as the meaning of the else-goal under the original substitution θ. Indeed, in Mercury, the test-goal may not bind any variables that are used in the else-goal. If, on the other hand, the test-goal succeeds with a number of solutions (represented by the substitutions $\theta_1, \ldots, \theta_n$), the meaning of the if-then-else is defined as the union of the meanings of the then-goal under each of the substitutions $\theta_1, \ldots, \theta_n$. The meaning of a conjunction is defined as the union of the meanings of the second conjunct, for each of the substitutions representing a solution of the first conjunct. Note that if the first conjunct fails, this union will be the empty set, denoting failure of the conjunction.

The meaning of a disjunction, on the other hand, is defined as the union of the meanings of the individual disjuncts each under the original environment θ. Consequently, the disjunction only fails when both the disjuncts fail (denoted by an empty set). If one of the disjuncts fails, the meaning of the disjunction equals the meaning of the other disjunct, and if both disjuncts succeed, the set of solutions of the disjunction equals the union of the solutions of the individual disjuncts. If the negated goal in a not-goal succeeds (the number of solutions being not important), the negation fails. If, on the other hand, the negated goal fails, the negation succeeds with a single solution being the original environment. Indeed, a negated goal in Mercury is not allowed to bind variables that are used outside

the negated goal. Note that $\mathcal{S}$ essentially defines an interpreter for Mercury: in the context of a type correct and well-moded program P, a well-moded goal G and an input substitution θ for G, $\mathcal{S}[\![G]\!]\theta$ represents the set of solutions found by left-to-right evaluation of G under θ.

Recall that Mercury is a logic programming language that allows to define and use functions in addition to predicates. The basic execution mechanism of Mercury is SLD-resolution, restricted by the mode system. An n-ary function in Mercury is treated as syntactic sugar for an $n + 1$-ary predicate, and function definitions and calls are converted into predicate definitions and calls during compilation (Somogyi, Henderson, Conway, Bromage, Dowd, Jeffery, Ross, Schachte, and Taylor 1996). This approach contrasts with the approach taken in *Functional Logic Programming* (see (Hanus 1994) for an overview) in which features from functional and logic programming are integrated in a much more thorough way. The expressivity from logic programming languages (logic variables, partial data structures, search) is combined with lazy and efficient functional computations. The operational semantics of functional logic programs is based on narrowing, a generalisation of term-rewriting where matching is replaced by unification (Reddy 1985). See (Hanus 1994) for an overview of functional logic languages and their implementations.

3.2 A Domain of Binding-times

Binding-time analysis can be seen as an application of abstract interpretation over a domain of *binding-times*. A binding-time abstracts a value by specifying at what time during the computation (either early, during specialisation or late, during evaluation of the residual program) the value becomes known. In their most basic form, the binding-time of a value is either *static* or *dynamic*, denoting a value that is known during specialisation or a value that is known only during evaluation of the residual program, respectively.

It is recognised (Jones, Gomard, and Sestoft 1993) that for a logic programming language, approximating values by either *static* or *dynamic* is too coarse grained in general. Indeed, most logic programs use a lot of *structured* data, where data values are represented by structured terms. Consequently, the input to the specialiser usually consists of a partially instantiated term: a term that is less instantiated than it would be at run-time. Approximating a partially instantiated term by *dynamic* usually results in too much information loss, possibly resulting in missed specialisation opportunities. Therefore, we use the structural information from the type system to represent more detailed binding-times, capable of distinguishing between the computation stages in which *parts* of a value (according to that value's type) become known.

Let us recall the basic notions from Mercury's type system. A type is any term from $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ and for every type constructor in $\Sigma_{\mathcal{T}}$, we have a type rule that

defines the constructor. Moreover, the type rules indicate what function symbols (from Σ) are associated with the defined type constructor. As such, a type can be represented by an AND-OR tree in the usual way: a type corresponds with an OR-node, with a child for each of the function symbols associated to the type constructor's type rule. Each such child is again an AND-node with the type trees corresponding to the functor's arguments as children.

Example 3.9 *Consider the following type definition. The type trees for the types transportation and list(T) are depicted in Fig. 3.3. In this figure, function symbols are put in italic print, whereas type constructors are not.* AND*-nodes are identified by connecting the lines to their children with an arc (or a circle in case of a single child node).*

$$\begin{array}{l} \mathit{transportation} \rightarrow \mathit{land}(\mathit{land\_vehicle})\ ;\ \mathit{air}(\mathit{air\_vehicle}). \\ \mathit{land\_vehicle} \rightarrow \mathit{bike}\ ;\ \mathit{car}. \\ \mathit{air\_vehicle} \rightarrow \mathit{plane}(\mathit{ptype})\ ;\ \mathit{helicopter}. \\ \mathit{ptype} \rightarrow \mathit{prop}\ ;\ \mathit{jet}. \\ \\ \mathit{list}(T) \rightarrow []\ ;\ [T \mid \mathit{list}(T)]. \end{array}$$

Figure 3.3: Type trees for *transportation* and $list(T)$.

Note that, by definition, a recursive type corresponds to an infinite type tree. Given a type tree for a type t, its OR-nodes are of particular interest, as they represent the (types of the) possible subterms of any term of type t. Hence, in

what follows, we will refer to these OR-nodes simply as *type nodes*. A type node is uniquely identified in a type tree by the path from the root of the tree towards the type node. A path in a type tree can be described by a sequence over $\Sigma \times \mathbb{N}$. The empty sequence $\langle\rangle$ denotes the root of the type tree, whereas a sequence $\langle(f,i)\rangle$ denotes the type tree obtained by selecting the root's child node associated to the functor $f/m \in \Sigma$ and again the i-th child of this AND-node. A sequence $\langle(f_1,i_1),\ldots,(f_n,i_n)\rangle$ denotes the type node obtained by first selecting the type node identified by $\langle(f_1,i_1)\rangle$ and then, recursively in the type tree identified by this node, the type node identified by $\langle(f_2,i_2),\ldots,(f_n,i_n)\rangle$. In what follows, we denote the set of all such sequences over $\Sigma \times \mathbb{N}$ by *TPath*. Given $\delta, \epsilon \in \mathit{TPath}$, we denote with $\delta \bullet \epsilon$ the sequence obtained by concatenating ϵ to δ. Given a type t and a path δ in t's type tree, we denote with t^δ the type in $\mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$ that is represented by the type node identified by δ in the type tree for t.

Example 3.10 *Consider the type trees depicted in Fig. 3.3. If δ denotes the sequence $\langle([],1)\rangle$, γ the sequence $\langle([],2)\rangle$ and t the type list(T), then we have*

$$\begin{aligned}
&\gamma \bullet \delta = \langle([],2),([],1)\rangle \\
&t^\delta = T \\
&t^\gamma = \mathit{list}(T) \\
&t^{\gamma\bullet\delta} = T
\end{aligned}$$

We can now formally define a type tree as follows:

Definition 3.5 *Given a type $t \in \mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$, the* type tree *of t, denoted by $\mathcal{L}_t$, is a set of sequences from TPath and is recursively defined as:*

- $\langle\rangle \in \mathcal{L}_t$
- *if $t = h(\overline{T})\theta$ with $h(\overline{T}) \to f_1(\overline{t}_1);\ldots;f_k(\overline{t}_k)$ a type rule and θ a type substitution, then for all $i \in \{1\ldots k\}$, and if $f_i/m \in \Sigma$, for each $j \in \{1\ldots m\}$, $\langle(f_i,j)\rangle \bullet \delta \in \mathcal{L}_t$ where $\delta \in \mathcal{L}_{(t_{i_j})\theta}$ and t_{i_j} denotes the j-th type in $\overline{t_i}$.*

Example 3.11 *Reconsider the type definition from Example 3.9. The first-order type trees for transportation and list(T) are defined by the sets denoted by $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively.*

$$\mathcal{L}_1 = \left\{ \begin{array}{l} \langle\rangle \\ \langle(\mathit{land},1)\rangle \\ \langle(\mathit{air},1)\rangle \\ \langle(\mathit{air},1),(\mathit{plane},1)\rangle \end{array} \right\} \quad \mathcal{L}_2 = \left\{ \begin{array}{l} \langle\rangle \\ \langle([],1)\rangle \\ \langle([],2)\rangle \\ \langle([],2),([],1)\rangle \\ \langle([],2),([],2)\rangle \\ \langle([],2),([],2),([],1)\rangle \\ \langle([],2),([],2),([],2)\rangle \\ \langle([],2),([],2),([],2),([],1)\rangle \\ \ldots \end{array} \right\}$$

In Mercury, terms are typed. We will denote the fact that $\tau \in \mathcal{T}(\Sigma, \mathcal{V})$ is a term of type t by $\tau : t$ wherever appropriate. A type tree for a type t describes all possible terms of type t. Indeed, a particular term $\tau : t$ can be seen as a finite subtree of $\mathcal{L}_t$ that is obtained by pruning in each OR-node of $\mathcal{L}_t$ all but one alternatives (fixing the functor of a subterm), or pruning *all* alternatives when the subterm is a variable. As such, we can use (a finite subset of) sequences in $\mathcal{L}_t$ to denote each of a term $\tau : t$'s subterms. For a term $\tau : t$ and a sequence $\delta \in \mathcal{L}_t$, we denote with τ^δ the subterm identified by δ in τ, if it exists.

Example 3.12 *Consider a term $\tau = air(plane(X))$ of type transportation, defined in Example 3.9. If δ_1 denotes the sequence $\langle(air, 1)\rangle$ and δ_2 the sequence $\langle(air, 1), (plane, 1)\rangle$, τ^{δ_1} denotes the subterm $plane(X)$ and τ^{δ_2} the subterm X.*

Practical systems approximate an infinite type tree by a finite type *graph*. A type graph is obtained from a type tree by folding, on a branch, a recursive occurrence of a type node back to an ancestor node of the same type. Figure 3.4 depicts a finite type graph for the $list(T)$ type. Formally, type graphs can be obtained

Figure 3.4: Type graph for $list(T)$.

from type trees, by imposing an equivalence relation on the set of sequences from *TPath* for a given type. Let us therefore define the relation $\equiv$ that can be shown to be an equivalence relation.

Definition 3.6 *The relation $\equiv$ is defined in TPath as the least transitive relation such that*

$$\text{if } \delta = \alpha \bullet \epsilon \text{ and } t^\delta = t^\alpha \text{ then } \alpha \equiv \delta.$$

Informally, two type nodes in a type tree are equivalent if either one of the nodes is an ancestor of the other while both denote the same type, or the nodes have a common ancestor such that all three nodes denote the same types. We have the following:

Proposition 3.1 *The relation $\equiv$ is an equivalence relation on TPath.*

Proof Immediate from Definition 3.6.

□

Proposition 3.2 *Given a type tree $\mathcal{L}_t$ with $\alpha, \beta \in \mathcal{L}_t$. If $\alpha \equiv \beta$ then there exists $\gamma \in \mathcal{L}_t$ such that $\alpha = \gamma \bullet \epsilon_1$, $\beta = \gamma \bullet \epsilon_2$ and $\alpha \equiv \gamma \equiv \beta$.*

Proof Assume $\alpha = \beta \bullet \epsilon$, in which case proof follows if we take γ to be β, ϵ_1 to be ϵ and ϵ_2 the empty sequence $\langle\rangle$. Otherwise, due to transitivity and symmetry, there exists γ such that $\alpha \equiv \gamma$ and $\beta \equiv \gamma$. By definition 3.6, we have that $\alpha = \gamma \bullet \epsilon_1$ and $\beta = \gamma \bullet \epsilon_2$.

□

In what follows, we restrict our attention to (polymorphic) types that are not defined in terms of a strict instance of itself. That is, we assume for any type t and $\delta \in \mathcal{L}_t$ that $t \not\lessdot t^\delta$ (where ¡ denotes the strict instance relation). This is a natural condition and is related to the polymorphism discipline of definitional genericity (Lakshman and Reddy 1991). For any such type t, the equivalence relation $\equiv$ partitions the (possibly infinite set) $\mathcal{L}_t$ into a finite number of equivalence classes. For any $\delta \in \mathcal{L}_t$, the equivalence class of δ is defined as

$$[\delta] = \{\gamma \in \mathcal{L}_t \mid \delta \equiv \gamma\}.$$

The minimal element of an equivalence class $[\delta]$ exists and is defined as follows.

$$\overline{[\delta]} = \alpha \in [\delta] \text{ such that } \forall \beta \in [\delta] : \beta = \alpha \bullet \epsilon \text{ for some } \epsilon \in \mathit{TPath}$$

Next, we define, for a type t, its *type graph* as the finite set of minimal elements of the equivalence classes of $\mathcal{L}_t$:

Definition 3.7 *For a type $t \in \mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$, we denote t's type graph by $\mathcal{L}_t^{\equiv}$ which is defined as*

$$\mathcal{L}_t^{\equiv} = \{\overline{[\delta]} \mid \delta \in \mathcal{L}_t\}.$$

Example 3.13 *For the types transportation and list(T) from Example 3.9, the associated type graphs are as follows:*

$$\mathcal{L}^{\equiv}_{transportation} = \left\{ \begin{array}{l} \langle\rangle \\ \langle(land, 1)\rangle \\ \langle(air, 1)\rangle \\ \langle(air, 1), (plane, 1)\rangle \end{array} \right\} \qquad \mathcal{L}^{\equiv}_{list(T)} = \left\{ \begin{array}{l} \langle\rangle \\ \langle([], 1)\rangle \end{array} \right\}$$

A type graph $\mathcal{L}_t^{\equiv}$ provides a finite approximation of the structure of terms of type t: every node in the $\mathcal{L}_t^{\equiv}$ abstracts a number of subterms of the term. For the $list(T)$ type from above, values are abstracted by two nodes in the type graph: the root node, identified by the sequence $\langle\rangle$, represents all subterms of type $list(T)$ – in other words the skeleton of the list – whereas the node identified by the sequence $\langle([], 1)\rangle$ represents all subterms of type T, being the elements of the list. Note that due to the particular definition of $\equiv$, two subterms of a same type are not necessarily abstracted by the same node in $\mathcal{L}_t^{\equiv}$. This is the case when $\mathcal{L}_t$ contains two type nodes representing the same type without them being equivalent, as in the next example.

Example 3.14 *Consider the type* $pair(T)$ *defined as*

$$pair(T) \longrightarrow (T - T).$$

A term of the type $pair(T)$ *is a term* $(A - B)$ *where* A *and* B *are terms of type* T*. For* $t = pair(T)$*,*

$$typetree_t = \mathcal{L}_t^{\equiv} = \left\{ \begin{array}{c} \langle\rangle \\ \langle(-), 1\rangle \\ \langle(-), 2\rangle \end{array} \right\}$$

Although $\langle(-), 1\rangle$ *and* $\langle(-), 2\rangle$ *denote subterms of the same type (*T*), they are not equivalent according to Definition 3.6.*

The ability to distinguish between two occurrences of the same type in $\mathcal{L}_t^{\equiv}$ allows a more precise characterisation of terms of type t. Using Example 3.14 for example, it allows to distinguish between both elements of a pair. By associating an abstract value to each of the nodes in $\mathcal{L}_t^{\equiv}$, we obtain an abstract characterisation of terms of type t, based on the structure of the term. For binding-time analysis, we are interested in the time a (part of a) value becomes known in the computation process. We use the abstract values $\mathcal{B} = \{static, dynamic\}$ to denote, respectively, an early binding (a specialisation-time value) and a late binding (a run-time value). A binding-time associates a value from $\mathcal{B}$ to each of the nodes in a type graph.

Definition 3.8 *A* binding-time *for a type* $t \in \mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ *is a function*

$$\beta : TPath \mapsto \mathcal{B}$$

such that $\forall \delta \in dom(\beta)$ *holds that* $\beta(\delta) = dynamic$ *implies that* $\beta(\delta') = dynamic$ *for all* $\delta' \in dom(\beta)$ *with* $\delta' = \delta \bullet \epsilon$ *for some* $\epsilon \in TPath$*. The set of all binding-times is denoted by* $\mathcal{BT}$*.*

Note that a binding-time for a type t can be represented graphically by labelling the type nodes in t's type graph by *static* or *dynamic*. The relation between terms and the binding-times that approximate them is given by the following abstraction function.

Definition 3.9 *The* binding-time abstraction *is a function* $\alpha : \mathcal{T}(\Sigma, \mathcal{V}) \mapsto \mathcal{BT}$ *and is defined as follows:*

$$\alpha(\tau : t) = \left\{ (\delta, v) \;\middle|\; \begin{array}{l} \delta \in \mathcal{L}_t^{\equiv} \text{ and } v = dynamic \text{ if } \exists \text{ a subterm } \tau^{\delta'} \text{ in } \tau \text{ with} \\ \delta = [\overline{\delta'}] \bullet \epsilon \text{ for } \epsilon \in TPath(\text{possibly } \langle\rangle) \text{ and } \tau^{\delta'} \text{ a variable,} \\ v = static \text{ otherwise} \end{array} \right\}$$

If a term $\tau : t$ contains a subterm $\tau^{\delta'}$ that is a variable, then the binding-time abstraction associates the value *dynamic* to the node in $\mathcal{L}_t^{\equiv}$ that represents this subterm and to all its descendant nodes in $\mathcal{L}_t^{\equiv}$.

Example 3.15 *Given the following terms of type $list(T)$ as defined in Example 3.9, their binding-time abstraction is:*

$$\begin{array}{lcl} \alpha([]) & = & \{(\langle\rangle, static), (\langle([],1)\rangle, static)\} \\ \alpha([X_1, X_2] & = & \{(\langle\rangle, static), (\langle([],1)\rangle, dynamic)\} \\ \alpha(X) & = & \{(\langle\rangle, dynamic), (\langle([],1)\rangle, dynamic)\} \\ \alpha([X|Y]) & = & \{(\langle\rangle, dynamic), (\langle([],1)\rangle, dynamic)\} \end{array}$$

Since the term [] does not contain any variable, it is abstracted by a binding-time specifying that the list's skeleton as well as its elements are static. A term $[X_1, X_2]$ is approximated by a binding-time specifying that the list's skeleton is static, but its elements are dynamic. A variable is abstracted by a binding-time specifying that the list's skeleton as well as its elements are dynamic. Also a term $[X|Y]$ is approximated by a binding-time stating that its list skeleton as well as its elements are dynamic due to the presence of the variable subterm $Y : list(T)$.

The following example shows why we define a binding-time to associate *dynamic* with the descendants of a *dynamic* node:

Example 3.16 *Consider the type definition for a tree of integers:*

$$inttree \longrightarrow nil; t(int, inttree, inttree).$$

The type graph of $t = inttree$, $\mathcal{L}_t^{\equiv}$ contains only two nodes: $\langle\rangle$ denoting the tree's skeleton, and $\langle t, 1\rangle$ denoting the integer elements in the tree. We have

$$\alpha(t(0, X, t(1, nil, nil))) = \{(\langle\rangle, dynamic), (\langle t, 1\rangle, dynamic)\}.$$

Although all subterms of type int in the term $t(0, X, t(1, nil, nil))$ are non-variable terms, we can not abstract them to static. Indeed, the variable X in the term, being of type inttree, possibly represents some unknown integer elements.

To make our approximations suitable for a binding-time analysis, we define a partial order relation on $\mathcal{BT}$:

Definition 3.10 *Let $\beta, \beta' \in \mathcal{BT}$ such that $dom(\beta) \subseteq dom(\beta')$ or $dom(\beta') \subseteq dom(\beta)$. We say that β* covers *β', denoted by $\beta \succeq \beta'$ if and only if $\forall \delta \in dom(\beta) \cap dom(\beta')$ holds that $\beta'(\delta) = dynamic$ implies $\beta(\delta) = dynamic$.*

If a binding-time β covers another binding-time β', then β is "at least as dynamic" as β'. Note that the relationship between $dom(\beta)$ and $dom(\beta')$ implies that the *covers* relation is only defined between two binding-times that are derived from types t and t' such that either t is an instance of t' or t' is an instance of t.

Example 3.17 *Recall the binding-times obtained by abstracting the terms in Example 3.15. We have that*

$$\alpha(X) \succeq \alpha([X_1, X_2]) \succeq \alpha([])$$

In what follows, we extend the notion of the $\succeq$ relation to include the elements $\{\top, \bot\}$ such that $\top \succeq \beta$ and $\beta \succeq \bot$ for all $\beta \in \mathcal{BT}$. If we denote with $\mathcal{BT}^+$ the set $\mathcal{BT}^+ = \mathcal{BT} \cup \{\top, \bot\}$, $(\mathcal{BT}^+, \succeq)$ forms a complete lattice. Wherever appropriate, we use $\bot$ and $\top$ to denote, for a particular type, a binding-time in which all nodes are mapped to *static*, respectively a binding-time in which all nodes are mapped to *dynamic*. In what follows, we use the following notation. If β denotes a binding-time for a type t and $\delta \in \mathcal{L}_t^{\equiv}$, then β^δ denotes the binding-time for a type t^δ that is obtained as follows:

$$\beta^\delta = \left\{ \ (\gamma, \beta(\overline{[\delta \bullet \gamma]})) \ \middle| \ \gamma \in \mathcal{L}_{t^\delta}^{\equiv} \ \right\}.$$

Moreover, if $\beta_1, \ldots, \beta_n$ are binding-times for types $t_1, \ldots, t_n$ and t is a type such that $\langle (f,1) \rangle, \ldots, \langle (f,n) \rangle \in \mathcal{L}_t^{\equiv}$, we denote with $f(\beta_1, \ldots, \beta_n)$ the *least* binding-time, say β, such that $\beta^{\overline{[\langle (f,i) \rangle]}} \succeq \beta_i$ for all i.

3.3 Binding-time Analysis by Abstract Interpretation

In what follows, we devise a polyvariant specialisation process like described in Section 2.2.2 in which we implement the *local* control strategy (the specialisation of a single procedure call into a specialised predicate by reducing its body goal) in an off-line way. To accommodate such an off-line control strategy, we develop a polyvariant binding-time analysis. The final output of the analysis is an annotated version of the program in which each of the original procedures may occur in several annotated versions. Each such version represents the instructions how to reduce a call to the procedure with a call pattern that is approximated by the binding-times with respect to which the procedure was annotated. Correctness of the analysis ensures that if a particular call occurs during specialisation, a version of the called procedure – annotated with respect to the particular call's binding-time abstraction – is available.

In what follows, we present a binding-time analysis as an application of abstract interpretation in which the program is evaluated over the domain of binding-times. A following section shows how an annotated version of the program can be derived from the result of abstract interpretation and deals with the correctness of the analysis.

3.3.1 Basic Binding-time Analysis

Before defining the actual analysis, we present some necessary environments, representing the output of binding-time analysis at a particular level in the analysis.

Definition 3.11 *A* binding-time environment *is a function* $\mathcal{V} \mapsto \mathcal{BT}^+$. *The set of all binding-time environments is denoted by* $\mathcal{BT}Env$.

A binding-time environment associates a binding-time to a program variable. We use a binding-time environment during analysis to represent the result of binding-time analysis at a single program point: every variable that is relevant at that program point is mapped to a unique binding-time that approximates the variable's value during specialisation at that particular program point.

To handle procedure calls during analysis, we are interested in the relation between the binding-times of a procedure's input arguments (constituting the call pattern with respect to which we want to analyse the procedure) and the binding-times of its output arguments, reflecting their binding-time approximations at the time the procedure exits. To model this relation for a polyvariant analysis (where a procedure is analysed a number of times, each time with respect to a particular call pattern) we introduce the notion of a *procedure environment* as a mapping from a binding-time environment to another binding-time environment and a *program environment* mapping a procedure symbol to such a procedure environment.

Definition 3.12 *A* procedure environment *is a function* $\mathcal{BTE}nv \mapsto \mathcal{BTE}nv$. *A* program environment *is a function* $\Pi \mapsto (\mathcal{BTE}nv \mapsto \mathcal{BTE}nv)$. *The set of all such program environments is denoted by* $\mathcal{PE}nv$.

Before defining the actual analysis, we introduce the following notations. For any function $\phi : A \mapsto B$ and elements $a \in A$, $b \in B$, we denote the *update* of ϕ with respect to a and b as $\phi[a/b]$, which is a new function $\phi[a/b] : A \mapsto B$ defined as

$$\phi[a/b] \quad = \quad \begin{cases} \phi \cup \{(a,b)\} & \text{if } a \notin dom(\phi) \\ \phi \setminus \{(a,\phi(a))\} \cup \{(a,b)\} & \text{otherwise} \end{cases}$$

Starting from the *covers* relation, we define the following *greater than* relations on the newly introduced environments. Given two binding-time environments, say π and π', we define $\pi' \sqsupseteq \pi$ if and only if $\forall X \in dom(\pi)$: $X \in dom(\pi')$ and $\pi'(X) \succeq \pi(X)$. Likewise, for two program environments Ψ and Ψ', we define $\Psi' \sqsupseteq \Psi$ if and only if $\forall p \in dom(\Psi)$ holds that $p \in dom(\Psi')$ and $\forall(\pi_i, \pi_o) \in \Psi(p)$, there exists $(\pi_i, \pi'_o) \in \Psi'(p)$ such that $\pi'_o \sqsupseteq \pi_o$. An essential operator is the *least upper bound* of two functions: when a partial order exists on B, we define for all functions $f, g : A \mapsto B$ their least upper bound $f \sqcup g$ to be a function $A \mapsto B$ with $dom(f \sqcup g) = dom(f) \cup dom(g)$, where $f \sqcup g$ is defined as

$$\forall x \in dom(f \sqcup g) : (f \sqcup g)(x) = \begin{cases} f(x) \sqcup g(x) & \text{if } x \in dom(f) \cap dom(g) \\ f(x) & \text{if } x \in dom(f) \text{ and } x \notin dom(g) \\ g(x) & \text{if } x \in dom(g) \text{ and } x \notin dom(f) \end{cases}$$

In what follows, we present the analysis piecewise. We start by the analysis of a single atom, which we extend to the analysis of a goal, and a complete program in the next subsections. We start by the analysis of a single atom. Input is a binding-time environment that represents binding-times for the atom's input variables, and a program environment that represents the result of binding-time

analysis so far for the complete program. The result of analysing an atom is the initial binding-time environment updated with binding-times for the atom's output arguments, and a possibly updated program environment. Indeed, when analysing a procedure call, information from the program environment is used to derive the binding-times of the call's output arguments. If such a call was not yet analysed before, the program environment is updated with the new call pattern for which the procedure should be analysed. The analysis is denoted by a function

$$\mathcal{A} : Atom \times \mathcal{BT}Env \times \mathcal{P}Env \mapsto \mathcal{BT}Env \times \mathcal{P}Env.$$

which is defined in Fig. 3.5

$$\begin{array}{lcl}
\mathcal{A}[\![X \Rightarrow f(Y_1,\ldots,Y_n)]\!]\pi\Psi & = & (\pi \cup \{(Y_1, \beta^{\langle(f,1)\rangle}),\ldots,(Y_n,\beta^{\langle(f,n)\rangle})\},\Psi) \\
 & & \text{where } \beta = \pi(X) \\
\mathcal{A}[\![X == Y]\!]\pi\Psi & = & (\pi,\Psi) \\
\mathcal{A}[\![X := Y]\!]\pi\Psi & = & (\pi \cup \{(X,\pi(Y))\},\Psi) \\
\mathcal{A}[\![X \Leftarrow f(Y_1,\ldots,Y_n)]\!]\pi\Psi & = & (\pi \cup \{(X, f(\beta_1,\ldots,\beta_n))\},\Psi) \\
 & & \text{where } \beta_i = \pi(Y_i), \text{ for each } i \\
\mathcal{A}[\![p(X_1,\ldots,X_n)]\!]\pi\Psi & = & (\pi \cup \pi_{out},\Psi') \\
 & & \text{where, if } \langle F_1,\ldots,F_n\rangle = \mathcal{A}rgs(p), \\
 & & \pi_{out} = \{(X_i,\Psi'(p)(\pi_{in})(F_i)) \mid F_i \in \mathtt{out}(p)\} \\
 & & \pi_{in} = \{(F_i,\pi(X_i) \mid F_i \in \mathtt{in}(p)\} \\
 & & \text{and } \Psi' = \Psi[p/\Psi(p) \sqcup Out] \\
 & & \text{with } Out = \{(\pi_{in},\{(F_i,\bot) \mid F_i \in \mathtt{out}(p)\})\}
\end{array}$$

Figure 3.5: Definition of $\mathcal{A}$, for analysing a single atom.

Handling unification is straightforward: the binding-time environment π is updated with the binding times for the unification's output argument(s), derived from the binding-times of the unification's input argument(s). This derivation is a straightforward abstraction of the program meaning function $\mathcal{S}$ with respect to a unification; the effect being that the unification is performed using abstract values from $\mathcal{BT}^+$ instead of concrete values from $\mathcal{T}(\Sigma,\mathcal{V})$. Since no new calls are encountered during analysis of a unification, the program environment Ψ remains unchanged. When analysing a procedure call to p, the binding-times of p's output arguments are to be found in the program environment, since this environment represents the result of analysis so far. Before retrieving them, however, we update this program environment with the procedure environment $\{(\pi_{in},\{(F_i,\bot) \mid F_i \in \mathtt{out}(p)\})\}$, associating a binding-time environment in which each of the called procedure's output arguments is initialised to $\bot$ with the call pattern represented

by π_{in} (constructed from the call's input arguments). The use of $\sqcup$ in the update ensures that if p was already analysed with respect to the call pattern represented by π_{in}, the resulting exit pattern (already registered in $\Psi(p)$), is preserved and used in the construction of the binding times for the call's actual output arguments. If it was not, the call's output arguments are initialised to $\bot$. The definitions of π_{in} and π_{out} provide the mapping between the formal procedure arguments $\langle F_1, \ldots, F_n \rangle$ and the actual arguments $\langle X_1, \ldots, X_n \rangle$ of the call.

Example 3.18 *Reconsider the* `append/3` *procedure from Example 3.5. Assume that π is a binding-time environment $\pi = \{X/\beta_l\}$ where β_l is a binding-time of the form*

$$\beta_l = \{(\langle\rangle, \mathit{static}), (\langle[|], 1\rangle, \mathit{dynamic})\}$$

approximating those terms of type $list(T)$ that are bound at least to a list skeleton. Then we have for any program environment Ψ:

$$\mathcal{A}[\![X \Rightarrow [E|Es]]\!]\pi\Psi = (\pi', \Psi)$$

where

$$\pi' = \left\{ \begin{array}{l} X/\{(\langle\rangle, \mathit{static}), (\langle[|], 1\rangle, \mathit{dynamic})\} \\ E/\{(\langle\rangle, \mathit{dynamic})\} \\ Es/\{(\langle\rangle, \mathit{static}), (\langle[|], 1\rangle, \mathit{dynamic})\} \end{array} \right\}$$

Since we consider only well-moded programs, we can be sure that a variable that is input to an atom (be it a unification or a predicate call) is associated with a binding-time in π. For the same reason, we know that the atom's output variable(s) are not represented in π, so we can safely add the newly derived binding-time(s) to the binding-time environment.

3.3.2 Incorporating Specialisation Control

The binding-times produced by binding-time analysis represent a variable's state of instantiation during specialisation. Therefore, the bindings computed for an atom's output variables are only valid when the atom is actually reduced during specialisation and values for its output variables are computed by the specialiser. When the atom is residualised, on the other hand, it is not evaluated and no output at all is computed during specialisation. In the latter case, the atom's output variables need to be bound to $\top$ by binding-time analysis, correctly reflecting their instantiatedness state (which is completely *dynamic*) during specialisation.

To that extent, we introduce a control aspect in the binding-time analysis, reflecting the control strategy followed by the specialiser. Our analysis models a rather conservative specialisation strategy, in the sense that during specialisation, no atoms are reduced that are under *dynamic control*. An atom is under dynamic control, if it is not certain to be evaluated, since its evaluation depends on the success or failure of another goal, while this result is unknown at specialisation-time.

Example 3.19 *Consider the following code fragment*

```
if X ⇒ [] then p(X) else q(X)
```

If X's binding-time does not allow to decide whether or not the test $X \Rightarrow []$ will succeed during specialisation, both atoms $p(X)$ and $q(X)$ are under dynamic control. Indeed, the specialiser has no means of knowing which of the branches will be taken during the second stage of the computation.

The general idea of this control strategy is as follows: if during specialisation only atoms are reduced that are not under dynamic control, only atoms are reduced that would also be evaluated by an equivalent single stage computation. Indeed, their being evaluated depends only on goals that are – during specialisation– sufficiently reduced in order to decide success or failure. Hence, no atoms are "speculatively" reduced, guaranteeing termination of the reduction process (constituting local termination) under the assumption that the equivalent single stage computation terminates. To model such behaviour, we introduce the auxiliary function

$$\mathcal{C} : \mathit{Goal} \times \mathcal{PP} \times \mathcal{BTE}nv \times \mathcal{P}Env \mapsto \{\top, \bot\}.$$

The intended meaning of $\mathcal{C}[\![G]\!]\eta\pi\Psi = \top$ is that the atom in G identified by the program point η is under dynamic control when G is reduced with respect to input that is approximated by the binding-time environment π. If the atom is not under dynamic control, $\mathcal{C}[\![G]\!]\eta\pi\Psi = \bot$. Note that the notion of dynamic control is *relative* with respect to the surrounding goal. An atom identified by η might for example be under dynamic control with respect to a goal G, $\mathcal{C}[\![G]\!]\eta\pi\Psi = \top$ whereas the same atom might not be under dynamic control at all in a subgoal G' of G, that is $\mathcal{C}[\![G']\!]\eta\pi\Psi = \bot$. We will define $\mathcal{C}$ later on, but first use $\mathcal{C}$ to define the analysis function that analyses non-atomic goals. Basically, performing binding-time analysis of a non-atomic goal consists of performing an analysis of the individual subgoals, and combining their computed binding-times in a suitable way. Hence, we introduce the analysis function

$$\mathcal{G} : \mathit{Goal} \times \mathcal{BTE}nv \times \mathcal{P}Env \times \mathit{Proc} \times \mathcal{BTE}nv \mapsto \mathcal{BTE}nv \times \mathcal{P}Env$$

which is defined in Fig. 3.6. Like $\mathcal{A}$, the input for analysing a goal includes a binding-time environment and a program environment, reflecting respectively the binding-times of the goal's input variables and the result of binding-time analysis so far. Apart from the goal to be analysed, $\mathcal{G}$ requires the procedure we are currently analysing together with a binding-time environment π_{in} for the procedure's input arguments, reflecting the call pattern for which we are currently analysing the procedure. When incorporating the result of analysing an atomic subgoal, the analysis needs this procedure and call pattern in order to decide whether or not the atom is under dynamic control with respect to the procedure's body. If this

$$
\begin{array}{lcl}
\mathcal{G}[\![A]\!]\pi\Psi p\pi_{in} & = & (\pi_o, \Psi') \\
 & & \text{where } \pi_o = \{(V, \pi'(V) \sqcup \tau_\eta) \mid V \in \mathtt{out}(A)\} \\
 & & \text{if } (\pi', \Psi') = \mathcal{A}[\![A]\!]\pi\Psi \\
 & & \text{and } \eta = \mathcal{P}p(A), \tau_\eta = \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi \\
\mathcal{G}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\pi\Psi p\pi_{in} & = & (\pi_2 \sqcup \pi_3, \Psi_3) \\
 & & \text{where} \left\{ \begin{array}{l} (\pi_1, \Psi_1) = \mathcal{G}[\![G_1]\!]\pi\Psi p\pi_{in} \\ (\pi_2, \Psi_2) = \mathcal{G}[\![G_2]\!]\pi_1\Psi_1 p\pi_{in} \\ (\pi_3, \Psi_3) = \mathcal{G}[\![G_3]\!]\pi\Psi_2 p\pi_{in} \end{array} \right. \\
\mathcal{G}[\![G_1\ ,\ G_2]\!]\pi\Psi p\pi_{in} & = & (\pi_2, \Psi_2) \\
 & & \text{where} \left\{ \begin{array}{l} (\pi_1, \Psi_1) = \mathcal{G}[\![G_1]\!]\pi\Psi p\pi_{in} \\ (\pi_2, \Psi_2) = \mathcal{G}[\![G_2]\!]\pi_1\Psi_1 p\pi_{in} \end{array} \right. \\
\mathcal{G}[\![G_1\ ;\ G_2]\!]\pi\Psi p\pi_{in} & = & (\pi_1 \sqcup \pi_2, \Psi_2) \\
 & & \text{where} \left\{ \begin{array}{l} (\pi_1, \Psi_1) = \mathcal{G}[\![G_1]\!]\pi\Psi p\pi_{in} \\ (\pi_2, \Psi_2) = \mathcal{G}[\![G_2]\!]\pi\Psi_1 p\pi_{in} \end{array} \right. \\
\mathcal{G}[\![\mathit{not}(G)]\!]\pi\Psi p\pi_{in} & = & \mathcal{G}[\![G]\!]\pi\Psi p\pi_{in}
\end{array}
$$

Figure 3.6: Definition of $\mathcal{G}$ for non-atomic goals.

is the case, it updates the current binding-time environment by associating the atom's output variables with $\top$. If the atom is not under dynamic control, the atom's output variables are associated with their binding-time as computed by $\mathcal{A}$. Using $\{\bot, \top\}$ to denote dynamic control allows to incorporate this result in the computed binding-times by simple use of the least upper bound operator. Note that the program environment is updated independently of the fact whether the atom is under dynamic control or not. Indeed, if the atom is a procedure call, the particular procedure must be analysed with respect to the binding-times from the call, either to guide the reduction process in case the call is reduced or to create a new specialised version of the predicate in case the call is residualised.

The binding-time analysis of non-atomic goals is quite straightforward and again abstracts the semantic function $\mathcal{S}$: the binding-time environment produced by an if-then-else goal is the least upper bound of the binding-time environments of both the then- and else-goals. The binding-time environment resulting from the test-goal is used as input to the analysis of the then-goal. Since in Mercury, the test-goal of an if-then-else can not export bindings to the else-goal, analysing the latter is done with the original binding-time environment as input. The program environment is threaded through the complete if-then-else goal, as it should record each of the calls occurring during analysis. When analysing a conjunction, both conjuncts are analysed and the result of analysing the first conjunct is used as input when analysing the second. The binding-time environment resulting from a disjunction equals the least upper bound of the binding-time environments that

are obtained by analysing each of the disjuncts separately. The initial program environment, however, is again threaded through the construct and possibly updated in every disjunct. The result of analysing a negated goal is simply the result of analysing the goal. In Mercury, a goal $not(G)$ may not bind variables occurring outside G.

Example 3.20 *Reconsider the definition of* `append/3` *from Example 3.5. Let* G_1 *and* G_2 *denote the first, respectively second disjunct in* `append/3`*'s body goal, that is*

$$\begin{array}{lcl} G_1 & = & (X \Rightarrow [], Z := Y) \\ G_2 & = & (X \Rightarrow [E|Es], append(Es, Y, R), Z := [E|R]) \end{array}$$

Let β_l *be a binding-time of the form*

$$\beta_l = \{(\langle\rangle, static), (\rangle[|], 1\rangle, dynamic)\}$$

approximating values of type $list(T)$ *that are instantiated up to a list skeleton and let* π *be the binding-time environment* $\pi = \{X/\beta_l, Y/\beta_l\}$*. Then, if* $\Psi(\texttt{append}) = \emptyset$*, we have that*

$$\mathcal{G}[\![G_1]\!]\pi\Psi\texttt{append}\pi = (\ \{\ X/\beta_l, Y/\beta_l, Z/\beta_l\ \}, \quad \Psi\ )$$

and

$$\mathcal{G}[\![G_2]\!]\pi\Psi\texttt{append}\pi = \left(\ \left\{\ \begin{array}{l} X/\beta_l, Y/\beta_l, Z/\bot \\ E/\{(\langle\rangle, dynamic)\}, Es/\beta_l \\ R/\bot \end{array}\ \right\}, \quad \Psi'\ \right)$$

where Ψ' *is such that* $\Psi'(\texttt{append}) = \{(\pi, \{Z/\bot\})\}$*. Moreover, we have that*

$$\mathcal{G}[\![(G_1; G_2)]\!]\pi\Psi\texttt{append}\pi = \left(\ \left\{\ \begin{array}{l} X/\beta_l, Y/\beta_l, Z/\beta_l \\ E/\{(\langle\rangle, dynamic)\}, Es/\beta_l \\ R/\bot \end{array}\ \right\}, \quad \Psi'\ \right)$$

Remains the definition of the dynamic context function $\mathcal{C}$. Recall that the intended meaning of $\mathcal{C}[\![G]\!]\eta\pi\Psi = \top$ is that the atom in G identified by the program point η is under dynamic control when G is reduced with respect to input that is approximated by the binding-time environment π. An atom is under dynamic control when the fact whether it will be evaluated depends on the success or failure of another goal, while the latter can not be determined during specialisation. Goals of which the evaluation depends on the success or failure of another goal are the then- and else-goals (and their subgoals) in an if-then-else – depending on success or failure of the test-goal– and the second conjunct in a conjunction, which is only evaluated when the first goal succeeds. For approximating success or failure of a goal at analysis-time, we introduce yet another auxiliary function

$$\mathcal{R} : Goal \times \mathcal{BT}Env \times \mathcal{P}Env \mapsto \{\bot, \top\}$$

Given a goal G, a binding-time environment π for that goal's input variables and a program environment Ψ representing the results of analysis so far, $\mathcal{R}$ computes whether or not reduction of that goal with respect to values that are approximated by the binding-time environment possibly results in residual code that might fail when evaluated at run-time. If $\mathcal{R}[\![G]\!]\pi\Psi = \bot$, specialisation of G is guaranteed to result in either success or failure *at specialisation-time*, or in some residual code which is guaranteed *not* to fail when later on evaluated at run-time. If specialisation of a goal possibly results in some residual code that might either succeed or fail at run-time, $\mathcal{R}[\![G]\!]\pi\Psi = \top$. The definition of $\mathcal{R}$ is to be found in Fig. 3.7. Deconstruction and test unifications can be reduced to *true* or *fail* during

$$
\begin{array}{lcl}
\mathcal{R}[\![X \Rightarrow f(\overline{Y})]\!]\pi\Psi & = & \begin{cases} \bot & \text{if } \pi(X)(\langle\rangle) = \mathit{static} \\ \top & \text{otherwise} \end{cases} \\
\mathcal{R}[\![X == Y]\!]\pi\Psi & = & \begin{cases} \bot & \text{if } \pi(X)(\langle\rangle) \sqcup \pi(Y)(\langle\rangle) = \mathit{static} \\ \top & \text{otherwise} \end{cases} \\
\mathcal{R}[\![X := Y]\!]\pi\Psi & = & \bot \\
\mathcal{R}[\![X \Leftarrow f(\overline{Y})]\!]\pi\Psi & = & \bot \\
\mathcal{R}[\![p(X_1, \ldots, X_n)]\!]\pi\Psi & = & \mathcal{R}[\![\mathcal{B}\mathit{ody}(p)]\!]\pi_{in}\Psi \\
 & & \text{where, if } \langle F_1, \ldots, F_n\rangle = \mathcal{A}\mathit{rgs}(p) \\
 & & \pi_{in} = \{(F_i, \pi(X_i)) \mid F_i \in \mathtt{in}(p)\} \\
\mathcal{R}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\pi\Psi & = & \mathcal{R}[\![G_1]\!]\pi\Psi \sqcup \mathcal{R}[\![G_2]\!]\pi'\Psi \sqcup \mathcal{R}[\![G_3]\!]\pi\Psi \\
 & & \text{where } (\pi', \_) = \mathcal{G}[\![G_1]\!]\pi\Psi \\
\mathcal{R}[\![(G_1\ ,\ G_2)]\!]\pi\Psi & = & \mathcal{R}[\![G_1]\!]\pi\Psi \sqcup \mathcal{R}[\![G_2]\!]\pi'\Psi \\
 & & \text{where } (\pi', \_) = \mathcal{G}[\![G_1]\!]\pi\Psi \\
\mathcal{R}[\![(G_1\ ;\ G_2)]\!]\pi\Psi & = & \mathcal{R}[\![G_1]\!]\pi\Psi \sqcup \mathcal{R}[\![G_2]\!]\pi\Psi \\
\mathcal{R}[\![\mathit{not}(G)]\!]\pi\Psi & = & \mathcal{R}[\![G]\!]\pi\Psi
\end{array}
$$

Figure 3.7: The definition of $\mathcal{R}$.

specialisation when the unification's input variables are bound to an outermost constructor (represented during binding-time analysis by a binding-time that maps the root node $\langle\rangle$ to *static*). Otherwise, the unification is residualised and whether it will succeed or fail at run-time can not be determined during analysis, as the necessary input is still unavailable. If assignment and construction unifications are residualised, the residual atoms always succeed at run-time, hence the result of $\mathcal{R}$ is $\bot$. Handling a procedure call during specialisation does not result in code that might fail at run-time when reducing the body goal does not result in residual code that can fail. For the remaining goals it holds that they do not reduce to code that can possibly fail if the same holds for each of their subgoals, under the appropriate binding-time environments, which are threaded exactly as in the definition of $\mathcal{G}$.

Example 3.21 *Reconsider the* `append/3` *procedure from Example 3.5. Assume*

that π is a binding-time environment $\pi = \{X/\beta_l, Y/\beta_d\}$ where

$$\begin{array}{lcl} \beta_l & = & \{(\langle\rangle, static), (\langle[|], 1\rangle, dynamic)\} \\ \beta_d & = & \{(\langle\rangle, dynamic), (\langle[|], 1\rangle, dynamic)\} \end{array}$$

(β_l approximating those terms of type $list(T)$ that are bound at least to a list skeleton, and β_d approximating every term (including a free variable) of type $list(T)$). Then we have for any program environment Ψ:

$$\mathcal{R}[\![X \Rightarrow [E|Es]]\!]\pi\Psi = \bot.$$

Indeed, X's binding-time guarantees that X is bound, during specialisation, to a term that is instantiated at least up to a list skeleton. Hence, the deconstruction can be performed during specialisation, resulting to true or fail depending on the particular outermost functor of X's value. On the other hand,

$$\mathcal{R}[\![Y \Rightarrow [E'|Es']]\!]\pi\Psi = \top.$$

Indeed, according to Y's binding-time, Y might be a free variable during specialisation. Hence, the deconstruction can not be annotated reducible; and once residualised, the atom can still succeed or fail at runtime.

Given the definition of $\mathcal{R}$, we can finally define the dynamic context function. Its definition is depicted in Fig. 3.8. First, for every goal G holds that G itself is

$$\begin{array}{lcl}
\mathcal{C}[\![A]\!]\eta\pi\Psi & = & \bot \text{ for } A \in Atom \\
\mathcal{C}[\![if\, G_1\, then\, G_2\, else\, G_3]\!]\eta\pi\Psi & = & \begin{cases} \mathcal{R}[\![G_1]\!]\pi\Psi \sqcup \mathcal{C}[\![G_2]\!]\eta\pi\Psi & \text{if } \eta \in \mathcal{P}ps(G_2) \\ \mathcal{R}[\![G_1]\!]\pi\Psi \sqcup \mathcal{C}[\![G_3]\!]\eta\pi\Psi & \text{if } \eta \in \mathcal{P}ps(G_3) \\ \mathcal{C}[\![G_1]\!]\eta\pi\Psi & \text{if } \eta \in \mathcal{P}ps(G_1) \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{C}[\![(G_1\ ,\ G_2)]\!]\eta\pi\Psi & = & \begin{cases} \mathcal{R}[\![G_1]\!]\pi\Psi \sqcup \mathcal{C}[\![G_2]\!]\eta\pi\Psi & \text{if } \eta \in \mathcal{P}ps(G_2) \\ \mathcal{C}[\![G_1]\!]\eta\pi\Psi & \text{if } \eta \in \mathcal{P}ps(G_1) \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{C}[\![(G_1\ ;\ G_2)]\!]\eta\pi\Psi & = & \begin{cases} \mathcal{C}[\![G_i]\!]\eta\pi\Psi & \text{if } \eta \in \mathcal{P}ps(G_i) \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{C}[\![not(G)]\!]\eta\pi\Psi & = & \begin{cases} \mathcal{C}[\![G]\!]\eta\pi\Psi & \text{if } \eta \in \mathcal{P}ps(G) \\ \bot & \text{otherwise} \end{cases}
\end{array}$$

Figure 3.8: Definition of $\mathcal{C}$.

not under dynamic control in G: hence, if $\eta \in \mathcal{PP}$ is the program point associated to G itself, $\mathcal{C}[\![G]\!]\eta\pi\Psi = \bot$. The latter is always the case if G is an atom (since η can only identify the atom itself). If η identifies an atom in the then-goal, respectively

else-goal of an if-then-else, the atom is under dynamic control with respect to the if-then-else if either reduction of the test-goal results in some residual code that can possibly fail, or if it is under dynamic control with respect to the then-goal, respectively else-goal. If the atom is a subgoal in the test-goal, it is under dynamic control with respect to the if-then-else if it is under dynamic control with respect to the test-goal. An atom in the second conjunct of a conjunction is under dynamic control with respect to the conjunction if either reducing the first conjunct results in residual code that can possibly fail, or the atom is under dynamic control with respect to the second conjunct. An atom in the first conjunct of a conjunction is under dynamic control with respect to the conjunction if it is under dynamic control with respect to this first conjunct. An atom in a disjunct of a disjunction is under dynamic control with respect to the disjunction if it is under dynamic control with respect to its particular disjunct, and an atom in a negated goal is under dynamic control with respect to the negation, if it is under dynamic control with respect to the negated goal.

Example 3.22 *Reconsider the* `append/3` *procedure from Example 3.5. Assume that* π *is a binding-time environment such that* $dom(\pi) \ni X$ *and*

$$\pi(X) = \{(\langle\rangle, dynamic), (\langle[|], 1\rangle, dynamic)\}.$$

We then have that

$$\mathcal{C}[\![(X \Rightarrow []_{\eta_1}, Z := Y_{\eta_2})]\!]\eta_2\pi\Psi = \top.$$

Indeed, since $\mathcal{R}[\![X \Rightarrow []]\!]\pi\Psi = \top$*, the assignment* $Z := Y$ *is under dynamic control with respect to the conjunction.*

Our analysis function $\mathcal{G}$ is capable of analysing a procedure's body with respect to a particular binding-time environment, reflecting a particular call pattern. In order to perform a polyvariant binding-time analysis, we need to perform analysis of a complete program: each of the program's procedures must be analysed with respect to each of the binding-time environments reflecting the call pattern of a call occurring during analysis. Analysing a program requires a program environment that records, for each predicate, the binding-time environments with respect to which that predicate must be analysed and results in a possibly updated program environment. The program analysis function Δ is defined in Definition 3.13.

Definition 3.13 *The program analysis function* $\Delta : \wp(Proc) \times \mathcal{PE}nv \mapsto \mathcal{PE}nv$ *is defined as follows:*

$$\Delta(P, \Psi) = \begin{cases} \Psi & \text{if } P = \emptyset \\ \Delta(P_s, \delta(G, \Psi(p), \Psi) & \text{if } P = \{(p(\overline{X}) \leftarrow G) :: P_s\} \end{cases}$$

where the auxiliary function

$$\delta : Goal \times (\mathcal{BTE}nv \mapsto \mathcal{BTE}nv) \times \mathcal{PE}nv \mapsto \mathcal{PE}nv$$

repeatedly analyses a goal for a number of binding-time environments for the goal's input arguments:

$$\delta(G,S,\Psi) = \begin{cases} \Psi & \text{if } S = \emptyset \\ \delta(G, S_s, \Psi'[p/\Psi'(p) \sqcup \{(\pi_i, \pi'_o)\}]) & \text{if } S = \{(\pi_i,\pi_o) :: S_s\} \\ & \text{and } (\pi'_o, \Psi') = \mathcal{G}[\![G]\!]\pi_i\Psi \end{cases}$$

Given a program P, the result of binding-time analysis is a program environment Ψ such that Ψ is a solution to the equation

$$\Psi = \Delta(P, \Psi). \tag{3.1}$$

In general, binding-time analysis is to be performed with respect to the binding-time abstraction of the (atomic) partial deduction query. Hence we start the analysis from an initial program environment Ψ_0 that contains a single binding-time environment representing the initial call. If the initial call is $p(\overline{t})$ and the procedure p is defined as $p(\overline{F}) \leftarrow G$, the initial program environment Ψ_0 is such that

$$\Psi_0(p) = \{(\ \{(F_i, \alpha(t_i)) \mid F_i \in \mathtt{in}(p)\},\ \{(F_i, \bot) \mid F_i \in \mathtt{out}(p)\}\ )\}$$

and $\Psi_0(q) = \emptyset$ for all $q \neq p$. In other words, the program environment contains a single entry, for the initially called procedure. This entry maps a binding-time environment containing binding-time approximations of the initial call's input arguments to a binding-time environment in which the procedure's output arguments are initialised to $\bot$. Subsequent program environments are constructed by repeatedly applying Δ:

$$\Psi_i = \Delta(P, \Psi_{i-1}), \forall i > 0 \tag{3.2}$$

In the next subsection, we define a crucial correctness property of program environments and prove the existence of a least fixed point of equation 3.2, being the most precise and correct solution to (3.1).

Example 3.23 *Reconsider the* `append/3` *procedure from Example 3.5. Let* β_l *denote a binding-time of the form*

$$\beta_l = \{(\langle\rangle, \mathit{static}), (\langle[|], 1\rangle, \mathit{dynamic})\}$$

approximating those terms of the type $\mathit{list}(T)$ *that are at least bound to a list skeleton. If* `append([X,Y], [Z], R)` *denotes an initial call, the initial program environment* Ψ_0 *is such that*

$$\Psi_0(\mathtt{append}) = (\ \{X/\beta_l, Y/\beta_l\},\ \ \{\ Z/\bot\ \}\ ).$$

According to equation (3.2) and Example 3.20, we have that

$$\Psi_1(\mathtt{append}) = \left(\ \{X/\beta_l, Y/\beta_l\},\ \ \left\{ \begin{array}{l} X/\beta_l, Y/\beta_l, Z/\beta_l \\ E/\{(\langle\rangle, \mathit{dynamic})\}, Es/\beta_l \\ R/\bot \end{array} \right\}\ \right).$$

Reapplying Δ *results in*

$$\Psi_2(\texttt{append}) = \left(\{X/\beta_l, Y/\beta_l\}, \quad \left\{ \begin{array}{l} X/\beta_l, Y/\beta_l, Z/\beta_l \\ E/\{(\langle\rangle, dynamic)\}, Es/\beta_l \\ R/\beta_l \end{array} \right\} \right)$$

which can be verified to be the least fixed point to (3.2) and hence a solution to (3.1).

3.3.3 Congruence

A program environment relates, for each of the procedures, a binding-time environment containing binding-times for a procedure's input arguments, with a binding-time environment containing binding-times for all of the procedure's variables. Each of the latter binding-times approximate the value the associated variable will have during specialisation, *at the procedure's exit point*. Due to possible $\sqcup$ operations that are performed during abstract interpretation of the procedure's body, this binding-time is not necessarily the best approximation of the variable's value at any program point in the procedure's body goal. However, the binding-time of a variable at a particular program point can be computed from the binding-times of the procedure's input arguments in a straightforward way, by simulating the analysis function $\mathcal{G}$ until the program point of interest is reached and returning the binding-time environment as it exists at that program point. We therefore introduce the function

$$\Delta_{pp} : Goal \times \mathcal{PP} \times \mathcal{BTE}nv \times \mathcal{P}Env \times Proc \times \mathcal{BTE}nv \mapsto \mathcal{BTE}nv$$

with the intended meaning that if during analysis of a procedure p with respect to a binding-time environment π_{in} the goal G (being a subgoal of $\mathcal{B}ody(p)$) is analysed with respect to the binding-time environment π, then $\pi_\eta = \Delta_{pp}[\![G]\!]\eta\pi\Psi p\pi_{in}$ denotes the binding-time environment as it exists at program point $\eta \in \mathcal{P}ps(G)$. The definition of Δ_{pp} is based on the syntactic structure of goals, like $\mathcal{G}$, and is depicted in Fig. 3.9.

Our interest in using Δ_{pp} is to compute a binding-time environment that contains the binding-times for a subgoal's input arguments as they are used when analysing the particular subgoal. Likewise, we can define a complementary function, Δ^+_{pp} that takes the same input as Δ_{pp} but computes the binding-time environment as it exists *after* analysing the particular subgoal.

Definition 3.14 *The function* Δ^+_{pp} *has the signature*

$$\Delta^+_{pp} : Goal \times \mathcal{PP} \times \mathcal{BTE}nv \times \mathcal{P}Env \times Proc \times \mathcal{BTE}nv \mapsto \mathcal{BTE}nv$$

and is defined as

$$\Delta^+_{pp}[\![G]\!]\eta\pi\Psi p\pi_{in} = \mathcal{G}[\![G']\!]\pi_\eta\Psi p\pi_{in}$$

where $\pi_\eta = \Delta_{pp}[\![G]\!]\eta\pi\Psi p\pi_{in}$ *and* G' *is the subgoal of* G *such that* $\mathcal{P}p(G) = \eta$.

$$
\begin{array}{lcl}
\Delta_{pp}[\![A]\!]\eta\pi\Psi p\pi_{in} & = & \pi \\
\Delta_{pp}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\eta\pi\Psi p\pi_{in} & = & \left\{ \begin{array}{ll} \Delta_{pp}[\![G_1]\!]\eta\pi\Psi p\pi_{in} & \text{if } \eta \in \mathcal{P}p(G_1) \\ \Delta_{pp}[\![G_2]\!]\eta\pi'\Psi p\pi_{in} & \text{if } \eta \in \mathcal{P}p(G_2) \\ \Delta_{pp}[\![G_3]\!]\eta\pi\Psi p\pi_{in} & \text{if } \eta \in \mathcal{P}p(G_3) \\ \pi & \text{otherwise} \end{array} \right. \\
 & & \text{where } (\pi', \_) = \mathcal{G}[\![G_1]\!]\pi\Psi p\pi_{in} \\
\Delta_{pp}[\![(G_1\ ,\ G_2)]\!]\eta\pi\Psi p\pi_{in} & = & \left\{ \begin{array}{ll} \Delta_{pp}[\![G_1]\!]\eta\pi\Psi p\pi_{in} & \text{if } \eta \in \mathcal{P}p(G_1) \\ \Delta_{pp}[\![G_2]\!]\eta\pi'\Psi p\pi_{in} & \text{if } \eta \in \mathcal{P}p(G_2) \\ \pi & \text{otherwise} \end{array} \right. \\
 & & \text{where } (\pi', \_) = \mathcal{G}[\![G_1]\!]\pi\Psi p\pi_{in} \\
\Delta_{pp}[\![(G_1\ ;\ G_2)]\!]\eta\pi\Psi p\pi_{in} & = & \left\{ \begin{array}{ll} \Delta_{pp}[\![G_1]\!]\eta\pi\Psi p\pi_{in} & \text{if } \eta \in \mathcal{P}p(G_1) \\ \Delta_{pp}[\![G_2]\!]\eta\pi\Psi p\pi_{in} & \text{if } \eta \in \mathcal{P}p(G_2) \\ \pi & \text{otherwise} \end{array} \right. \\
\Delta_{pp}[\![\mathit{not}(G)]\!]\eta\pi\Psi p\pi_{in} & = & \left\{ \begin{array}{ll} \Delta_{pp}[\![G]\!]\eta\pi\Psi p\pi_{in} & \text{if } \eta \in \mathcal{P}p(G) \\ \pi & \text{otherwise} \end{array} \right.
\end{array}
$$

Figure 3.9: The definition of Δ_{pp}.

In order to be useful for program specialisation, the program environment resulting from binding-time analysis must be *congruent* (Jones, Gomard, and Sestoft 1993). Intuitively, congruence means that if there is a possible data flow (through unification or parameter binding) from a subvalue of X to a subvalue of Y in the program, then the binding-time approximation of the particular subvalue of Y must be *at least as dynamic* as the binding-time approximation of the particular subvalue of X. The general idea behind congruence is that values can not be characterised as known during specialisation (*static*) if they are constructed from values that are possibly unknown during specialisation (*dynamic*). The notion of congruence is formally defined in the following definition.

Definition 3.15 *Given a program P, a program environment Ψ is* congruent *when for each procedure $p(\overline{F}) \leftarrow G$ in P and for each atomic subgoal A identified by program point η of G and for all $(\pi_i, \pi_o) \in \Psi(p)$ the following holds: if*

$$\pi_\eta = \Delta_{pp}[\![\mathcal{B}\mathit{ody}(p)]\!]\eta\pi_i\Psi p\pi_i$$

and

$$\pi^+_\eta = \Delta^+_{pp}[\![\mathcal{B}\mathit{ody}(p)]\!]\eta\pi_i\Psi p\pi_i$$

then it holds that

- *if A is of the form $Y \Rightarrow f(X_1, \ldots, X_n)$ then $\forall i \in \{1, \ldots, n\} : \beta_{X_i} \succeq \beta_Y^{\langle (f,i) \rangle}$ where $\beta_{X_i} = \pi^+_\eta(X_i)$ and $\beta_Y = \pi_\eta(Y)$*

- *if A is of the form $Y \Leftarrow f(X_1, \ldots, X_n)$ then $\forall i \in \{1, \ldots, n\} : \beta_Y^{\langle f,i \rangle} \succeq \beta_{X_i}$ where $\beta_{X_i} = \pi_\eta(X_i)$ and $\beta_Y = \pi_\eta^+(Y)$*
- *if A is of the form $X := Y$ then $\beta_X \succeq \beta_Y$ where $\beta_X = \pi_\eta^+(X)$ and $\beta_Y = \pi_\eta(Y)$*
- *if A is of the form $q(\overline{X})$ and $\langle F_1, \ldots, F_n \rangle = \mathcal{A}rgs(q)$ then there exists an element $(\pi_q, \pi'_q) \in \Psi(q)$ such that*
  - *for all $F_i \in \mathtt{in}(q) : \beta_{F_i} \succeq \beta_{X_i}$ where $\beta_{F_i} = \pi_q(F_i)$ and $\beta_{X_i} = \pi_\eta(X_i)$*
  - *for all $F_i \in \mathtt{out}(q) : \beta_{X_i} \succeq \beta_{F_i}$ where $\beta_{F_i} = \pi'_q(F_i)$ and $\beta_{X_i} = \pi_\eta^+(X_i)$*

Now, let us return to the result of binding-time analysis as defined in Equation 3.1 on page 75. If our binding-time analysis is to be as precise as possible, we are interested in the *least solution* to that equation. Correctness requires this solution to be congruent. Starting from an initial program environment, Ψ_0, the least solution to Equation 3.1 can be found by repeatedly computing, for all $i > 0$, $\Psi_i = \Delta(P, \Psi_{i-1})$ until a fixed point is reached. During a first analysis (using δ), this environment is updated by the newly computed binding-times for the initial procedure's output arguments, and with environments for the procedure calls encountered during analyses. An important correctness result is that repeatedly reapplying the analysis on the newly obtained program environments reaches a fixed point, which is the least congruent solution to Equation 3.1.

Theorem 3.1 *Let P be a program, and $p(\overline{t})$ an initial call. If Ψ_0 is defined as before, namely*

$$\Psi_0(p) = \{(\ \{(F_i, \alpha(t_i)) \mid F_i \in \mathtt{in}(p)\},\ \{(F_i, \bot) \mid F_i \in \mathtt{out}(p)\}\ )\}$$

and for all $i > 0, \Psi_i = \Delta_{pp}(P, \Psi_{i-1})$, there exists $n \in \mathbb{N}$ such that $\Psi_n = \Psi_{n-1}$ and Ψ_n is the least congruent solution to Equation 3.1.

Proof For any update of Ψ into Ψ', be it by $\mathcal{A}$, $\mathcal{G}$ or the auxiliary function δ, holds that $\Psi' \sqsupseteq \Psi$ since the update is performed using $\sqcup$. By the way the program environment is threaded through Δ via δ, $\mathcal{G}$ and $\mathcal{A}$, $\Psi_i \sqsupseteq \Psi_{i-1}$, for all $i > 0$ in $\Psi_i = \Delta(P, \Psi_{i-1})$. As such, repeated applications of Δ yield a monotonic sequence of program environments, implying the existence of a Ψ such that $\Psi = \Delta(P, \Psi)$. More, there exist $n \in \mathbb{N}$ such that $\Psi_n = \Delta(P, \Psi_n)$. Indeed, the number of types used in P is finite, hence so is $\mathcal{BT}^+$and the number of possible program environments for P. By definition, $\mathcal{A}$ constructs a congruent binding-time environment, and the use of $\sqcup$ in $\mathcal{G}$ to combine binding-time environments preserves congruence. So Ψ is a congruent solution. Moreover, it is the least congruent solution, since all exit patterns recorded in Ψ start from $\bot$ and are changed to a higher value (using $\sqcup$) only when congruence so requires.

□

In this section, we have described a binding-time analysis for Mercury by abstract interpretation. The result is a congruent program environment from which the binding-time of every program variable at every program point can be derived.

3.4 From Binding-times to Specialisation

In this section we define how a congruent program environment can be transformed into instructions for an off-line program specialiser. We also show correctness of the binding-time analysis with respect to a particular specialiser.

3.4.1 An Annotated Program

A suitable way to represent the result of binding-time analysis – both for a human inspector and a program specialiser – is to add annotations to the original source code. These annotations are instructions telling the specialiser what it should do with a particular goal: either *reduce* the goal, or *residualise* it. To record these instructions in a source program, we introduce a two-level syntax, as in (Jones, Gomard, and Sestoft 1993): the syntactic domains $2Atom$, $2Goal$ and $2Proc$ are defined as in Definition 3.2, except that each construct appears in a *static* as well as *dynamic* variant: For a goal $G \in Goal$, G^s and G^d denote the static, respectively dynamic variant of G in $2Goal$. A static annotation is a directive to the specialiser to reduce this goal, a dynamic one the instruction to residualise it. An annotated program simply is a set of annotated procedures. Figures 3.10 through 3.12 define how to derive a version of a program P that is annotated with respect to a congruent program environment Ψ. In this definition, a rule of the form

$$\Psi, p, \pi_{in} \vdash G \rhd G'$$

should be read as follows: given a program environment Ψ, a procedure p and a binding-time environment π_{in} containing binding-times for p's input arguments, if G is a subgoal of $\mathcal{B}ody(p)$, the goal G' is an annotated version of G with respect to Ψ and π_{in}. To express the validity of a rule r given a condition C, we use the notation

$$\frac{C}{r}$$

Figure 3.10 defines the annotation of atomic goals. In this definition, the binding-time environment π_η denotes the binding-time environment at the program point η, that is

$$\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi p\pi_{in}.$$

A test identified by program point η in the body of a predicate p is annotated reducible (*static*) if at program point η the involved variables are bound to an

$$\frac{\pi_\eta(X) \sqcup \pi_\eta(Y) \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi \neq \top}{\Psi, p, \pi_{in} \vdash X ==_\eta Y \rhd X ==^s_\eta Y}$$

$$\frac{\pi_\eta(X) \sqcup \pi_\eta(Y) \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi = \top}{\Psi, p, \pi_{in} \vdash X ==_\eta Y \rhd X ==^d_\eta Y}$$

$$\frac{\pi_\eta(X) \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi \neq \top}{\Psi, p, \pi_{in} \vdash X \Rightarrow_\eta f(\overline{Y}) \rhd X \Rightarrow^s_\eta f(\overline{Y})}$$

$$\frac{\pi_\eta(X) \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi = \top}{\Psi, p, \pi_{in} \vdash X \Rightarrow_\eta f(\overline{Y}) \rhd X \Rightarrow^d_\eta f(\overline{Y})}$$

$$\frac{\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi\Psi \neq \top}{\Psi, p, \pi_{in} \vdash X :=_\eta Y \rhd X :=^s_\eta Y} \qquad \frac{\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi\Psi = \top}{\Psi, p, \pi_{in} \vdash X :=_\eta Y \rhd X :=^d_\eta Y}$$

$$\frac{\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi\Psi \neq \top}{\Psi, p, \pi_{in} \vdash X \Leftarrow_\eta f(\overline{Y}) \rhd X \Leftarrow^s_\eta f(\overline{Y})}$$

$$\frac{\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi\Psi = \top}{\Psi, p, \pi_{in} \vdash X \Leftarrow_\eta f(\overline{Y}) \rhd X \Leftarrow^d_\eta f(\overline{Y})}$$

$$\frac{\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi\Psi \neq \top}{\Psi, p, \pi_{in} \vdash q(\overline{X})_\eta \rhd q_S(\overline{X})^s_\eta} \quad \text{where } S = \langle \pi_\eta(X_{i_1}), \ldots, \pi_\eta(X_{i_k}) \rangle \text{ if } \mathtt{in}(q) = \langle F_{i_1}, \ldots, F_{i_k} \rangle$$

$$\frac{\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi\Psi = \top}{\Psi, p, \pi_{in} \vdash q(\overline{X})_\eta \rhd q_S(\overline{X})^d_\eta} \quad \text{where } S = \langle \pi_\eta(X_{i_1}), \ldots, \pi_\eta(X_{i_k}) \rangle \text{ if } \mathtt{in}(q) = \langle F_{i_1}, \ldots, F_{i_k} \rangle$$

Figure 3.10: Annotation of atoms.

outermost functor and the test is not under dynamic control. Otherwise, the test is annotated to residualise (*dynamic*). Annotating a deconstruction is analogously: if at η the deconstructed variable is bound to an outermost functor and the deconstruction is not under dynamic control, it is annotated reducible, otherwise it is annotated residualisable. An assignment or a construction unification is annotated reducible if it is not under dynamic control, otherwise it is annotated to residualise.

The same holds for a procedure call: if the call is not under dynamic control, it is annotated reducible, otherwise it is annotated to residualise. Note that in both cases the called procedure symbol is renamed into a procedure symbol that is uniquely determined by the call pattern, expressed as the sequence of binding-times associated to the call's input variables. Indeed, since our binding-time analysis is polyvariant, we create a particularly annotated version of a procedure, for each of its call patterns encountered during analysis. Hence, when a call is annotated, it should refer to the right annotated version of the procedure, namely the one that is annotated with respect to a binding-time environment in which the procedure's input arguments are bound to the binding-times contained in S. Note that the conditions under which an atom is annotated reducible equal the conditions under which the binding-times of the atom's output variables were incorporated during construction of the program environment Ψ. This is natural since the annotations simply make the specialisation strategy that was assumed by the analysis explicit.

The same observation holds for the annotation of structured goals, to be found in Figure 3.11. An if-then-else goal is annotated reducible if its test goal reduces during specialisation either to *true*, *fail* or residual code which is guaranteed not to fail, and the if-then-else goal itself is not under dynamic control. Otherwise it is annotated to residualise. A not-goal is annotated static if the negated goal can during specialisation be reduced to *true*, *fail* or residual code that is guaranteed not to fail, and the not-goal itself is not under dynamic control. Otherwise, it is annotated to residualise. Conjunctions as well as disjunctions are unconditionally annotated reducible, since the specialiser treats them as connectives and always considers their subgoals for reduction.

An annotated program consists of a set of annotated procedures. An annotated procedure is obtained by annotating the procedure's body goal, and renaming the procedure symbol so that its name is uniquely associated with the binding-times of the procedure's input arguments with respect to which the procedure is annotated. Each of the original program's procedures is annotated once for every binding-time environment for its input arguments that was encountered during analysis. The binding-time environments for which an annotated version must be created are found in the program environment. The rules for annotating a procedure and a complete program are depicted in Fig. 3.12. Note that the annotate symbol $\triangleright$ is overloaded to denote the annotation of a procedure, respectively a program. In

$$\frac{\mathcal{R}[\![G_1]\!]\pi_{in}\Psi \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi = \bot \quad \Psi, p, \pi_{in} \vdash G_1 \rhd G_1' \quad \Psi, p, \pi_{in} \vdash G_2 \rhd G_2' \quad \Psi, p, \pi_{in} \vdash G_3 \rhd G_3'}{\Psi, p, \pi_{in} \vdash \mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3 \rhd \mathit{if}^s\, G_1'\, \mathit{then}\, G_2'\, \mathit{else}\, G_3'}$$

$$\frac{\mathcal{R}[\![G_1]\!]\pi_{in}\Psi \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi = \top \quad \Psi, p, \pi_{in} \vdash G_1 \rhd G_1'}{\Psi, p, \pi_{in} \vdash \mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3 \rhd \mathit{if}^d\, G_1'\, \mathit{then}\, G_2\, \mathit{else}\, G_3}$$

$$\frac{\mathcal{R}[\![G]\!]\pi_{in}\Psi \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi = \bot, \Psi, p, \pi_{in} \vdash G \rhd G'}{\Psi, p, \pi_{in} \vdash \mathit{not}(G) \rhd \mathit{not}^s(G')}$$

$$\frac{\mathcal{R}[\![G]\!]\pi_{in}\Psi \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi = \top, \Psi, p, \pi_{in} \vdash G \rhd G'}{\Psi, p, \pi_{in} \vdash \mathit{not}(G) \rhd \mathit{not}^d(G')}$$

$$\frac{\Psi, p, \pi_{in} \vdash G_1 \rhd G_1' \quad \Psi, p, \pi_{in} \vdash G_2 \rhd G_2'}{\Psi, p, \pi_{in} \vdash (G_1\ ,\ G_2) \rhd (G_1'\ ,\ G_2')^s}$$

$$\frac{\Psi, p, \pi_{in} \vdash G_1 \rhd G_1' \quad \Psi, p, \pi_{in} \vdash G_2 \rhd G_2'}{\Psi, p, \pi_{in} \vdash (G_1\ ;\ G_2) \rhd (G_1'\ ;\ G_2')^s}$$

Figure 3.11: Annotation of structured goals.

the rule for annotating a program,

$$\Psi \vdash P \rhd P'$$

denotes that P' is an annotated version of P with respect to the program environment Ψ. The following definition links an annotated goal with the binding-times of the goal's input variables with respect to which it was annotated.

Definition 3.16 *Given a program P and a congruent program environment Ψ for P. Let p be a procedure defined in P, and G a subgoal of $\mathcal{B}ody(p)$, identified by the program point η. We say that G_a is an* annotated version of G with respect to the binding-time environment π_η *if and only if*

$\exists(\pi_{in}, \_) \in \Psi(p)$ *such that* $\Psi, p, \pi_{in} \vdash G \rhd G_a$ *and* $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi p\pi_{in}$.

We conclude this section with the annotated version of the `append/3` procedure from Example 3.23:

$$\frac{\Psi, p, \pi_{in} \vdash B \rhd B' \text{ and } S = \langle \pi_{in}(F_{i_1}), \ldots, \pi_{in}(F_{i_k}) \rangle \text{ if } \mathtt{in}(p) = \langle F_{i_1}, \ldots, F_{i_k} \rangle}{\Psi, p, \pi_{in} \vdash p(\overline{F}) \leftarrow B \rhd p_S(\overline{F}) \leftarrow B'}$$

$$\frac{\left.\begin{array}{l} p(\overline{F}) \leftarrow B \in P \\ (\pi_{in}, \_) \in \Psi(p) \\ \Psi, p, \pi_{in} \vdash p(\overline{F}) \leftarrow B \rhd p_S(\overline{F}) \leftarrow B' \end{array}\right\} \Rightarrow \quad p_S(\overline{F}) \leftarrow B' \in P'}{\Psi \vdash P \rhd P'}$$

Figure 3.12: Annotation of procedures and programs.

Example 3.24 *The annotated version of* `append/3` *with respect to* Ψ_2 *and* $\pi_{in} = \{X/\beta_l, Y/\beta_l\}$ *from Example 3.23 is*

```
append(X,Y,Z):-
  (X⇒ˢ [], Z:=ˢ Y ;
   X⇒ˢ [E|Es], append(Es,Y,R)ˢ, Z⇐ˢ [E|R]).
```

3.4.2 From Annotation to Specialisation

If Ψ is a congruent program environment for a program P, the annotated version of P with respect to Ψ is *well-annotated* (Jones, Gomard, and Sestoft 1993). Well-annotatedness ensures that if a goal is reduced with respect to *partial* values that are correctly approximated by the binding-times for which the goal was annotated, the reduction process that follows the annotations "can not go wrong". In our logic programming setting, this involves the following:

- If a unification (be it a test, deconstruction, assignment or construction) is reduced by the specialiser, the involved variables are bound to values that are instantiated enough to perform the particular unification.
- If the specialiser is instructed to reduce a structured goal and doing so requires to know whether a particular subgoal succeeds or fails (as is the case with a negation, an if-then-else or a conjunction), then knowledge about success or failure of the particular subgoal is available after its reduction.

Reducing a goal is very similar to evaluating the goal. Both processes require an environment that binds the goal's variables to some values, and both may produce new bindings during the process. The difference, of course, is that reduction only considers those (sub)goals that are annotated reducible, and residualises the others. As such, the result of reduction can be in the form of bindings, residual

code, or a combination of both. Formally, we can define the reduction process by the semantic function

$$\mathcal{T} : 2Goal \times Subst \mapsto \wp(2Goal \times Subst).$$

Given an annotated goal G and a substitution θ providing partial values (in the form of partially instantiated terms) for the goal's input variables, $\mathcal{T}[\![G]\!]\theta$ is either the empty set $\emptyset$ denoting failure of G under θ, or a set of pairs of the form (G', θ'). Each such pair consists of a residual (annotated) goal G' and a substitution θ' that is an update of θ and includes the bindings that were produced by the reduction process. If the residual goal is *true*, it means that the goal G was completely reduced resulting in success with the computed answer substitution θ'. Since the substitution θ' is an update of θ, we will often represent a substitution resulting from $\mathcal{T}[\![G]\!]\theta$ as $\theta\sigma$, in order to concretise the bindings – the substitution σ – that were created by the reduction.

During specialisation, the result of reduction needs to be converted back to a Mercury goal in superhomogeneous form. Doing so involves removing the remaining (dynamic) annotations from a residual goal, and converting the bindings that were created during reduction to some code that produces these values (lifting). For a substitution θ, we denote with $\overline{\theta}$ the goal that is defined as the conjunction of *constructions* that "create" θ.

Example 3.25 *Let θ be a substitution $\theta = \{X/[], Y/[X_1, X_2]\}$. Then $\overline{\theta}$ represents the goal*

$$X \Leftarrow [], Y_1 \Leftarrow [], Y_2 \Leftarrow [X_2|Y_1], Y \Leftarrow [X_1|Y_2].$$

Due to the the syntactic form of unifications in superhomogeneous form, $\overline{\theta}$ can contain variables that are originally not present in θ. In what follows we assume that these intermediate variables created by $\overline{\theta}$ are fresh and used nowhere else such that for substitutions θ, σ with $dom(\theta) \cap dom(\sigma) = \emptyset$, it holds that

$$\mathcal{S}[\![\overline{\theta}]\!]\sigma = \{\theta\sigma\}$$

modulo the newly introduced variables of $\overline{\theta}$. In other words, the result of evaluating the code that "creates" a substitution θ with respect to an input substitution σ, that is $\mathcal{S}[\![\overline{\theta}]\!]\sigma$ – with $\mathcal{S}$ being Mercury's semantic function – equals a single substitution $\theta\sigma$ (modulo the fresh variables introduced by $\overline{\theta}$).

Converting the result of reduction to a Mercury goal is denoted by the *lifting* function that is defined in Definition 3.17.

Definition 3.17 *The* lifting *function*

$$\phi : \wp(2Goal \times Subst) \mapsto Goal$$

is recursively defined as follows:

$$\begin{array}{lcl} \phi(\emptyset) & = & fail \\ \phi((G, \theta) :: S_s) & = & (\ (\overline{\theta}\ ,\ G)\ ;\ \phi(S_s)\ ) \end{array}$$

The lifting function basically creates a disjunction, with a single disjunct for each success element resulting from the reduction. For each such element (θ, G), it creates a conjunction: the first conjunct being the goal that creates θ, the second one being the residual goal G (without the annotations).

Figure 3.13 shows the definition of the reduction function $\mathcal{T}$. As reduction

$$
\begin{array}{lcl}
\mathcal{T}[\![X ==^d Y]\!]\theta_s & = & \{(X == Y, \theta_s)\} \\
\mathcal{T}[\![X \Rightarrow^d f(\overline{Y})]\!]\theta_s & = & \{(X \Rightarrow f(\overline{Y}), \theta_s)\} \\
\mathcal{T}[\![X :=^d Y]\!]\theta_s & = & \{(X := Y, \theta_s)\} \\
\mathcal{T}[\![X \Leftarrow^d f(\overline{Y})]\!]\theta_s & = & \{(X \Leftarrow f(\overline{Y}), \theta_s)\} \\
\mathcal{T}[\![p^d_S(\overline{X})]\!]\theta_s & = & \{(p_S(\overline{X}), \theta_s)\} \\
\mathcal{T}[\![X ==^s Y]\!]\theta_s & = & \begin{cases} \{(true, \theta)\} & \text{if } \mathcal{S}[\![X == Y]\!]\theta_s = \{\theta\} \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{T}[\![X \Rightarrow^s f(\overline{Y})]\!]\theta_s & = & \begin{cases} \{(true, \theta)\} & \text{if } \mathcal{S}[\![X \Rightarrow f(\overline{Y})]\!]\theta_s = \{\theta\} \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{T}[\![X :=^s Y]\!]\theta_s & = & \{(true, \theta)\} \text{ where } \{\theta\} = \mathcal{S}[\![X := Y]\!]\theta_s \\
\mathcal{T}[\![X \Leftarrow f(\overline{Y})]\!]\theta_s & = & \{(true, \theta)\} \text{ where } \{\theta\} = \mathcal{S}[\![X \Leftarrow f(\overline{Y})]\!]\theta_s \\
\mathcal{T}[\![p^s_S(\overline{X})]\!]\theta_s & = & \left\{((C, \overline{\rho^o_{B|D_1}}), \rho^o_{B|D_2}\theta') \;\middle|\; (C, \theta') \in \mathcal{T}[\![B\rho]\!]\rho_B\theta_s\right\} \\
 & & \text{where } B = \mathcal{B}ody(p) \\
 & & \text{and } D_1 = dom(\rho^o_B) \setminus dom(\theta') \\
 & & \text{and } D_2 = dom(\theta') \\
\mathcal{T}[\![not^d(G')]\!]\theta_s & = & \{(not(\phi(\mathcal{T}[\![G']\!]\theta_s)), \theta_s)\} \\
\mathcal{T}[\![not^s(G')]\!]\theta_s & = & \begin{cases} \{(true, \theta_s)\} & \text{if } \mathcal{T}[\![G']\!]\theta_s = \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{T}[\![if^d\ G_1\ then\, G_2\ else\, G_3]\!]\theta_s & = & \{(if\, \phi(\mathcal{T}[\![G_1]\!]\theta_s)\ then\, G_2\ else\, G_3, \theta_s)\} \\
\mathcal{T}[\![if^s\ G_1\ then\, G_2\ else\, G_3]\!]\theta_s & = & \left\{\begin{array}{l} \mathcal{T}[\![G_3]\!]\theta_s \text{ if } \mathcal{T}[\![G_1]\!]\theta_s = \emptyset \\ \left\{ ((C, D), \theta_s\gamma\eta) \;\middle|\; \begin{array}{l}(C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s \\ (D, \theta_s\gamma\eta) \in \mathcal{T}[\![G_2]\!]\theta_s\gamma\end{array} \right\} \end{array}\right. \\
\mathcal{T}[\![(G^a_1\ ,\ G^a_2)]\!]\theta_s & = & \left\{\begin{array}{l} \emptyset \text{ if } \mathcal{T}[\![G_1]\!]\theta_s = \emptyset \\ \left\{ ((C, D), \theta_s\gamma\eta) \;\middle|\; \begin{array}{l}(C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s \\ (D, \theta_s\gamma\eta) \in \mathcal{T}[\![G_2]\!]\theta_s\gamma\end{array} \right\} \end{array}\right. \\
\mathcal{T}[\![(G^a_1\ ;\ G^a_2)]\!]\theta_s & = & \mathcal{T}[\![G_1]\!]\theta_s \cup \mathcal{T}[\![G_2]\!]\theta_s
\end{array}
$$

Figure 3.13: The definition of $\mathcal{T}$.

approximates normal evaluation, the definition of $\mathcal{T}$ can be seen as a non standard interpreter for Mercury. A call to the reducer matches a single rule in $\mathcal{T}$'s definition which shows the residual code and substitutions resulting from reducing the goal. Reducing a dynamically annotated goal results in the corresponding residual goal, together with the current substitution. Reducing a statically an-

notated unification boils down to plain evaluation of the unification. Note that, although the substitution θ_s provides input to the semantic function $\mathcal{S}$, it is not necessarily an input substitution, in the sense that it might bind variables to partially instantiated terms. However, well-annotatedness ensures that the terms in θ_s are sufficiently instantiated to perform the unification. The result of reducing a statically annotated procedure call is the result of reducing its body goal (under the appropriate renaming substitutions). Each resulting residual goal is conjoined with the necessary code to associate the bindings constructed by the residual goal with the call's appropriate original output variables. Reducing a static negation results in failure if reducing the negated goal results in success, and vice versa. If reducing the test-goal of an if-then-else results in failure, the result of reducing the if-then-else equals the result of reducing the else-goal. If reducing the test-goal results in success (in the form of a set of pairs ($2^{Goal \times Subst}$), then the then-goal is reduced under the appropriate substitution and the results of reducing the test-goal and then-goal are combined appropriately into a disjunction of conjunctions of the residual goals. If reducing the first conjunct of a conjunction results in failure, reducing the complete conjunction also fails. Otherwise, the results from reducing both conjuncts are combined appropriately into a disjunction of conjunctions of the residual goals. The result of reducing a disjunction is defined as the union of reducing each disjunct separately.

The process of reducing a goal G^a can be depicted by a proof tree, being a generalised AND-OR tree in which the nodes are labelled by a $\mathcal{T}$-application. Since the $\mathcal{T}$-application of a dynamically annotated atom does not result in any more calls to the reducer, these $\mathcal{T}$-applications become leafs of the tree, as do the $\mathcal{T}$-applications of a statically annotated unification. The $\mathcal{T}$-application of a dynamically annotated structured goal (being either a negation or an if-then-else) has a single child node, being the root of the $\mathcal{T}$-tree that represents the reduction of the negated goal, respectively test-goal. A node representing the $\mathcal{T}$-application of a statically annotated procedure call has a single child node, being the root of the $\mathcal{T}$-tree representing the reduction of the procedure's body. Also the $\mathcal{T}$-application of a statically annotated negation results in a single child node, namely the root of the $\mathcal{T}$-tree representing the reduction of the negated goal. The $\mathcal{T}$-application of a statically annotated if-then-else results in an AND-node, of which the first child is the root of the $\mathcal{T}$-tree representing the reduction of the test-goal. If this $\mathcal{T}$-tree represents failure, the AND-node has a second child being the $\mathcal{T}$-tree representing the reduction of the else-goal. If, on the other hand, the $\mathcal{T}$-tree for the test-goal has n leafs representing success (thus having a residual goal that is either *true* or guaranteed to succeed at run-time), the AND-node has a second child which is an OR-node that combines the n $\mathcal{T}$-trees representing reduction of the then-goal, each time with respect to a different substitution resulting from the reduction of the test-goal. The $\mathcal{T}$-application of a conjunction also results in an AND-node, with a first child being the $\mathcal{T}$-tree representing the reduction of the first conjunct. If this

$\mathcal{T}$-tree has n leafs representing success of the first conjunct, the AND-node again has a second child analogously as in the case of an if-then-else. The $\mathcal{T}$-application of a disjunction results in an OR-node with two children: the $\mathcal{T}$-trees representing reduction of each disjunct. In what follows, we call the proof tree depicting the reduction of an annotated goal G with respect to a substitution θ the $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G]\!]\theta$.

Example 3.26 *Consider the* `append/3` *procedure with respect to a binding-time environment* $\pi_0 = \{X/\beta_l, Y/\bot\}$. *It can be verified that the annotated version of* `append/3` *with respect to* π_0 *corresponds with the annotated version from Example 3.24 (created with respect to the binding-time environment* $\{X/\beta_l, Y/\beta_l\}$*). Reducing the goal* $append(X, Y, Z)$ *with respect to the substitution* $\theta_s = \{X/[]\}$ *results in the* $\mathcal{T}$*-tree depicted in Fig. 3.14*

Figure 3.14: A sample $\mathcal{T}$-tree.

3.4.3 Correctness

In what follows, we establish an important result, proving correctness of our binding-time analysis with respect to the reduction process as defined above. Our exposition is along the lines of Gomard's work on partial evaluation of the Lambda Calculus (Gomard 1992). In our setting, correctness involves the following:

- termination of the reduction process (local termination).
- reduction preserves the semantics of the original program. Intuitively, this

ensures that one obtains the same result by running the program's computations in two stages as by an equivalent single stage computation.

Termination of the reduction process is guaranteed in case the code that is reduced does not contain a statically controlled infinite loop. In other words, reduction with respect to a substitution θ_s might not terminate, but only in case full evaluation does not terminate for every input substitution that is an instance of θ_s. This effect is due to the annotation behaviour: if the code contains a dynamically controlled loop, the annotations guarantee that the loop body (being under dynamic control) will not be reduced at all. Although this termination result is weaker than usual termination results of (on-line) local control strategies in partial deduction, it is generally accepted in the context of partial evaluation of functional programs (for example Similix (Bondorf and Jørgensen 1993)) exhibits the same (local) termination behaviour).

Before we can prove the equivalence of a two-stage computation with a single stage computation, we need to scrutinise the relation that must exist between the input of a single stage computation, and both inputs of the corresponding dual stage computation. A first requirement is that the single stage input *agrees* with the dual stage inputs (the terminology is from Gomard). In a logic programming context, this means that combining the two substitutions representing the staged input equals the input substitution of the single stage computation.

Definition 3.18 *Given $\theta, \theta_s, \theta_d \in Subst$. We say that θ agrees with θ_s and θ_d if and only if*

- $dom(\theta_d) \subseteq (dom(\theta) \setminus dom(\theta_s)) \cup range(\theta_s)$
- $\theta = (\theta_s\theta_d)_{|dom(\theta)}$

Example 3.27 *Reconsider the* `append/3` *procedure. The input substitution*

$$\{X/[a,b], Y/[c]\}$$

agrees with $\theta_s = \{X/[X_1, X_2]\}$ and $\theta_d = \{X_1/a, X_2/b, Y/[c]\}$.

A next requirement is that the early (static) input that is used when reducing a goal correctly corresponds with that goal's annotations. In other words, the static input must match with the binding-time environment for which the goal was annotated. Again using the terminology of (Gomard 1992), we introduce the notion of a binding-time environment *suiting* a substitution:

Definition 3.19 *A binding-time environment π* suits *a substitution θ if and only if*

$$dom(\pi) \supseteq dom(\theta) \text{ and } \forall X \in dom(\pi) \text{ holds that } \pi(X) \succeq \alpha(\theta(X))$$

where α is the binding-time abstraction defined in Definition 3.9.

Example 3.28 *The binding-time environment* $\{X/\beta_l, Y/\top\}$ *from Example 3.26 suits the substitution* θ_s *from Example 3.27.*

Well-annotatedness translates to the following property at the level of the reduction process: if π and π' represent respectively the binding-time environments at the entry and exit points of a well-annotated goal, and π suits a substitution θ_s, then π' suits any substitution $\theta_s\gamma$ that is the result of reducing the goal with respect to θ_s. Put differently, if the static input matches with the binding-time environment for which the goal was analysed (annotated), the result of reduction matches with the binding-time environment being the result of analysing the goal.

Proposition 3.3 *Consider a goal* G*, binding-time environment* π *and* G_a *being an annotated version of* G *with respect to* π*. Let* π' *be the binding-time environment resulting from analysing* G*, that is* $(\Psi, \pi') = \mathcal{G}[\![G_a]\!]\pi\Psi$ *given a congruent program environment* Ψ*. Then, if* θ_s *is a substitution such that* π *suits* θ_s*, it holds* $\forall(C, \theta_s\gamma) \in \mathcal{T}[\![G_a]\!]\theta_s$ *that* π' *suits* $\theta_s\gamma$*.*

Proof Let us consider the case where G is a deconstruction of the form $X \Rightarrow f(Y_1, \ldots, Y_n)$. The proof is analogous for the other kinds of unifications. Well-annotatedness ensures that if bindings are created by the analysis, the atom is annotated *static* and will be reduced by $\mathcal{T}$. In particular, well-annotatedness implies for the above atom that $X \in dom(\pi)$ and $\pi(X) = f(\beta_1, \ldots, \beta_n)$. Since π suits θ_s we have that $\pi(X) \succeq \alpha(\theta_s(X))$, and hence $\theta_s = f(t_1, \ldots, t_n)$ such that $\forall i : \beta_i \succeq \alpha(t_i)$. Now, well-modedness ensures that $\{Y_1, \ldots, Y_n\} \not\subseteq dom(\pi)$ and consequently $\{Y_1, \ldots, Y_n\} \not\subseteq dom(\theta_s)$. By definition of $\mathcal{G}$ (through $\mathcal{A}$), we have that $dom(\pi') = dom(\pi) \cup \{Y_1, \ldots, Y_n\}$ and $\forall i : \pi'(Y_i) = \beta_i$. By definition of $\mathcal{T}$, $dom(\theta_s\gamma) = dom(\theta_s) \cup \{Y_1, \ldots, Y_n\}$ and $\forall i : \theta_s\gamma(Y_i) = t_i$, and the result follows.

The proof for the other kinds of unifications is analogous, and the proof for the other goals is straightforward since a procedure call only introduces renamings and by the fact that $\mathcal{G}$ and $\mathcal{T}$ thread environments in exactly the same way during analysis or reduction of a structured goal.

□

The main correctness result we are about to prove states that, under given conditions, evaluating a goal with respect to an input substitution has the same result – in terms of failure and success with respect to an output substitution – as first reducing the appropriately annotated goal with respect to some partial input, and evaluating the specialised goal under the remaining input. Formally:

Theorem 3.2 *Given a program* P *and a congruent program environment* Ψ *for* P *such that* $\Psi \vdash P \rhd P_a$*. For every procedure* $p(\overline{F}) \leftarrow B \in P$ *and for every goal* G *identified by program point* η *in* B*, if*

- *G_a is an annotated version of G with respect to a binding-time environment π*
- *θ is an input substitution for G and θ agrees with substitutions θ_s and θ_d*
- *π suits θ_s*

then it holds that

$$\mathcal{S}[\![G]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d.$$

Note that correctness, as stated in Theorem 3.2 is independent of the order in which solutions for a goal are computed. This is allowed for Mercury since, being a pure language, the semantics of a program is independent of this order. Also note that the correctness statement assumes that the single stage as well as the dual stage computation terminates, since only then both sides of the equation are defined. The following proposition will be useful in the proof of Theorem 3.2. It states that the result of a two stage computation is invariant of the fact whether the first stage input is included in the second stage input, or not. Formally, if θ_s represent the first stage input and θ_d the second stage input, we have that $\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_s\theta_d = \mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d$.

Proposition 3.4 *Let G_a be an annotated goal and θ_s, θ_d as in the condition of Theorem 3.2. Then, we have that*

$$\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_s\theta_d = \mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d.$$

Proof The result of reduction, $\mathcal{T}[\![G_a]\!]\theta_s$, either is the empty set – denoting failure of G_a under θ_s – or a nonempty set of pairs $(C, \theta_s\gamma)$ representing a residual goal (possibly *true*) and substitution $\theta_s\gamma$. We consider each case separately.

1. First, assume $\mathcal{T}[\![G_a]\!]\theta_s = \emptyset$. In this case, we have that $\mathcal{S}[\![\mathit{fail}]\!]\theta_s\theta_d = \mathcal{S}[\![\mathit{fail}]\!]\theta_d = \emptyset$.
2. Next, assume that $\mathcal{T}[\![G_a]\!]\theta_s = \{(C_1, \theta_s\gamma_1), \ldots, (C_n, \theta_s\gamma_n)\}$ with $n \geq 1$. By definition of ϕ and $\mathcal{S}$ with respect to a disjunction, we have that

$$\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_s\theta_d \quad = \quad \left\{ \; \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![(\overline{\theta_s\gamma}, C)]\!]\theta_s\theta_d \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_a]\!]\theta_s \end{array} \right\}$$

   By the definition of $\mathcal{S}$ with respect to a conjunction and the fact that $\mathcal{S}[\![\overline{\theta_s\gamma}]\!]\theta_s\theta_d = \{\theta_s\gamma\theta_s\theta_d\}$ we rewrite the above into

$$\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_s\theta_d \quad = \quad \left\{ \; \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![C]\!]\theta_s\gamma\theta_s\theta_d \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_a]\!]\theta_s \end{array} \right\}$$

Now, γ represents the bindings constructed by reducing G_a with respect to θ_s: if a variable $X \in range(\gamma)$, then $X \notin dom(\theta_s)$ and consequently, $\theta_s\gamma\theta_s = \theta_s\gamma$ and the above becomes

$$\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_s\theta_d \quad = \quad \left\{ \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![C]\!]\theta_s\gamma\theta_d \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_a]\!]\theta_s \end{array} \right\}$$

which we can again transform into

$$\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_s\theta_d \quad = \quad \left\{ \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![(\overline{\theta_s\gamma}, C)]\!]\theta_d, \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_a]\!]\theta_s \end{array} \right\}$$

which becomes, using the definition of ϕ:

$$\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_s\theta_d \quad = \quad \mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d \ .$$

This concludes the proof.

□

In order to prove correctness of the analysis (Theorem 3.2), we first provide a number of lemma's that prove the result expressed by Theorem 3.2 in a number of well defined cases, given some conditions. Later on, we glue these results together by reasoning over the $\mathcal{T}$-tree build by the reduction. The correctness proof is inspired on Gomard's proof of semantics preserving reduction for LambdaMix (Gomard 1992). The proof is more involved, however, due to the fact that Mercury – being a logic programming language – deals with named procedures, non determinism and success or failure of the involved goals which is naturally reflected in the specialisation strategy and hence the correctness proof. We start by proving Theorem 3.2 in case the reduced goal is a unification.

Lemma 3.1 *Let G be an atomic goal of the form $X == Y$, $X \Rightarrow f(\overline{Y})$, $X \Leftarrow f(\overline{Y})$ or $X := Y$. Let G_a be an annotated version of G with respect to a binding-time environment π and let θ, θ_s and θ_d be substitutions as in the condition of Theorem 3.2. It then holds that*

$$\mathcal{S}[\![G]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d.$$

Proof Let us first consider the case where the atomic goal is annotated dynamic. In that case, by definition, $\mathcal{T}[\![G_a]\!]\theta_s = \{(G, \theta_s)\}$, and $\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![\ (\overline{\theta_s}\ ,\ G)\ ]\!]\theta_d$ by definition of ϕ. Using the definition of $\mathcal{S}$, we then derive that

$$\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d = \bigcup_{\sigma \in \mathcal{S}[\![\overline{\theta_s}]\!]\theta_d} \mathcal{S}[\![G]\!]\sigma$$

Now, since $\mathcal{S}[\![\overline{\theta_s}]\!]\theta_d = \{\theta_s\theta_d\}$, we have that

$$\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![G]\!]\theta_s\theta_d = \mathcal{S}[\![G]\!]\theta.$$

Let us now consider the case where the atomic goal is annotated static. Since G_a is a statically annotated unification, we have that $\mathcal{T}[\![G_a]\!]\theta_s = \mathcal{S}[\![G]\!]\theta_s$. The latter either equals the empty set (denoting failure) or a singleton of the form $\theta_s\sigma$ where σ represents the (possibly empty) update to θ_s originating from the unification. We consider each case separately.

1. First, assume that $\mathcal{S}[\![G]\!]\theta_s = \emptyset$. Then also $\mathcal{S}[\![G]\!]\theta_s\theta_d = \mathcal{S}[\![G]\!](\theta_s\theta_d) = \mathcal{S}[\![G]\!]\theta = \emptyset$, and hence $\mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![fail]\!]\theta_d = \emptyset = \mathcal{S}[\![G]\!]\theta$.
2. Now, assume $\mathcal{S}[\![G]\!]\theta_s = \{\theta_s\sigma\}$. Since $dom(\sigma) = \mathtt{out}(G)$, $dom(\sigma) \cap range(\theta_s) = \emptyset$ and we have that $\theta_s\sigma = \theta_s \cup \sigma$ and consequently $\{\theta_s\sigma\theta_d\}_{|\mathcal{V}(G)} = \{\theta_s\theta_d \cup \sigma\theta_d\}_{|\mathcal{V}(G)} = \{\theta \cup \sigma\theta_d\}_{|\mathcal{V}(G)}$. Next, assume G is of the form $X \Rightarrow f(Y_1, \ldots, Y_n)$. The proof is analogous for the other unifications. We then have that $\sigma = \{Y_1/t_1, \ldots, Y_n/t_n\}$ where $X/f(t_1, \ldots, t_n) \in \theta_s$ and $\mathcal{S}[\![G]\!]\theta = \theta\{Y_1/s_1, \ldots, Y_n/s_n\}$ where $X/f(s_1, \ldots, s_n) \in \theta$. Since θ agrees with θ_s and θ_d, we have that $\forall i : s_i = t_i\theta_d$ and $\mathcal{S}[\![G]\!]\theta = \theta\{Y_1/t_1\theta_d, \ldots, Y_n/t_n\theta_d\} = \{\theta \cup \sigma\theta_d\}_{|\mathcal{V}(G)}$, concluding the proof.

□

For structured goals, we will prove Theorem 3.2 by induction on the structure of the goal. We start by proving the induction step: if the property expressed by Theorem 3.2 holds for the subgoals of a structured goal, the property also holds for the structured goal itself. To ease notation, we introduce, as in (Gomard 1992), a name $\mathcal{H}$ for this property: for a goal G, $\mathcal{H}(G)$ expresses that if the conditions of Theorem 3.2 are satisfied with respect to a goal G and substitutions θ, θ_s and θ_d, then $\mathcal{S}[\![G]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d$, for G_a an annotated version of G with respect to a binding-time environment that suits θ_s. In what follows, we prove the induction step in a separate lemma for each of the structured goals.

Lemma 3.2 *Let G be a goal of the form $not(G')$. If G_a is an annotated version of G with respect to a binding-time environment π and θ, θ_s and θ_d substitutions as in the condition of Theorem 3.2, then if $\mathcal{H}(G')$ holds, then it holds that*

$$\mathcal{S}[\![G]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d.$$

Proof Let us first consider the case where the negation is annotated to residualise. By definition, $\mathcal{T}[\![not^d(G')]\!]\theta_s = \{(not(\phi(\mathcal{T}[\![G']\!]\theta_s)), \theta_s)\}$, and hence we have that

$$\begin{aligned}
\mathcal{S}[\![\phi(\mathcal{T}[\![not^d(G')]\!]\theta_s)]\!]\theta_d &= \mathcal{S}[\![\phi(\{(not(\phi(\mathcal{T}[\![G']\!]\theta_s)), \theta_s)\})]\!]\theta_d \\
&= \mathcal{S}[\![(\overline{\theta_s}, not(\phi(\mathcal{T}[\![G']\!]\theta_s)))]\!]\theta_d \\
&= \mathcal{S}[\![not(\phi(\mathcal{T}[\![G']\!]\theta_s))]\!]\theta_s\theta_d
\end{aligned}$$

by the definitions of ϕ and $\mathcal{S}$ on a conjunction combined with the fact that $\mathcal{S}[\![\overline{\theta_s}]\!]\theta_d = \theta_s\theta_d$. Now, by definition of $\mathcal{S}$ of a negation, we have that

$$\mathcal{S}[\![not(\phi(\mathcal{T}[\![G']\!]\theta_s))]\!]\theta_s\theta_d = \begin{cases} \{\theta_s\theta_d\} & \text{if } \mathcal{S}[\![\phi(\mathcal{T}[\![G']\!]\theta_s)]\!]\theta_s\theta_d = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

Due to Proposition 3.4, we have $\mathcal{S}[\![\phi(\mathcal{T}[\![G']\!]\theta_s)]\!]\theta_s\theta_d = \mathcal{S}[\![\phi(\mathcal{T}[\![G']\!]\theta_s)]\!]\theta_d$. By the induction hypothesis, $\mathcal{H}(G')$, the latter equals $\mathcal{S}[\![G']\!]\theta$ and the above right hand side can be rewritten into

$$\begin{cases} \{\theta\} & \text{if } \mathcal{S}[\![G']\!]\theta = \emptyset \\ \emptyset & \text{otherwise} \end{cases} \quad = \quad \mathcal{S}[\![not(G')]\!]\theta$$

by definition of $\mathcal{S}$ with respect to a negation, which concludes this case of the proof.

Now we prove the second case, where the negation is annotated reducible. Reduction of G' either results in failure (indicated by the empty set), in success or in some residual code (the latter cases both indicated by a nonempty set). We consider each case separately:

1. Assume $\mathcal{T}[\![G']\!]\theta_s = \emptyset$. By definition of $\mathcal{T}$, it follows that $\mathcal{T}[\![not^s(G')]\!]\theta_s = \{(true, \theta_s)\}$. In this case, we have that

   $$\mathcal{S}[\![\phi(\mathcal{T}[\![not(G')]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![\overline{\theta_s}]\!]\theta_d = \{\theta_s\theta_d\} = \{\theta\}.$$

   Now, proof follows if also $\mathcal{S}[\![not(G')]\!]\theta = \{\theta\}$ which is (by definition of $\mathcal{S}$) the case if $\mathcal{S}[\![G']\!]\theta = \emptyset$. And indeed, since G' is annotated with respect to the same binding-time environment as the goal $not(G')$, we have that θ, θ_s and θ_d also satisfy the conditions of Theorem 3.2 with respect to the goal G'. Moreover $\mathcal{H}(G')$ holds, meaning that $\mathcal{S}[\![G']\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G']\!]\theta_s)]\!]\theta_d$ and we have that

   $$\mathcal{S}[\![G']\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G']\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![fail]\!]\theta_d = \emptyset$$

   by the assumption that $\mathcal{T}[\![G']\!]\theta_s = \emptyset$.

2. The proof is analogously in case we assume that $\mathcal{T}[\![G']\!]\theta_s = S$ with S being a non empty set. By definition of $\mathcal{T}$, it follows that $\mathcal{T}[\![not^s(G')]\!]\theta_s = \emptyset$. In this case, we have that

   $$\mathcal{S}[\![\phi(\mathcal{T}[\![not(G')]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![fail]\!]\theta_d = \emptyset.$$

   Now, proof follows if also $\mathcal{S}[\![not(G')]\!]\theta = \emptyset$ which is (by definition of $\mathcal{S}$) the case if $\mathcal{S}[\![G']\!]\theta \neq \emptyset$. And indeed, since G' is annotated with respect to the same binding-time environment as the goal $not(G')$, we have that θ, θ_s and θ_d also satisfy the conditions of Theorem 3.2

with respect to the goal G'. Moreover $\mathcal{H}(G')$ holds, meaning that $\mathcal{S}[\![G']\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G']\!]\theta_s)]\!]\theta_d$, and we have that

$$\mathcal{S}[\![G']\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G']\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![\phi(S)]\!].$$

Since we assume $not^s(G')$ to be well-annotated, S – being the result of reducing the negated goal – is such that further evaluating $\phi(S)$ does not result in failure, that is $\mathcal{S}[\![\phi(S)]\!]\theta_d \neq \emptyset$, which concludes the proof.

□

Lemma 3.3 *Let G be a goal of the form $if\, G_1\, then\, G_2\, else\, G_3$. If G_a is an annotated version of G with respect to a binding-time environment π and θ, θ_s and θ_d substitutions as in the condition of Theorem 3.2, then if $\mathcal{H}(G_1)$, $\mathcal{H}(G_2)$ and $\mathcal{H}(G_3)$ holds, then it holds that*

$$\mathcal{S}[\![G]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d.$$

Proof We first consider the case where the if-then-else goal is annotated to residualise. By definition,

$$\mathcal{T}[\![if^d\, G_1\, then\, G_2\, else\, G_3]\!]\theta_s = \{(if\, \phi(\mathcal{T}[\![G_1]\!]\theta_s)\, then\, G_2\, else\, G_3), \theta_s)$$

and hence we have that

$$\begin{aligned}
&\mathcal{S}[\![\phi(\mathcal{T}[\![if^d\, G_1\, then\, G_2\, else\, G_3]\!]\theta_s)]\!]\theta_d \\
&= \mathcal{S}[\![\phi(\{(if\, \phi(\mathcal{T}[\![G_1]\!]\theta_s)\, then\, G_2\, else\, G_3, \theta_s)\})]\!]\theta_d \\
&= \mathcal{S}[\![if\, \phi(\mathcal{T}[\![G_1]\!]\theta_s)\, then\, G_2\, else\, G_3]\!]\theta_s\theta_d
\end{aligned}$$

by the definitions of ϕ and $\mathcal{S}$ on a conjunction combined with the fact that $\mathcal{S}[\![\overline{\theta_s}]\!]\theta_d = \theta_s\theta_d$. Now, by the fact that $\theta_s\theta_d = \theta$ and the definition of $\mathcal{S}$, the above equals $\mathcal{S}[\![if G_1\, then G_2\, else G_3]\!]\theta$ if $\mathcal{S}[\![G_1]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G_1]\!]\theta_s)]\!]\theta$. This is indeed the case, since the latter equals $\mathcal{S}[\![\phi(\mathcal{T}[\![G_1]\!]\theta_s)]\!]\theta_d$ by Proposition 3.4. Moreover, G_1 is annotated with respect to the same binding-time environment as G, hence θ, θ_s and θ_d also satisfy the condition of Theorem 3.2 with respect to G_1 and $\mathcal{H}(G_1)$ holds.

Next, we consider the case where the if-then-else goal is annotated reducible. In this case, the test goal is reduced which can result in failure (indicated by the empty set), in success or in some residual code (the latter cases both indicated by a nonempty set). We consider each case separately:

1. Assume that $\mathcal{T}[\![G_1]\!]\theta_s = \emptyset$. By definition of $\mathcal{T}$, it follows that

$$\mathcal{T}[\![if\, G_1\, then\, G_2\, else\, G_3]\!]\theta_s = \mathcal{T}[\![G_3]\!]\theta_s.$$

In this case, we have that

$$\mathcal{S}[\![\phi(\mathcal{T}[\![\mathit{if}\, G_1\ \mathit{then}\, G_2\ \mathit{else}\, G_3]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![\phi(\mathcal{T}[\![G_3]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![G_3]\!]\theta$$

since G_3 is annotated with respect to the same binding-time environment as G, hence θ, θ_s and θ_d also satisfy the conditions of Theorem 3.2 with respect to G_3 and $\mathcal{H}(G_3)$ holds. Now, proof follows if also $\mathcal{S}[\![\mathit{if}\, G_1\ \mathit{then}\, G_2\ \mathit{else}\, G_3]\!]\theta = \mathcal{S}[\![G_3]\!]\theta$, which is (by definition of $\mathcal{S}$) the case if $\mathcal{S}[\![G_1]\!]\theta = \{\}$. And indeed, since θ, θ_s and θ_d also satisfy the conditions of Theorem 3.2 with respect to the goal G_1 and $\mathcal{H}(G_1)$ holds, we have that

$$\mathcal{S}[\![G_1]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G_1]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![\phi(\emptyset)]\!]\theta_d = \mathcal{S}[\![\mathit{fail}]\!]\theta_d = \emptyset.$$

2. Assume that $\mathcal{T}[\![G_1]\!]\theta_s = S \neq \emptyset$. In this case, we have by definition of $\mathcal{S}$

$$\mathcal{S}[\![\mathit{if} G_1\ \mathit{then} G_2\ \mathit{else} G_3]\!]\theta = \{\theta' \mid \theta' \in \mathcal{S}[\![G_2]\!]\sigma' \text{ and } \sigma' \in \mathcal{S}[\![G_1]\!]\theta\} \quad (3.3)$$

Now, θ agrees with θ_s and θ_d and satisfies the conditions of Theorem 3.2 with respect to G_1. Since $\mathcal{H}(G_1)$ holds, we have that

$$\begin{aligned} \mathcal{S}[\![G_1]\!]\theta &= \mathcal{S}[\![\phi(\mathcal{T}[\![G_1]\!]\theta_s)]\!]\theta_d \\ &= \left\{ \theta_s\gamma\theta_d\sigma \;\middle|\; \begin{array}{l} \theta_s\gamma\theta_d\sigma \in \mathcal{S}[\![C]\!]\theta_s\gamma\theta_d \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s \end{array} \right\} \end{aligned}$$

Indeed, since $\phi(\mathcal{T}[\![G_1]\!]\theta_s)$ is a disjunction with each disjunct of the form $(\overline{\theta_s\gamma}, C)$ (being an element of $\mathcal{T}[\![G_1]\!]$), $\mathcal{S}[\![\phi(\mathcal{T}[\![G_1]\!]\theta_s)]\!]\theta_d$ is the union of $\mathcal{S}$-applications of the individual disjuncts. For such a particular disjunct, $\mathcal{S}[\![(\overline{\theta_s\gamma}, C)]\!]\theta_d = \mathcal{S}[\![C]\!]\theta_s\gamma\theta_d$ by definition of $\mathcal{S}$ with respect to a conjunction and the fact that $\mathcal{S}[\![\overline{\theta_s\gamma}]\!]\theta_d = \{\theta_s\gamma\theta_d\}$. Hence, we can rewrite (3.3) into

$$\mathcal{S}[\![\mathit{if} G_1\ \mathit{then} G_2\ \mathit{else} G_3]\!]\theta = \left\{ \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![G_2]\!]\theta_s\gamma\theta_d\sigma, \\ \theta_s\gamma\theta_d\sigma \in \mathcal{S}[\![C]\!]\theta_s\gamma\theta_d, \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s \end{array} \right\} \quad (3.4)$$

In (3.4), each such $\theta_s\gamma\theta_d\sigma$ agrees with $\theta_s\gamma$ and $\theta_d\sigma$ and, by Proposition 3.3 they satisfy the condition of Theorem 3.2 with respect tot the goal G_2. Indeed, being the result of reducing G_1, $\theta_s\gamma$ is such that the binding-time environment that results from analysing G_1, which is the same binding-time environment with respect to which G_2 is annotated, suits $\theta_s\gamma$. Moreover, since $\mathcal{H}(G_2)$ holds, we have that

$$\begin{aligned} \mathcal{S}[\![G_2]\!]\theta_s\gamma\theta_d\sigma &= \mathcal{S}[\![\phi(\mathcal{T}[\![G_2]\!]\theta_s\gamma)]\!]\theta_d\sigma \\ &= \bigcup_{(D, \theta_s\gamma\eta) \in \mathcal{T}[\![G_2]\!]\theta_s\gamma} \mathcal{S}[\![D]\!]\theta_s\gamma\eta\theta_d\sigma \end{aligned}$$

and we rewrite (3.4) into

$$\mathcal{S}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\theta = \left\{ \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![D]\!]\theta_s\gamma\eta\theta_d\sigma, \\ (D, \theta_s\gamma\eta) \in \mathcal{T}[\![G_2]\!]\theta_s\gamma, \\ \theta_s\gamma\theta_d\sigma \in \mathcal{S}[\![C]\!]\theta_s\gamma\theta_d, \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s \end{array} \right\} \quad (3.5)$$

Now, since $(D, (\theta_s\gamma)\eta) \in \mathcal{T}[\![G_2]\!](\theta_s\gamma)$ and the program is well-moded, η does not instantiate variables from the range of $(\theta_s\gamma)$ (and vice versa) and we have that $(\theta_s\gamma)\eta = \eta(\theta_s\gamma)$ and consequently

$$\mathcal{S}[\![D]\!]\theta_s\gamma\eta\theta_d\sigma = \mathcal{S}[\![D]\!]\eta\theta_s\gamma\theta_d\sigma = \mathcal{S}[\![(\overline{\eta}, D)]\!]\theta_s\gamma\theta_d\sigma$$

Substituting this result into (3.5) results in

$$\mathcal{S}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\theta = \left\{ \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![(\overline{\eta}, D)]\!]\theta_s\gamma\theta_d\sigma, \\ \theta_s\gamma\theta_d\sigma \in \mathcal{S}[\![C]\!]\theta_s\gamma\theta_d, \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s, \\ (D, \theta_s\gamma\eta) \in \mathcal{T}[\![G_2]\!]\theta_s\gamma, \end{array} \right\}$$

which can, using the definition of $\mathcal{S}$ with respect to a conjunction be rewritten into

$$\mathcal{S}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\theta = \left\{ \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![(C, (\overline{\eta}, D))]\!]\theta_s\gamma\theta_d, \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s, \\ (D, \theta_s\gamma\eta) \in \mathcal{T}[\![G_2]\!]\theta_s\gamma, \end{array} \right\}$$

And again into

$$\mathcal{S}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\theta = \left\{ \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![(\overline{\theta_s\gamma}, (C, (\overline{\eta}, D)))]\!]\theta_d, \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s, \\ (D, \theta_s\gamma\eta) \in \mathcal{T}[\![G_2]\!]\theta_s\gamma, \end{array} \right\}$$

Since $dom(\eta) \subseteq \mathtt{out}(G_2)$ and the goal is well-moded, $dom(\eta) \cap \mathcal{V}(C) = \emptyset$ and we can reorder the conjunction $(C, (\overline{\eta}, D))$ into $(\overline{\eta}, (C, D))$ which results in

$$\mathcal{S}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\theta = \left\{ \theta' \;\middle|\; \begin{array}{l} \theta' \in \mathcal{S}[\![(\overline{\theta_s\gamma\eta}, (C, D))]\!]\theta_d, \\ (C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s, \\ (D, \theta_s\gamma\eta) \in \mathcal{T}[\![G_2]\!]\theta_s\gamma, \end{array} \right\}$$

Using the definition of ϕ, we can rewrite the right hand side of the above into

$$\mathcal{S}[\![\phi(\left\{ ((C, D), \theta_s\gamma\eta) \;\middle|\; \begin{array}{l} (C, \theta_s\gamma) \in \mathcal{T}[\![G_1]\!]\theta_s, \\ (D, \theta_s\gamma\eta) \in \mathcal{T}[\![G_2]\!]\theta_s\gamma, \end{array} \right\})]\!]\theta_d$$

Due to the definition of $\mathcal{T}$ with respect to an if-then-else, we obtain

$$\mathcal{S}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, G_3]\!]\theta_s)]\!]\theta_d$$

concluding the proof.

□

The following two lemmas prove the induction step in case of a conjunction and a disjunction. Recall that both constructs only appear in a static variant.

Lemma 3.4 *Let G be a goal of the form (G_1, G_2). Let $(G_1^a, G_2^a)^s$ be an annotated version of this goal with respect to a binding-time environment π and θ, θ_s and θ_d substitutions as in the condition of Theorem 3.2, then if $\mathcal{H}(G_1)$ and $\mathcal{H}(G_2)$ hold, then it holds that*

$$\mathcal{S}[\![(G_1, G_2)]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![(G_1^a, G_2^a)^s]\!]\theta_s)]\!]\theta_d$$

Proof By definition of $\mathcal{S}$ and $\mathcal{T}$, it holds that

$$\begin{aligned} &\mathcal{S}[\![(G_1, G_2)]\!]\theta = \mathcal{S}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, \mathit{fail}]\!]\theta \\ \text{and} \quad &\mathcal{T}[\![(G_1^a, G_2^a)^s]\!]\theta_s = \mathcal{T}[\![\mathit{if}\, G_1^a\, \mathit{then}\, G_2^a\, \mathit{else}\, \mathit{fail}]\!]\theta_s \end{aligned} \tag{3.6}$$

By Lemma 3.3, we have that

$$\mathcal{S}[\![\mathit{if}\, G_1\, \mathit{then}\, G_2\, \mathit{else}\, \mathit{fail}]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![\mathit{if}\, G_1^a\, \mathit{then}\, G_2^a\, \mathit{else}\, \mathit{fail}]\!]\theta_s)]\!]\theta_d. \tag{3.7}$$

Combining (3.6) and (3.7) results in

$$\mathcal{S}[\![(G_1, G_2)]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![(G_1^a, G_2^a)^s]\!]\theta_s)]\!]\theta_d$$

which concludes the proof.

□

Lemma 3.5 *Let G be a goal of the form $(G_1; G_2)$. Let $(G_1^a; G_2^a)^s$ be an annotated version of this goal with respect to a binding-time environment π and θ, θ_s and θ_d substitutions as in the condition of Theorem 3.2, then if $\mathcal{H}(G_1)$ and $\mathcal{H}(G_2)$ hold, then it holds that*

$$\mathcal{S}[\![(G_1; G_2)]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![(G_1^a; G_2^a)^s]\!]\theta_s)]\!]\theta_d$$

Proof By definition of $\mathcal{S}$ with respect to a disjunction, we have that

$$\mathcal{S}[\![(G_1\ ;\ G_2)]\!]\theta = \mathcal{S}[\![G_1]\!]\theta \cup \mathcal{S}[\![G_2]\!]\theta.$$

Now, since θ agrees with θ_s and θ_d and they satisfy the condition of Theorem 3.2 with respect to both G_1 and G_2 and $\mathcal{H}(G_1)$ and $\mathcal{H}(G_2)$ hold, we rewrite the above into

$$\begin{aligned}\mathcal{S}[\![(G_1\ ;\ G_2)]\!]\theta &= \mathcal{S}[\![\phi(\mathcal{T}[\![G_1^a]\!]\theta_s)]\!]\theta_d \cup \mathcal{S}[\![\phi(\mathcal{T}[\![G_2^a]\!]\theta_s)]\!]\theta_d \\ &= \mathcal{S}[\![(\phi(\mathcal{T}[\![G_1^a]\!]\theta_s)\ ;\ \phi(\mathcal{T}[\![G_2^a]\!]\theta_s)]\!]\theta_d\end{aligned}$$

again by the definition of $\mathcal{S}$ with respect to a disjunction. Now, the result of applying ϕ is a disjunction. By definition of ϕ, the above again rewrites to

$$\begin{aligned}\mathcal{S}[\![(G_1\ ;\ G_2)]\!]\theta &= \mathcal{S}[\![\phi(\mathcal{T}[\![G_1^a]\!]\theta_s \cup \mathcal{T}[\![G_2^a]\!]\theta_s)]\!]\theta_d \\ &= \mathcal{S}[\![\phi(\mathcal{T}[\![(G_1^a\ ;\ G_2^a)]\!]\theta_s)]\!]\theta_d\end{aligned}$$

by definition of $\mathcal{T}$ with respect to a disjunction.

□

Together, lemma's 3.2 through 3.5 prove the induction step in the proof of Theorem 3.2: if $\mathcal{H}(G')$ holds for any subgoal G' of a structured goal G, also $\mathcal{H}(G)$ holds. In what follows, we will complete the proof of Theorem 3.2 for any goal G by reasoning over the $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G]\!]\theta_s$, first proving that the theorem holds for the leaves of the tree, and by induction (using lemma's 3.2 through 3.5) that it also holds for any other node in the tree. However, the crucial point in the proof remains the handling of procedure calls. We therefore define first the auxiliary notion of a *partial $\mathcal{T}$-tree*, which is a $\mathcal{T}$-tree in which no procedure calls are reduced.

Definition 3.20 *For a goal G and substitution θ_s as in the condition of Theorem 3.2, we define the* partial $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G]\!]\theta_s$ *as the largest $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G]\!]\theta_s$ such that if the tree contains a $\mathcal{T}$-application of a procedure call, this node is a leaf node of the tree.*

As a consequence of its definition, a partial $\mathcal{T}$-tree rooted in a node that represents a procedure call has this root node as its only node. As the last preparation for the final proof of Theorem 3.2, we prove the following lemma, relating the induction hypothesis on several goals, as long as they are not related through procedure calls.

Lemma 3.6 *For any goal G. If G_a is an annotated version of this goal and θ_s a substitution as in the condition of Theorem 3.2, then if $\mathcal{H}(L)$ holds for any goal L in a leaf of the partial $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G_a]\!]\theta_s$, then also $\mathcal{H}(G)$ holds.*

Proof If $\mathcal{H}(L)$ holds for any goal L in a leaf of the partial $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G_a]\!]\theta_s$, $\mathcal{H}(G')$ holds for any goal G' in the partial $\mathcal{T}$-tree (including G),

since G' is composed of (a subset of) the goals in the leaf nodes by conjunctions, disjunctions, if-then-elses or negations only. Proof follows by repeatedly applying Lemma's 3.2 through 3.5 based on the structure of G'. □

Note that a complete $\mathcal{T}$-tree can be completely covered by a number of partial $\mathcal{T}$-trees that are connected by means of the reduction of a predicate call. The last auxiliary notion we introduce relates a goal G in a complete $\mathcal{T}$-tree to the maximal number of $\mathcal{T}$-applications of a procedure call in any branch of the complete $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G]\!]\theta_s$. Formally, we define the notion of the *call height* of a node in a $\mathcal{T}$-tree as follows:

Definition 3.21 *Given a goal G and a substitution θ_s as in the condition of Theorem 3.2. The* call height *of the node $\mathcal{T}[\![G]\!]\theta_s$ (or simply G) is defined as*

$$callh(\mathcal{T}[\![G]\!]\theta_s) = b + max \left\{ callh(\mathcal{T}[\![G']\!]\theta') \;\middle|\; \begin{array}{l} \mathcal{T}[\![G']\!]\theta' \text{ is a direct descendant} \\ \text{of } \mathcal{T}[\![G]\!]\theta_s \text{ in the } \mathcal{T}\text{-tree} \\ \text{rooted in } \mathcal{T}[\![G]\!]\theta_s \end{array} \right\}$$

where

$$b = \begin{cases} 1 & \text{in case } G \text{ is of the form } p_S(\overline{X}) \\ 0 & \text{otherwise} \end{cases}$$

The proof of Theorem 3.2 is now straightforward. We prove that $\mathcal{H}(G)$ holds for any goal G and substitution θ_s as in the condition of Theorem 3.2 by induction on the call height of G.

Proof (Theorem 3.2)

First, let us assume that $callh(G) = 0$. In this case, the partial $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G]\!]\theta_s$ does not contain a $\mathcal{T}$-application of a procedure call at all. The proof is immediate since the leafs of the tree must be unifications for which the property $\mathcal{H}$ holds by Lemma 3.1, and it follows by Lemma 3.6 that $\mathcal{H}(G)$ also holds.

Now, suppose that $\mathcal{H}(G')$ holds for any goal G' with $callh(G') < k$ for some $k \in \mathbb{N}$. We prove that $\mathcal{H}(G)$ holds when $callh(G) = k$. Consider the partial $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G]\!]\theta_s$. The leafs of this partial $\mathcal{T}$-tree are either $\mathcal{T}$-applications of unifications, or $\mathcal{T}$-applications of a procedure call G'' with $callh(G'') \leq k$. In either case, for such a leaf L, we have that $\mathcal{H}(L)$ holds:

- if L represents a unification, $\mathcal{H}(L)$ holds by Lemma 3.1.
- Assume L is a procedure call with $callh(L) < k$. By induction hypothesis, we have that $\mathcal{H}(L)$ holds.
- Assume L is a procedure call with $callh(L) = k$.

1. Either L is a dynamically annotated call, $p_S^d(\overline{X})$ (only in case $k = 1$). By definition, $\mathcal{T}[\![p_S^d(\overline{X})]\!]\theta_s = \{(p_S(\overline{X}), \theta_s)\}$ and, for θ_d and θ as in the condition of Theorem 3.2,

$$\mathcal{S}[\![\phi(\mathcal{T}[\![p_S^d(\overline{X})]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![(\overline{\theta_s}\ ,\ p_S(\overline{X}))]\!]\theta_d.$$

by definition of ϕ. Using the definition of $\mathcal{S}$, we then derive that

$$\mathcal{S}[\![\phi(\mathcal{T}[\![p_S^d(\overline{X})]\!]\theta_s)]\!]\theta_d = \bigcup_{\sigma\ \in\ \mathcal{S}[\![\overline{\theta_s}]\!]\theta_d} \mathcal{S}[\![p_S(\overline{X})]\!]\sigma\ .$$

Now, since $\mathcal{S}[\![\overline{\theta_s}]\!]\theta_d = \{\theta_s\theta_d\} = \{\theta\}$, we have that

$$\mathcal{S}[\![\phi(\mathcal{T}[\![p_S^d(\overline{X})]\!]\theta_s)]\!]\theta_d = \mathcal{S}[\![p_S(\overline{X})]\!]\theta_s\theta_d = \mathcal{S}[\![p_S(\overline{X})]\!]\theta.$$

2. Or L is a statically annotated call $p_S^s(\overline{X})$. The partial $\mathcal{T}$-tree that is rooted in $\mathcal{T}[\![G]\!]\theta_s$'s only child node, $\mathcal{T}[\![B\rho]\!]\rho_B\theta_s$ is such that $callh(B\rho) < k$ and by induction hypothesis, we have that $\mathcal{H}(B\rho)$ holds with respect to $\rho_B\theta_s$. Moreover, by definition of $\mathcal{T}$, we have that $\mathcal{S}[\![\phi(\mathcal{T}[\![p_S^s(\overline{X})]\!]\theta_s)]\!]\theta_d$ equals

$$\mathcal{S}[\![\phi(\Big\{\ ((C, \overline{\rho^o_{B|D_1}}), \rho^o_{B|D_2}\theta')\ \Big|\ (C,\theta') \in \mathcal{T}[\![B\rho]\!]\rho_B\theta_s\ \Big\})]\!]\theta_d$$

where $D_1 = dom(\rho^o_B) \setminus dom(\theta')$ and $D_2 = dom(\theta')$. The above can be rewritten, by definition of ϕ, into the following set:

$$\left\{\theta''\ \middle|\ \theta'' \in \mathcal{S}[\![(\overline{\rho^o_{B|D_2}\theta'}, (C, \overline{\rho^o_{B|D_1}}))]\!]\theta_d \text{ and } (C,\theta') \in \mathcal{T}[\![B\rho]\!]\rho_B\theta_s\right\}.$$

Using the definition of $\mathcal{S}$ with respect to a conjunction and the fact that $\mathcal{S}[\![\overline{\rho^o_{B|D_2}\theta'}]\!]\theta_d = \rho^o_{B|D_2}\theta'\theta_d$, the above equals

$$\left\{\ \theta''\ \middle|\ \theta'' \in \mathcal{S}[\![(C, \overline{\rho^o_{B|D_1}})]\!]\rho^o_{B|D_2}\theta'\theta_d \text{ and } (C,\theta') \in \mathcal{T}[\![B\rho]\!]\rho_B\theta_s\ \right\}$$

and, again using the definition of $\mathcal{S}$ with respect to a conjunction, into

$$\left\{\ \theta''\ \middle|\ \begin{array}{l}\theta'' \in \mathcal{S}[\![\overline{\rho^o_{B|D_1}}]\!]\theta''' \text{ where}\\ \theta''' \in \mathcal{S}[\![C]\!]\rho^o_{B|D_2}\theta'\theta_d \text{ and } (C,\theta') \in \mathcal{T}[\![B\rho]\!]\rho_B\theta_s\end{array}\ \right\}.$$

Now, since θ''' is of the form $\rho^o_{B|D_2}\theta'\theta_d\sigma$, we have that

$$\mathcal{S}[\![\overline{\rho^o_{B|D_1}}]\!]\theta''' = \rho^o_B\theta'\theta_d\sigma = \rho^o_B(\rho^o_{B|D_2}\theta'\theta_d\sigma) = \rho^o_B\theta'''$$

and by the definition of $\mathcal{S}$ with respect to a conjunction, the above equals

$$\left\{ \begin{array}{l|l} \rho^o_B\theta''' & \theta''' \in \mathcal{S}[\![(\overline{\rho^o_{B|D_2}}, C)]\!]\theta'\theta_d \\ & \text{where } (C,\theta') \in \mathcal{T}[\![B\rho]\!]\rho_B\theta_s \end{array} \right\}.$$

Employing the fact that $\theta'\theta_d = \mathcal{S}[\![\overline{\theta'}]\!]\theta_d$ and again applying the definition of $\mathcal{S}$ with respect to a conjunction, we rewrite the above into

$$\left\{ \begin{array}{l|l} \rho^o_B\theta''' & \theta''' \in \mathcal{S}[\![(\overline{\theta'}, (\overline{\rho^o_{B|D_2}}, C))]\!]\theta_d \\ & \text{and } (C,\theta') \in \mathcal{T}[\![B\rho]\!]\rho_B\theta_s \end{array} \right\}$$

which, by definition of ϕ equals

$$\left\{ \rho^o_B\theta''' \;\middle|\; \theta''' \in \mathcal{S}[\![\phi(\mathcal{T}[\![B\rho]\!]\rho_B\theta_s)]\!]\theta_d \right\}.$$

Applying the induction hypothesis, $\mathcal{H}(B\rho)$ results into

$$\left\{ \rho^o_B\theta''' \;\middle|\; \theta''' \in \mathcal{S}[\![B\rho]\!]\rho_B\theta \right\} = \mathcal{S}[\![p^s_S(\overline{X})]\!]\theta$$

by definition and concludes the proof.

Having shown that $\mathcal{H}(L)$ holds for each of the leafs of the partial $\mathcal{T}$-tree rooted in $\mathcal{T}[\![G]\!]\theta_s$, we can conclude (using Lemma 3.6) that also $\mathcal{H}(G)$ holds.

□

3.5 Discussion

In this Chapter, we have developed a binding-time analysis for a subset of the logic programming language Mercury. More precisely, we have considered first-order Mercury programs that are not partitioned into modules, and have assumed that each of the defined predicates is type correct and well-moded. Moreover, we have assumed that a predicate's arguments are partitioned in a set of *input* and *output* arguments. The latter restriction ensures that each unification in a well-moded program is directional, and binds one or more variables to ground terms. Consequently, it does not allow a program to unify two non-ground values. The effect is a simplified execution mechanism (and consequently a simplified analysis) since it only has to deal with unifications having an instantaneous effect: a variable is assigned a value once and for all; the value is never modified by unifications occurring later during the execution. At first sight, imposing the latter restriction on the programs to be analysed may seem overly conservative as it prevents the analysis of programs that manipulate partially instantiated structures. However, although Mercury (Somogyi, Henderson, and Conway 1996) allows some relaxation in the declared modes it does not allow a free variable to be shared between terms

and hence only allows a limited use of partially instantiated structures. Adding the fact that the vast majority of predicates can be declared having *input* and *output* arguments only (Somogyi, Henderson, and Conway 1996; Drabent 1987), diminishes the need for an analysis to handle the more involved modes possible in Mercury. Some of the other restrictions we have imposed on the programs that can be analysed are lifted in Chapter 4, in which we propose a binding-time analysis for Mercury that is capable of dealing with the higher-order and (to some extent) the modular characteristics of the language. Hence, the discussion of these more involved issues in relation to binding-time analysis is postponed until the latter chapter.

The remainder of the discussion is organised as follows. In a first part, we discuss several aspects of the proposed binding-time analysis and indicate the existing relations with other work, mainly in the field of partial evaluation of functional languages. In a second part, we discuss aspects of the binding-time analysis related to its use for off-line program specialisation.

3.5.1 Binding-time Analysis and Related Work

Although binding-time analyses have been developed for a variety of languages, most of the research on binding-time analysis was performed with respect to the functional programming paradigm. According to (Jones, Gomard, and Sestoft 1993), the first such binding-time analysis was developed by Jones, Sestoft and Søndergaard (Jones, Sestoft, and Søndergaard 1985; Sestoft 1986) and acted on a first-order language with atomic binding-times. Especially for languages like Mercury, in which data is represented by structured terms, such an atomic division – stating that a value as a whole is either *static* or *dynamic* – is considered too coarse grained. Indeed, even if the language does not allow the use of partially instantiated terms at runtime, it is most likely that partial data is represented by partially instantiated terms at specialisation-time. Hence the need for a more refined domain of binding-times, capable of expressing that *parts* of a value are *static* or *dynamic*. Our proposed domain of binding-times, $(\mathcal{BT}^+, \succeq)$, allows such more detailed representations by associating a value from $\{$*static*, *dynamic*$\}$ to each OR-node in a value's associated type graph. Type correctness ensures that during binding-time analysis only values of (instances of) the same type are compared (even in the presence of polymorphism) and $(\mathcal{BT}^+, \succeq)$ is a complete lattice.

Strongly related to our domain of binding-times is the domain proposed and used by Launchbury and Mogensen. Launchbury (Launchbury 1990) defines a system of types and derives a finite domain of *projections* over each type. Such a projection maps a value to a part of the value that is definitely static, as such "blanking" out the dynamic part. Consider for example the following type definition of a pair:

```
:- type pair(T1,T2) ---> (T1 - T2).
```

Four projections would be associated to the above type definition (Launchbury 1990): *id* mapping a value (t_1, t_2) to itself – thus representing a completely static value, *abs* mapping the value to X – thus representing a completely dynamic value, and the projections $left$ and $right$, mapping (t_1, t_2) to t_1, respectively t_2 denoting that the left, respectively right subterm of the term is definitely static. Note the similarity with our representation of the corresponding binding-times (graphically depicted in Fig. 3.15) mapping the nodes of the value's type graph to the constants *static* and *dynamic*. Projections have been used to denote partially static values

Figure 3.15: The labelled type graphs for `pair(T1,T2)`.

during binding-time analysis of functional languages (Launchbury 1990; Mogensen 1989). Similix – a partial evaluation system for Scheme – takes another approach. Its binding-time analysis (Bondorf and Jørgensen 1993; Bondorf 1990) uses a simple domain of atomic binding-times $\{\bot, S, D, Ps_\xi, Cl_n\}$. Of these, the value S denotes an atomic static value, Cl_n is a family of binding-times describing a static function value of arity n and Ps_ξ is a family of binding-times describing a static constructor value. The reason that the analysis can employ such a simple domain of binding-times while still being able to distinguish partially static structures is that all flow of structure (either first-order constructed values or, being a higher-order analysis, closures) has been resolved by a separate *flow analysis* (Bondorf and Jørgensen 1993).

Our proposed binding-time analysis is *polyvariant*, meaning that a single predicate is analysed several times: once for every distinct call pattern encountered during the analysis. Polyvariance is particularly powerful in combination with a refined domain of binding-times like $\mathcal{BT}$: different analyses are performed for a predicate with respect to different call patterns, even if the call patterns differ only in a single binding-time characterisation of a single argument. Such polyvariance at the level of the individual binding-time characterisations enables to maximally propagate the binding-time information conveyed in a structured binding-time. Our polyvariant analysis is guaranteed to create a finite program environment, since the possible number of call patterns of a procedure is finite. Indeed, it has a finite number of arguments and the finite type graph of each of these arguments can be labelled only in a finite number of ways. Initially, most developed binding-time

analyses were monovariant, allowing only a single binding-time to be associated to each function argument. An example is Similix's initial binding-time analysis (Bondorf 1990). More recent analyses are usually polyvariant. Polyvariance has been obtained in different ways: either by duplicating a function's definition a sufficient number of times and performing a monovariant analysis (Rytz and Gengler 1992) or by associating a set of binding-time descriptions to each function (Consel 1993a). The binding-time analysis presented in this chapter belongs to the latter category. Other binding-time analyses obtain polyvariance by solving a constraint system that is polymorphic in the involved binding-times (Henglein and Mossin 1994). We reconsider such analyses in detail in Chapter 4.

The binding-time analysis presented in this chapter is formulated as an application of abstract interpretation. Starting from an initial program environment, it constructs a monotonically increasing sequence of program environments, until a fixed point is reached. This is a classic approach taken when formulating binding-time analyses (Jones, Sestoft, and Søndergaard 1985; Sestoft 1986; Mogensen 1989; Rytz and Gengler 1992). A basic correctness requirement of the computed program environment is congruence: if a value is constructed using a (partially) dynamic value, the relevant part in the former value's binding-time must also be dynamic. The congruence requirement in the context of Mercury programs is formally defined in Definition 3.15 and congruence is ensured by the way unifications and predicate calls are handled during analysis. Apart from producing a congruent program environment, binding-time analysis must produce a well-annotated program. Well-annotatedness ensures that if a particular construct is annotated to be reduced by the specialiser, then the involved variables' values are instantiated enough to actually perform the operation. Guaranteeing well-annotatedness of a Mercury program is far from trivial because it not only involves reasoning over the computed binding-times, but also on the success/failure characteristics of the involved goals. Indeed, apart from possibly producing a number of output values, a goal in Mercury can either succeed (a number of times) or fail. Success or failure of a goal influences the specialisation process, since it determines the control flow in a predicate which in turn determines the binding-times computed for subsequent goals in the control flow graph. Our analysis guarantees well-annotatedness by incorporating the result of annotation (and hence a particular specialisation strategy) into the computation of binding-times: if a binding-time is produced in a goal that depends – according to the control flow – on another goal of which the success or failure can not be decided by the specialiser, the particular binding-time will be made dynamic. Incorporating such a specialisation strategy requires the explicit propagation of success/failure information, which complicates the formulation of the analysis functions considerably.

Contrary to the field of partial evaluation in functional programming in which a lot of attention has been paid to off-line partial evaluation using binding-time analysis to implement local control, most efforts in the context of partial evaluation

of logic programming have concentrated on on-line techniques. Automatic, on-line control techniques are usually defined using some well-founded (Bruynooghe, De Schreye, and Martens 1992; Martens, De Schreye, and Horváth 1994; Martens and De Schreye 1996) or well-quasi (Bol 1993; Sahlin 1993; Leuschel 1999; De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999) measurement on the constructed SLD-trees. On-line control has also been considered in the context of supercompilation of functional languages by Sørensen and Glück (1995). Similar control techniques have been developed within the framework of specialisation of functional logic languages (Alpuente, Falschi, Julián, and Vidal 1997; Albert, Alpuente, Falaschi, and Julian 1998).

3.5.2 The Role of Binding-time Analysis in Specialisation

In the context of logic programming, the basic task of a specialiser is twofold: it should be able to *reduce* a particular goal, and to create *specialised* versions of a procedure. Creating a version of a procedure $p(\overline{X}) \leftarrow G$ with respect to an input environment θ_s is in itself trivial: it consists of reducing the procedure's body goal with respect to θ_s and recording the result of reduction into a new procedure definition for p (usually renamed with respect θ_s). Such a specialiser creates residual procedures that are specialised versions of the program's original procedures. This is not always the case with other partial evaluators. For example Similix (Bondorf 1991; Bondorf and Jørgensen 1993) is capable of replacing other expressions (not necessarily function calls) with a call to a residual function whose body consists of the residual code after reducing the original expression. Therefore, it supports the insertion of *specialisation points* – either by hand or automatically – into the subject program. Consequently, the residual functions created by Similix are not necessarily specialised versions of the program's original functions but are newly created ones, similar to the ones created, for example, by (on-line) supercompilation (Sørensen and Glück 1995). In any case, two aspects play a crucial role in the termination of the process: local termination involves termination of the reduction process, global termination involves guaranteeing that no infinite number of specialised procedures is created.

In section 3.4.2, we have defined a reducer for Mercury, embodied by the function $\mathcal{T}$. The reducer accentuates the off-line nature of the process since it does not take any control decisions, but rather trivially follows the annotations computed by the binding-time analysis. The strategy implemented by our binding-time analysis towards controlling termination of the reduction process is similar to the strategy followed by Similix (Bondorf 1991; Bondorf and Jørgensen 1993). The basic idea is to guarantee that the reducer never evaluates a goal "speculatively". That is, during reduction of an initial goal G with respect to partial input, it will only reduce those (sub)goals that would definitely be evaluated when evaluating the initial goal with respect to the complete input. Such a strategy prevents infinite loops that are due to speculative evaluation. It does not, however, prevent the

existence of statically controlled infinite loops. Since the existence of such a loop would also make plain evaluation of the program (with complete input) loop infinitely, nontermination of the reducer in this case is usually accepted (Sestoft 1988; Jones, Gomard, and Sestoft 1993; Bondorf and Jørgensen 1993). Similix implements this strategy by inserting specialisation points at dynamic conditionals as such ensuring that if the test of an if-then-else can not completely be evaluated, neither of the branches is reduced but rather the expression is replaced by a residual call whose body is a specialised version of the original expression. In our Mercury binding-time analysis, this strategy is reflected in the annotations of a conjunction and an if-then-else. According to these annotations, the second conjunct of a conjunction will only be reduced if reduction of the first conjunct results in success. Likewise, neither branch of an if-then-else goal will be reduced if success or failure of the test-goal can not be decided during reduction. Instead of creating a single specialisation point for the dynamic construction (be it a conjunction or an if-then-else), specialised versions of the procedure calls occurring in the dynamic goal can be created.

Section 3.4.3 shows correctness of the binding-time analysis with respect to the reducer $\mathcal{T}$. Theorem 3.2 states that *if* the reduction process terminates, the residual goal computes (for instances of the partial input) the same result as the original goal would for the complete input. This is an important correctness result as it basically proves that the partial evaluation equation (Equation 1.1 from Chapter 1) holds for a partial evaluation system comprising the defined binding-time analysis and the trivial reducer $\mathcal{T}$. The correctness proof is inspired on Gomard's proof showing the correctness of LambdaMix (Gomard 1992) but is more involved due to the handling of success/failure information and nondeterminism which are inherent to a logic programming language.

Although the binding-time analysis guarantees local termination (for those programs that do not contain a statically controlled infinite loop), global termination is not ensured at all. The following Example illustrates the global termination problem.

Example 3.29 *Consider the procedure*

```
p(list(T)::in, T::in, list(T)::in, list(T)::out) is det.
```

A call `p(L,E,[],R)` *deconstructs a list L, and builds a new list R of the same length, but in which all elements are initialised to the value E. The procedure uses an accumulator.*

```
p(L,E,Acc,R):-    Acc2⇐^s [E|Acc],
                  if^d L⇒^d [X|Xs] then p(Xs,E,Acc2,R)^d
                                   else R:=^d Acc2.
```

The procedure defined in Example 3.29 is well-annotated given as binding-times

that L is *dynamic* and the other input arguments are completely *static*. Building a specialised procedure for the call `p(L,a,[],R)` results in the residual call `p(L,a,[a],R)` for which in turn a specialised procedure is constructed, once more resulting in a residual call `p(L,a,[a,a],R)` and so on. Global termination can be achieved by specialising appropriate *generalisations* of residualised calls, mapping several residualised calls to the same (specialised) procedure. Many off-line partial evaluators expose this kind of termination problems, examples being Similix (Bondorf and Jørgensen 1993), Schism (Consel 1993b) – a partial evaluator for a Scheme subset – and C-mix (Andersen 1993), a partial evaluator for C. One particular approach towards solving the (local as well as global) control problem in an off-line setting is the work by Andersen and Holst (Andersen and Holst 1996), based on Holst's earlier work (Holst 1991). This work develops an analysis for a higher-order untyped strict functional language that computes an approximation of the program's control- and data flow during partial evaluation, providing information about which values grow or shrink along the possible evaluation paths. This information is then used to change some of the computed binding-times from *static* to *dynamic* such that partial evaluation of the program only enters finitely many different configurations and guarantees, in combination with memoization, termination of the process. In an on-line setting, a wealth of techniques to guarantee (global) termination have been developed. Sørensen and Glück, for example, devise an algorithm to control the process of supercompilation (Sørensen and Glück 1995). Although the technique does not distinguish explicitly between local and global control levels, it can be seen as focusing on global control (as it controls the creation of specialised functions) while local control is limited to performing one-step unfoldings. Strongly related with the control of supercompilation is the control of on-line partial deduction in a logic programming setting, where a strong emphasis has been made on the distinction between local and global control. See (Leuschel and Bruynooghe 2001) and – to some extent – Chapter 2 for an overview. Distinguishing between local and global control in an off-line setting allows, in addition to the local control which is guaranteed by binding-time analysis, to implement global control on the level of the specialiser, when concrete input is available. Such a system, actually being a blend of off- and on-line control, contrasts with a purely off-line approach as envisioned in Holst's work (Holst 1991), in which *all* control decisions to guarantee termination are taken by an analysis that is run prior to the actual specialisation.

In the following chapter, we give an alternative formulation of the Mercury binding-time analysis developed in this chapter. Reformulating the analysis in the framework of constraint normalisation provides a – discutably – cleaner handling of polyvariance and enables to extend the analysis to deal (to some extent) with the modular structure of Mercury programs. We also alter the analysis to deal with Mercury's higher-order features. In this section, we have discussed the general observations behind the domain of binding-times we employ and the strategy

implemented by the binding-time analysis. Since these observations largely hold also for the reformulated analysis, we will not repeat their discussion in the next chapter. We do, however, postpone the discussion of some more involved issues – also applying to the analysis described in this chapter – until then.

Chapter 4

On Binding-time Analysis for Modular and Higher-order Mercury

The First Rule of Program Optimisation:
Don't do it.
The Second Rule of Program Optimisation:
Don't do it yet.

– *Michael Jackson.*

In this chapter, we discuss an alternative way of performing binding-time analysis for Mercury, that allows to upgrade the analysis defined in the previous chapter into an analysis for a fuller subset of Mercury, in which a program is build from several modules and in which programs can employ the higher-order capabilities of the language.

4.1 Introduction and Motivation

In the previous chapter, we discussed the fact that Mercury was designed as a logic programming language specifically tuned towards the creation of large-scale, real-world applications. When writing large applications, the programmer usually is encouraged to write general code, that can be used in a number of different situ-

ations, and to abstract concrete data representations by hiding the representation behind a number of procedure calls. To support the programmer employing these software engineering capabilities, Mercury provides a flexible module system that supports the composition of a program from several modules, rigorously defining how modules should interact with each other.

4.1.1 Mercury's Module System

The basic module system of Mercury is simple. A module consists of an *interface* part and an *implementation* part. The interface part contains those type definitions and procedure declarations that the module provides (or *exports*) towards other modules. In other words, the types and procedures declared in the interface part of a module are visible and can be used (or *imported*) by other modules.

Apart from the implementation of the procedures that are declared in the module's interface, its implementation part possibly contains additional type definitions and the declaration and implementation of additional procedures. These types and procedures are only visible in the implementation part of this module, and can not be used by other modules.

The interface part of a module rigorously defines how this module is to be used by other modules: it defines what procedures this module exports and their associated type- and mode declaration says how to call them. Types can be declared in two ways in a module's interface. Either the type definition is part of the interface, in what case the type and its representation are exported by the module, or the type is defined *abstract* in the interface, in what case only its name is exported by the module. In the former case, a module that imports the type can construct and deconstruct terms of the particular type, whereas in the latter case the importing module can only perform assignments and equality tests between terms of the particular type.

If a module wants to use entities that are defined in another module, it must explicitly import the module, in what case it imports the complete interface of the module. The imported entities can be used in the importing module as if they were declared in it (apart from the abstract types). Both the interface and implementation parts of a module can import the interfaces of other modules. A Mercury program is then defined as a set of Mercury modules. Note that the way in which the modules import each other impose a hierarchy on the modules that constitute a program. Following the terminology of (Puebla and Hermenegildo 1999), we use the notation $\mathit{imports}(M, M')$ to indicate that the module M imports the interface of M' and $\mathit{imported}(M)$ to denote the set of modules that are imported by M, that is: $\mathit{imported}(M) = \{M' \mid \mathit{imports}(M, M')\}$. With the notion $\mathit{dependent}(M)$ we refer to the set of modules on whose interface part M *depends* (either directly,

or indirectly through *imported*(M)). More formally:

$$\mathit{dependent}(M) = \left\{ M' \;\middle|\; \begin{array}{l} \mathit{imports}(M, M') \text{ or } \exists M'' \text{ such that} \\ \mathit{imports}(M, M'') \text{ and } M' \in \mathit{dependent}(M'') \end{array} \right\}$$

If we graphically represent a module by a box, and denote $\mathit{imports}(M, M')$ by an arrow from M towards M', Fig. 4.1 shows an example of a module hierarchy in Mercury. In the example, we have that $\mathit{imported}(M_1) = \{M_2, M_3, M_5\}$ and

Figure 4.1: A sample module hierarchy.

$\mathit{dependent}(M_1) = \{M_2, M_3, M_4, M_5\}$. Note that in Mercury, the *imports* relation is not transitive contrary to the *dependent* relation. Indeed, when a module M imports the interface of a module M', it becomes dependent on the interfaces imported by M' (and those imported therein) but it does not import these itself. While in Mercury modules may depend on each other in a circular way, we restrict our attention to programs in which no circularities exist between the modules, that is: for any module M, it holds that $M \notin \mathit{dependent}(M)$. We discuss circular dependencies later in this chapter.

The module system described above is to some extent a simplification of Mercury's real module system, in which modules can be constructed from submodules. While submodules do provide extra means to the programmer to control encapsulation and visibility of declarations, they do not pose additional conceptual difficulties and we do not consider them in the remainder of this work.

4.1.2 Analysing a Multi Module Program

It is only recently that analysis of modular programs has gained some attention in the logic programming field. For example (Puebla and Hermenegildo 1999) discusses some (mainly practical) issues in the analysis of multi module programs. It describes a number of "scenarios" in order to deal with multi module programs, from which we consider two extremes.

Dealing with several modules as one. A first, straightforward way of dealing with a multi module program is to convert the program into a monolithic program. Basically, this is achieved by merging the code of the individual modules and performing some renamings in order to prevent name clashes. Analysing a multi module program boils down to analysing a monolithic program and can hence – in theory – be performed by any standard analysis like the binding-time analysis from the previous chapter. Apart from some practical issues – like for example memory consumption – that may prevent successful analysis of a huge monolithic programs, other considerations are to be made:

- Since the analysis requires a monolithic program to be constructed, the complete source of all modules comprising a program P must be available to the analysis. While this may not be a problem in case the program is build uniquely using self-written modules, it may well be a problem when modules are used that are supplied by a third party which does not provides the module's source code.

- When a module is imported in more than one program, that module's procedures are likely to be analysed over and over, each time a program importing the module is analysed. This problem is not inherently due to the use of modules. Indeed, a call-dependent analysis of a program, whether modular or not, in general repeats the analysis of a single predicate when this predicate is used in – sometimes only slightly – different contexts, usually expressed by different call patterns. Even when the predicate is used in different contexts and reanalysis make sense, it often involves repeated computations that are invariant over the particular context and they could, in principle, be avoided. However, the problem becomes more apparent when multi module programs are analysed; at least when analysis is done by combining the modules in a monolithic program. Indeed, the procedures from an imported module can appear in different programs that are analysed in isolation. Hence, a single procedure usually is repeatedly analysed, even with respect to the same context. Since the use of modules is often encouraged in order to improve reusability of code, this problem is likely to occur frequently.

Dealing with a single module. Another extreme for an analysis is to be capable of dealing with a single module in isolation of the other modules. As (Puebla and Hermenegildo 1999) remarks, this ability allows to deal with *incomplete* programs – programs for which not all modules are as yet present – and allows an analysis tool to be more efficient. Indeed, it is natural to expect that processing a single module is more efficient than processing the whole program. However, less than optimal results may be obtained (Puebla and Hermenegildo 1999), of course due to the lack of analysis results for the imported modules.

In practice, tools that employ a processing scheme in which a single module

M is considered in isolation, do require some information to be available about the modules that are imported in M. A typical example is a compiler: in general, compiling a module M requires the *interfaces* of *imported*(M) to be available to the compiler as well. Compiling these interfaces (or simply loading them when they are precompiled) suffices to compile M. The source files of *imported*(M) do not need to be available to the compiler.

In (Puebla and Hermenegildo 1999), a third scenario is discussed: one that only considers a single module at a time, but not in isolation. That is, the analysis tool is able to suspend the analysis of a module M_1 when information from a module M_2 is needed, process the module M_2 and later on resume the analysis of M_1. Such a processing scheme most likely must work through all the modules in *dependent*(M) when processing M. We return to this scenario when we discuss the handling of circular module dependencies.

In this chapter, we devise a binding-time analysis that can handle a modular Mercury program while dealing with a single module in isolation. Like a compiler needs the interfaces of the imported modules, our analysis needs the *result* of analysing *imported*(M) when analysing M. Since we consider module hierarchies without circular dependencies, such an analysis can analyse the module hierarchy bottom-up, ensuring that the results of analysing *imported*(M) are available when M itself is analysed. Note that in such a scheme, each module is analysed only once, regardless whether the module is imported in several other modules. In the module hierarchy of Fig. 4.1, for example, the modules M_4 and M_5 are analysed once but the result of analysing M_4 is used twice: when analysing M_2 and when analysing M_3 – the latter analysis also incorporating the results of analysing M_5. Next, the result of analysing M_2, M_3 and again M_5 is used to analyse the top level module M_1.

4.2 Revisiting Binding-time Analysis for Mercury: a First-order Setting

In what follows, we consider first-order Mercury programs in superhomogeneous form like in the previous chapter. The only difference now is that a program is defined as a set of modules, that is $P = \{M_1, \ldots, M_n\}$. To be precise, during analysis we do not consider a single *module* in isolation, but rather a single *analysis unit*. Such an analysis unit consists of a module M, together with the predicate declarations from the *interfaces* of *imported*(M) and the type definitions from the interfaces of *dependent*(M). We also assume that abstract types are not used in an analysis unit. Indeed, while such an abstract type may be an excellent tool for the software engineer, there are few needs for them once the program has passed type checking, hence we assume their definition to be available just as for regular types.

The use of such an analysis unit (which roughly corresponds with the unit that is used for compiling the module) ensures that the analysis has enough knowledge about the module it is analysing: the declarations of the predicates that are called in this module together with the definitions of the types used in this module. In what follows, if we refer to the analysis of a "module", we refer to the analysis of the corresponding analysis unit.

4.2.1 Symbolically representing binding-times and their relations

In order to make the binding-time analysis as much modular as possible, we devise an analysis that works in two phases. In a first phase, we represent binding-times and the relations that exist between them in a symbolic way. Doing so enables to perform a large part of the data-flow analysis on this symbolic representation and hence independent of a particular call pattern. It is only in the second phase that call patterns are combined with the symbolic information derived from the first phase, computing the actual binding-times. The first phase of the analysis hence is *call independent* whereas the second phase is *call dependent*. Obviously, the call independent phase of the analysis does not need to be repeated in case a procedure is called with different call patterns and consequently, the result of a module's call independent analysis can be used regardless the context the module is used in, and must not be repeated when the module is used in different programs.

Recall from Chapter 3 that the binding-time analysis associates a binding-time with a variable at the program point where the variable is initialised. If a variable is initialised in both branches of a disjunction, or if-then-else, it associates a possibly different binding-time with the variable in each of the branches. Starting from the point where the two branches merge, it considers only a single binding-time for the variable which is defined as the least upper bound of the binding-times in the individual branches. Hence, to compute the binding-time of a variable at any program point in a procedure, it suffices to consider the binding-times of that variable at the program points where that variable is initialised, and take the least upper bound of an appropriate subset of these. Using the necessary definitions and notation from Chapter 3, we can formally express this as follows: for a procedure p that is analysed with respect to the binding-time environment π_{in}, we have that

$$\Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_{in}\Psi p\pi_{in}(V) = \bigsqcup_{\eta' \in \mathtt{reach}(V,\eta)} \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta'\pi_{in}\Psi p\pi_{in}(V)$$

for $V \in \mathcal{V}(p)$ and Ψ a congruent program environment.

To symbolically represent the binding-time of a variable at a particular program point, we introduce the concept of a *binding-time variable*, the set of which is denoted by $\mathcal{V}_{\mathcal{BT}}$. We will denote elements of this set as variables subscribed by a program point. If V is a variable occurring in a goal G, and η is a program point

identifying an atom in G, then the binding-time variable $V_\eta \in \mathcal{V}_{\mathcal{BT}}$ symbolically represents the binding-time of V at program point η. Given a sequence $\delta \in TPath$, we use the notation V_η^δ to denote the subvalue identified by δ in the binding-time of V at program point η.

Example 4.1 *Reconsider the definition of* `append/3` *from the previous Chapter, depicted in Fig. 4.2. The example shows the program points associated to the atomic subgoals by subscripting the atom with a natural number and the program points associated to a structured subgoal by subscripting the goal with the characters 'c' for conjunction and 'd' for disjunction, again accompanied by a natural number. Given the definition in Fig. 4.2, the binding-time variables* X_0, Z_2, Z_5 *and* Z_0

```
append(X,Y,Z)0:-
  ((X⇒[]1, Z:=Y2)c1 ;
   (X⇒[E|Es]3, (append(Es, Y, R)4, Z⇐[E|R]5)c2)c3)d1 .
```

Figure 4.2: `append/3` in superhomogeneous form

denote, respectively the binding-time of X *at the program point* 0 *and the binding-times of* Z *at the program points* $2, 5$ *and* 0.

Recall from Chapter 3 that a variable's binding-time is influenced by the fact whether or not the atom in which it is constructed is under dynamic control. Computing whether a goal is under dynamic control requires an approximation of another goal's success characteristics, which are in turn computed from the binding-times of the latter goal's input variables. Recall that this control information (resulting from applications of the functions $\mathcal{R}$ and $\mathcal{C}$) is represented by elements of the set $\{\bot, \top\}$. Since these values are elements of the set of binding-times, $\mathcal{BT}^+$, we can represent also this information symbolically, by considering a number of extra binding-time variables. These are of the form $\mathcal{C}_\eta$ and $\mathcal{R}_\eta$ for every program point η. Their intended meaning is as follows: if $\mathcal{C}_\eta = \top$, the goal identified by η is under dynamic control in the procedure's body, which is not the case if $\mathcal{C}_\eta = \bot$. Likewise, if $\mathcal{R}_\eta = \bot$, the goal identified by η reduces either to *true* or *fail* during specialisation, or to some residual code which is guaranteed not to fail at run-time. If, on the other hand, $\mathcal{R}_\eta = \top$, the goal identified by η possibly reduces to residual code that can fail at run-time. Note that these binding-time variables – which we will refer to as *control variables* – are boolean in the sense that they will only assume a value that is either $\bot$ or $\top$.

Binding-times are computed from other binding-times (according to the *covers* relation) combined – using a least upper bound operator – with the appropriate control information. Symbolically, we can represent these dependencies by a number of constraints between the involved binding-time variables. In general:

Definition 4.1 *A* binding-time constraint *is a constraint of the following form:*

$$V_\eta^\delta \succeq X_{\eta'}^\gamma \qquad V_\eta^\delta \succeq \top$$

$$V_\eta^\delta \succeq^* X_{\eta'}^\gamma \qquad V_\eta^\delta \succeq^* \top$$

where $V_\eta, X_{\eta'} \in \mathcal{V}_{\mathcal{BT}}$ *and* $\delta, \gamma \in TPath$*. The set of all binding-time constraints is denoted by* $\mathcal{BTC}$*.*

A constraint of the form $V_\eta^\delta \succeq X_{\eta'}^\gamma$ denotes that the binding-time represented by V_η^δ must be at least as dynamic as (or *cover*) the binding-time represented by $X_{\eta'}^\gamma$. Note that such a constraint requires the types of V and X, denoted by t_V and t_X to be such that t_V^δ and t_X^γ are instances of one another, in order for their binding-times to be comparable. The intended meaning of a constraint of the form $V_\eta^\delta \succeq^* X_{\eta'}^\gamma$ is that the binding-time represented by V_η^δ is at least as dynamic as the binding-time value associated to the node identified by γ in the binding-time represented by $X_{\eta'}^\gamma$. Note that such a constraint does not require t_V^δ and t_X^γ to be of comparable types; it simply expresses that if the node identified by γ in the binding-time represented by $X_{\eta'}$ is *dynamic*, so must be the node identified by δ in V_η and by definition of a binding-time, so must be all its descendant nodes. Remark that we also allow constraints of which the right-hand side is the constant $\top$. Although we occasionally also consider constraints of which the right-hand side is the constant $\bot$, we do not explicitly mention these in the definition, as these constraints are superfluous: for any $X_\eta \in \mathcal{V}_{\mathcal{BT}}$ and $\delta \in TPath$, it holds by definition that $X_\eta^\delta \succeq \bot$.

Example 4.2 *Reconsider the definition of* `append/3` *in Fig. 4.2. Some examples of binding-time constraints between binding-time variables from* `append/3` *and their intended meaning are:*

$Z_2 \succeq Y_0$	*the binding-time associated to* Z *at program point* 2 *is at least as dynamic as the binding-time associated to* Y *at program point* 0
$E_3 \succeq X_0^{\langle[],1\rangle}$	*the binding-time associated to* E *at program point* 3 *is at least as dynamic as the subvalue denoted by* $\langle[],1\rangle$ *of the binding-time associated to* X *at program point* 0
$Z_5^{\langle[],1\rangle} \succeq E_3$	*the subvalue denoted by* $\langle[],1\rangle$ *in the binding-time of* Z *at program point* 5 *is at least as dynamic as the binding-time associated to* E *at program point* 3
$\mathcal{R}_3 \succeq^* X_0$	*the atom at program point* 3 *reduces to true, fail or code that is guaranteed to succeed if* X_0 *represents a binding-time in which the root node* $\langle\rangle$ *is bound to static*
$\mathcal{C}_4 \succeq \mathcal{R}_3$	*the atom at program point* 4 *is under dynamic control if the atom at program point* 3 *possibly reduces to code that might fail*

A set of binding-time constraints is called a binding-time constraint system (or simply a constraint system). The link between a binding-time constraint system and the actual binding-times it represents is formalised as a solution to the constraint system.

Definition 4.2 *A* solution *to a binding-time constraint system $\mathcal{C}$ is a substitution $\sigma : \mathcal{V}_{\mathcal{BT}} \mapsto \mathcal{BT}$ mapping binding-time variables to binding-times such that*

- *for every constraint $V_\eta^\delta \succeq \top \in \mathcal{C}$ and $V_\eta^\delta \succeq^* \top \in \mathcal{C}$ it holds that $\sigma(V_\eta)^\delta \succeq \top$*
- *for every constraint $V_\eta^\delta \succeq X_{\eta'}^\gamma \in \mathcal{C}$ it holds that $\sigma(V_\eta)^\delta \succeq \sigma(X_{\eta'})^\gamma$*
- *for every constraint $V_\eta^\delta \succeq^* X_{\eta'}^\gamma \in \mathcal{C}$ it holds that $\sigma(X_{\eta'})(\gamma) = dynamic \Rightarrow \sigma(V_\eta)^\delta \succeq \top$*

Given two solutions σ and σ' to $\mathcal{C}$, we define that $\sigma \sqsupseteq \sigma'$ if for all $V_\eta \in dom(\sigma')$ it holds that $V_\eta \in dom(\sigma)$ and $\sigma(V_\eta) \succeq \sigma'(V_\eta)$. A solution σ is a minimal solution *for $\mathcal{C}$ if for every solution σ' for $\mathcal{C}$ it holds that $\sigma' \sqsupseteq \sigma$.*

We will sometimes use a constraint of the form $V_\eta^\delta \succeq X_{\eta'}^{\gamma'} \sqcup Y_{\eta''}^{\gamma''}$ (analogously for $\succeq^*$) as shorthand notation for the set of constraints $\{V_\eta^\delta \succeq X_{\eta'}^{\gamma'}, V_\eta^\delta \succeq Y_{\eta''}^{\gamma''}\}$. Indeed, from Definition 4.2 it can be seen that in any solution σ satisfying the latter two constraints, it holds that $\sigma(V_\eta)^\delta \succeq \sigma(X_{\eta'}^{\gamma'}) \sqcup \sigma(Y_{\eta''}^{\gamma''})$.

Example 4.3 *Consider the following binding-time constraint system and its least solution. For sake of simplicity, we assume that all binding-time variables are boolean and range over the set $\{dynamic, static\}$.*

Binding-time constraint system	*Minimal solution*
$X_{\eta_1} \succeq \top$ $R_{\eta_3} \succeq X_{\eta_2}$ $Y_{\eta_4} \succeq X_{\eta_1}$ $Y_{\eta_4} \succeq R_{\eta_3}$	$\left\{ \begin{array}{ll} (X_{\eta_1}, dynamic) & (X_{\eta_2}, static) \\ (R_{\eta_3}, static) & (Y_{\eta_4}, dynamic) \end{array} \right\}$

The next subsections develop our binding-time analysis in a constraint setting. In the first, call independent phase we build a binding-time constraint system for each procedure such that a solution for the system with respect to a particular call pattern can efficiently be computed. The actual computation of solutions is performed in the second, call dependent phase of the analysis.

4.2.2 Symbolic data flow analysis

The binding-time analysis from Chapter 3 basically performs an abstract interpretation of the program over the domain of binding-times. Essentially, it constructs a program environment that records, for each relevant procedure, a set of

associations between binding-times of the procedure's input arguments with the corresponding binding-times for the procedure's output arguments. Polyvariance is ensured precisely because a program environment stores a *set* of such associations for each procedure. In our symbolic setting, the relation that exist between the binding-times of a procedure's input and output arguments can be represented by a single set of binding-time constraints; each such constraint constraining the binding-time of an output argument in function of the binding-time of an input argument. Polyvariance is immediate, since the binding-times of the input arguments are represented symbolically and hence can be instantiated by any call pattern. Binding-time constraints with a procedure's input argument on their right-hand side are said to be in normal form:

Definition 4.3 *A binding-time constraint is in* normal form *with respect to a procedure $p \in Proc$ if it is either of the form*

- $V_\eta^\delta \succeq \top$
- $V_\eta^\delta \succeq X_{\eta_0}^\gamma$ *with* $X \in \mathtt{in}(p)$ *and* η_0 *the program point associated to p's head atom.*

and analogously for constraints of this form using $\succeq^*$.

In the remainder of this chapter we assume, as before, that the program point η_0 identifies a procedure's head atom whereas η_b identifies the procedure's body goal.

Example 4.4 *Reconsider the binding-time constraints from Example 4.2. The constraints*

$$Z_2 \succeq Y_0 \qquad E_3 \succeq X_0^{\langle [],1\rangle} \qquad \mathcal{R}_3 \succeq^* X_0$$

are in normal form with respect to `append/3`*, whereas the constraints*

$$Z_5^{\langle [],1\rangle} \succeq E_3 \qquad \mathcal{C}_4 \succeq \mathcal{R}_3$$

are not.

A program environment can thus be represented by associating a set of binding-time constraints in normal form to each procedure.

Definition 4.4 *A* symbolic program environment *is a function $Proc \mapsto \wp(\mathcal{BTC})$. The set of all such symbolic program environments is denoted by $\mathcal{PE}nv_S$.*

Our symbolic data flow analysis builds a symbolic program environment much in the same way the binding-time analysis from Chapter 3 builds a concrete program environment. In what follows, we present the analysis functions that abstractly interpret a procedure over the domain $(\wp(\mathcal{BTC}), \subseteq)$. The analysis functions require a symbolic program environment that represent the result of analysis so far, and

update the environment with newly derived information until a fixed point is reached.

Recall from Chapter 3 that the binding-time analysis incorporates the following control strategy: if an atom is under dynamic control, it will not be unfolded by the specialiser and hence the binding-times of its output arguments must be made $\top$ by the analysis. Just like we postpone the computation of concrete binding-times until the second, call dependent phase of the binding-time analysis, we defer also the computation of concrete values for the control variables (corresponding to the values computed by the $\mathcal{C}$ and $\mathcal{R}$ functions of Chapter 3 and implementing the specialisation strategy) to this second phase. We model their influence on the computed binding-times by incorporating their symbolic representations into the constraints derived in the first phase of the analysis.

We are now ready to formally specify the first, call independent phase of the binding-time analysis that set up a binding-time constraint system for each of the procedures of an analysis unit. We do this by means of three analysis functions: $\mathcal{A}_S$ that derives the constraints introduced by a single atom, $\mathcal{G}_S$ that derives the constraints introduced by a goal and Δ_S that analyses a complete analysis unit. Their signatures are as follows:

$$\mathcal{A}_S : Atom \times \mathcal{P}Env_S \mapsto \wp(\mathcal{BTC})$$

$$\mathcal{G}_S : Goal \times \mathcal{P}Env_S \mapsto \wp(\mathcal{BTC})$$

$$\Delta_S : \wp(Proc) \times \mathcal{P}Env_S \mapsto \mathcal{P}Env_S$$

We start by the definition of the $\mathcal{G}_S$ function, which can be found in Fig. 4.3. Recall that we use the notation G_η as shorthand for a goal G identified by program point η. Analysing a goal requires, apart from the goal to be analysed, a symbolic program environment that contains the result of analysis so far. This environment is needed for the analysis of an atomic goal, which is handled by the analysis function $\mathcal{A}_S$ which we will define later on. The set of binding-time constraints of a structured goal is defined as the union of the sets of constraints associated to the individual subgoals, together with the constraints on the control variables that are introduced by the particular construct. In order not to overload the definition, we define the latter set by means of an auxiliary function, $\mathcal{G}'_S$. The definition of this auxiliary function is found in Fig. 4.4. The constraints on control variables introduced by a structured goal are simple, as they merely reflect the propagation of the control variable's value, either from the goal to its subgoals (in case of the control variable $\mathcal{C}$) or from the goal's subgoals to the goal itself (in case of $\mathcal{R}$). The binding-time variables denoting dynamic control denote that a goal is under dynamic control *with respect to the procedure's body*. The negated goal in a negation is under dynamic control only if the negation itself is. On the other hand, the negation reduces to true, fail, or residual code which is guaranteed to succeed if the negated goal does. Note that the negation also reduces to true, fail or residual

$$
\begin{array}{lcl}
\mathcal{G}_S[\![A]\!]\mu & = & \mathcal{A}[\![A]\!]\mu \\
\mathcal{G}_S[\![not_\eta(G_{\eta'})]\!]\mu & = & \mathcal{G}_S[\![G_{\eta'}]\!]\mu \;\cup\; \mathcal{G}'_S[\![not_\eta(G_{\eta'})]\!] \\
\mathcal{G}_S[\![if_\eta\, G'_{\eta'}\ then\, G''_{\eta''}\ else\, G'''_{\eta'''}]\!]\mu & = & \left[\begin{array}{c} \mathcal{G}_S[\![G'_{\eta'}]\!]\mu \cup \mathcal{G}_S[\![G''_{\eta''}]\!]\mu \cup \mathcal{G}_S[\![G'''_{\eta'''}]\!]\mu \\ \cup \\ \mathcal{G}'_S[\![if_\eta\, G'_{\eta'}\ then\, G''_{\eta''}\ else\, G'''_{\eta'''}]\!] \end{array}\right] \\
\mathcal{G}_S[\![(G'_{\eta'}, G''_{\eta''})_\eta]\!]\mu & = & \left[\begin{array}{c} \mathcal{G}_S[\![G'_{\eta'}]\!]\mu \;\cup\; \mathcal{G}_S[\![G''_{\eta''}]\!]\mu \\ \cup \\ \mathcal{G}'_S[\![(G'_{\eta'}, G''_{\eta''})_\eta]\!] \end{array}\right] \\
\mathcal{G}_S[\![(G'_{\eta'}; G''_{\eta''})_\eta]\!]\mu & = & \left[\begin{array}{c} \mathcal{G}_S[\![G'_{\eta'}]\!]\mu \;\cup\; \mathcal{G}_S[\![G''_{\eta''}]\!]\mu \\ \cup \\ \mathcal{G}'_S[\![(G'_{\eta'}; G''_{\eta''})_\eta]\!] \end{array}\right]
\end{array}
$$

Figure 4.3: The definition of $\mathcal{G}_S$.

code that is guaranteed to succeed (in casu fail) when the negated goal reduces to residual code that is guaranteed to succeed. The propagation in the other constructs is similar: the subgoals of an if-then-else are under dynamic control if the if-then-else is under dynamic control. Moreover, both the then and else goals are under dynamic control if the test goal possibly reduces to residual code which could fail at run time. If each of the if-then-else's subgoals reduces to true, fail or code that is guaranteed to succeed, so does the if-then-else. The subgoals of a conjunction are under dynamic control if the conjunction itself is. Moreover, the second conjunct is under dynamic control if the first conjunct possibly reduces to residual code that could fail. If both conjuncts reduce to true, fail or code that is guaranteed to succeed, so does the conjunction. To conclude, if a disjunction is under dynamic control, so are both disjuncts. If both disjuncts reduce to true, fail or code that is guaranteed to succeed, so does the disjunction.

Example 4.5 *Reconsider the definition of* `append/3` *in Example 4.1. The body goal contains the following structured subgoals: a conjunction identified by program point* c_1 *with the atomic conjuncts identified by program points* 1 *and* 2*, a second conjunction identified by* c_2 *with the atomic conjuncts identified by program points* 4 *and* 5*, a third conjunction identified by* c_3 *with the conjuncts identified by program points* 3 *and* c_2 *and a disjunction identified by program point* d_1 *with the disjuncts identified by* c_1 *and* c_3*. The binding-time constraints that are associated to each*

$$
\begin{array}{lcl}
\mathcal{G}'_S[\![not_\eta(G_{\eta'})]\!] & = & \left\{ \begin{array}{cc} \mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta & \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'} \end{array} \right\} \\
\mathcal{G}'_S[\![if_\eta\, G'_{\eta'}\ then\, G''_{\eta''}\ else\, G'''_{\eta'''}]\!] & = & \left\{ \begin{array}{ccc} \mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta & \mathcal{C}_{\eta''} \succeq \mathcal{C}_\eta & \mathcal{C}_{\eta'''} \succeq \mathcal{C}_\eta \\ & \mathcal{C}_{\eta''} \succeq \mathcal{R}_{\eta'} & \mathcal{C}_{\eta'''} \succeq \mathcal{R}_{\eta'} \\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'} & \mathcal{R}_\eta \succeq \mathcal{R}_{\eta''} & \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'''} \end{array} \right\} \\
\mathcal{G}'_S[\![(G'_{\eta'}, G''_{\eta''})_\eta]\!] & = & \left\{ \begin{array}{ccc} \mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta & \mathcal{C}_{\eta''} \succeq \mathcal{C}_\eta & \mathcal{C}_{\eta''} \succeq \mathcal{R}_{\eta'} \\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'} & \mathcal{R}_\eta \succeq \mathcal{R}_{\eta''} & \end{array} \right\} \\
\mathcal{G}'_S[\![(G'_{\eta'}; G''_{\eta''})_\eta]\!] & = & \left\{ \begin{array}{cc} \mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta & \mathcal{C}_{\eta''} \succeq \mathcal{C}_\eta \\ \mathcal{R}_\eta \succeq \mathcal{R}_{\eta'} & \mathcal{R}_\eta \succeq \mathcal{R}_{\eta''} \end{array} \right\}
\end{array}
$$

Figure 4.4: The auxiliary function $\mathcal{G}'_S$.

of these structured goals are as follows:

(c_1)	$\mathcal{C}_1 \succeq \mathcal{C}_{c_1}$	$\mathcal{R}_{c_1} \succeq \mathcal{R}_1$
	$\mathcal{C}_2 \succeq \mathcal{C}_{c_1}$	$\mathcal{R}_{c_1} \succeq \mathcal{R}_2$
	$\mathcal{C}_2 \succeq \mathcal{R}_1$	
(c_2)	$\mathcal{C}_4 \succeq \mathcal{C}_{c_2}$	$\mathcal{R}_{c_2} \succeq \mathcal{R}_4$
	$\mathcal{C}_5 \succeq \mathcal{C}_{c_2}$	$\mathcal{R}_{c_2} \succeq \mathcal{R}_5$
	$\mathcal{C}_5 \succeq \mathcal{R}_4$	
(c_3)	$\mathcal{C}_3 \succeq \mathcal{C}_{c_3}$	$\mathcal{R}_{c_3} \succeq \mathcal{R}_3$
	$\mathcal{C}_{c_2} \succeq \mathcal{C}_{c_3}$	$\mathcal{R}_{c_3} \succeq \mathcal{R}_{c_2}$
	$\mathcal{C}_{c_2} \succeq \mathcal{R}_3$	
(d_1)	$\mathcal{C}_{c_1} \succeq \mathcal{C}_{d_1}$	$\mathcal{R}_{d_1} \succeq \mathcal{R}_{c_1}$
	$\mathcal{C}_{c_3} \succeq \mathcal{C}_{d_1}$	$\mathcal{R}_{d_1} \succeq \mathcal{R}_{c_3}$

The binding-time constraints that are associated to an atomic goal are somewhat more involved. Apart from binding-time constraints on the atom's output variables, analysing an atom also possibly results in a binding-time constraint on the control variable $\mathcal{R}_\eta$, indicating under what conditions the atom can be reduced to true, fail, or code that is guaranteed to succeed. Moreover, when creating the binding-time constraints on the atom's output variables, the control variable $\mathcal{C}_\eta$ must be taken into account, in order to guarantee that the particular binding-time is made $\top$ in case the atom is under dynamic control.

$$
\begin{array}{lcl}
\mathcal{A}_S[\![X == Y_\eta]\!]\mu & = & \left[\begin{array}{c} \{\mathcal{R}_\eta \succeq^* X_{\eta'} \mid \eta' \in \mathtt{reach}(X,\eta)\} \\ \cup \\ \{\mathcal{R}_\eta \succeq^* Y_{\eta'} \mid \eta' \in \mathtt{reach}(Y,\eta)\} \end{array}\right] \\
\mathcal{A}_S[\![X := Y_\eta]\!]\mu & = & \left[\begin{array}{c} \{X_\eta \succeq Y_{\eta'} \sqcup \mathcal{C}_\eta \mid \eta' \in \mathtt{reach}(Y,\eta)\} \\ \cup \\ \{\mathcal{R}_\eta \succeq \bot\} \end{array}\right] \\
\mathcal{A}_S[\![X \Rightarrow f(\overline{Y})_\eta]\!]\mu & = & \left[\begin{array}{c} \bigcup_{Y_i \in \overline{Y}} \{Y_{i_\eta} \succeq X_{\eta'}^{\overline{[\langle f,i\rangle]}} \sqcup \mathcal{C}_\eta \mid \eta' \in \mathtt{reach}(X,\eta)\} \\ \cup \\ \{\mathcal{R}_\eta \succeq^* X_{\eta'} \mid \eta' \in \mathtt{reach}(X,\eta)\} \end{array}\right] \\
\mathcal{A}_S[\![X \Leftarrow f(\overline{Y})_\eta]\!]\mu & = & \left[\begin{array}{c} \bigcup_{Y_i \in \overline{Y}} \{X_\eta^{\overline{[\langle f,i\rangle]}} \succeq Y_{i_{\eta'}} \sqcup \mathcal{C}_\eta \mid \eta' \in \mathtt{reach}(Y^i,\eta)\} \\ \cup \\ \{\mathcal{R}_\eta \succeq \bot\} \end{array}\right] \\
\mathcal{A}_S[\![q(\overline{X})_\eta]\!]\mu & = & \rho_\eta^q(\overline{X}, \mu(q))
\end{array}
$$

where $\rho_\eta^q : \langle\mathcal{V}\rangle \times \wp(\mathcal{BTC}) \mapsto \wp(\mathcal{BTC})$ denotes the renaming of a constraint set with respect to a sequence of variables, and is defined as

$$
\rho_\eta^q(\overline{X}, S) = \left[\begin{array}{c}
\left\{ X_{i_\eta}^\delta \succeq X_{j_{\eta'}}^\gamma \sqcup \mathcal{C}_\eta \;\middle|\; \begin{array}{l} F_{i_{\eta_0}}^\delta \succeq F_{j_{\eta_0}}^\gamma \in S \\ \text{and } \eta' \in \mathtt{reach}(X_j,\eta) \end{array} \right\} \\
\cup \\
\left\{ X_{i_\eta}^\delta \succeq \top \;\middle|\; F_{i_{\eta_0}}^\delta \succeq \top \in S \right\} \\
\cup \\
\left\{ \mathcal{R}_\eta \succeq X_{i_{\eta'}}^\delta \;\middle|\; \begin{array}{l} \mathcal{R}_{\eta_b} \succeq F_{i_{\eta_0}}^\delta \in S \\ \text{where } \eta_b = \mathcal{P}p(\mathcal{B}ody(q)) \end{array} \right\} \\
\cup \\
\left\{ \mathcal{R}_\eta \succeq \top \;\middle|\; \mathcal{R}_{\eta_b} \succeq \top \in S \right\}
\end{array}\right]
$$

Figure 4.5: The definition of $\mathcal{A}_S$.

Note that, when creating a binding-time constraint on one of the atom's output variables V, this binding-time is represented by V_η since the binding-time is created in the atom identified by program point η. If, on the other hand, the binding-time of one of the atom's *input* variables V is considered, this binding-time is represented by the least upper bound of the binding-time variables $\{V_{\eta'} \mid \eta' \in \texttt{reach}(V, \eta)\}$.

A test does not have any output variables, so it only creates constraints on control variables. The atom reduces to true, fail or code that is guaranteed to succeed when both input variables are bound to an outermost functor. An assignment $X := Y$ introduces the constraints specifying that the binding-time of X at program point η must be at least as dynamic as the least upper bound of the binding-times of Y as they are constructed at the relevant program points. Moreover, if the assignment is under dynamic control, X_η must be assigned the value $\top$. This is guaranteed by adding $\sqcup\, \mathcal{C}_\eta$ to the right-hand side of the constraints on X_η. Even if an assignment is not reduced, it can never fail at run time. Hence the (superfluous) constraint $\mathcal{R}_\eta \succeq \bot$. A deconstruction introduces some binding-time constraints indicating that the binding-time of the newly introduced variables must be at least as dynamic as the corresponding subvalue in the binding-time of the deconstructed variable. Also in this case, the least upper bound with $\mathcal{C}_\eta$ guarantees that, if the deconstruction is under dynamic control, the newly introduced binding-time variables will be forced to have the value $\top$. If the deconstructed variable is bound to at least an outermost functor, the deconstruction reduces to true or fail at specialisation time. Otherwise, a residualised deconstruction can either succeed or fail at run time which is reflected that in that case $\mathcal{R}_\eta$ will have the value $\top$. When handling a construction on the other hand, the binding-time of the constructed variable is constrained by the binding-times of the variables used in the construction. Again, if the construction is under dynamic control, the constructed binding-time is guaranteed to be $\top$ by the use of the least upper bound with $\mathcal{C}_\eta$. Even when residualised, a construction can never fail, so again the (superfluous) constraint $\mathcal{R}_\eta \succeq \bot$ is introduced.

Example 4.6 *Reconsider the definition of* `append/3` *in Example 4.1. The constraints that are associated to the unifications in* `append/3`*'s body goal are as follows. The numbers in the left hand side column denote the particular unification's program point.*

(1)	$\mathcal{R}_1 \succeq^* X_0$	
(2)	$\mathcal{R}_2 \succeq \bot$	$Z_2 \succeq Y_0$
(3)	$\mathcal{R}_3 \succeq^* X_0$	$E_3 \succeq X_0^{\langle [\,],1\rangle}$ $Es_3 \succeq X_0^{\langle\rangle}$
(5)	$\mathcal{R}_5 \succeq \bot$	$Z_5^{\langle [\,],1\rangle} \succeq E_3$ $Z_5^{\langle\rangle} \succeq R_4$

The constraints introduced by a procedure call are all obtained by renaming the relevant constraints from the symbolic program environment. First of all,

since the constraints from the symbolic program environment are in normal form, the constraints on the procedure's formal output arguments have a right-hand side that is either $\top$, or a formal input argument. Renaming such a constraint with respect to the actual arguments of the call involves renaming the formal output argument F_i with X_{i_η} (η being the program point identifying the call) and the eventual formal input argument F_j with each of the binding-time variables $\{X_{j_{\eta'}} \mid \eta' \in \texttt{reach}(X_j, \eta)\}$ since again the binding-time of the input variable X_j at program point η is the least upper bound of those $X_{j_{\eta'}}$. As with the other atoms, the created binding-times are constrained to be $\top$ when the procedure call is under dynamic control. A constraint on the control variable $\mathcal{R}_\eta$ is obtained by renaming the constraints on the control variable $\mathcal{R}_{\eta_b}$ in the symbolic program environment. These constraints express the conditions – in terms of the procedure's formal input arguments – under which the procedure's body goal reduces to true, fail, or code that is guaranteed to succeed. Renaming this constraint, one obtains an equivalent condition on $\mathcal{R}_\eta$ (the call) in terms of the call's actual input arguments.

Example 4.7 *Reconsider the definition of* `append/3` *in Example 4.1. Assume the procedure call* $append(Es, Y, R)$ *at program point* 4 *is analysed with respect to a symbolic program environment* μ *such that*

$$\mu(append) = \left\{ \begin{array}{c} Z_0 \succeq Y_0 \\ Z_0^{\langle [],1\rangle} \succeq X_0^{\langle [],1\rangle} \\ \mathcal{R}_{d_1} \succeq X_0 \end{array} \right\}$$

The constraints associated to the procedure call $append(Es, Y, R)$ *at program point* 4 *are then*

$$\left\{ \begin{array}{c} R_4 \succeq Y_0 \\ R_4^{\langle [],1\rangle} \succeq Es_3^{\langle [],1\rangle} \\ \mathcal{R}_4 \succeq Es_3 \end{array} \right\}$$

Note that formal input and output arguments are both referred to by their occurrence in η_0, denoting the procedure's head atom. Since a procedure's output arguments are usually initialised at several program points in the procedure's body, this requires to add the constraints

$$\left\{ \; F_{i_{\eta_0}} \succeq F_{i_{\eta'}} \;\middle|\; \eta' \in \texttt{reach}(F_i, \eta_0) \; \right\}$$

for each output argument F_i to the constraints that are generated for a procedure body, something we make explicit in the definition of the analysis function Δ_S that analyses a complete analysis unit.

Example 4.8 *Reconsider the definition of* `append/3` *in Example 4.1. The extra constraints associated to* `append/3`*'s output arguments are*

$$Z_0 \succeq Z_2 \qquad Z_0 \succeq Z_5$$

Analysing a module consists of analysing in turn each of the procedure's body goals with respect to a current symbolic program environment, and update the program environment with the obtained result. In order to do so, the set of binding-time constraints resulting from the analysis of a body goal must be converted to normal form, before the program environment can be updated. Doing so is possible due to the well-modedness of Mercury procedures. Indeed, in a well-moded procedure it is possible to trace data flow back to the procedure's input arguments, and consequently to express the binding-time constraints on any variable in function of the binding-times of the procedure's input arguments. We will define the transformation of a constraint system into a constraint system that is in normal form shortly, but define Δ_S first. For any constraint system $\mathcal{C}$, let $\mathcal{C}_N$ denote the equivalent constraint system in normal form.

Definition 4.5 *The module analysis function* $\Delta_S : \wp(Proc) \times \mathcal{PE}nv_S \mapsto \mathcal{PE}nv_S$ *is defined as follows:*

$$\Delta_S(M,\mu) = \begin{cases} \mu & \text{if } M = \emptyset \\ \Delta_S(M_s, \mu[p/\mathcal{C}_N]) & \text{if } M = \{p(\overline{X}) \leftarrow G\} :: M_s\} \\ & \text{and } \mathcal{C} = \mathcal{G}[\![G]\!]\mu \cup \mathcal{O}_p \end{cases}$$

where

$$\mathcal{O}_p = \bigcup_{F_i \in \mathtt{out}(p)} \left\{ \; F_{i_{\eta_0}} \succeq F_{i_{\eta'}} \;\middle|\; \eta' \in \mathtt{reach}(F_i, \eta_0) \; \right\}$$

Given an analysis unit M and a symbolic program environment μ, Δ_S computes a new symbolic program environment by analysing the body of each procedure p in M, normalising the resulting constraint set, and replacing $\mu(p)$ with the newly obtained constraints in normal form. The result of call independent analysis of an analysis unit M is then a symbolic program environment μ that is a solution to the equation

$$\mu = \Delta_S(M, \mu) \tag{4.1}$$

Again, we are interested in the least solution to equation 4.1, that is the solution which minimally constrains the binding-time variables. Such a minimal solution can be found as follows. Starting from an initial symbolic program environment μ_0 that is defined such that for every procedure p, $\mu(p) = \emptyset$, one computes

$$\mu_i = \Delta_S(M, \mu_{i-1})$$

for all $0 < i \leq n$ where $n \in \mathbb{N}$ is the smallest n such that $\mu_n = \mu_{n-1}$.

Example 4.9 *Reconsider the definition of* `append/3` *in Example 4.1. Let us denote, with* $\mathcal{C}_i$ *the set of constraints as they are derived in the* i*'th application round of* Δ_S*, that is,*

$$\mathcal{C}_i = \mathcal{G}_S[\![\mathcal{B}ody(append)]\!]\mu_{i-1} \cup \mathcal{O}_{append}$$

where

$$\mathcal{O}_{append} = \{Z_0 \succeq Z_2, Z_0 \succeq Z_5\}.$$

Starting from a symbolic program environment μ_0 *such that* $\mu_0(append) = \emptyset$*, we derive in the first round the following set of constraints*

$$\mathcal{C}_0 = \left\{ \begin{array}{lll} Z_2 \succeq Y_0 & \mathcal{R}_{c_1} \succeq \mathcal{R}_1 & \mathcal{C}_1 \succeq \mathcal{C}_{c_1} \\ E_3 \succeq X_0^{\langle [|], 1\rangle} & \mathcal{R}_{c_1} \succeq \mathcal{R}_2 & \mathcal{C}_2 \succeq \mathcal{C}_{c_1} \\ Es_3 \succeq X_0^{\langle\rangle} & \mathcal{R}_{c_2} \succeq \mathcal{R}_4 & \mathcal{C}_2 \succeq \mathcal{R}_1 \\ Z_5^{\langle [|], 1\rangle} \succeq E_3 & \mathcal{R}_{c_2} \succeq \mathcal{R}_5 & \mathcal{C}_4 \succeq \mathcal{C}_{c_2} \\ Z_5^{\langle\rangle} \succeq R_4 & \mathcal{R}_{c_3} \succeq \mathcal{R}_3 & \mathcal{C}_5 \succeq \mathcal{C}_{c_2} \\ \mathcal{R}_1 \succeq^* X_0 & \mathcal{R}_{c_3} \succeq \mathcal{R}_{c_2} & \mathcal{C}_5 \succeq \mathcal{R}_4 \\ \mathcal{R}_2 \succeq \bot & \mathcal{R}_{d_1} \succeq \mathcal{R}_{c_1} & \mathcal{C}_3 \succeq \mathcal{C}_{c_3} \\ \mathcal{R}_3 \succeq^* X_0 & \mathcal{R}_{d_1} \succeq \mathcal{R}_{c_3} & \mathcal{C}_{c_2} \succeq \mathcal{C}_{c_3} \\ \mathcal{R}_5 \succeq \bot & \mathcal{C}_{c_2} \succeq \mathcal{R}_3 & \mathcal{C}_{c_1} \succeq \mathcal{C}_{d_1} \\ Z_0 \succeq Z_2 & Z_0 \succeq Z_5 & \mathcal{C}_{c_3} \succeq \mathcal{C}_{d_1} \end{array} \right\}$$

If we denote with $\mathcal{C}_{append}$ *the union of the sets of constraints derived for the unifications and structured subgoals of* `append/3`*'s body goal (see Examples 4.5 and 4.6), we have that* $\mathcal{C}_0$ *simply equals* $\mathcal{C}_{append} \cup \mathcal{O}_{append}$*. In fact, the set of constraints derived in the i'th round also consists of the set* $\mathcal{C}_{append} \cup \mathcal{O}_{append}$ *but augmented with the renamed constraints from* μ_{i-1} *for the procedure calls in the body goal. The derived sets of constraints are depicted in Fig. 4.6*

Remains to define for a constraint system $\mathcal{C}$, its equivalent constraint system in normal form, $\mathcal{C}_N$. A set of binding-time constraints can be brought to normal form by repeatedly applying the rewrite rules from Definition 4.7 until a fixed point is reached. Rewriting a constraint involves unfolding the (subvalue of the) binding-time variable in its right-hand side with respect to a constraint on (a subvalue of) this variable. If we consider two subvalues of a binding-time variable, say X_η^δ and X_η^γ, one of them is a subvalue of the other if either δ is an extension of γ or vice versa. This is captured by the following definition:

Definition 4.6 *We define* $ext : TPath \times TPath \mapsto TPath \times TPath$ *as follows:*

$$ext(\gamma, \delta) = \left\{ \begin{array}{ll} (\langle\rangle, \epsilon) & \text{if } \gamma = \delta \bullet \epsilon \\ (\epsilon, \langle\rangle) & \text{if } \gamma \bullet \epsilon = \delta \\ \text{undefined} & \text{otherwise} \end{array} \right.$$

Note that if $ext(\gamma, \delta) = (\epsilon, \epsilon')$ then $\gamma \bullet \epsilon = \delta \bullet \epsilon'$. Unfolding a constraint $X_\eta^\gamma \succeq Y_{\eta'}^\delta$ with respect to another constraint result in a new constraint on (a subvalue of) X_η^γ, with as right hand side the appropriate subvalue of the right hand side of the constraint that was used for unfolding. To denote a subvalue of a constraint's

i	$\mathcal{C}_i$	$\mu_i(append)$
1	$\mathcal{C}_{append}$ $\cup$ $\mathcal{O}_{append}$	$Z_2 \succeq Y_0$ $E_3 \succeq X_0^{\langle[],1\rangle}$ $Es_3 \succeq X_0$ $Z_5^{\langle[],1\rangle} \succeq X_0^{\langle[],1\rangle}$ $\mathcal{R}_1 \succeq^* X_0$ $\mathcal{R}_3 \succeq^* X_0$ $\mathcal{R}_{c_1} \succeq^* X_0$ $\mathcal{R}_{c_3} \succeq^* X_0$ $\mathcal{C}_2 \succeq^* X_0$ $\mathcal{C}_4 \succeq^* X_0$ $\mathcal{C}_5 \succeq^* X_0$ $Z_0 \succeq Y_0$ $Z_0^{\langle[],1\rangle} \succeq X_0^{\langle[],1\rangle}$ $\mathcal{R}_{d_1} \succeq^* X_0$
2	$\mathcal{C}_{append}$ $\cup$ $\mathcal{O}_{append}$ $\cup$ $\left\{ \begin{array}{c} R_4 \succeq Y_0 \\ R_4^{\langle[],1\rangle} \succeq Es_3^{\langle[],1\rangle} \\ \mathcal{R}_4 \succeq^* Es_0 \end{array} \right\}$	$Z_2 \succeq Y_0$ $E_3 \succeq X_0^{\langle[],1\rangle}$ $Es_3 \succeq X_0$ $Z_5^{\langle[],1\rangle} \succeq X_0^{\langle[],1\rangle}$ $Z_5 \succeq Y_0$ $\mathcal{R}_1 \succeq^* X_0$ $\mathcal{R}_3 \succeq^* X_0$ $\mathcal{R}_{c_1} \succeq^* X_0$ $\mathcal{R}_{c_2} \succeq^* X_0$ $\mathcal{R}_{c_3} \succeq^* X_0$ $\mathcal{C}_{c_2} \succeq^* X_0$ $\mathcal{C}_2 \succeq^* X_0$ $\mathcal{C}_4 \succeq^* X_0$ $\mathcal{C}_5 \succeq^* X_0$ $Z_0 \succeq Y_0$ $Z_0^{\langle[],1\rangle} \succeq X_0^{\langle[],1\rangle}$ $\mathcal{R}_{d_1} \succeq^* X_0$
3	idem	idem

Figure 4.6: Analysing `append/3` using Δ_S.

right hand side S, we use the notation $S^{\overline{\bullet\epsilon}}$. If S denotes a variable X_η^γ, then $S^{\overline{\bullet\epsilon}}$ equals $X_\eta^{\overline{[\gamma\bullet\epsilon]}}$. Otherwise, if S denotes the constants $\bot$ or $\top$, $S^{\overline{\bullet\epsilon}}$ simply equals S. Note the use of the minimal element of the equivalence class, $\overline{[\gamma \bullet \eta]}$, to denote an element of the appropriate type graph $\mathcal{L}_t^{\equiv}$ (rather than the type tree $\mathcal{L}_t$).

Definition 4.7 *Given two binding-time constraint systems $\mathcal{C}$ and $\mathcal{C}'$. We say that $\mathcal{C}'$ is a rewriting of $\mathcal{C}$ if each of the following rules hold:*

$$\{X_\eta^\gamma \succeq Y_{\eta'}^\delta, Y_{\eta'}^{\delta'} \succeq S\} \subseteq \mathcal{C} \text{ and } ext(\delta,\delta') = (\epsilon,\epsilon') \Rightarrow \{X_\eta^{\overline{[\gamma\bullet\epsilon]}} \succeq S^{\bullet\epsilon'}\} \subseteq \mathcal{C}' \quad (4.2)$$

$$\begin{array}{c} \{X_\eta^\gamma \succeq Y_{\eta_0}^\delta\} \subseteq \mathcal{C} \Rightarrow \{X_\eta^\gamma \succeq Y_{\eta_0}^\delta\} \subseteq \mathcal{C}' \\ \{X_\eta^\gamma \succeq \top\} \subseteq \mathcal{C} \Rightarrow \{X_\eta^\gamma \succeq \top\} \subseteq \mathcal{C}' \end{array} \quad (4.3)$$

$$\{X_\eta^\gamma \succeq Y_{\eta'}, Y_{\eta'} \succeq^* S\} \subseteq \mathcal{C} \Rightarrow \{X_\eta^\gamma \succeq^* S\} \subseteq \mathcal{C}' \quad (4.4)$$

$$\{Y_\eta \succeq^* X_{\eta'}^\delta, X_{\eta'}^{\delta'} \succeq S\} \subseteq \mathcal{C} \text{ and } ext(\delta,\delta') = (\langle\rangle,\epsilon) \Rightarrow Y_\eta \succeq^* S^{\overline{\bullet\epsilon}} \quad (4.5)$$

We say that a rewriting $\mathcal{C}'$ of $\mathcal{C}$ is a minimal *rewriting, denoted by $\mathcal{C} \rhd \mathcal{C}'$ if and only if for each rewriting $\mathcal{C}''$ of $\mathcal{C}$ holds that $\mathcal{C}' \subseteq \mathcal{C}''$.*

The following proposition states that repeatedly constructing a minimal rewriting of an initial constraint system reaches a finitary fixed point, which is in normal form.

Proposition 4.1 *Let $\mathcal{C}_0$ denote a constraint system associated to the binding-time variables of a procedure p. There exist a finite amount of constraint systems $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ such that $\mathcal{C}_0 \rhd \mathcal{C}_1 \rhd \mathcal{C}_2 \rhd \ldots \rhd \mathcal{C}_n$ and the constraints from $\mathcal{C}_n$ are in normal form with respect to p.*

Proof Consider a sequence $\mathcal{C}_0 \rhd \mathcal{C}_1 \rhd \mathcal{C}_2 \rhd \ldots$ Since $\mathcal{C}_0$ is a set of binding-time constraints associated to a single procedure p, the involved constraints do not contain loops (the data flow in p is well-moded and also the constraints on the control variables of p do not contain loops by construction). Since, for $i > 0$, $\mathcal{C}_i$ is a minimal rewriting of $\mathcal{C}_{i-1}$, the constraints of $\mathcal{C}_i$ are either the result of unfolding a constraint from $\mathcal{C}_{i-1}$ with respect to another constraint of $\mathcal{C}_{i-1}$ (rewrite rules 4.2, 4.4 and 4.5) or constraints taken unchanged from $\mathcal{C}_{i-1}$ (rewrite rule 4.3). Consequently, also $\mathcal{C}_i$ does not contain loops, and hence unfolding can not continue infinitely and there must exist $n \in \mathbb{N}$ such that $\mathcal{C}_n = \mathcal{C}_{n-1}$. Since $\mathcal{C}_n$ contains only constraints that are taken unchanged from $\mathcal{C}_{n-1}$ (by rewrite rule 4.3), the constraints of $\mathcal{C}_n$ are in normal form.

□

4.2.3 From a symbolic program environment to binding-times

The previous section introduces the first, call independent phase of the binding-time analysis. In what follows, we define how the result of this first phase can be used to compute a congruent program environment. Moreover, we prove the obtained program environment to be equivalent with the one that is obtained by the analysis from Chapter 3. First we define how a constraint system in normal form can be used to obtain binding-times for a procedure's binding-time variables, given binding-times for its input arguments.

Definition 4.8 *Let $\mathcal{C}$ denote a constraint system in normal form with respect to a predicate p and $\sigma_0 : \mathcal{V}_{\mathcal{BT}} \mapsto \mathcal{BT}$ be a substitution with $dom(\sigma_0) = \{V_{\eta_0} | V \in \mathtt{in}(p)\}$. The* solution of $\mathcal{C}$ induced by σ_0 *is the substitution σ such that $dom(\sigma) = \{V_\eta | V \in \mathcal{V}(p), \eta \in \mathtt{init}(V)\}$ and such that $\forall V_\eta \in dom(\sigma)$ it holds that*

$$\sigma(V_\eta) = \begin{cases} \sigma_0(V_\eta) & \text{if } V \in \mathtt{in}(p) \\ \bot \sqcup S_{V_\eta} \sqcup S'_{V_\eta} & \text{if } V : t \in (\mathcal{V}(p) \setminus \mathtt{in}(p)) \end{cases}$$

where

$$\begin{array}{lcl} S_{V_\eta} & = & \{(\gamma \bullet \epsilon, \mathit{dynamic}) \mid V_\eta^\gamma \succeq \top \in \mathcal{C} \text{ and } \epsilon \in \mathcal{L}_{\overline{t^\gamma}}^{\equiv}\} \\ S'_{V_\eta} & = & \{(\gamma \bullet \epsilon, \beta^\delta(\epsilon)) \mid V_\eta^\gamma \succeq X_{\eta_0}^\delta \in \mathcal{C}, \beta = \sigma_0(X_{\eta_0}) \text{ and } \epsilon \in \mathcal{L}_{\overline{t^\gamma}}^{\equiv}\} \end{array}$$

It can be easily verified that a solution induced by σ_0 is the *minimal* solution σ such that $\sigma_{|dom(\sigma_0)} = \sigma_0$ (we say in this case that the solution is minimal *with respect to* σ_0). This guarantees that there does not exist a solution to the constraint system in which binding-time variables are associated with less dynamic binding-times. In other words, the solution induced by σ_0 is the "best" binding-time characterisation of a procedure's variables (with respect to the binding- times of the procedure's input variables from σ_0).

Example 4.10 *Let μ denote the symbolic program environment with $dom(\mu) = \{append\}$ from Example 4.9. Let β_l denote a binding-time associated to the type* `list(T)` *of the form*

$$\beta_l = \{(\langle\rangle, \mathit{static}), (\langle[|], 1\rangle, \mathit{dynamic})\}$$

approximating those terms of type `list(T)` *that are at least bound to a list skeleton. If σ_0 denotes the substitution $\{X_0/\beta_l, Y_0/\beta_l\}$, the solution of $\mu(append)$ induced by σ_0 is the substitution σ defined as*

$$\sigma = \left\{ \begin{array}{lll} X_0/\beta_l & Y_0/\beta_l & Z_0/\beta_l \\ Z_2/\beta_l & Z_5/\beta_l & E_3/\{(\langle\rangle, \mathit{dynamic})\} \\ Es_3/\beta_l & & \end{array} \right\}$$

and each of the control variables mappend to $(\langle\rangle, static)\}$.

If σ is an induced solution to the binding-time constraints of a procedure p, σ associates a binding-time to binding-time variable rather than a program variable. Recall from section 4.2.1 that, in order to compute a program variable's binding-time at a particular program point η, it suffices to take the least upper bound of the appropriate binding-time variable's values, that is

$$\bigsqcup_{\eta' \in \mathtt{reach}(X,\eta)} \sigma(X_{\eta'}).$$

Now, suppose that the initial procedure call for which a binding-time analysis needs to be performed is $p(t_1, \ldots, t_n)$. If μ represents the symbolic program environment resulting from call *independent* analysis, the remaining call *dependent* phase of the analysis consists of computing the solution of $\mu(p)$ induced by a substitution σ_0 – associating p's input arguments with the binding-times from the initial call – and repeat this process, for each call in p's body goal with respect to a call pattern consisting of the binding-times of the call's input arguments as computed by the current solution. Formally, we represent the result of the call dependent analysis phase by an *induced* program environment. Such an induced program environment is a mapping from a procedure to a set of binding-time substitutions. Note that, due to the use of binding-time variables, this differs slightly from the representation of the program environment produced by the analysis of Chapter 3. We return to the relation between the results of both analyses shortly.

Definition 4.9 *If μ is a symbolic program environment representing the result of call independent analysis and $p(t_1, \ldots, t_n)$ is an initial call, the program environment* induced by μ and $p(t_1, \ldots, t_n)$ *is denoted by $\Psi^{\mu}_{p(\overline{t})}$ and defined as the smallest mapping $Proc \mapsto \wp(\mathcal{V}_{\mathcal{BT}} \mapsto \mathcal{BT})$ such that*

- *$\sigma_0 \in \Psi^{\mu}_{p(\overline{t})}(p)$ where σ_0 is the solution to $\mu(p)$ induced by*

  $$\left\{ \; (V_{\eta_0}, \alpha(t)) \;\middle|\; V \in \mathtt{in}(p) \text{ and } t \text{ the term in } p(\overline{t}) \text{ corresponding with } V \; \right\}$$

- *if $\sigma \in \Psi^{\mu}_{p(\overline{t})}(q)$ for some q and $r(X_1, \ldots, X_n)$ is a procedure call in $\mathcal{B}ody(q)$ at program point η, then $\exists \sigma' \in \Psi^{\mu}_{p(\overline{t})}(r)$ where σ' is the solution to $\mu(r)$ induced by*

  $$\left\{ \; (V_{i_{\eta_0}}, \beta_i) \;\middle|\; V_{i_{\eta_0}} \in \mathtt{in}(p) \; \right\}$$

  where

  $$\beta_i = \bigsqcup_{\eta' \in \mathtt{reach}(X,\eta)} \sigma(X_{\eta'}).$$

Note that such an induced program environment can be constructed by a simple algorithm:

- Given a procedure p and a binding-time substitution σ_0 with $dom(\sigma_0) = \mathtt{in}(p)$, compute the solution σ induced by σ_0.
- Construct, for each call $r(\overline{X})$ occurring in $\mathcal{B}ody(p)$ a binding-time substitution σ_0' mapping the called procedure's *formal* input arguments, $\mathcal{A}rgs(r)$, to the binding-times of the call's *actual* input arguments (computed by σ). Repeat this process for each such procedure r with respect to σ_0'.

Note that such an algorithm processes every procedure body only once for every encountered call pattern. The relation between an induced program environment $\Psi^{\mu}_{p(\overline{t})}(q)$ and the annotated versions that are created for q is immediate, since there is a one-to-one correspondence between an entry $\sigma \in \Psi^{\mu}_{p(\overline{t})}(q)$ and an annotated version of q, since the annotations are created in function of the involved variable's binding-times which are in turn determined by σ. Hence, in what follows, we refer by the notion of "annotation phase" to the call dependent phase of binding-time analysis, be it the construction of an (induced) program environment, or the effective creation of annotated versions.

4.2.4 On the modularity of the approach

Let us now return to the topic of modularity. Binding-time analysis is by nature a call dependent process. Indeed, annotated versions of a procedure are created according to the calls that occur to that procedure. If a call to a procedure with respect to some call pattern results in a call to another procedure, than also a version of the latter with respect to the binding-times from the call's arguments must be created.

However, in the previous sections we have developed a binding-time analysis that is to be performed in two phases. The first phase of the process performs the data flow analysis in a symbolic way, by computing constraints that express binding-time relations between the variables of a procedure. Being call independent, generating the constraints associated to a procedure's body only involves renaming the constraints that are associated to the procedures that are called from within the body. It does not require the binding-times of the call's arguments. For a program that is divided into several modules, this means that the constraint generating phase of the analysis can be performed on a single module, without considering any particular call to the module. It only requires the constraints associated to the procedures from the interfaces of the imported modules. More formally, let μ^M denote the symbolic program environment associated to a module M and let $\overline{\mu}^M$ denote this environment restricted to the procedures from M's interface. If we then define, for each module M

$$\mu_0^M = \bigcup_{M' \in \mathit{imported}(M)} \overline{\mu}^{M'}$$

$$\mu_i^M = \Delta_S(M, \mu_{i-1}^M) \text{ for all } i > 0$$

We define the result of the (first phase) analysis for a module M as μ_n^M with $n \in \mathbb{N}$ the smallest such n for which $\mu_n^M = \mu_{n-1}^M$.

If we consider module hierarchies without circularities, this phase of the analysis can be performed in a bottom-up way. Reconsider the module hierarchy from Fig. 4.1. The result of bottom-up analysis of this hierarchy is depicted in Fig. 4.7. First, the modules at the bottom level, M_4 and M_5 are analysed. Since these modules do not import any other modules, they can be analysed starting from an empty initial program environment. The rounded boxes in the figure denote the symbolic program environment resulting from analysing a particular module. The shaded part of the box represent this environment restricted to the module's interface procedures. Subsequently, the modules M_2 and M_3 can be analysed, since

Figure 4.7: Bottom-up analysis of the module hierarchy.

their analysis only requires the constraints from the interface procedures of M_4, respectively M_4 and M_5. Finally, since now the result is available of analysing M_2, M_3 and M_5, the module M_1 can be analysed. Note that in this process, each module is analysed only once. If a module, like M_5 in the example, is imported in more than one module, analysing the latter modules only requires the result of analysing the former.

Example 4.11 *Consider the following definition of the* `reverse/2` *procedure:*

```
rev(A,B):-
     A = [], B = [] ;
     A = [E|Es], rev(Es, Bs), append(Bs, [E], B).
```

If the definition of `append/3` *is given in a different module, analysis of the module comprising* `rev/2` *starts from the program environment* μ_0 *with* $dom(\mu_0) = \{append, rev\}$ *and defined as follows:*

$$\mu_0(rev) = \emptyset$$
$$\mu_0(append) = \{Z \succeq Y, Z^{\langle[],1\rangle} \succeq X^{\langle[],1\rangle}\}$$

with $\mu_0(append)$ *the simplified result of an earlier analysis of* `append/3` *(See Example 4.9). The result of analysing the* `rev/2` *predicate (again restricted to the program variables and associating only a single program point with each variable) is as follows:*

$$\{B \succeq \bot, B^{\langle[],1\rangle} \succeq A^{\langle[],1\rangle}\}$$

the first (superfluous) constraint results from the first disjunct, the second from the second disjunct. In the minimal solution to the above constraint set, the elements of B *will be characterised as known during specialisation only if the elements of* A *are.*

The second phase of the analysis consists of annotating procedures, which requires computing solutions to the constraint systems that are derived in the first phase. It is the second phase of the process that reflects the call-dependent nature of binding-time analysis. Indeed, when creating an annotated version of a procedure, say p, the analysis might encounter a call to another procedure. The binding-times of the call's arguments are immediate, from the solution to the constraint system associated to p, but the called predicate must in turn be annotated with respect to these computed binding-times. When caller and callee are part of a same module, this is not a problem since the source code and constraint systems of both procedures are available. Keeping track of the call patterns for which a procedure was annotated and linking this call pattern with the right annotated version is sufficient to guarantee that a procedure is not annotated twice with respect to the same call pattern.

The story is different, though, when caller and callee are in different modules. The source code of the called procedure is to be found in an imported module, its constraint system in the latter module's symbolic program environment. Consequently, one may conclude that performing the second phase of the analysis on a multi module program requires the complete source of the program (all modules), together with symbolic program environments of all modules. This corresponds to analysing a multi module program as a monolithic program, and thus at first sight the problems mentioned in Section 4.1 are still present in our analysis. However, we note the following:

- The remaining problem is mostly a problem of storage – keeping all the module's source code and constraint systems available to the annotation process. Computing the least solution to a constraint system with respect to

an initial substitution is a rather cheap process, as it merely consists of performing a substitution on the constraints and computing least upper bounds. Moreover, since all dependencies of other procedures are incorporated during generation of the constraint system, computing a solution for a constraint system is independent of the fact whether it is computed in context of a single module, or a multi-module program.

- In principle, one could also guarantee that the second phase of the analysis can be performed one module at a time, bottom-up in the module hierarchy. Indeed, since the representation of binding-times is finite for each type, the number of call patterns with respect to which a procedure can be annotated is also finite. Hence, one could create *all* possible annotated versions of a module's interface predicates. During annotation, if a call is encountered to a procedure from an imported module, the binding-times of the call's arguments must be computed, and the right annotated version of the called predicate, depending on the computed call pattern, must be selected. If this is the case, modules can be annotated one at a time, bottom-up in the module hierarchy. In this case, the only information required from the imported modules is a mapping, for each interface procedure, between call patterns and the associated renamings of the procedure.

Note that for a practical system, it might be worthwhile to consider a tradeoff between storage space and analysis time. One could analyse a number of a module's interface procedures with respect to their "frequently occurring" call patterns and store a mapping between the call pattern and the name of the particular annotated version for later use. When annotating, in another module, a procedure that contains such a call, it is sufficient to consult the stored mapping, and rename the call to refer to the right annotated version, without the need to (re)annotate the called procedure with respect to this particular call pattern. On the other hand, if a call is encountered to another module's interface procedure and no suitable annotated version is stored for that module, it must be created, requiring the (re)analysis of some of the imported module's procedures (possibly propagating down the module hierarchy). The disadvantage of such a combined system is of course the extra bookkeeping that is required to associate a call pattern with a particular annotated version, when it is available.

The above exposition assumes that no circular dependencies are present in the module hierarchy. We discuss a modular approach in the presence of such circularities in Section 4.4.4, after having extended the analysis to deal with Mercury's higher-order features.

4.3 Correctness of the Analysis

In this section, we prove correctness of the constraint approach for binding-time analysis, by proving it equivalent with respect to the analysis of Chapter 3. In what follows, we assume a (monolithic) program P, and define the following relation between a program environment as derived by the analysis of Chapter 3 and a symbolic program environment as derived by the constraint approach of the current chapter.

Definition 4.10 *Given a program environment $\Psi \in \mathcal{PE}nv$ and a symbolic program environment $\mu \in \mathcal{PE}nv_S$. We say that Ψ and μ are* nearly output equivalent *if the following holds for all $p \in dom(\Psi)$ and $\pi_0 \in dom(\Psi(p))$:*

- *let σ be the minimal solution of $\mu(p)$ with respect to $\sigma_0 = \{(V_{\eta_0}, \pi_0(V) \mid V \in \mathtt{in}(p)\}$.*
- *$\sigma(\mathcal{R}_{\eta_b}) \succeq \mathcal{R}[\![\mathcal{B}ody(p)]\!]\pi_0\Psi$ where η_b is the program point identifying $\mathcal{B}ody(p)$*
- *$\sigma(V_{\eta_0}) \succeq \pi(V)$ for all $V \in \mathtt{out}(p)$ where $\pi = (\Psi(p))(\pi_0)$.*

We say that Ψ and μ are output-equivalent *if $\sigma(V_{\eta_0}) = \pi(V)$ for all such p, π_0 and V.*

Definition 4.10 states that Ψ and μ are nearly output equivalent if, for each procedure p and binding-time environments $(\pi_0, \pi) \in \Psi(p)$ the binding-time recorded in π for each output variable V of p is approximated by the the binding-time that is associated to V_{η_0} in σ, where σ is the minimal solution of $\mu(p)$ with respect to a substitution $\sigma_0 = \{(V_{\eta_0}, \pi_0(V) \mid V \in \mathtt{in}(p)\}$ that associates to p's input variables to the same binding-times as in the initial binding-time environment π_0. If the recorded binding-times are equal, Ψ and μ are said to be *output equivalent*. Intuitively, output equivalence states that the information recorded in Ψ is also retrievable from μ. This does not necessarily hold the other way round. Indeed, being the result of a call-independent analysis, μ represents possibly *more* information than Ψ, since least solutions can be computed for $\mu(p)$ with respect to an initial substitution σ_0 that has no corresponding binding-time environment π_0 in $dom(\Psi(p))$.

In what follows, we will prove that the program environment resulting from the analysis of Chapter 3 and the symbolic program environment resulting from the call-independent analysis are output equivalent. As a consequence, we can prove that they both associate the same binding-time to any variable at any program point. The correctness result is formally stated in the following theorem:

Theorem 4.1 *Let P be a monolithic program with an associated initial call $p(\overline{t})$. Let Ψ_0 denote a program environment such that*

$$\Psi_0(p) = \{(\{\ (F_i, \alpha(t_i)) \mid F_i \in \mathtt{in}(p)\ \}, \{\ (F_i, \bot) \mid F_i \in \mathtt{out}(p)\ \})\}$$

and $\Psi_0(q) = \emptyset$ *for all procedures* $q \neq p$ *and* μ_0 *a symbolic program environment such that* $\mu_0(p) = \emptyset$ *for all procedures* p*. For each* $i \in \mathbb{N}$*,* $i > 0$*, let*

$$\Psi_i = \Delta(P, \Psi_{i-1}) \quad \textit{and} \quad \mu_i = \Delta_S(P, \mu_{i-1}).$$

If n *is the smallest natural number such that* $\Psi_n = \Psi_{n-1}$*, then it holds that also* $\mu_n = \mu_{n-1}$ *and* Ψ_n *and* μ_n *are output equivalent.*

To prove Theorem 4.1, we will first prove for each i, that Ψ_i and μ_i are nearly output equivalent. In other words, we prove that the information that is derived in the i'th round of the analysis of Chapter 3 is approximated by the appropriate least solutions of the constraint system that is derived in the i'th round of the call independent analysis. Furthermore, we prove that if Ψ_n is the fixed point of the analysis, the appropriate least solutions of the constraint systems in μ_n contain the same binding-times as Ψ_n. The proof of Theorem 4.1 is accomplished by induction on the number of analysis rounds. The main body of the proof comprises the induction step: given a program environment Ψ_i and symbolic program environment μ_i that are nearly output equivalent, we prove that also Ψ_{i+1} and μ_{i+1} are nearly output equivalent. In what follows, we will always compare the binding-times (or control information) derived from Ψ for a procedure p and initial binding-time environment π_0 with the appropriate binding-times derived from the minimal solution of $\mu(p)$ induced by a substitution σ_0 mapping p's input arguments to their respective binding-times from π_0. However, to ease the formulation we will simply consider binding-times (or control information) "derived from Ψ" versus those "derived from μ". To ease the proof, we introduce a number of auxiliary lemma's. The first two such lemmas consider the relation between the control information that is derived from a program environment Ψ and a symbolic program environment μ that are nearly output equivalent. The first lemma states that the result of computing – from Ψ – whether or not a goal reduces during specialisation to *true*, *fail* or residual code that is guaranteed to succeed is approximated by the control information derived from μ under the assumption that the same holds for the atomic subgoals of the goal.

Lemma 4.1 *Let* Ψ *be a program environment that is (nearly) output equivalent with a symbolic program environment* μ *and let* p*,* π_0 *and* σ_0 *be as in Definition 4.10, and* σ *the minimal solution with respect to* σ_0 *of* $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$*. Let* G_η *be a subgoal of* $\mathcal{B}ody(p)$*. If it holds that*

$$\mathcal{R}[\![A_{\eta'}]\!]\pi_{\eta'}\Psi \preceq \sigma(\mathcal{R}_{\eta'})$$

for every atomic subgoal $A_{\eta'}$ *of* G_η *where* $\pi_{\eta'} = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta' p \pi_0 \Psi$*, then it holds that*

$$\mathcal{R}[\![G_\eta]\!]\pi_\eta\Psi \preceq \sigma(\mathcal{R}_\eta)$$

where $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta p \pi_0 \Psi$

Proof We prove this lemma by structural induction. For atomic goals, the proof is immediate by the condition of the lemma. For non-atomic goals, we prove that the lemma holds under the assumption that it holds for the goal's subgoals. we consider each case separately, but the proof is analogous in each case.

- Suppose $G_\eta = not(G'_{\eta'})$. By definition of $\mathcal{R}$ we have that

$$\mathcal{R}[\![not_\eta(G'_{\eta'})]\!]\pi_\eta\Psi = \mathcal{R}[\![G'_{\eta'}]\!]\pi_\eta\Psi \preceq \sigma(\mathcal{R}_{\eta'})$$

  by induction hypothesis and the fact that π_η also equals the binding-time environment at program point η': $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta' p\pi_0\Psi$. Now, proof follows since $\sigma(\mathcal{R}_{\eta'}) = \sigma(\mathcal{R}_\eta)$ due to the fact that σ is the least solution to $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and $\mathcal{R}_\eta \succeq \mathcal{R}_{\eta'}$ is the only constraint on $\mathcal{R}_\eta$ in $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$.

- Suppose $G_\eta = (G'_{\eta'}; G''_{\eta''})$. By definition of $\mathcal{R}$, we have that

$$\mathcal{R}[\![(G'_{\eta'}; G''_{\eta''})]\!]\pi_\eta\Psi = \mathcal{R}[\![G'_{\eta'}]\!]\pi_\eta\Psi \sqcup \mathcal{R}[\![G''_{\eta''}]\!]\pi_\eta\Psi.$$

  Since $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta' p\pi_0\Psi = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta'' p\pi_0\Psi$, the induction hypothesis hold and we rewrite the above into

$$\mathcal{R}[\![(G'_{\eta'}; G''_{\eta''})]\!]\pi_\eta\Psi \preceq \sigma(\mathcal{R}_{\eta'}) \sqcup \sigma(\mathcal{R}_{\eta''}) = \sigma(\mathcal{R}_\eta)$$

  since σ is the least solution to $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and $\mathcal{R}_\eta \succeq \mathcal{R}_{\eta'}$ and $\mathcal{R}_\eta \succeq \mathcal{R}_{\eta''}$ are the only constraints on $\mathcal{R}_\eta$ in the set $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$.

- Suppose $G_\eta = (G'_{\eta'}, G''_{\eta''})$. By definition of $\mathcal{R}$, we have that

$$\mathcal{R}[\![(G'_{\eta'}, G''_{\eta''})]\!]\pi_\eta\Psi = \mathcal{R}[\![G_{\eta'}]\!]\pi_\eta \sqcup \mathcal{R}[\![G_{\eta''}]\!]\pi_{\eta''}$$

  where $\pi'' = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta'' p\pi_0\Psi$. By induction hypothesis, we rewrite the above into

$$\mathcal{R}[\![(G'_{\eta'}, G''_{\eta''})]\!]\pi_\eta\Psi \preceq \sigma(\mathcal{R}_{\eta'}) \sqcup \sigma(\mathcal{R}_{\eta''}) = \sigma(\mathcal{R}_\eta)$$

  since σ is the least solution of $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and $\mathcal{R}_\eta \succeq \mathcal{R}_{\eta'}$ and $\mathcal{R}_\eta \succeq \mathcal{R}_{\eta''}$ are the only constraints on $\mathcal{R}_\eta$ in the set $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$.

- Suppose $G_\eta = \mathit{if}\, G'_{\eta'}\ \mathit{then}\, G''_{\eta''}\ \mathit{else}\, G'''_{\eta'''}$. By definition of $\mathcal{R}$, we have that

$$\mathcal{R}[\![\mathit{if}\, G'_{\eta'}\ \mathit{then}\, G''_{\eta''}\ \mathit{else}\, G'''_{\eta'''}]\!]\pi_\eta\Psi$$
$$=$$
$$\mathcal{R}[\![G'_{\eta'}]\!]\pi_\eta\Psi \sqcup \mathcal{R}[\![G''_{\eta''}]\!]\pi_{\eta''}\Psi \sqcup \mathcal{R}[\![G'''_{\eta'''}]\!]\pi_\eta\Psi$$

where $\pi_{\eta''} = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta'' p \pi_0 \Psi$. Since also $\Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta' p \pi_0 \Psi = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta''' p \pi_0 \Psi = \pi_\eta$ and the induction hypothesis hold, we rewrite the above into

$$\mathcal{R}[\![if G'_{\eta'} \; then G''_{\eta''} \; else G'''_{\eta'''}]\!]\pi_\eta \Psi \preceq \sigma(\mathcal{R}_{\eta'}) \sqcup \sigma(\mathcal{R}_{\eta''}) \sqcup \sigma(\mathcal{R}_{\eta'''}) = \sigma(\mathcal{R}_\eta)$$

since σ is the least solution to $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and $\mathcal{R}_\eta \succeq \mathcal{R}_{\eta'}$, $\mathcal{R}_\eta \succeq \mathcal{R}_{\eta''}$ and $\mathcal{R}_\eta \succeq \mathcal{R}_{\eta'''}$ are the only constraints on $\mathcal{R}_\eta$ in $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$.

□

Before we can state a second auxiliary lemma, we need to reason about the relative positions of the atoms and goals in a procedure's body goal. We therefore introduce the following notion:

Definition 4.11 *Given two program points η, η' identifying atoms in a procedure p's body goal. We say that η* precedes *η', denoted by $\eta < \eta'$ if and only if there exist a control flow path in p of the form $\langle \ldots \eta, \ldots, \eta' \ldots \rangle$ (that is, η is positioned at the left hand side of η').*

The precedes relation is extended in a straightforward way to deal with the relative position of goals:

Definition 4.12 *Given two subgoals G_1 and G_2 identified by η_1 and η_2 of a procedure p's body goal. We say that η_1* precedes *η_2, denoted by $\eta_1 < \eta_2$ if and only if the following holds: if η'_2 denotes an atomic subgoal of G_2, then for each η'_1 identifying an atomic subgoal in G_1 it holds that $\eta'_1 < \eta'_2$.*

Given Ψ and μ that are nearly output equivalent, the second lemma states that the result of computing – from Ψ – whether a goal G at program point η is under dynamic control with respect to the procedure's body goal is approximated by $\sigma(\mathcal{C}_\eta)$, with σ being the appropriate minimal solution to μ, under the condition that for each atomic subgoal $A_{\eta'}$ that precedes G_η in the procedure's body holds that the result of computing – from Ψ – whether $A_{\eta'}$ can be reduced to *true*, *fail* or code that is guaranteed to succeed is approximated by $\sigma(\mathcal{R}_{\eta'})$.

Lemma 4.2 *Let Ψ be a program environment that is (nearly) output equivalent with a symbolic program environment μ and let p, π_0 and σ_0 be as in Definition 4.10, and σ the minimal solution with respect to σ_0 of $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$. Let G_η be a subgoal of $\mathcal{B}ody(p)$. If for each atomic subgoal $A_{\eta'}$ of $\mathcal{B}ody(p)$ with η' preceding η holds that $\mathcal{R}[\![A_{\eta'}]\!]\pi_{\eta'}\Psi \preceq \sigma(\mathcal{R}_{\eta'})$, then*

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta)$$

where $\pi_{\eta'} = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta'\pi_0\Psi p \pi_0$

Proof We prove this by structural induction on p's body goal. For the body goal itself with its associated program point η_b, it holds by definition of $\mathcal{C}$ that $\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta_b\pi_0 = \bot$. Since σ is the minimal solution to $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and no constraints exist on $\mathcal{C}_{\eta_b}$ in $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$, also $\sigma(\mathcal{C}_{\eta_b}) = \bot$.

Now, we prove that if $\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta)$ for a subgoal G_η of $\mathcal{B}ody(p)$, then also $\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta'\pi_0\Psi \preceq \sigma(\mathcal{C}_{\eta'})$ holds for any syntactic subgoal $G'_{\eta'}$ of G_η. Again, we prove this for each syntactic case separately.

- Suppose G_η is of the form $not(G'_{\eta'})$. From the definition of $\mathcal{C}$ we have that

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta'\pi_0\Psi = \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi \sqcup \mathcal{C}[\![not_\eta(G'_{\eta'})]\!]\eta'\pi_\eta\Psi$$

  where $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta p\pi_0\Psi$. By induction hypothesis, we rewrite the above into

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta'\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta) \sqcup \mathcal{C}[\![not_\eta(G'_{\eta'})]\!]\eta'\pi_\eta\Psi$$

  and again, using the fact that $\mathcal{C}[\![not_\eta(G'_{\eta'})]\!]\eta'\pi_\eta\Psi = \mathcal{C}[\![G'_{\eta'}]\!]\eta'\pi_\eta\Psi = \bot$ (from the definition of $\mathcal{C}$) into

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta'\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta) \sqcup \bot = \sigma(\mathcal{C}_{\eta'})$$

  the latter due to the fact that σ is the least solution to $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and $\mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta$ is the only constraint on $\mathcal{C}_{\eta'}$ in $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$.

- Suppose G_η is of the form $(G'_{\eta'}; G''_{\eta''})$. We prove the lemma for the subgoal $G'_{\eta'}$. The proof for the other subgoal $G''_{\eta''}$ is analogous. From the definition of $\mathcal{C}$ we have that

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta'\pi_0\Psi = \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi \sqcup \mathcal{C}[\![(G'_{\eta'}; G''_{\eta''})]\!]\eta'\pi_\eta\Psi$$

  where $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta p\pi_0\Psi$. By induction hypothesis and the fact that $\mathcal{C}[\![(G'_{\eta'}; G''_{\eta''})]\!]\eta'\pi_\eta\Psi = \mathcal{C}[\![G'_{\eta'}]\!]\eta'\pi_\eta\Psi = \bot$, we rewrite the above into

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta'\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta) \sqcup \bot = \sigma(\mathcal{C}_{\eta'})$$

  the latter due to the fact that σ is the least solution to $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and $\mathcal{C}_{\eta'} \succeq \mathcal{C}_\eta$ is the only constraint on $\mathcal{C}_{\eta'}$ in $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$.

- Suppose G_η is of the form $(G'_{\eta'}, G''_{\eta''})$. For the first subgoal $G'_{\eta'}$, the proof is identical as above, since the first disjunct of a disjunction and the first conjunct of a conjunction are treated identical by the $\mathcal{C}$

function and constraint generation. For the second conjunct, we have from the definition of $\mathcal{C}$ that

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta''\pi_0\Psi = \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi \sqcup \mathcal{C}[\![(G'_{\eta'}, G''_{\eta''})]\!]\eta''\pi_\eta\Psi$$

where $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta p\pi_0\Psi$. By induction hypothesis and the fact that $\mathcal{C}[\![(G'_{\eta'}, G''_{\eta''})]\!]\eta''\pi_\eta\Psi = \mathcal{R}[\![G'_{\eta'}]\!]\pi_\eta\Psi \sqcup \bot$ we rewrite the above into

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta''\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta) \sqcup \mathcal{R}[\![G'_{\eta'}]\!]\pi_\eta\Psi \sqcup \bot$$

Since each atomic subgoal of $G'_{\eta'}$ precedes $G'''_{\eta''}$, we conclude from the condition in the lemma together with Lemma 4.1 that $\mathcal{R}[\![G'_{\eta'}]\!]\pi_{\eta'}\Psi \preceq \sigma(\mathcal{R}_{\eta'})$. Now, $\pi_{\eta'} = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta' p\pi_0\Psi = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta p\pi_0\Psi = \pi_\eta$ and hence we rewrite the above into

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta''\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta) \sqcup \sigma(\mathcal{R}_{\eta'}) = \sigma(\mathcal{C}_{\eta''})$$

the latter due to the fact that σ is the least solution to $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and $\mathcal{C}_{\eta''} \succeq \mathcal{C}_\eta$ and $\mathcal{C}_{\eta''} \succeq \mathcal{R}_{\eta'}$ are the only constraints on $\mathcal{C}_{\eta''}$ in $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$.

- Suppose G_η is of the form $\mathit{if}\, G'_{\eta'}\, \mathit{then}\, G''_{\eta''}\, \mathit{else}\, G''''_{\eta''''}$. Again, the proof for the first subgoal is identical to the proof of the first conjunct in a conjunction since the analysis treats both subgoals identical. The proof for the second subgoal is identical to the proof of the second conjunct in a conjunction, again since analysis treats both subgoals in an identical way. Now, we prove that the lemma also holds for the third subgoal. From the definition of $\mathcal{C}$ we have that

$$\begin{array}{c}\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta'''\pi_0\Psi \\ = \\ \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi \sqcup \mathcal{C}[\![\mathit{if}\, G'_{\eta'}\, \mathit{then}\, G''_{\eta''}\, \mathit{else}\, G''''_{\eta''''}]\!]\eta'''p\pi_\eta\Psi\end{array}$$

since $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta'''p\pi_0\Psi$. By induction hypothesis and the fact that

$$\begin{array}{ll}\mathcal{C}[\![\mathit{if}\, G'_{\eta'}\, \mathit{then}\, G''_{\eta''}\, \mathit{else}\, G''''_{\eta''''}]\!]\eta'''\pi_\eta\Psi & = \mathcal{R}[\![G'_{\eta'}]\!]\pi_\eta\Psi \sqcup \mathcal{C}[\![G''''_{\eta''''}]\!]\eta'''\pi_\eta\Psi \\ & = \mathcal{R}[\![G'_{\eta'}]\!]\pi_\eta\Psi \sqcup \bot\end{array}$$

we rewrite the above into

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta'''\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta) \sqcup \mathcal{R}[\![G'_{\eta'}]\!]\pi_\eta\Psi \sqcup \bot$$

and use identical reasoning as in the proof of the second conjunct of a conjunction.

□

The previous lemmas define under what conditions the control information derived from Ψ is approximated by the control information derived from μ. In addition, the following lemma relates, for an atom in the procedure's body, the approximation of control information with the approximation of the binding-times of the atom's variables derived from Ψ and μ. It states that if the binding-times derived from Ψ for an atom A_η's input arguments are approximated by the binding-times of these variables derived from μ, than the result of computing – from Ψ – whether reducing the atom results in *true*, *fail* or code that is guaranteed to succeed is approximated by $\sigma(\mathcal{R}_\eta)$. Moreover, if the result of computing – from Ψ – whether the atom is under dynamic control with respect to the procedure's body is approximated by $\sigma(\mathcal{C}_\eta)$, it states that the binding-times computed from Ψ for the atom's output arguments are approximated by the binding-times for those arguments computed from μ.

Lemma 4.3 *Let Ψ be a program environment that is (nearly) output equivalent with a symbolic program environment μ and let p, π_0 and σ_0 be as in Definition 4.10, and σ the minimal solution with respect to σ_0 of $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$. If A_η is an atomic subgoal of $\mathcal{B}ody(p)$, and $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta p \pi_0 \Psi$, the following holds: if*

$$\forall V \in \mathtt{in}(A_\eta) : \pi_\eta(V) \preceq \bigsqcup_{\eta' \in \mathtt{reach}(V,\eta)} \sigma(V_{\eta'})$$

then

$$\mathcal{R}[\![A_\eta]\!]\pi_\eta\Psi \preceq \sigma(\mathcal{R}_\eta)$$

and if $\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta)$, then

$$\forall V \in \mathtt{out}(A_\eta) : \Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta p \pi_0 \Psi(V) \preceq \sigma(V_\eta).$$

Proof We consider each kind of atom separately.

- Suppose A_η is of the form $X := Y$. From the definition of $\mathcal{R}$, we have that $\mathcal{R}[\![A_\eta]\!]\pi_\eta\Psi = \bot$. Since σ is the least solution of $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and $\mathcal{R}_\eta \succeq \bot$ is the only constraint on $\mathcal{R}_\eta$ in $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$, also $\sigma(\mathcal{R}_\eta) = \bot$. Moreover,

  $$\sigma(X_\eta) = \bigsqcup_{\eta' \in \mathtt{reach}(Y,\eta)} \sigma(Y_{\eta'}) \sqcup \sigma(\mathcal{C}_\eta).$$

  Using the conditions from the lemma, we rewrite the above into

  $$\sigma(X_\eta) \succeq \pi_\eta(Y) \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi.$$

  The right hand side of the above equals X's value in the binding-time environment constructed by $\mathcal{G}[\![A_\eta]\!]\pi_\eta\Psi$, hence the above rewrites into

  $$\sigma(X_\eta) \succeq \Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta p \pi_0 \Psi(X).$$

- Suppose A_η is of the form $X \Leftarrow f(\overline{Y})$. From the definition of $\mathcal{R}$, $\mathcal{R}[\![A_\eta]\!]\pi\Psi = \bot$. Since σ is the least solution of $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0 and $\mathcal{R}_\eta \succeq \bot$ is the only constraint on $\mathcal{R}_\eta$ in $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$, also $\sigma(\mathcal{R}_\eta) = \bot$. Moreover, $\sigma(X_\eta)$ is the least dynamic value such that all of the constraints on X_η are satisfied. Since there are no constraints on the root node, $(\sigma(X_\eta))(\langle\rangle) = \bot$ and for each $Y_i \in \overline{Y}$:

$$(\sigma(X_\eta))^{\langle f,i\rangle} = \bigsqcup_{\eta' \in \mathtt{reach}(Y_i,\eta)} \sigma(Y_{i_{\eta'}}) \sqcup \sigma(\mathcal{C}_\eta).$$

  Using the conditions from the lemma, we rewrite the above into

$$(\sigma(X_\eta))^{\langle f,i\rangle} \succeq \pi_\eta(Y_i) \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi.$$

  Again, the right-hand side of the above equals the binding-time constructed by $\mathcal{G}[\![A_\eta]\!]\pi_\eta\Psi$ for $X^{\langle f,i\rangle}$, which concludes the proof for this case.

- Suppose A_η is of the form $X \Rightarrow f(\overline{Y})$. Since σ is the least solution to $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0, and since the only constraints on $\mathcal{R}_\eta$ are those created by the deconstruction, we have that

$$\sigma(\mathcal{R}_\eta) = \bigsqcup_{\eta' \in \mathtt{reach}(X,\eta)} (\sigma(X_{\eta'}))(\langle\rangle) = \left(\bigsqcup_{\eta' \in \mathtt{reach}(X,\eta)} X_{\eta'}\right)(\langle\rangle)$$

  From the condition of the lemma, the above can be rewritten into

$$\sigma(\mathcal{R}_\eta) \succeq (\pi_\eta(X))(\langle\rangle) = \mathcal{R}[\![A_\eta]\!]\pi_\eta\Psi$$

  the latter by the definition of $\mathcal{R}$.
  Moreover, for each such $Y_i \in \overline{Y}$, we have that

$$\sigma(Y_{i_\eta}) = \bigsqcup_{\eta' \in \mathtt{reach}(X,\eta)} \sigma(X_{\eta'})^{\langle f,i\rangle} \sqcup \sigma(\mathcal{C}_\eta).$$

  Again from the condition of the lemma, we rewrite the above into

$$\sigma(Y_{i_\eta}) \succeq (\pi_\eta(X))^{\langle f,i\rangle} \sqcup \mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi$$

  the right hand side equaling the binding-time constructed by $\mathcal{G}[\![A_\eta]\!]\pi_\eta\Psi$ for Y_i, concluding this case of the proof.

- Suppose A_η is of the form $X == Y$. Since σ is the least solution to $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ with respect to σ_0, and since the only constraints on $\mathcal{R}_\eta$ are $\mathcal{R}_\eta \succeq^* X$ and $\mathcal{R}_\eta \succeq^* Y$, we have that

$$\sigma(\mathcal{R}_\eta) = \bigsqcup_{\eta' \in \mathtt{reach}(X,\eta)} (\sigma(X_{\eta'}))(\langle\rangle) \sqcup \bigsqcup_{\eta' \in \mathtt{reach}(Y,\eta)} (\sigma(Y_{\eta'}))(\langle\rangle).$$

  Similar reasoning as in the case of a deconstruction concludes the proof.

- Suppose A_η is of the form $q(\overline{X})$. Let us introduce a binding-time environment and binding-time substitution σ_{c_0} as follows:

$$\pi_c = \{(F_i, \pi_\eta(X_i)) \mid F_i \in \mathtt{in}(q)\}$$

$$\sigma_{c_0} = \{(F_{i_{\eta_0}}, \pi_c(F_i)) \mid F_i \in \mathtt{in}(q)\}.$$

By these definitions and the condition of the lemma, we have that

$$\forall F_i \in \mathtt{in}(q) : \sigma_{c_0}(F_{i_{\eta_0}}) = \pi_\eta(X_i) \preceq \bigsqcup_{\eta' \in \mathtt{reach}(X_i,\eta)} \sigma(X_{i_{\eta'}}) \tag{4.6}$$

Now, for every constraint of the form $\mathcal{R}_{\eta_b} \succeq F_{i_{\eta_0}}$ in $\mu(q)$, we have the constraints $\{\mathcal{R}_\eta \succeq X_{i_{\eta'}} \mid \eta' \in \mathtt{reach}(X_i, \eta)\}$ in the constraints constructed for p. If we denote with σ_c the minimal solution of $\mu(q)$ with respect to σ_{c_0}, we have that

$$\sigma_c(\mathcal{R}_{\eta_b}) \preceq \sigma(\mathcal{R}_\eta) \tag{4.7}$$

due to (4.6) and the fact that both σ and σ_{c_0} are minimal solutions. Likewise, for each constraint of the form $F_{i_{\eta_0}} \succeq F_{j_{\eta_0}}$ in $\mu(q)$, we have the constraints $\{X_\eta \succeq X_{j_{\eta'}} \mid \eta' \in \mathtt{reach}(X_j, \eta)\}$ in the constraints constructed for p. Again, by (4.6) and the fact that both σ and σ_{c_0} are minimal solutions, we have that

$$\sigma_c(F_i) \preceq \sigma(X_{i_\eta}). \tag{4.8}$$

Now, we have that

$$\mathcal{R}[\![q(\overline{X})_\eta]\!]\pi_\eta\Psi = \mathcal{R}[\![\mathcal{B}ody(q)]\!]\pi_c\Psi \preceq \sigma_c(\mathcal{R}_{\eta_b})$$

since Ψ and μ are nearly output equivalent. Combining this result with (4.7), we obtain

$$\mathcal{R}[\![q(\overline{X})_\eta]\!]\pi_\eta\Psi \preceq \sigma(\mathcal{R}_\eta).$$

First, assume that $\pi_c \in dom(\Psi(q))$. By definition we have for each $X_i \in \mathtt{out}(q(\overline{X}))$ with F_i the corresponding formal argument in $\mathcal{A}rgs(q)$ that

$$\Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi p\pi_0(X_i) = \Psi(q)(\pi_0)(F_i).$$

and since Ψ and μ are nearly output equivalent, $\Psi(q)(\pi_0)(F_i) \preceq \sigma_c(F_i)$. Combining both results and using (4.8), we obtain

$$\Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi p\pi_0(X_i) = \Psi(q)(\pi_0)(F_i) \preceq \sigma_c(F_i) \preceq \sigma(X_{i_\eta})$$

which concludes the case where $\pi_c \in dom(\Psi(q))$.
Next, assume that $\pi_c \notin dom(\Psi(q))$. In this case, we have by definition that

$$\Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi p\pi_0(X_i) = \bot \preceq \sigma(X_{i_\eta}). \qquad \square$$

Lemma's 4.1 through 4.3 can be glued together into the following result, constituting the final auxiliary lemma before we can prove the induction step in the proof of Theorem 4.1. It states that, given Ψ and μ being nearly output equivalent, the binding-times computed from Ψ of a procedure's output arguments are approximated by the binding-times computed from the appropriate minimal solution for μ.

Lemma 4.4 *Let Ψ be program environment that is (nearly) output equivalent with a symbolic program environment μ and let p, π_0, and σ_0 be as in Definition 4.10, and σ the minimal solution with respect to σ_0 of $\mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$. If $\pi = \mathcal{G}[\![\mathcal{B}ody(p)]\!]\pi_0\Psi$, then it holds*

$$\forall V \in \mathtt{out}(p) : \pi(V) \preceq \sigma(V_{\eta_0}).$$

Proof Since

$$\pi(V) = \bigsqcup_{\eta' \in \mathtt{reach}(V,\eta_0)} \Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta'\pi_0\Psi p\pi_0$$

and

$$\sigma(V_{\eta_0}) = \bigsqcup_{\eta' \in \mathtt{reach}(V,\eta_0)} \sigma(V_{\eta'})$$

it suffices to prove that for each such η': $\sigma(V_{\eta'}) \succeq \Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta'\pi_0\Psi p\pi_0$. The above follows immediately if we prove that for each atom A_η in $\mathcal{B}ody(p)$ and for all $V \in \mathtt{out}(A_\eta)$ holds that $\sigma(V_\eta) \succeq \Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi p\pi_0$. We prove, by induction on the position of the program point η in the procedure's body, that for each atom A at program point η the following holds:

1. $\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta)$
2. $\mathcal{R}[\![A_\eta]\!]\pi_\eta\Psi \preceq \sigma(\mathcal{R}_\eta)$
3. for all $V \in \mathtt{out}(A_\eta)$: $\Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi p\pi_0(V) \preceq \sigma(V_\eta)$

Let Σ denote an ordering of $\mathcal{P}ps(\mathcal{B}ody(p))$ such that $\forall \eta, \eta' \in \mathcal{P}ps(\mathcal{B}ody(p))$: if there exists a control flow path in p in which $\eta < \eta'$, then $\eta < \eta'$ in Σ.

First, we prove the base case, that is we consider an atom A_η such that there does not exist an $\eta' < \eta$ on any control flow path in p:

- The condition of Lemma 4.2 is satisfied, and we have that

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi\Psi \preceq \sigma(\mathcal{C}_\eta).$$

- First, note that $\Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta p\pi_0\Psi = \pi_0$. Moreover, $\mathtt{in}(A_\eta) \subseteq \mathtt{in}(p)$ and hence, by definition of σ we have that $\forall V \in \mathtt{in}(A_\eta) : \sigma(V_{\eta_0}) = \pi_0(V)$. Since for each such V, $\mathtt{reach}(V, \eta) = \{\eta_0\}$, the condition of Lemma 4.3 is satisfied, its result concluding the proof of the base case.

Now, suppose the theorem holds for each $\eta' < \eta$ in Σ. We prove that it also holds for η. By induction hypothesis, $\mathcal{R}[\![A_{\eta'}]\!]\pi_{\eta'} \preceq \sigma(\mathcal{R}_{\eta'})$ – with $\pi_{\eta'} = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta' p \pi_0 \Psi$ – for each atomic subgoal $A_{\eta'}$ such that η' precedes η satisfying the condition of Lemma 4.2 and hence resulting into

$$\mathcal{C}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi \preceq \sigma(\mathcal{C}_\eta)$$

which proves the first item. The two remaining items follow immediately, if we can prove that the condition of Lemma 4.3 holds. This is indeed the case. First, we have for all $V \in \mathtt{in}(A_\eta)$ that

$$(\Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta p \pi_0 \Psi)(V) = \bigsqcup_{\eta' \in \mathtt{reach}(V,\eta)} (\Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta' p \pi_0 \Psi)(V).$$

Due to well-modedness and the ordering on Σ, for each such η' the induction hypothesis holds, and hence we rewrite the above into

$$(\Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta p \pi_0 \Psi)(V) \preceq \bigsqcup_{\eta' \in \mathtt{reach}(V,\eta)} \sigma(V_{\eta'})$$

which satisfies the condition of Lemma 4.3, concluding the proof.

□

Given Lemma 4.4, the proof of the induction step in the proof of Theorem 4.1 is straightforward:

Corollary 4.1 *Let Ψ be a program environment that is (nearly) output equivalent with a symbolic program environment μ and let p, π_0 and σ be as in Definition 4.10. If we define*

$$\begin{aligned} \Psi' &= \Delta(P, \Psi) \\ \mu' &= \Delta_S(P, \mu) \end{aligned}$$

then Ψ' and μ' are nearly output equivalent.

Proof By definition, Ψ' and μ' are such that for each procedure p, $\mu'(p) = \mathcal{G}_S[\![\mathcal{B}ody(p)]\!]\mu$ and for all $(\pi_0, \pi) \in \Psi'(p)$ holds:

- if $\pi_0 \in dom(\Psi(p))$, $(\pi, \_) = \mathcal{G}[\![\mathcal{B}ody(p)]\!]\pi_0 \Psi p \pi_0$
- if $\pi_0 \notin dom(\Psi(p))$, $\pi = \{(F, \bot) \mid V \in \mathtt{out}(p)\}$

The result follows immediately from Lemma 4.4 and Definition 4.10 for those $\pi_0 \in dom(\Psi(p))$. For those $\pi_0 \notin dom(\Psi(p))$, the result is immediate since for all $V \in \mathtt{out}(p)$: $\pi(V) = \bot$.

□

Remains the proof of Theorem 4.1.

Proof (Theorem 4.1) To prove Theorem 4.1, we first prove that the program environment and symbolic program environment derived in each iteration step are nearly output equivalent. We prove this by induction on the number of iterations.

- Firstly, Ψ_0 and μ_0 are nearly output equivalent since Ψ contains only a single entry binding the initial predicate's output arguments to $\bot$, ensuring that the condition for nearly output equivalence is satisfied. This proves the base case of the induction.
- Next, assume that Ψ_i and μ_i are nearly output equivalent. By Corollary 4.1, also Ψ_{i+1} and μ_{i+1} are nearly output equivalent.

Now, let n be the smallest natural number such that $\Psi_n = \Psi_{n-1}$. The existence of such a finitary fixed point is guaranteed by Theorem 3.1. From the above, we have that Ψ_n and μ_n are nearly output equivalent. We now prove that they are also output equivalent. Suppose that Ψ_n and μ_n are *not* output equivalent for some predicate p and binding-time environment $\pi_0 \in dom(\Psi_n(p))$. If this is the case, it must be due to a predicate call $q(\overline{X})_\eta$ in $\mathcal{B}ody(p)$ such that

- either $\sigma(X_i) > \pi^+_\eta(X_i)$ for some $X_i \in \mathtt{out}(q(\overline{X}))$ where

$$\pi^+_\eta = \Delta^+_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi_{n-1}p\pi_0$$

  and σ the minimal solution with respect to σ_0 of $\mu_n(p)$,
- or $\sigma(\mathcal{R}_\eta) > \mathcal{R}[\![q(\overline{X})]\!]\pi_\eta\Psi_{n-1}$ where $\pi_\eta = \Delta_{pp}[\![\mathcal{B}ody(p)]\!]\eta\pi_0\Psi_{n-1}p\pi_0$.

Let us consider the first case, the second one is analogous. Since σ is the minimal solution of $\mu_n(p)$, there must exist a constraint on X_i in $\mu_n(p)$ that is not satisfied in π^+_η. Now, since the only constraints that exist on X_i are constraints that express congruence, this contradicts with the fact that Ψ_n is a congruent program environment (Theorem 3.1) and concludes the proof.

□

This correctness result states the equivalence between the binding-time analysis from Chapter 3 and the analysis from the present chapter. Consequently, the result of the current analysis can be used to construct an annotated version of a program analogous to the way annotated versions are created in the previous chapter. In the following section, we extend the constraint based analysis to deal with Mercury's higher-order features.

4.4 Binding-time Analysis for Mercury: a Higher-order Setting

The binding-time analysis described in the previous sections of this chapter deals with a first-order subset of Mercury. In this section, we alter the analysis such that it can deal with the higher-order aspects of the language. First, we discuss the basic issues involved with higher-order Mercury, and show later on how the binding-time analysis can be adapted to deal with them. Finally, we discuss the topic of modularity in a higher-order context.

4.4.1 Higher-order Mercury

Mercury is a higher-order language in which *closures* can be created, passed as arguments of predicate calls, and in turn be called themselves. To describe the higher-order features of the language, it suffices to extend the definition of superhomogeneous form (see Definition 3.2) with two new kinds of atoms:

- A *higher-order unification* which is of the form $X \Leftarrow p(V_1, \ldots, V_k)$ where $X, V_1, \ldots, V_k \in \mathcal{V}$ and $p/n \in \Pi$ with $k \leq n$.
- A *higher-order call* which is of the form $X(V_1, \ldots, V_n)$ where $X, V_1, \ldots, V_n \in \mathcal{V}$.

A higher-order unification $X \Leftarrow p(V_1, \ldots, V_k)$ constructs a closure from an n-arity procedure p by *currying* the first k arguments (with $k \leq n$). The result of the construction is assigned to the variable X and denotes a procedure of arity $n - k$. Such a closure can be called by a higher-order call of the form $X(V_{k+1}, \ldots, V_n)$ where $V_{k+1}, \ldots, V_n$ are the $n - k$ remaining arguments. The effect of evaluating the conjunction $X \Leftarrow p(V_1, \ldots, V_k), X(V_{k+1}, \ldots, V_n)$ equals the effect of evaluating $p(V_1, \ldots, V_n)$.

When writing Mercury code, the programmer can also use lambda expressions to construct closures. These can, however, be converted into a regular procedure definition which is then again used to construct the closure as above. The Melbourne Mercury compiler does this conversion as part of the translation into superhomogeneous form. Note that closures can not be constructed from other closures: once a closure is created, one can only call it or pass it as an argument to another procedure.

Example 4.12 *Consider the definition in Mercury of the* `map/3` *predicate and its possible use to reverse a list of lists, as depicted in Fig. 4.8. The predicate* `map/3` *converts the first list of type T to a new list of type T using the predicate provided as its second argument.*

Note that in order to represent higher-order *types* it suffices to add a special type constructor, *pred*, to $\Sigma_{\mathcal{T}}$. This constructor is special in the sense that it can be

```
:- pred map(list(T),pred(T,T),list(T)).
:- mode map(in,in(pred(in,out)),out) is det.
map([],P,[]).
map([X|Xs],P,[Y|Ys]):-
     P(X,Y), map(Xs,P,Ys).

:- pred rev(list(T),list(T)).
:- mode rev(in,out) is det.
rev([],[]).
rev([X|Xs],R):-
     rev(Xs,Rs), append(Rs, [X], R).

:- pred revl(list(list(T)),list(list(T))).
:- mode revl(in,out) is det.
revl(L1,L2):-
     map(L1,rev,L2).
```

Figure 4.8: The definition and sample use of `map/3`.

used with any arity and it has no type rule associated with it. Consequently, a higher-order type corresponds with a leaf node in a type tree. In what follows we represent higher-order types as $pred(t_1, \ldots, t_k)$ with $t_1, \ldots, t_k$ types. In Example 4.12, the same constructor *pred* is also used in the definition of a higher-order *mode* – specifying for an argument of a higher-order type what arguments are input, respectively output. We do not discuss higher-order types and modes in detail, and assume that higher-order types are not used in the definition of other types.

4.4.2 The necessity of closure information

The basic problem when analysing a procedure involving higher-order calls, is that the control flow in the procedure is determined by the value of the higher-order variables. Consequently, without knowing (an approximation of) these values, it is impossible to compute meaningful data dependencies between the procedure's variables. The call independent phase of the binding-time analysis is confronted with this problem during the creation of a constraint system that reflects these data dependencies. Reconsider the definition of the `map/3` predicate, now in superhomogeneous form.

Example 4.13 *The definition of* `map/3` *in superhomogeneous form is as follows:*

```
:- pred map(list(T),pred(T,T),list(T)).
:- mode map(in,in(pred(in,out)),out) is det.
map(L, P, R):-
     (L⇒[], R⇐[] ;
      L⇒[E|Es], P(E,N), map(Es,P,Rs), R⇐[N|Rs]).
```

In Example 4.13, we know from the mode declaration that E is input and N is output from the call to the higher-order variable P. To compute a meaningful binding-time for N (and all variables depending on N), we need to consider the dependencies that most likely exist between N and E. In case of a first-order call, these dependencies are obtained by computing the data dependencies between the called procedure's input and output arguments. In case of a higher-order call, the information about the particular procedure that is called is missing. Without this information, we can only approximate N's binding-time by $\top$ (corresponding with the worst-case assumption that the value is possibly unknown during specialisation), resulting in the following constraints generated for the `map/3` predicate:

$$\begin{array}{lll} R \succeq \top & R \succeq R_s & R^{\langle [],1\rangle} \succeq N \\ E \succeq L^{\langle [],1\rangle} & E_s \succeq L & N \succeq \top \end{array} \tag{4.9}$$

Note that in order to simplify the examples, we do not distinguish between several occurrences of a variable in the different branches of the disjunction, but rather consider a single binding-time for each variable. Every solution of this constraint system will approximate the elements of `map`'s output list by $\top$, regardless the binding-times of the input list.

However, consider a call to `map/3` that binds P for example to the predicate `rev/2` defined in Example 4.12. In this case, we can create a more precise set of constraints for `map/3` *with respect to the fact that* $P =$ `rev/2`, in what case the call $P(E, N)$ can, during constraint generation, be treated as a first-order call $rev(E, N)$, resulting in the following constraints which are renamings of $\mu(rev)$ from Example 4.11.

$$N \succeq \bot \qquad N^{\langle [],1\rangle} \succeq E^{\langle [],1\rangle} \tag{4.10}$$

The least solution of these constraints will associate to N a binding-time that characterises the list skeleton of N as known during specialisation, and the elements of N known if and only if the elements of E are known. Normalising the set of constraints composed of (4.9) (minus $N \succeq \top$) and (4.10) and incorporating the constraints for the call $map(E_s, P, R_s)$ now results in the following constraint system

$$\begin{array}{lll} E \succeq L^{\langle [],1\rangle} & N^{\langle [],1\rangle} \succeq L^{\langle \langle [],1\rangle,\langle [],1\rangle\rangle} & R \succeq \bot \\ E_s \succeq L & N \succeq \bot & R^{\langle [],1\rangle} \succeq \bot \\ & & R^{\langle \langle [],1\rangle,\langle [],1\rangle\rangle} \succeq L^{\langle \langle [],1\rangle,\langle [],1\rangle\rangle} \end{array}$$

Informally, these constraints on `map/3`'s output argument R express that in the least solution, the binding-times of the elements of the output lists will be the same as the binding-times of the elements of the input lists. Note that this result is more precise then any solution resulting from (4.9). Of course, this more precise result for `map/3` is only valid for the case where `map/3` is called with its higher-order argument bound to `rev/2`.

4.4.3 Higher-order binding-time analysis

The general idea behind performing the binding-time analysis in a higher-order setting is as follows: instead of associating a single set of binding-time constraints with each procedure, we will possibly associate *several* sets of constraints with a single procedure, depending on the particular context the procedure is used in. Recall that binding-time constraints represent the data flow inside a predicate symbolically, hence constraint systems that are associated to a single procedure will differ only when different knowledge about the closures bound to the higher-order arguments is incorporated in the analysis.

Closure information can be derived by a separate, so-called *closure analysis* (Jones, Gomard, and Sestoft 1993; Palsberg 1995). Since closures can not only be constructed in a procedure, but also passed around by procedure calls, closure analysis is by definition a call dependent process. In what follows, we will first define a suitable representation for such closure information, and reformulate the first phase of our binding-time analysis so that it integrates the derivation of closure information with the derivation of binding-time constraint systems. Doing so basically transforms the process of building constraint systems into a call dependent process, since the closure information from a particular call pattern is taken into account. In order to use closures during binding-time analysis, where concrete values of the closure's curried arguments are approximated by binding-times, we introduce the notion of a *binding-time closure* as follows.

Definition 4.13 *A* binding-time closure *is a term of the form* $p(\beta_1, \ldots, \beta_k)$ *where* $p/n \in \Pi$, $k \leq n$ *and* $\beta_1, \ldots, \beta_k \in \mathcal{BT}^+$. *The set of all such binding-time closures is denoted by* $\mathcal{C}los$.

If $p/n \in \Pi$, $p(\beta_1, \ldots, \beta_k)$ approximates a procedure of arity $n-k$, being an instance of p in which the first k arguments are fixed and approximated by the binding-times $\beta_1, \ldots, \beta_k$.

Example 4.14 *Given the traditional* `append/3` *procedure and* β_l *being a binding-time approximating terms of type* `list(T)` *that are instantiated at least up to a list skeleton, append, append*(β_l) *and append*$(\bot, \beta_l)$ *are examples of binding-time closures.*

Note that a single binding-time closure $p(\beta_1, \ldots, \beta_k)$ approximates a set of run-time closures, namely those that are constructed from the procedure p/n by currying the first k arguments with values that approximated by $\beta_1, \ldots, \beta_k$. In order to obtain a precise binding-time analysis, we will approximate the value of a higher-order variable with a *set* of binding-time closures. A singleton set $\{c\}$ describes that the higher-order variable under consideration is during specialisation definitely bound to a closure that is approximated by c. In general, a set $\{c_1, \ldots, c_n\}$ describes that the higher-order variable under consideration is bound during specialisation to a closure that is approximated either by c_1, c_2,..., or c_n.

Approximating the value of a higher-order value by a set of binding-time closures allows to keep a high degree of precision. Indeed, a higher-order variable might be assigned a value in different branches of a disjunction, and used later on, at a program point where the branches have been merged. If, during analysis, the value of the higher-order value is approximated by a single binding-time closure, its approximation might need to be lifted to $\top$ after the disjunction, for example when both closures are constructed using a different procedure symbol. However, it is not at all unlikely that two such closures still have very similar data flow relations; a fact that can not be exploited when the value is lifted to $\top$, but can when a higher-order variable is approximated by a set of such closures.

To make this representation explicit, we alter the definition of the domain $\mathcal{B}$ from Chapter 3. Instead of containing only the values *static* and *dynamic*, we now include a value $static(S)$ with S being a set of binding-time closures. Note that, if we define $dynamic > static$ as before and $static(S_1) > static(S_2)$ if and only if $S_1 \supseteq S_2$, $\mathcal{B}$ is still partially ordered. For completeness sake, we repeat the definition of a binding-time from Chapter 3.

Definition 4.14 *A* binding-time *for a type $t \in \mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ is a relation $\beta : TPath \times \mathcal{B}$ such that $\forall \delta \in dom(\beta)$ holds that $\beta(\delta) = dynamic$ implies that $\beta(\delta') = dynamic$ for all $\delta' \in dom(\beta)$ with $\delta' = \delta \bullet \epsilon$ for some $\epsilon \in TPath$. The set of all binding-times is denoted by $\mathcal{BT}$.*

Since the binding-times now include higher-order binding-times, we alter the definition of the partial order relation on $\mathcal{BT}$:

Definition 4.15 *Let $\beta, \beta' \in \mathcal{BT}$ such that $dom(\beta) \subseteq dom(\beta')$ or $dom(\beta') \subseteq dom(\beta)$. We say that β* covers *β', denoted by $\beta \succeq \beta'$ if and only if $\forall \delta \in dom(\beta) \cap dom(\beta')$ holds that*

- *$\beta'(\delta) = dynamic$ implies $\beta(\delta) = dynamic$, and*
- *$\beta'(\delta) = static(S')$ implies $\beta(\delta) = static(S)$ and $S' \supseteq S$.*

Note that, with this new definition, the covers relation remains only defined between two binding-times that are derived from types that are instances of each other. In case of higher-order binding-times this means that both sets of binding-time closures contain closures of identical arity and types of the arguments. Like before, we denote with $\mathcal{BT}^+$ the set $\mathcal{BT} \cup \{\top, \bot\}$, and $(\mathcal{BT}^+, \succeq)$ forms a complete lattice.

In what follows, we devise the binding-time analysis that associates to a procedure a number of constraint systems with respect to the closure information from a particular call. Recall that closure information is available from the procedure arguments' binding-times. Hence, we can model the result of analysis as a function that associates a constraint system to a procedure in combination with a binding-time environment comprising binding-times for the procedure's arguments and as such constituting the procedure's call pattern.

Definition 4.16 *A* symbolic higher-order program environment *is a function*

$$Proc \times \mathcal{BTE}nv \mapsto \wp(\mathcal{BTC}).$$

in which $\mathcal{BTE}nv$ denotes the set of binding-time environments, defined in Definition 3.11. The set of all such functions is denoted by $\mathcal{PE}nv_S^h$.

We can construct the binding-time constraint systems much in the same way as in the first-order case. However, since the process now is call dependent, we need to introduce call dependent counterparts of the analysis functions $\mathcal{A}_S$, $\mathcal{G}_S$ and Δ_S from Section 4.2.2, which we name $\mathcal{A}_S^h$, $\mathcal{G}_S^h$ and Δ_S^h respectively. Their signatures are as follows:

$$\mathcal{A}_S^h : Atom \times Proc \times \mathcal{BTE}nv \times \mathcal{PE}nv_S^h \mapsto \wp(\mathcal{BTC}) \times \mathcal{PE}nv_S^h$$

$$\mathcal{G}_S^h : Goal \times Proc \times \mathcal{BTE}nv \times \mathcal{PE}nv_S^h \mapsto \wp(\mathcal{BTC}) \times \mathcal{PE}nv_S^h$$

$$\Delta_S^h : \wp(Proc) \times \mathcal{PE}nv_S^h \mapsto \mathcal{PE}nv_S^h$$

Note that the analysis functions that analyse an atom, respectively a goal now take a procedure and binding-time environment as argument, denoting for what particular procedure and call pattern the constraints must be created. The result of these analysis functions is a set of binding-time constraints as before, but now combined with a possibly updated symbolic program environment. Indeed, the latter can be updated with newly encountered procedure calls that must be analysed. In what follows, we discuss each of the analysis functions in turn, but rely heavily on the material and definitions from Section 4.2.2, focusing on the difference introduced by the higher-order analysis.

We start by the most involved analysis function, which is $\mathcal{A}_S^h$, used to analyse a single atom. The definition of $\mathcal{A}_S^h$ is found in Fig. 4.9. The data flow dependencies that are created by a first-order unification remain the same as in the first-order case, and such a unification does not update the program environment. Handling a higher-order construction results in a number of constraints on the higher-order variable, such that in the least solution this higher-order variable is bound to a set of closures that approximate the constructed closure. Also in this case the program environment remains unchanged. Handling a first-order call is analogous to the first-order case, except for the fact that a *particular* set of constraints must be selected from the program environment for renaming. In order to do so, the minimal solution σ is computed for the constraint system that is associated with the procedure p and binding-time environment π_0 for which we are analysing the procedure. From this minimal solution, the binding-times of the call's arguments are computed and used to construct a new binding-time environment π. The program environment is updated by adding the latter binding-time environment to the domain of $\mu^h(q)$. The use of the least upper bound ensures that if q was already analysed with respect to π, the latter's constraint system is kept.

$$
\begin{array}{lcl}
\mathcal{A}^h_S[\![U]\!]p\pi_0\mu^h & = & (\mathcal{C}_U, \mu^h) \text{ for a first-order unification } U \\
 & & \text{and } \mathcal{C}_U \text{ the constraints associated to } U \text{ by } \mathcal{A}_S \\
\mathcal{A}^h_S[\![X \Leftarrow p(\overline{X})_\eta]\!]p\pi_0\mu^h & = & \left(\{X \succeq \mathit{static}(p(\overline{X_{\eta_i}})) \mid X_{i_{\eta_i}} \in \mathtt{reach}(X_i, \eta)\}, \mu^h \right) \\
\mathcal{A}^h_S[\![q(\overline{X})]\!]p\pi_0\mu^h & = & (\rho^q_\eta(\overline{X}, \mu^{h'}(q, \pi)), \mu^{h'}) \\
 & & \text{with } \rho^q_\eta \text{ the renaming from Fig. 4.5} \\
 & & \pi = \{(F_i, \beta_i) \mid F_i \in \mathtt{in}(q)\} \\
 & & \beta_i = \bigsqcup_{\eta' \in \mathtt{reach}(X_i, \eta)} \sigma(X_{i_{\eta'}}) \\
 & & \text{and } \sigma \text{ the least solution of } \mu^{h'}(p, \pi_0) \\
 & & \text{with } \mu^{h'} = \mu^h[p / \mu^h(p) \sqcup (\pi, \{\})] \\
\mathcal{A}^h_S[\![P(\overline{X})]\!]p\pi_0\mu^h & = & \begin{cases} (\{X_{i_\eta} \succeq \bot \mid X_i \in \mathtt{out}(q(\overline{X}))\}, \mu^h) & \text{if } \mathcal{C} = \emptyset \\ (\mathcal{C}, \mu^{h'}) & \text{otherwise} \end{cases} \\
 & & \text{where } \mathcal{C} = \bigcup_{q(\beta_1, \ldots, \beta_k) \in S} \rho^q_\eta(Q, \mu^h(q, \pi)) \\
 & & \text{with } \rho^q_\eta \text{ the renaming from Fig. 4.5} \\
 & & S = \bigsqcup_{\eta' \in \mathtt{reach}(P, \eta)} \sigma(P_{\eta'}) \\
 & & \pi \in \mathcal{BT}\mathit{Env} \text{ such that } \mathit{dom}(\pi) = \mathtt{in}(q) \\
 & & \text{and } \pi(F_i) = \begin{cases} \beta_i & \text{if } 1 \leq i \leq k \\ \gamma_i & \text{if } i > k \end{cases} \\
 & & \text{where } \gamma_i = \bigsqcup_{\eta' \in \mathtt{reach}(X_i, \eta)} \sigma(X_{i_{\eta'}}) \\
 & & Q = \langle P^{\langle \mathit{pred}, 1 \rangle}, \ldots, P^{\langle \mathit{pred}, k \rangle} \rangle \bullet \overline{X} \\
 & & \text{and } \sigma \text{ the least solution of } \mu^h(p, \pi_0) \\
 & & \text{with } \mu^{h'} = \mu^h[p / \mu^h(p) \sqcup (\pi, \{\})]
\end{array}
$$

Figure 4.9: The definition of $\mathcal{A}^h_S$.

Otherwise, an empty constraint system is added to the program environment. In any case, the result of handling the procedure call is the set of constraints associated to q and π in the program environment, renamed with respect to the actual argument variables.

Handling a higher-order call is a bit more complicated. Again, the minimal solution σ is computed for the constraint system that is associated with the procedure p and binding-time environment π_0 for which we are currently analysing the procedure. If σ associates a set of binding-time closures to the higher-order variable P, the call is handled as the union of a number of first-order calls. For each such binding-time closure $q(\beta_1, \ldots, \beta_k)$, the binding-times from the closure, $\beta_1, \ldots, \beta_k$ are combined with the binding-times of the higher-order call's arguments, computed from σ, to constitute the call's complete call pattern π. The constraints

associated to q with respect to this complete call pattern are retrieved from the (appropriately updated) program environment and renamed in the following particular manner. Those formal input arguments F_i of q that are curried in the closure are renamed by the binding-time variable $P^{\langle pred,i\rangle}$. Indeed, the binding-time variable P restricted to the subvalue $\langle pred, i\rangle$ denotes the i'th binding-time in the closure associated to P. The remaining (uncurried) formal input arguments are renamed by the appropriate actual argument variable from $\overline{X}$.

An important observation is the following. The constraint systems associated to two call patterns π and π' of a single procedure differ only when π and π' differ with respect to their higher-order variables. Consequently, the program environment should be updated with respect to a newly encountered call only when the new call pattern is different in its higher-order variables from any previously recorded call pattern. Likewise, when selecting the appropriate constraint system from the environment, only the higher-order variables from the call pattern must be taken into account when comparing a call's call pattern with the call patterns recorded in the program environment.

The constraints that are associated to a structured goal do not differ from those in the first-order setting. Hence, the definition of $\mathcal{G}_S^h$ is analogous to the definition of $\mathcal{G}_S$, apart from the fact that the procedure and binding-time environment with respect to which we are currently analysing the procedure must be provided as arguments, and that a possibly updated program environment must be threaded through, in order to record all the calls that are encountered during analysis of a structured goal. For a goal G in a procedure p and a binding-time environment π_0, the intended meaning of $\mathcal{G}_S^h[\![G]\!]p\pi_0\mu^h = (\mathcal{C}, \mu^{h'})$ is that analysing G with respect to π_0 and a current program environment μ^h results in the constraint system $\mathcal{C}$ and a program environment $\mu^{h'}$ which is the original program environment μ^h updated with the calls encountered during analysis of G. We do not give a detailed definition of $\mathcal{G}_S^h$. Remains the definition of the analysis function Δ_S^h:

Definition 4.17 *The program analysis function* $\Delta_S^h : \wp(Proc)\times\mathcal{P}Env_S^h \mapsto \mathcal{P}Env_S^h$ *is defined as follows:*

$$\Delta_S^h(M,\mu^h) = \begin{cases} \mu^h & \text{if } M = \emptyset \\ \Delta_S^h(Ms, \delta_S(G,\mu^h(p),\mu^h)) & \text{if } M = \{(p(\overline{F}) : -G) :: Ms\} \end{cases}$$

where the auxiliary function $\delta_S : Goal \times (\mathcal{BT}Env \mapsto \wp(\mathcal{BTC})) \times \mathcal{P}Env_S^h \mapsto \mathcal{P}Env_S^h$ *is defined as*

$$\delta_S(G,S,\mu^h) = \begin{cases} \mu^h & \text{if } S = \emptyset \\ \delta_S(G, Ss, \mu^{h'}[p/\mu^{h'}(p) \sqcup \{(\pi_0,\mathcal{C}')\}]) & \text{if } S = \{(\pi_0,\mathcal{C}) :: Ss\} \\ & (\mathcal{C}', \mu^{h'}) = \mathcal{G}_S^h[\![G]\!]p\pi_0\mu^h \end{cases}$$

4.4.4 What about modularity?

In a higher-order setting, the constraint generation phase of our binding-time analysis is a call dependent process. Indeed, the data flow dependencies in a procedure are determined by the closures contained in the procedure's call pattern. This suggests that the advantage of modularity, associated to the constraint based technique in a first-order setting, might no longer hold in a higher-order setting. However, to some extent the analysis can still be performed in a bottom-up, modular way.

Consider an analysis unit M, and an initial program environment ${\mu^h}_0$ such that $dom({\mu^h}_0) = \mathit{exported}(M)$ and $\forall p \in dom({\mu^h}_0)$ holds that ${\mu^h}_0(p) = \{(F_i, \top) \mid F_i \in \mathtt{in}(p)\}$. The analysis then proceeds as usual, constructing a number of subsequent program environments until a fixed point is reached. That is, we define

$$ {\mu^h}_i = \Delta^h_S(M, {\mu^h}_{i-1}) $$

for all $i > 0$. At first sight, it might seem strange to start the analysis using a binding-time environment of the form $\pi = \{(F_i, \top) \mid F_i \in \mathtt{in}(p)\}$. However, recall that only the higher-order parts of the call patterns influence the resulting constraint systems. Hence, for those procedures that have no higher-order arguments, the constraint system derived by the call dependent analysis equals the one derived by the call independent analysis of Section 4.2, and it can readily be used by other modules importing these procedures from M. In fact, the same holds for procedures that do have higher-order arguments. Indeed, since the analysis starts from a call pattern $\pi = \{(F_i, \top) \mid F_i \in \mathtt{in}(p)\}$, no higher-order information from outside the module is assumed, and the result of analysis, ${\mu^h}_n$ (for $n \in \mathbb{N}$ such that ${\mu^h}_n = {\mu^h}_{n-1}$), can be used when analysing other modules that import M. The fact that M's exported procedures are analysed with respect to a *most general* call pattern ensures that the constraint sets associated to them correctly approximate the data-flow inside the procedure, no matter what the actual (higher-order) arguments are. Of course, the results will not be optimal, as the caller's closure information is not taken into account.

Note that this precision loss only occurs when closure information is "lost" over a module boundary. This is the case when closure information is available for the arguments of a call to an imported procedure but it is not used, since the imported procedure was analysed independently with respect to a most general call pattern. The call dependent nature of the process ensures that closure information that is constructed in a module M, is propagated inside M itself.

Example 4.15 *Reconsider the* `revl/2` *predicate from Example 4.12 that reverses each of the lists in its first argument. Analysing* `revl/2` *with respect to the binding time* $\top$ *for its single input argument will still result in the binding-time constraints*

$$ \begin{array}{l} L_2^{\langle [\,],1\rangle} \succeq \bot \\ L_2^{\langle ([\,],1),([\,],1)\rangle} \succeq L_1^{([\,],1),([\,],1)\rangle} \end{array} $$

since the binding-time closure rev is propagated when analysing the `map/3` *predicate.*

4.5 Discussion

We conclude this chapter by discussing several issues related to the binding-time analysis developed in this chapter, in particular concentrating on the constraint-based approach in association with its modular and higher-order extensions. A more general discussion related to the domain of binding-times, correctness and termination can be found in Section 3.5 of Chapter 3 and will not be repeated here.

4.5.1 Constraints, Modules and Higher-orderedness

Constraint based (binding-time) analysis has been considered before. In (Henglein 1991), Henglein develops such a constraint-based (higher-order) binding-time analysis for λ-calculus by viewing the problem as a type inference problem for annotated λ-terms in a two-level λ-calculus. A set of constraints capturing local binding-time requirements is created and transformed into a normal form. A solver is used to find a consistent minimal binding-time classification. The analysis is redeveloped, concentrating on the aspect of polyvariance, for a PCF-like language in (Henglein and Mossin 1994). Henglein's analysis is scaled up by Bondorf and Jørgensen in (Bondorf and Jørgensen 1993), where they construct three (monovariant) analyses to be used in the partial evaluator Similix (Bondorf 1991). The advantages of doing binding-time analysis by constraint normalisation are summarised in (Bondorf and Jørgensen 1993) as follows. First of all, the constraint based approach is viewed as a more *elegant* description of the analysis since, contrary to the abstract interpretation approach, problem and solution are separated: the constraint system expresses the binding-time *requirements* on the involved variables, whereas actual binding-times are contained in a *solution* to the constraint system. Moreover, the constraint-based approach allows to easily express a bidirectional flow. This aspect is neatly illustrated in our binding-time analysis for Mercury: the context flow, represented by the binding-time variables $\mathcal{C}$ and $\mathcal{R}$, is expressed by a number of constraints that again express a local relation between a goal and its subgoals. Compare this with the definitions of $\mathcal{C}$ and $\mathcal{R}$ functions that compute, in the abstract interpretation approach of Chapter 3, the same information but whose definitions are quite involved, precisely due to the bidirectional information flow between $\mathcal{C}$- and $\mathcal{R}$ values. The third advantage mentioned in (Bondorf and Jørgensen 1993) of doing constraint-based analysis is the efficiency of its implementation. The flow analysis of Bondorf and Jørgensen (Bondorf and Jørgensen 1993) (being monovariant and equality-based) can be done in "almost linear time".

In this chapter, we have shown that a similar constraint-based approach is also feasible for Mercury. We have combined a constraint-based approach with the domain of binding-times developed in Chapter 3 as such obtaining an analysis that is polyvariant and able to deal with partially instantiated structures. The new analysis incorporates the same specialisation strategy as the former one and we have shown both analyses to compute the same binding-time for a variable at a particular program point. Upgrading binding-time analysis to deal with Mercury's higher-order constructs requires closure information. In the literature, also closure analysis has been formulated by means of abstract interpretation (Bondorf 1991; Consel 1990) as well as by constraint solving (Heintze 1994; Palsberg 1995; Henglein 1992). Bondorf and Jørgensen (Bondorf and Jørgensen 1993) develop a constraint-based flow analysis that traces higher-order flow as well as flow of constructed (first-order) values. In this chapter, we have combined closure analysis with binding-time analysis and used constraints to express the first-order as well as the higher-order data flow. We have enhanced the domain of binding-times to include a set of closures that represents the binding-time of a higher-order value, and formulated the constraint-generation phase as a call dependent process in which only the higher-order parts of the call pattern are used. During constraint generation, the constraints involving higher-order values are evaluated, and the resulting closure information is used to decide what constraints to incorporate, possibly propagating closure information down into the called procedures.

We have discussed in detail how the analysis can be applied to multi-module programs according to a one module at a time scenario in Sections 4.2.4 and 4.4.4. If we do not wish to propagate closure information over module boundaries, the constraint generation phase can be performed one module at a time, bottom-up in the module hierarchy. Remaining issues are precisely such propagation and the handling of circularities in the module hierarchy. Recent work (Bueno, de la Banda, Hermenegildo, Marriott, Puebla, and Stuckey 2000) presents a framework for the (call-dependent) analysis of multi-module programs that solves both problems. The key invariant in the approach of (Bueno, de la Banda, Hermenegildo, Marriott, Puebla, and Stuckey 2000) is that at each stage of the process, the analysis results are correct, but reanalysis may – when more information is available – produce more accurate results. The analysis performs some extra bookkeeping such that, when a module is analysed, it records both the call patterns occurring in the calls to the imported procedures, and the analysis results of the module's exported procedures. When the recorded information contains new calls (or calls with a more accurate call pattern) to the imported modules, the analysis may decide to reanalyse the relevant imported modules with respect to the more accurate call patterns. Likewise, the recording of more accurate analysis results for a module's exported procedures can trigger the reanalysis of those modules that would possibly profit from these more accurate results. Note that our binding-time analysis neatly fits such an approach: initially, a module's exported procedures are

analysed with respect to $\top$ (no closure information is available). The resulting binding-time constraint systems are correct, but could possibly be more precise, when the procedures are (re)analysed with respect to a more accurate call pattern. To the best of our knowledge, the binding-time analysis of modular programs has been considered only occasionally before. Henglein and Mossin (Henglein and Mossin 1994) note that a symbolic representation of binding-times allows a modular approach. Based on such a symbolic analysis, (Dussart, Heldal, and Hughes 1997) present a method to specialise a multi-module program – written in a simple yet higher-order functional language – by constructing, for each of the modules, a generating extension, while using only the result of a call-independent binding-time analysis. The analysis assumes that annotations indicating whether a function must be unfolded are given by hand and is restricted to module hierarchies without circular dependencies.

4.5.2 Implementation

To investigate the feasibility of the approach for Mercury, a prototype implementation of the binding-time analysis was created. The implementation, written in Mercury, interacts with the Melbourne Mercury Compiler (Somogyi, Henderson, and Conway 1996) to convert full Mercury programs to the requested format. Table 4.1 summarises some experiments that were conducted using the analysis. It reports, for some selected procedures, the number of constraints comprising that procedure's normal form and the time (in seconds) it took to generate that normal form. Note that the implementation was not optimised for speed. Hence, the provided timings should not be considered definitive; we include them for completeness, and as a means to indicate some of the implementation's bottlenecks which we believe to be relevant for any implementation of the developed binding-time analysis. First, note that for the examples that were covered in the text, the number of derived constraints might differ from the number of constraints mentioned in the text. This is due to the fact that the implementation derives a number of additional constraints and the fact that the Melbourne compiler's internal representation of superhomogeneous form differs somewhat from the one we defined in this text. The former considers, for example, flattened conjunctions and disjunctions and employs the notion of a *switch*. A switch can be seen as a deterministic disjunction, in which a single branch is taken, determined by the value of a variable. Handling these constructs does not introduce conceptual difficulties, but requires the creation of additional constraints.

The `map.plus1` module implements an example involving a call to `map/3` with respect to a known predicate (`plus1/2` – adding the constant value 1 to its first argument). The `dvisor` module implements one of the DPPD benchmarks (Leuschel 1996), namely a small expert system. Being a datalog program of which the procedure definitions are simple in structure (they do not contain much nested structures), the number of constraints as well as the time needed to generate them

Module	Procedure	Turns	Total time	Constraints
`map.plus1`	`map/3`	3	0.20	36
	`plus1/2`	2	0.02	13
	`p/2`	2	0.03	27
`dvisor`	`what_to_do_today/2`	2	0.09	17
	`kind_of_day/2`	2	0.06	63
	`kind_of_wheater/3`	2	0.01	23
	`proposal/3`	3	0.29	89
`adrlist`	`create/2`	3	0.40	43
	`get/3`	3	1.01	123
	`set/4`	3	1.50	121
`list`	`append/3 : 0`	3	0.12	34
	`member/2 : 0`	3	0.03	19
	`same_length/2 : 0`	3	0.09	29
	`insert/3 : 0`	2	0.00	5
	`delete/3 : 0`	3	0.31	56
	`merge/4 : 0`	3	3.51	176
	`merge_sort/2 : 0`	3	5.44	150
	`chunk_2/5 : 0`	5	7.18	223
	`hosort/5 : 0`	3	8.51	272

Table 4.1: Some benchmarks.

is fairly small. The `adrlist` example implements a small address book implemented as a list of a structured type. Note that the procedures `get/3` and `set/4` – although simple in implementation, decompose the list and the structure of the latter's elements, resulting in constraints on several subparts of the predicate's arguments. Hence the rather high number of constraints and the time needed to create them. The last entry in the table represents a selection of the Melbourne compiler's `list` module. Note that the time needed to construct the procedure's normal form (and the number of constraints therein) depends on how complicated the structure of the procedure's body goal is. Indeed, large procedures with a lot of nested subgoals result in a lot of constraints on the $\mathcal{C}$ and $\mathcal{R}$ variables associated to the subgoals. For completeness sake, the table with analysis results for the complete `list` module can be found in Appendix A.7.

4.5.3 Some Remaining Issues

Few binding-time analyses have been developed that are polyvariant, deal with partially instantiated data, modules and higher-order constructs for a realistic lan-

guage. Our binding-time analysis achieves this by combining a number of known techniques: partially instantiated structures are dealt with by incorporating a structured domain of binding-times, polyvariance and modularity are achieved by computing the binding-times symbolically and higher-order information is incorporated by propagating closure information during the symbolic phase of the analysis. However, a number of issues remain.

First of all, while the use of a structured domain of binding-times allows a fine-grained approximation of a value, it may also result in a large number of constraints that are created. Indeed, deconstructing a value results in a number of constraints on (subvalues of) this value, which will again be split in additional constraints if these subvalues are in turn deconstructed. Moreover, complicated (nested) goals result due to the employed specialisation strategy in additional constraints (on the $\mathcal{C}$ and $\mathcal{R}$ variables) that again constrain the binding-times of program variables. This, in combination with the use of a structured domain for binding-time representation may lead to a rather large amount of constraints that are created in which a lot of dependencies are duplicated. Some indication of this problem is given by the experimental results in the previous subsection (the `addrlist` example which uses a structured type and several of the `list` module's procedures: `merge`, `merge_sort`, `chunk` and `hosort` being quite complicated in structure). Hence, an efficient implementation must deal with this problem, employing a smart normalisation strategy that avoids the duplication of computations. An interesting topic for further research is the formal investigation of such an analysis' computational complexity. Complexity results are known for some binding-time analyses, but – to the best of our knowledge – not for a combined approach as ours.

The specialisation strategy that is incorporated in the analysis is a basic strategy – similar in behaviour to Similix (Bondorf and Jørgensen 1993) – that focuses on guaranteeing local termination (if the program does not contain a static loop, see the discussion of Chapter 3). The power of such a strategy, in particular with respect to its application for Mercury, must be evaluated. Coupling the binding-time analysis with a specialiser for Mercury seems an interesting way to go. Several improvements to the basic strategy are likely. For example, Mercury's determinism information might be used in order to decide whether residual code (being a residue of original code for which determinism is known) might fail at run-time (expressed by the $\mathcal{R}$ value). Exploiting this information may result in less body atoms to become characterised as being under dynamic control, hence creating new possibilities for reduction. Coupling the analysis with a specialiser for Mercury seems interesting for other reasons also. First of all, a specialiser guided by binding-time analysis might perform a number of optimisations that are currently implemented in isolation in the Melbourne Mercury compiler. Examples of such optimisations, currently based on heuristics, include the inlining of procedure bodies, higher-order call specialisation and specialisation of type-info's

(Dowd, Somogyi, Henderson, Conway, and Jeffery 1999). We expect that coupling the analysis with such a specialiser will reduce the need for some of these isolated optimisations, but will likely indicate a number of (necessary) refinements to the analysis as well. Moreover, while the use of binding-time analysis in combination with a general program specialiser will likely perform more aggressive transformations of the original source code, it remains an open question how to predict its effect on the final efficiency of the program. Some initial work on predicting the effectiveness of program specialisation are reported in (Albert, Antoy, and Vidal 2000).

Chapter 5

Binding-Time Annotations without Binding-Time Analysis

...when you have eliminated the impossible,
whatever remains, however improbable, must be the truth.

– Sherlock Holmes (Sir Arthur Conan Doyle)

In this chapter, we present a binding-time analysis for a pure, unmoded logic programming language (a pure subset of Prolog). The binding-time analysis is obtained by modifying a termination analyses such that it guarantees termination at specialisation-time instead of termination at run-time.

5.1 Introduction and Motivation

While in general an on-line partial deduction system can achieve better results than an off-line system, the off-line approach also offers a number of advantages. First of all, the separation of the process in a binding-time analysis followed by a specialisation phase makes the process conceptually easier to reason about, and results in a fairly simple (and efficient!) specialiser from which the burden of continuously monitoring the evaluation process has been removed. Also, the analysis output can be represented by annotations on the original source program, and

provides as such excellent feedback to the user providing clues to why an optimisation was (not) performed. In spite of these advantages, only few efforts have been made to construct an off-line partial evaluator for logic programming. In (Mogensen and Bondorf 1993), an off-line partial evaluator for a subset of Prolog is described. The partial evaluator, called LogiMix, performs polyvariant specialisation and was developed with the emphasis on dealing with non-logical predicates and side effects and obtaining self-application of the specialiser. LogiMix, being an off-line specialiser, requires the variables in the program to be annotated as either *static* (meaning that they are neither more nor less instantiated than they would be during full evaluation) or *dynamic*, and requires annotations whether or not to unfold calls to user defined predicates. However, no automatic binding-time analysis is described to provide the necessary annotations. Only recently, the off-line approach has regained attention in a logic programming setting: in (Jørgensen and Leuschel 1996), it is shown that the *cogen* approach (obtaining specialisation through the execution of a "generating extension" which is generated by a compiler generator) achieves very good specialisation of logic programs while being several orders of magnitude faster than current on-line systems. Also this approach, however, requires a binding-time analysis to provide the necessary input. As in the case of LogiMix, the annotations in (Jørgensen and Leuschel 1996) are created by hand.

In the previous chapters, we have developed a binding-time analysis for Mercury. This analysis, however, can not straightforwardly be generalised towards a binding-time analysis for a generic logic programming language (in case Prolog). Indeed, the propagation of binding-times and the unfolding strategy in the binding-time analysis for Mercury relies much on the presence of mode information. For Mercury, this is not a problem, as the mode information is also required for compilation and hence has to be provided by the programmer (at least for those predicates that can be called from outside a module). Although modern implementations of a more generic logic programming language like Ciao (Hermenegildo 1994b; Hermenegildo 1994a) and Sicstus (Andersson, Andersson, Boortz, Carlsson, Nilsson, Sjöland, and Widn 1993) offer the possibility to add mode information to a (Prolog) program, this information is usually provided either for debugging purposes or to help the compiler generate more efficient code. It is not a part of the language, and it is not required for compilation or execution.

One of the most studied analyses for unmoded logic programming languages is groundness analysis. Groundness analysis approximates the values a variable can take by *ground* in case it definitely has a ground value, or *possibly non-ground* otherwise. The domain for groundness analysis resembles much a (simple) domain for binding-time analysis, where variables are to be characterised *static* when the variable's (ground) value is definitely known, or *dynamic* otherwise. The main difference between a binding-time analysis and a groundness analysis is that groundness approximates the value of a variable *at runtime*, whereas binding-times approx-

imate the value of a variable *at specialisation time*. Hence, when approximating values during binding-time analysis, one has to take the annotations – specifying what predicate calls to unfold – into account. Indeed, if a predicate call will not be unfolded during specialisation, the variable bindings that would be created by the call, must not be taken into account during binding-time analysis, since the binding-time approximations need to reflect the variable's instantiation at specialisation-time.

Such an approach is followed in the work of Bruynooghe, Leuschel and Sagonas (Bruynooghe, Leuschel, and Sagonas 1998), which is, to the best of our knowledge, the first attempt to create an automatic binding-time analysis for an unmoded logic programming language. In this work, binding-time approximations of a program P w.r.t. a query Q are created by performing a traditional groundness analysis on a program that implements an on-line specialiser for P. The on-line specialiser for P is created by replacing the predicate calls in P by conditional statements, that perform the call only in case an unfolding condition is satisfied.

Example 5.1 *Consider the well-known* `append/3` *predicate, and an on-line specialiser for it as depicted in Fig. 5.1.*

`append/3`	on-line specialiser for `append/3`

`append([],Y,Y).`
`append([X|Xs],Y,[X|Zs]):-`
`  append(Xs,Y,Zs).`
	`append([],Y,Y).`
	`append([X|Xs],Y,[X|Zs]):-`
	`  (unf_append(Xs,Y,Zs) -> append(Xs,Y,Zs)`
	`  ; memoise_append(Xs,Y,Zs)).`

Figure 5.1: The `append/3` predicate and an on-line specialiser for it.

In the above example, the call `append(Xs,Y,Zs)` is replaced by the conditional statement

```
(unf_append(Xs,Y,Zs) -> append(Xs,Y,Zs)
    ; memoise_append(Xs,Y,Zs)).
```

The unfolding condition is implemented by a predicate that succeeds when a certain groundness condition on (a subset of) its arguments is satisfied. As noted in (Bruynooghe, Leuschel, and Sagonas 1998), this behaviour is rather unusual in the context of an on-line specialiser which usually bases its decisions on the complete unfolding history of the call, rather than only on the arguments of the call. Basing the unfolding decision on the availability of the actual arguments does make sense in an off-line setting where the unfold decisions are taken during binding-time analysis when no unfolding history is available. If the unfolding condition succeeds, the call is unfolded, otherwise the call is residualised (which is modeled by the `memoise_append` predicate in the above example).

Example 5.2 *In the* `append/3` *example, it makes sense to unfold a call to the* `append/3` *predicate when either the first or the third argument is ground. This*

condition is expressed by the following predicate definition:

```
unf_append(X,Y,Z):-ground(X).
unf_append(X,Y,Z):-ground(Z).
```

The next step in the binding-time analysis of (Bruynooghe, Leuschel, and Sagonas 1998) is to analyse the behaviour of the on-line specialiser, and use the results of analysis to make the unfolding decisions at compile time. Using a traditional groundness analysis on the on-line specialiser of Example 5.1 with respect to the unfolding condition of Example 5.2 and an initial call `append(ground,_,_)` stating that the first argument is definitely ground, would determine that the unfolding condition on the recursive call to `append` is satisfied, and the call will be annotated as reducible.

An important remaining issue in this binding-time analysis of (Bruynooghe, Leuschel, and Sagonas 1998) is the derivation of the unfolding conditions (the definition of the `unf_append` predicate in the above example). As it is noted in (Bruynooghe, Leuschel, and Sagonas 1998) they can either be given by the user, or provided by another analysis. The former approach is appealing for an advanced user, as it enables to control the unfolding by hand. If, on the other hand, the binding-time analysis should be able to run completely automatically, that is without any intervention from the user, it is mandatory that these unfold conditions can be generated by another analysis. As was suggested before in the literature (Bruynooghe, Leuschel, and Sagonas 1998; Decorte and Schreye 1998), it seems very natural to base the decision whether or not to unfold a predicate call onto the *termination properties* of the particular call. The motivation is obvious: if it can be shown that the (specialisation-time instance of the) call terminates under normal evaluation, unfolding the call will terminate during specialisation and the obtained binding-time annotations are safe.

Using termination as unfolding criterium is appealing also for another reason: there exist a vast amount of work on (automatic) termination analysis, in particular on termination analysis for logic programs. Most termination analysers start from a given program and query, and try to prove that the given query terminates (Lindenstrauss, Sagiv, and Serebrenik 1997; Mesnard 1996; Codish and Taboch 1999; Decorte, De Schreye, and Vandecasteele 1999). Other systems, however, like (Mesnard 1996), start from a program and derive, for each of the program's predicates, *termination conditions* guaranteeing that any call that satisfies these conditions terminates. For the `append/3` predicate, the system of (Mesnard 1996) (w.r.t. the termsize norm) derives that a call `append(X,Y,Z)` terminates if either `X` or `Z` is bound to ground term. Note that this condition can straightforwardly be used to implement the `unf_append/3` predicate required by the binding-time analysis of (Bruynooghe, Leuschel, and Sagonas 1998), as shown in Example 5.2.

However, appealing as the approach may seem, the use of *runtime* termination of a call as an unfold condition imposes considerable restrictions on the unfolding

possibilities. The fact that a call is marked reducible only in case it terminates under normal evaluation implies that only calls that can be *completely* unfolded to *true* or *fail* are unfolded. While this approach might be appropriate for some applications, it is not for general logic programs. Consider the following example, implementing a simple meta interpreter.

Example 5.3 *Consider the meta interpreter depicted in Fig. 5.2. The interpreter has the* `member/2` *and* `append/3` *predicates as object program. The clauses are numbered for later reference.*

```
1:solve([]).
2:solve([A|Gs]):- solve_atom(A), solve(Gs).

3:solve_atom(A):-clause(A,Body), solve(Body).

4:clause(member(X,Xs), [append(_,[X|_],Xs)]).
5:clause(append([],L,L), []).
6:clause(append([X|Xs],Y,[X|Zs]),[append(Xs,Y,Zs)]).
```

Figure 5.2: Vanilla meta interpreter

Suppose we want to specialise the meta interpreter from Example 5.3 with respect to the query `solve([member(X,Xs)])`. Any sensible termination analysis will indicate possible non-termination for this query, the reason being of course that, since an object level call `member(X,Xs)` does not terminate, neither will the meta call `solve([member(X,Xs)])`. Hence, if we take termination of a call as its unfolding condition, no call to `solve/1` will be unfolded when specialising the query `solve([member(X,Xs)])` in Example 5.3, resulting in a program that is far from optimally specialised.

Intuitively, from a specialisation point of view, we can see that it is perfectly safe to unfold all calls to the `solve/1` predicate as long as the intermediate calls to `solve_atom/1` are residualised. The idea is that the `solve/1` predicate in a sense only performs the parsing of an object goal (deconstructing a list of object atoms), which is terminating in Example 5.3. Thus, residualising the calls to `solve_atom/2` and unfolding the others results in the specialised program depicted in Fig. 5.3. Applying a standard structuring filtering transformation on the specialised program of Fig. 5.3 (Gallagher and Bruynooghe 1990) results in the program depicted in Fig. 5.4, which implements (a renaming of) the traditional `member/2` and `append/3` predicates. This example illustrates the need to be able to *partially* unfold a call: building a derivation in which some of the selected atoms are unfolded while others are residualised. Achieving this using runtime termination of a call as an unfolding criterium seems not feasible, since runtime termination assumes that *all* calls are unfolded. Yet termination of such partial unfolding remains an

```
solve([member(X,Xs)]):- solve_atom(member(X,Xs)).

solve_atom(member(X,Xs)):- solve_atom(append(A,[X|_], Xs)).

solve_atom(append([],[X|B],[X|B])).
solve_atom(append([E|Es],[X|B],[Z|Zs])):-
     solve_atom(append(Es, [X|B], Zs)).
```

Figure 5.3: Specialised Vanilla

```
solve([member(X,Xs)]):-
     sa_member(X,Xs).

sa_member(X,Xs) :-
     sa_append(A,[X|_],Xs).

sa_append([],[X|B],[X|B]).
sa_append([E|Es],[X|B],[Z|Zs]):-
     sa_append(Es, [X|B], Zs).
```

Figure 5.4: Applying structure filtering

important issue. In the remainder of this chapter, we try to merge these observations, and develop a binding-time analysis that does not incorporate the results of a separate termination analysis, but rather modifies a termination analysis, such that it takes the effect of residualising calls into account. As such the analysis proves "specialisation-time termination" rather than "runtime termination".

5.2 About Termination Analysis

We start by addressing some basic issues in the (automatic) termination analysis of logic programs. We focus in particular on the concepts and terminology of (Codish and Taboch 1999), as this work provides the necessary foundations for our binding-time analysis. For a survey on different aspects and techniques of termination analysis, we refer to (De Schreye and Decorte 1994). The termination of a query with respect to a program often depends on the particular selection rule used to construct the query's SLD derivation(s). In what follows, when we refer to the "termination" of a program, we refer to its termination with respect to the left-to-right selection rule.

5.2.1 Some Termination Analysis Basics

Basically, to prove termination of a query with respect to a particular logic program, it suffices to show that there is a strict decrease in size of subsequent predicate calls spawned by the query, where the size of a predicate call is measured over some well-founded domain. The size of a predicate call is measured by a so called *level mapping* (Cavedon 1989; Bezem 1990) and is usually computed as a linear combination of the sizes of (a subset of) the call's arguments. The size of an individual argument is measured by a so called *norm*, which is a function that maps a term to a natural number; an estimate of the "size" of the term.

In the termination analysis of (Codish and Taboch 1999), one is not only interested in concrete size estimates of the arguments of a particular call, but rather in expressing *size relations* between arguments of subsequent calls. To that extent, the notion of a *symbolic norm* (Lindenstrauss and Sagiv 1997; Codish and Taboch 1999) is used. A symbolic norm maps a term to an expression that possibly contains variables. To represent symbolic norms, we introduce the set of function symbols $\Sigma^{\|\cdot\|} = \mathbb{N} \cup \{+/2\}$.

Definition 5.1 *(From (Lindenstrauss and Sagiv 1997)) A* symbolic norm *is a function* $\|.\| : \mathcal{T}(\Sigma, V) \mapsto \mathcal{T}(\Sigma^{\|\cdot\|}, V)$ *such that*

$$\|t\| = \begin{cases} c + \sum_{i=1}^{n} a_i \|t_i\| & \text{if } t = f(t_1, \ldots, t_n) \\ t & \text{if } t \text{ is a variable} \end{cases}$$

where c *and* $a_1, \ldots, a_n$ *are non-negative integer constants depending only on* f/n.

Example 5.4 *Two symbolic norms that are often used are the* termsize *norm* $\|.\|_{ts}$*, counting the number of functors in a term, and the listlength norm* $\|.\|_{ll}$*, counting the number of elements in a list. These norms are defined as follows:*

$$\|t\|_{ts} = \begin{cases} 1 + \sum_{i=1}^{n} \|t_i\|_{ts} & \text{if } t = f(t_1, \ldots, t_n) \\ t & \text{if } t \text{ is a variable} \end{cases}$$

$$\|t\|_{ll} = \begin{cases} 1 + \|Xs\|_{ll} & \text{if } t = [X|Xs] \\ t & \text{if } t \text{ is a variable} \\ 0 & \text{otherwise} \end{cases}$$

Note that a symbolic norm can include variables in its value. An occurrence of a variable X in a symbolic norm means the "size" of X with respect to the given norm.

Example 5.5

$$\begin{array}{ll} \| f(a, b, g(c)) \|_{ts} = 5 & \| f(a, b, g(c)) \|_{ll} = 0 \\ \| [X_1, X_2, X_3] \|_{ts} = 4 + X_1 + X_2 + X_3 & \| [X_1, X_2, X_3] \|_{ll} = 3 \\ \| [X_1, X_2|X_3] \|_{ts} = 2 + X_1 + X_2 + X_3 & \| [X_1, X_2|X_3] \|_{ll} = 2 + X_3 \end{array}$$

Symbolic norms can be used to express size relations between arguments of a predicate call. In (Codish and Taboch 1999), for example, the atom `append(A,B,A+B)` specifies a relation in which the size of the third argument equals the sum of the sizes of the first two arguments. An important property on terms is the following:

Definition 5.2 *(From (Lindenstrauss and Sagiv 1997)) A term t is* instantiated enough *with respect to a symbolic norm $\|.\|$ if $\|t\|$ is a ground term.*

In Example 5.5, the term $[X_1, X_2, X_3]$ is instantiated enough with respect to the listlength norm (since $\| [X_1, X_2, X_3] \|_{ll} = 3$), but not with respect to the termsize norm (since $\| [X_1, X_2, X_3] \|_{ts} = 4 + X_1 + X_2 + X_3$, the latter expression still containing variables). The notion of a term being instantiated enough plays an important role in termination analysis. Indeed, if a term t is instantiated enough with respect to a given norm, this particular norm maps each instance of t to the same constant as it maps t itself to. Consequently, if a decrease can be shown between two terms t and t' that are both instantiated enough with respect to the norm under consideration, the decrease also holds between whatever instances of t and t' are considered.

As noted in (Codish and Taboch 1999), symbolic norms are similar to *semi-linear* norms as defined in (Bossi, Cocco, and Fabris 1994), with the difference that variables are mapped to variables (instead of to the constant 0). The notion of a term being instantiated enough is closely related to the notion of a term being "rigid" with respect to a semi-linear norm (Bossi, Cocco, and Fabris 1994; Decorte and Schreye 1998). For a detailed discussion on the role of rigidity in termination analysis, we refer to (Decorte and Schreye 1998).

Most systems for (automatic) termination analysis of logic programs rely explicitly or implicitly on the concept of a norm. Some practical systems often require the norm to be semi-linear and to be provided by the user (Codish and Taboch 1999; Lindenstrauss, Sagiv, and Serebrenik 1997; Mesnard 1996), while other systems try to derive a suitable semi-linear norm themselves such that termination for a given program can be proven (Decorte, de Schreye, and Fabris 1993; Bossi, Cocco, and Fabris 1994; Decorte, De Schreye, and Vandecasteele 1999).

Given a symbolic norm $\|.\|$, we define the *abstraction* of a program with respect to $\|.\|$ as the program that is obtained by replacing each term t in the program by $\|t\|$.

Example 5.6 *Consider the* `append/3` *predicate and its abstraction with respect to the list-length norm $\|.\|_{ll}$ (see Example 5.4) in Figure 5.5.*

Formally, an abstract program is syntactically defined over a first order constraint logic programming language denoted CLP($\mathcal{N}$) with predicate symbols Π and function symbols $\Sigma^{\|.\|}$ (Codish and Taboch 1999). Constraints in CLP($\mathcal{N}$) are conjunctions of the relations $\{=, \leq, \geq, <, >\}$ on $\mathcal{T}(\Sigma^{\|.\|}, V)$. For two atoms $A = p(t_1, \ldots, t_n)$ and $B = p(t'_1, \ldots, t'_n)$, we use $A = B$ as an abbreviation for the

append	abstract append/3 w.r.t. $\|.\|_{ll}$
`append([],Y,Y).`	`append(0, Y, Y).`
`append([X|Xs],Y,[X|Zs]):-` `append(Xs,Y,Zs).`	`append(1+Xs, Y, 1+Zs):-` `append(Xs, Y, Zs).`

Figure 5.5: The listlength abstraction of `append/3`

constraint $\bigwedge_{i=1}^{n}(t_i = t'_i)$. A clause in CLP($\mathcal{N}$) is of the form $H \leftarrow \mu, B_1, \ldots, B_n$ where μ is a constraint and $H, B_1, \ldots, B_n$ are atoms constructed with predicate symbols from Π and terms form $\mathcal{T}(\Sigma^{\|.\|}, V)$. For simplicity, we assume a normalised representation of clauses in CLP($\mathcal{N}$): the head of a clause contains only distinct variables. All unifications in the head are thus part of the constraint. For a program P, we denote its abstraction with respect to the norm $\|.\|$ by $P_{\|.\|}$. Again, we assume that each clause in $P_{\|.\|}$ has a unique number associated with it. Where appropriate, we will denote with $i : C$ the clause C with number i, we will often refer to a clause in a program by its associated number.

Example 5.7 *The CLP($\mathcal{N}$) program* $\texttt{append}_{\|.\|_{ll}}$ *in normalised form looks as follows:*

$$1 : append(X, Y, Z) \leftarrow X = 0, Y = Z.$$
$$2 : append(X, Y, Z) \leftarrow X = 1 + Xs, Z = 1 + Zs, append(Xs, Y, Zs).$$

Computations in CLP($\mathcal{N}$) are performed over $\mathbb{N}$ with the standard interpretations for $\{=, \leq, \geq, <, >\}$. This domain, denoted by $\mathcal{D}$ supports the classical operations (Jaffar and Maher 1994):

- A test for consistency: $\mathcal{D} \models \exists\mu$.
- Implication of one constraint by another: $\mathcal{D} \models \mu_0 \rightarrow \mu_1$.
- Projection of a constraint μ_0 onto a set of variables V: $\overline{\exists}_V(\mu_0)$.

5.2.2 Abstract Binary Unfoldings

We follow the approach of (Codish and Taboch 1999), and compute a finite approximation of the *abstract binary unfoldings semantics* (Codish and Taboch 1999) of a program P. The abstract binary unfoldings semantics of P consists of a (possibly infinite) set of abstract binary clauses. Where the abstract program expresses existing relations on the size of the arguments in the program, the associated abstract binary clauses express relations on the sizes of the arguments in *subsequent calls* that can occur in $P_{\|.\|}$. We slightly adapt some definitions of (Codish and Taboch 1999) to fit our needs. A first definition is that of a binary clause, which we generalise to the notion of a *labelled* abstract binary clause.

Definition 5.3 *A* labelled abstract binary clause *is a clause in CLP($\mathcal{N}$) that is either of the form $H \overset{i,j}{\leftarrow} \mu$, or $H \overset{i,j}{\leftarrow} \mu, B$ where $i, j \in \mathbb{N}$. The set of all such binary clauses is denoted by $\mathcal{BC}$. A clause of the first form, $H \overset{i,j}{\leftarrow} \mu$ is also referred to as a labelled constrained atom.*

The set of labelled abstract binary clauses of a program $P_{\|\cdot\|}$ is defined as the least fixed point of the operator $T_P^{\|\cdot\|}$ defined in Definition 5.4. The $T_P^{\|\cdot\|}$ operator is adapted from (Codish and Taboch 1999) such that it associates a label to each constructed binary clause, referring to how the binary clause was constructed. In what follows, we will often simply refer to "binary clauses" when we mean effectively "abstract labelled binary clauses". Also, we will drop the label from a binary clause when it is unimportant. In Definition 5.4, Id denotes the set of *identity* binary clauses, these are clauses of the form $p(X_1, \ldots, X_n) \leftarrow p(X_1, \ldots, X_n)$. Unfolding an atom with respect to an identity clause results in the atom itself.

Definition 5.4 $T_P^{\|\cdot\|} : \wp(\mathcal{BC}) \mapsto \wp(\mathcal{BC})$ *is defined as*

$$T_P^{\|\cdot\|}(I) = \left\{ H \overset{i,j}{\leftarrow} \mu, B \;\middle|\; \begin{array}{l} C = i : H \leftarrow \mu_0, B_1, \ldots, B_m \in P_{\|\cdot\|}, 1 \leq j \leq m, \\ \langle A_k \leftarrow \mu_k \rangle_{k=1}^{j-1} \ll_C I, \\ A_j \leftarrow \mu_j, B \ll_C I \cup Id, j < m \Rightarrow B \neq true \\ \mu' = \mu_0 \wedge \bigwedge_{k=1}^{j} (\mu_j \wedge \{B_k = A_k\}) \\ \mu = \overline{\exists}_{vars(\langle H, B \rangle)}(\mu') \end{array} \right\}$$

Given a set of binary clauses I, $T_P^{\|\cdot\|}(I)$ is a new set of binary clauses constructed by unfolding prefixes of clauses in $P_{\|\cdot\|}$. If $H \leftarrow \mu, B_1, \ldots, B_n$ is a clause in $P_{\|\cdot\|}$, for each $1 \leq j \leq m$, the body atoms $B_1, \ldots, B_{j-1}$ are unfolded with respect to constrained atoms in I and the corresponding instance of B_j is unfolded with respect to a binary clause $H_j \leftarrow \mu_j, B$ ($B \neq true$) from $I \cup Id$. Note that the use of the identity clause to "unfold" B_j results in a binary clause of the form $H \leftarrow \mu, B_j$ (which expresses that a call unifying with μ, H results in a call unifying with μ, B_j). Constrained atoms are allowed to unfold B_j only in case $j = m$; indeed, an answer is obtained only in case all body atoms are unfolded by a constrained fact. Note that the label associated to a clause constructed by $T_P^{\|\cdot\|}$ carries information on how the clause was constructed: a clause $H \overset{i,j}{\leftarrow} \mu, B$ is created by resolving the $j-1$ leftmost body atoms of the i'th clause of $P_{\|\cdot\|}$ with a constrained atom and the j'th atom with a binary clause.

In general, the least fixed point of $T_P^{\|\cdot\|}$, $\mathrm{lfp}(T_P^{\|\cdot\|})$ is an infinite set of binary clauses, as illustrated in the next example (from (Codish and Taboch 1999)).

Example 5.8 *Reconsider the* `append/3` *predicate from Example 5.7. The abstract binary unfoldings are computed as follows:*

$$(1)\quad (T_P^{\|\cdot\|})^1(\emptyset) = \left\{ \begin{array}{l} append(X,Y,Z) \overset{1,1}{\leftarrow} X = 0, Z = Y. \\ append(X,Y,Z) \overset{2,1}{\leftarrow} \\ \qquad X = 1 + Xs, Z = 1 + Zs, \\ \qquad append(Xs,Y,Zs). \end{array} \right\}$$

$$(2)\quad (T_P^{\|\cdot\|})^2(\emptyset) = \left\{ \begin{array}{l} append(X,Y,Z) \overset{2,1}{\leftarrow} X = 1, Z = 1 + Y. \\ append(X,Y,Z) \overset{2,1}{\leftarrow} \\ \qquad X = 2 + Xs, Z = 2 + Zs, \\ \qquad append(Xs,Y,Zs). \end{array} \right\} \cup (T_P^{\|\cdot\|})^1(\emptyset)$$

$$(3)\quad (T_P^{\|\cdot\|})^3(\emptyset) = \left\{ \begin{array}{l} append(X,Y,Z) \overset{2,1}{\leftarrow} X = 2, Z = 2 + Y. \\ append(X,Y,Z) \overset{2,1}{\leftarrow} \\ \qquad X = 3 + Xs, Z = 3 + Zs, \\ \qquad append(Xs,Y,Zs). \end{array} \right\} \cup (T_P^{\|\cdot\|})^2(\emptyset)$$

$\ldots$

To obtain a finitary analysis, different approaches exist to further approximate the abstract domain. One possibility is to derive affine inter-argument relations using linear equations (Verschaetse and De Schreye 1991; Karr 1976). In (Benoy and King 1997; Cousot and Halbwachs 1978), polyhedral approximations are used, combined with a convex hull operation as a least upper bound and a widening. Disjunctions of monotonicity and equality constraints are used in (Lindenstrauss and Sagiv 1997; Brodsky and Sagiv 1989).

Example 5.9 *Reconsider the abstract binary unfoldings of* `append/3` *from Example 5.8. Further abstracting using polyhedral approximations (Benoy and King 1997; Cousot and Halbwachs 1978) (thereby arbitrarily keeping one of the involved labels) results in the set*

$$lfp(T_P^{\|\cdot\|}) = \left\{ \begin{array}{l} append(X,Y,Z) \overset{1,1}{\leftarrow} Z = Y + X. \\ append(X,Y,Z) \overset{2,1}{\leftarrow} \\ \qquad Xs < X, Zs < Z, Ys = Y, \\ \qquad append(Xs,Ys,Zs). \end{array} \right\}$$

The binary clauses capture size relations that exist between the arguments of subsequent predicate calls. In order to be useful for termination analysis, these size relations must be combined with instantiation information, that specifies which of the arguments are instantiated enough with respect to the norm under consideration. Such instantiation information is obtained by a standard groundness analysis on the abstracted program. For example, a *Pos* based analysis on the abstracted program derives instantiation dependencies between the arguments of subsequent calls. A dependency of the form $X \rightarrow Y$ meaning that if X is instantiated enough, then so is Y. In what follows we consider, as in (Codish and Taboch 1999), an abstract domain that combines size relations and instantiation information. We

denote with mgu^α the abstract most general unifier over this domain, and denote with $\approx$ the equivalence of syntactic objects. For a program P, we denote with $\mathcal{B}_P$ the finite set of abstract binary clauses which approximates, with respect to some given abstraction, $\text{lfp}(T_P^{\|\cdot\|})$ over the combined abstract domain.

Note that computing $\mathcal{B}_P$ is a call independent process. In termination analysis, one is interested in the termination behaviour of a specific call with respect to the given program. First, we define the abstraction of a call $p(t_1, \ldots, t_n)$ with respect to a norm $\|.\|$ as $\|p(t_1, \ldots, t_n)\| = p(\|t_1\|, \ldots, \|t_n\|)$. The (possibly infinite) set of calls that arise during computation of an initial call Q in P can be approximated by a finite set of abstract calls, $calls_P^\alpha(Q)$, which is determined as follows:

Definition 5.5 *Given a program P, an abstract initial call Q and a finite set of abstract binary clauses $\mathcal{B}_P$ approximating $lfp(T_P^{\|\cdot\|})$.*

$$calls_P^\alpha(Q) = \left\{ \; B\theta \;\middle|\; \begin{array}{l} H \leftarrow B \in \mathcal{B}_P, \\ \theta = mgu^\alpha(Q, H) \end{array} \right\}$$

The work of (Codish and Taboch 1999) defines a sufficient condition to show termination of a call Q with respect to a program P. Given a finite approximation $\mathcal{B}_P$ of $\text{lfp}(T_P^{\|\cdot\|})$, it is sufficient to show for each call $C \in calls_P^\alpha(Q)$ a strict decrease in size from head to the single body atom for all recursive clauses of $\mathcal{B}_P$ that unify with the call C. Adding labels to the abstract binary clauses enables to reformulate the termination condition of (Codish and Taboch 1999) at the level of the *original* clauses of P. To that extent, we introduce the notion of a clause being *loop safe* in one of its body atoms. Intuitively, if we say that a clause $H \leftarrow B_1, \ldots, B_n$ is loop safe in it's i'th body atom with respect to a set of (abstract) calls S, this means that none of the calls in S unifying with H will spawn an infinite derivation through (instances of) this body atom.

Definition 5.6 *Given a program P, a set of abstract calls S and a finite approximation $\mathcal{B}_P$ of $lfp(T_P^{\|\cdot\|})$. Assume there exist a binary clause $\beta = h \overset{i,j}{\leftarrow} \mu, B \in \mathcal{B}_P$ and a call $C \in S$ such that $\theta = mgu^\alpha(H, C)$ and $B\theta \approx C$. Let $i_1, \ldots, i_k$ be the argument positions that are instantiated enough both in $H\theta$ and $B\theta$. We denote these arguments by $(H\theta)_{i_1}, \ldots, (H\theta)_{i_k}$ and $(B\theta)_{i_1}, \ldots, (B\theta)_{i_k}$. We say that the clause i of P is* loop safe *with respect to S in body atom i if for each such $\beta \in \mathcal{B}_P$ and $C \in S$ there exists a function f such that*

$$\mu \models f((H\theta)_{i_1}, \ldots, (H\theta)_{i_k}) > f((B\theta)_{i_1}, \ldots, (B\theta)_{i_k}).$$

Note that Definition 5.6 takes only those (abstract) calls and binary clauses into account such that applying the most general unifier of the head of the clause and the call on the body atom results in a recursive call with an equivalent call pattern. See (Codish and Taboch 1999) for details on why this is sufficient. Given

Definition 5.6 from above, we define for a clause i of P and a set of abstract calls S,

$$LS^i_{P,S} = \{\ j \mid \text{the clause } i \text{ is loop safe w.r.t. } S \text{ in body atom } j\ \}$$

Example 5.10 *Let P denote the program consisting of the* `append/3` *predicate with the abstract binary unfoldings from Example 5.9 and let Q denote the abstracted initial call* `append(0,Y,Z)`*. This call unifies (through mgu^α) only with the head of the binary clause labelled $(1,0)$, since unification of* `append(0,Y,Z)` *with the other clause, labelled $(2,1)$, fails due to the fact that the size constraints are not satisfied since no $Xs < 0$. Hence, we have that both clauses of* `append/3` *are loop safe with respect to $S = calls^\alpha_P(append(0,Y,Z)) = \{append(0,Y,Z)\}$. Or, we have that*

$$\begin{array}{rcl} LS^1_{P,S} & = & \emptyset \\ LS^2_{P,S} & = & \{1\} \end{array}$$

Now, we can reformulate an important result of (Codish and Taboch 1999).

Theorem 5.1 *Given a program P and initial goal Q. If each clause of P is loop safe with respect tot $calls^\alpha_P(Q)$ in each of its body atoms, then Q terminates with respect to P.*

Theorem 5.1 provides a sufficient condition for termination, formulated as a condition on each of the body atoms of the program's clauses. In what follows, we are interested in identifying those clauses and body atoms for which it can not be proven that they are loop safe. We define the notion of the *leftmost possibly looping atom* of a clause as follows:

Definition 5.7 *Given a program P, a set of abstract calls S and a finite approximation $\mathcal{B}_P$ of $lfp(T_P^{\|\cdot\|})$. Let $H \leftarrow B_1, \ldots, B_n$ be the i'th clause of P. We define its* leftmost possibly looping body *atom as follows*

$$LLA^i_{P,S} = \begin{cases} min(\{1,\ldots,n\} \setminus LS^i_{P,S}) & \text{if } (\{1,\ldots,n\} \setminus LS^i_{P,S}) \neq \emptyset \\ undefined & otherwise \end{cases}$$

Example 5.11 *Let P denote the program consisting of the* `append/3` *predicate with the abstract binary unfoldings from Example 5.9 and let Q denote the abstracted initial call* `append(X,0,Z)` *and $S = calls^\alpha_P(Q)$. Although we still have that $LS^1_{P,S} = \emptyset$, we also have that $LS^2_{P,S} = \emptyset$. Indeed, the second clause in the* `append` *program is not loop safe with respect to S in its only body atom, due to the existence of the binary clause labelled $(2,1)$ (see Example 5.9). Unifying this clause with the call* `append(X,0,Z)` *results in the binary clause*

$$append(X,0,Z) \leftarrow append(Xs,0,Zs).$$

Only the second argument of both atoms is instantiated enough, and there does not exist a function f such that $X > Xs, Z > Zs \models f(0) > f(0)$. Hence, we have that

$$LLA^2_{P,S} = 1.$$

In general, finding the leftmost looping atom of a clause (if it exists) is an undecidable problem. In practical systems, however, the function f in Definition 5.6 is fixed, and is usually defined as a linear combination of the involved arguments. When f is fixed, the test of Definition 5.6 can be evaluated, and consequently the sets $LS^i_{P,Q}$ and $LLA^i_{P,Q}$ can be computed. For any clause i in P, $LLA^i_{P,S}$ provides a safe approximation of the leftmost looping atom in the clause, since it is guaranteed that no atom to its left can be looping, when the program is evaluated with respect to an initial call Q and $S = calls^\alpha_P(Q)$.

5.3 From Termination analysis to Binding-time Analysis

Given a program P and an initial call Q, the basic task of binding-time analysis is to produce an annotated version of P with respect to Q such that specialising Q by following the annotations terminates. In what follows, we concentrate on the local control problem: specialising a single call (producing a single residual predicate by unfolding the call) terminates. Note that in the context of program specialisation, the call Q represents a call with possibly incomplete information as it denotes the specialisation-time call.

Annotating the body atoms of a clause usually consists of adding a label to each atom, specifying whether the atom is unfolded during specialisation, or residualised. In this work, however, we focus on the termination aspects of the unfolding, and hence employ a slightly different notion of annotations. In what follows, we simply replace atoms that should be residualised by *true*. Doing so permits to study the termination behaviour of a specialiser that simply unfolds the reducible atoms (and generates code for those that are residualised) by studying the termination behaviour of the annotated program under normal evaluation.

Definition 5.8 *Given a clause $H \leftarrow B_1, \ldots, B_n$. An* annotated version of *the clause is a clause $H \leftarrow B'_1, \ldots, B'_n$, where for each i such that $1 \leq i \leq n$ holds that either $B'_i = B_i$ or $B'_i = true$. An annotated version of a program $P = \bigcup_i C_i$ is a program $P' = \bigcup_i C'_i$ such that for every such clause C_i holds that C'_i is an annotated version of C_i.*

Note that, according to Definition 5.8, every clause is an annotated version of itself. When annotating a program, one generally wants to mark as much atoms reducible as possible, while guaranteeing termination of the unfolding. This is the main idea behind the analysis presented in this chapter. Suppose we have to annotate a program P with respect to an initial call Q. If we can prove that Q terminates with respect to P, the annotated version of P is simply P itself (every atom is annotated reducible). Hence, during specialisation, the goal Q will be completely unfolded, and specialisation of Q boils down to plain evaluation of Q

in P. If, on the other hand, termination of Q with respect to P can not be proven by our analysis, at least one of P's clauses must have a leftmost looping atom, and we mark this atom to be residualised (by replacing it by *true* in P). This process is repeated until the annotated program is loop safe. This is the main intuition behind the following algorithm.

Algorithm 5.1
Given a program P and initial call Q.
Let $P_0 = P$, $S_0 = calls^{\alpha}_{P}(Q)$, $k = 0$.
repeat
  if there exist a clause i in P_k such that $LLA^{i}_{P_k,S_k} = j$
  then
    let P_{k+1} be the program obtained by replacing the j'th body atom in the i'th clause in P_k by true and
    let $S_{k+1} = S_k \cup calls^{\alpha}_{P_{k+1}}(Q)$
  else
    $P_{k+1} = P_k$
until $P_k = P_{k+1}$
$P' = P_k$, $S' = S_k$

Note that Algorithm 5.1 is non deterministic: if several clauses i exist in P_k for which $LLA^{i}_{P_k,S_k}$ is defined, one of these must be selected for replacement by *true*. We return to this issue in Section 5.4. Also note the construction of the set S': starting from the program's initial abstract callset S_0, the abstract callset of the annotated program is added in each round of the algorithm. Doing so guarantees that the calls that are unfolded are correctly represented by an abstract call in S', but it also ensures that S' contains abstractions of the (concrete instances of the) calls that were replaced by *true* during the process. In other words, the set S' contains an abstraction of every call that is encountered (unfolded or residualised) during specialisation of P with respect to the initial call Q. An important observation is that Algorithm 5.1 terminates.

Proposition 5.1 *Algorithm 5.1 terminates for any program P and initial call Q.*

Proof The proof is immediate, since for each $k > 0$, P_k differs from P_{k-1} in the fact that a literal in a clause from P_{k-1} is replaced by *true* in P_k. Since there is only a finite number of literals in a program P, the result follows. □

The following example illustrates Algorithm 5.1 on the vanilla meta interpreter from this chapter's introduction.

Example 5.12 *Reconsider the vanilla meta interpreter with the sample object program depicted in Figure 5.2 and the initial call Q = solve([member(X,Xs)]).*

Let P_0 and S_0 be as in Algorithm 5.1. The labelled binary clauses associated to P_0 with respect to the listlength norm are depicted in Fig. 5.6. The constrained atoms are not listed. We have that clauses 2 *and* 3 *are the only ones that are not loop*

$\mathcal{B}_{P_0}$	Size relations	Instantiation relations
$solve\_atom(A) \overset{3,1}{\leftarrow} clause(B,C)$	$[A = B]$	[]
$solve\_atom(A) \overset{3,2}{\leftarrow} solve(B)$	[]	[]
$solve\_atom(A) \overset{3,2}{\leftarrow} solve\_atom(B)$	[]	[]
$solve(A) \overset{2,1}{\leftarrow} solve(B)$	[]	[]
$solve(A) \overset{2,2}{\leftarrow} solve(B)$	$[B < A]$	$[A \rightarrow B]$
$solve(A) \overset{2,1}{\leftarrow} solve\_atom(B)$	[]	[]

Figure 5.6: The labelled abstract binary clauses of P_0.

safe, as follows:

$$LLA^2_{P_0,S_0} = 1$$
$$LLA^3_{P_0,S_0} = 2$$

Let us annotate the second clause of P_0 by replacing its first body atom (the call to `solve_atom`*) by true. The resulting program, P_1 is depicted in Fig. 5.7, its associated binary clauses in Fig. 5.8. Now, there does not exist a clause i in P_1*

```
1:solve([]).
2:solve([A|Gs]):-true, solve(Gs).

3:solve_atom(A):-clause(A,Body), solve(Body).

4:clause(member(X,Xs), [append(_,[X|_],_Xs)]).
5:clause(append([],L,L), []).
6:clause(append([X|Xs],Y,[Z|Zs]),[append(Xs,Y,Zs)]).
```

Figure 5.7: Annotated vanilla meta interpreter.

with $LLA^i_{P_1,S_1}$ defined, hence Algorithm 5.1 reaches a fixed point and P_1 is loop safe.

The result of Algorithm 5.1 is itself a logic program P'. The following proposition states that evaluating the initial goal Q with respect to P' terminates.

Proposition 5.2 *Given a program P and initial goal Q. Let P' denote the annotated version of P obtained by applying Algorithm 5.1. Then Q terminates w.r.t. P'.*

$\mathcal{B}_{P_1}$	Size relations	Instantiation relations
$solve\_atom(A) \overset{3,1}{\leftarrow} clause(B,C)$	$[A = B]$	$[]$
$solve\_atom(A) \overset{3,2}{\leftarrow} solve(B)$	$[]$	$[]$
$solve(A) \overset{2,2}{\leftarrow} solve(B)$	$[B < A]$	$[A \rightarrow B]$

Figure 5.8: The labelled abstract binary clauses of P_1.

Proof If P' is the result of applying Algorithm 5.1, then P' is loop safe by the definition of the algorithm. Consequently, by Theorem 5.1, Q terminates w.r.t. P'.

□

Proposition 5.2 is an important result, as it implies local termination of a specialiser that simply follows the annotations. First, we define an unfolding rule that follows the annotations computed by the binding-time analysis. In what follows, let P' denote the program resulting from applying Algorithm 5.1 on P and an atomic partial deduction query Q.

Definition 5.9 *An atom A in a goal at the leaf of an SLD-tree is* selectable *unless it is an instance of an atom in P that is replaced by true in P'. The unfolding rule U_{bta} unfolds the leftmost selectable atom in each goal of the SLD-tree under construction. If no atom is selectable, no further unfolding is performed.*

Now, the SLD-tree built by U_{bta} for $P \cup \{Q\}$ is finite since U_{bta} unfolds – apart from Q – only atoms that are instances of an atom in P'. Moreover, this holds for every atom A that is abstracted by an atom in the set S' resulting from Algorithm 5.1.

Corollary 5.1 *Let P be a program and Q an atomic partial deduction query. Let P' and S' denote the result of applying Algorithm 5.1 on P and Q. If A is an atom whose abstraction is in S', the SLD-tree constructed by U_{bta} for $P \cup \{\leftarrow A\}$ is finite.*

Proof Immediate by the definition of U_{bta} and Proposition 5.2.

□

Being an unfolding rule, U_{bta} can be casted in the generic partial deduction algorithm of Chapter 2 that constructs, starting from the initial call Q, a set of atoms $\mathcal{A}$ for which finite SLD-trees are constructed such that the corresponding resultants are $\mathcal{A}$-closed. If P' and S' denote the result of applying Algorithm 5.1, Corollary 5.1 ensures *local* termination of the process. Indeed, U_{bta} constructs a

Figure 5.9: The SLD-trees constructed by U_{bta}.

finite SLD-tree for Q and, by definition of Algorithm 5.1 and construction of the set S', every atom in $\mathcal{A}$ is abstracted by an atom in S'. Hence, U_{bta} builds a finite SLD-tree for each of the atoms in $\mathcal{A}$.

Example 5.13 *Reconsider the vanilla meta interpreter with a sample object program depicted in Figure 5.2 and the initial atomic partial deduction query Q = solve([member(X,Xs)]). Applying the generic partial deduction algorithm from Chapter 2 using U_{bta} for local control and a simple variant check for global control results in the set*

$$\mathcal{A} = \left\{ \begin{array}{l} \texttt{solve([member(X,Xs)])} \\ \texttt{solve\_atom(member(X,Xs))} \\ \texttt{solve\_atom(append(A,[X|\_],Xs))} \end{array} \right\}$$

with the partial SLD-trees depicted in Fig. 5.9 and the residual program that was depicted in Fig. 5.3 that was depicted in this chapter's introduction.

Although the binding-time analysis guarantees local termination, it does not diminish the need for global control during the specialisation process. Indeed, the abstract callset S' is a finite set, but an infinite number of concretisations of the calls in S' can be constructed during specialisation, resulting in an infinite set $\mathcal{A}$.

5.4 Discussion and Related Work

We now discuss some of the issues raised by our binding-time analysis. For a general discussion on related work regarding binding-time analysis for functional languages, we refer to the discussion sections of Chapters 3 and 4 in which a binding-time analysis for the logic programming language Mercury was developed.

5.4.1 General Discussion

The binding-time analysis implemented by Algorithm 5.1 is *monovariant*: it creates only a single annotated version of every predicate. This is not a problem for correctness, as the annotations are safe in the sense that they guarantee termination of U_{bta} of *every* call that is specialised during the process. However, the resulting annotations are likely to be suboptimal, since calls in which different arguments are instantiated enough are likely to exposure a different termination behaviour, and hence they might profit from being unfolded differently.

Recall that Algorithm 5.1 implements some nondeterministic behaviour, since in each round it may be necessary to make a choice between different clauses having a leftmost possibly looping atom. When a choice has to be made between several leftmost possibly looping atoms that indicate different loops in the program, only one of the loops is broken no matter what atom is replaced by *true*, and the other loop will still be indicated by a leftmost possibly looping atom in the next round of the algorithm. When a choice has to be made between several leftmost possibly looping atoms that indicate the *same* loop in the program, the loop will be broken no matter what particular atom is chosen to be replaced by *true*. This was illustrated in Example 5.12, where the program P_0 has two such clauses, both indicating the same loop between the mutually recursive predicates `solve/1` and `solve_atom/1`.

$$LLA^2_{P_0,S_0} = 1$$
$$LLA^3_{P_0,S_0} = 2$$

The example shows that choosing the first one, clause number 2, leads to excellent specialisation results, as all the meta interpretation overhead is removed after specialisation according to the computed annotations. When the second option is chosen, different SLD-trees are built. Since the only residualised atom is the call `solve(Body)` in the third clause, all the constructed SLD-trees (except the first) will have an instance of this atom as their root. Although for Example 5.12, the residual program will be essentially the same, this observation can not be generalised. The key issue in this example is that when the first option is chosen, the roots of the constructed SLD-trees are instances of `solve_atom(A)`, representing an object level *atom*. Contrarywise, when the second option is taken, the roots of the constructed SLD-trees are instances of `solve(Body)`, representing an object level *conjunction*. When meta interpreters of this kind are specialised, this distinction may determine whether or not all meta interpretation is removed by the

specialisation process, and hence determines the obtained degree of specialisation. We discuss the specialisation of this kind of meta interpreters and the importance of local and global control therein in greater detail in Chapter 7. In our present setting, we conclude that, when the same loop is indicated by several leftmost looping atoms, the particular choice made in Algorithm 5.1 might determine the obtained degree of specialisation. Whether it is possible to guide the algorithm towards making a "good" choice, and if so how to do it, is a topic for further research.

Being essentially an iteration of termination analyses, our binding-time analysis relies on the particular norm and level mapping that are chosen. During analysis, a term is abstracted to *true* if it is "sufficiently instantiated" (or "rigid") with respect to the norm, or to *false* otherwise. Although some work exists towards automating the process of choosing a suitable norm (Decorte, De Schreye, and Vandecasteele 1999), most systems require the norm to be selected by the user. The particular norm that is used by the system determines the granularity of a term that is considered *static* for binding-time analysis. The use of the *termsize* norm, for example, corresponds to distinguishing, during binding-time analysis, between definitely ground terms and possibly non ground terms. The use of the *listlength* norm, on the other hand, enables to consider a term *static* when it is instantiated up to a list skeleton. In, (Leuschel, Jørgensen, Vanhoof, and Bruynooghe 2001), the notion of a *binding-type* is defined. Binding-types are defined much in the same way types are, but possibly using `static`, `dynamic` and `nonvar` as primitive type constructors. Like the type of a variable denotes the set of terms that can be bound to the variable at run-time, the binding-type of a variable denotes the set of terms that can be bound to the variable at specialisation-time. The following binding-type, for example, denotes the set of lists that are at least instantiated up to a list skeleton:

```
:- type listskel ---> [] ; [dynamic | listskel].
```

Recall that the definition of a (semi-linear) norm specifies, for a type definition, what subterms of a term of that type are taken into account for computing the term's norm. By replacing those subterms that are not taken into account by `dynamic` in the type definition, one obtains a binding-type that approximates precisely those terms that are rigid with respect to the norm. For example, the binding-type `listskel` defines exactly the set of terms that are instantiated enough with respect to the *listlength* norm (see the definition of the listlength norm in Example 5.4). Hence, by transforming each call in the abstract callset (the set S' output from Algorithm 5.1) such that the arguments of the call that are instantiated enough with respect to the norm are replaced with the binding-type associated to the norm and those that are not with *dynamic*, it is possible to derive, for every predicate, the *most dynamic* binding-type classification for which the predicate terminates. This classification is called a *division* in the terminology of (Leuschel, Jørgensen, Vanhoof, and Bruynooghe 2001). We return to the approach of (Leuschel, Jørgensen,

Vanhoof, and Bruynooghe 2001) in Section 5.4.2, where we report on some experiments that were performed with the described binding-time analysis. Scrutinising the exact relation between a norm and the corresponding domain of binding-times is a topic for further research.

5.4.2 Application and Experimentation

Table 5.1 summarizes a number of experiments that were run with a semi automatic implementation of the described binding-time analysis. The second column (*Round1*) presents the timings for termination analysis (Codish and Taboch 1999) of the original program (in which all calls are annotated static, i.e. reducible). In case the outcome of the analysis is possible non-termination, the third column (*Round2*) presents the timings for termination analysis of the program in which the problematic call is annotated dynamic. None of the benchmarks required more than two rounds of the termination analysis to derive a terminating annotated program. The benchmarks are all small examples taken from the DPPD library (Leuschel 1996), their code and the resulting annotated versions can be found in Appendix A. The benchmarks are reported in (Leuschel, Jørgensen, Vanhoof, and Bruynooghe 2001) and were run under SICStus Prolog 3.7.1 on a Sun Ultra E450 server with 256Mb RAM operating under SunOS 5.6.

Benchmark	Round 1	Round 2	LOGEN	Total	MIXTUS
`ex_depth`	240.0	230.0	4.4	474	200
`match`	470.0	180.0	2.4	652	50
`map.rev/reduce`	200.0	–	4.3	204	100
`parser`	100.0	50.0	–	150	–
`regexp1-3`	740.0	280.0	15.1	1035	670
`transpose`	210.0	150.0	7.0	367	290

Table 5.1: Timings (in ms) for the binding-time analysis and full specialisation.

The annotated versions resulting from the binding-time analysis were provided to the LOGEN system (Jørgensen and Leuschel 1996; Leuschel, Jørgensen, Vanhoof, and Bruynooghe 2001). The LOGEN system implements the so-called "cogen" approach for logic programming, in which specialisation is performed in two steps. Given a program, a specification of what inputs will be *static* and annotations specifying which calls should be unfolded, it generates a *generating extension* for the program at hand. Running this generating extension with particular values for the *static* inputs results in the specialised program. The fourth column in Table 5.1 contains the total time needed to produce the generating extension using LOGEN and to run it on the partial deduction query. The final column contains

the specialisation time of MIXTUS (Sahlin 1993) – a well-known on-line specialiser for Prolog – as a reference point. The only missing factor to make the binding-time analysis fully automatic, is the annotation of a particular call as dynamic after a run of the termination analysis. For the moment, this annotation is done by hand. However, we do believe the timings from Table 5.1 to be relevant as the termination analysis constitutes by all means the hard part of the analysis. Since the termination analysis pinpoints the call that is to be made residual, the time needed to actually annotate this call as dynamic can be neglected when compared with the timings for the termination analysis itself. Table 5.1 shows that binding-time analysis is the most expensive operation in the specialisation process. However, recall that the results of binding-time analysis can be used to perform *several* specialisations (with respect to values approximated by the binding-times from the partial deduction query). For the considered benchmarks, the cost of binding-time analysis will be recovered after a few specialisations compared with MIXTUS.

5.4.3 Future Work

Traditionally, binding-time analyses are described as an application of abstract interpretation: specialisation-time values are approximated by elements of a suitable abstract domain, which are propagated throughout the program. The hard part of the analysis, is that it must incorporate the effect of residualising particular calls in the program as well as deciding *what* calls to residualise. An example of such a binding-time analysis is (Bruynooghe, Leuschel, and Sagonas 1998). In this work, calls are residualised based upon the evaluation (at analysis time) of a condition, that is either provided by the user, or by another analysis. The main problem with such an approach, however, is that the unfolding conditions are expressed in isolation. That is, the condition whether or not to unfold a particular call does not take the effect of the conditions on other calls into account. Hence, the use of termination conditions as unfolding conditions implies that calls will either be completely reduced, or not at all. To the best of our knowledge, the binding-time analysis presented in this chapter is the first binding-time analysis for (a subset of) Prolog that can be fully automated while allowing more liberal unfoldings. The analysis takes a rather unusual approach towards binding-time analysis. Well-known techniques from termination analysis are used to annotate a program in subsequent steps, until it can be proven that *reduction* rather than *full evaluation* of a call terminates. The analysis is superior to known approaches based on run-time termination, as it allows to annotate predicates such that calls are *partially* unfolded.

Preliminary experiments show that the approach is feasible and results in more liberal unfoldings than with other (runtime-)termination based techniques like (Bruynooghe, Leuschel, and Sagonas 1998). Examples are the meta interpreter presented in this chapter's introduction and the `regexp` benchmark from above,

since in these programs, the ability to partially unfold a predicate call is crucial to achieve a fair amount of specialisation. In contrast with (Bruynooghe, Leuschel, and Sagonas 1998) – being a polyvariant analysis – our binding-time analysis is monovariant. This is not an issue in the benchmarks presented above, but definitely is when satisfying specialisation results are to be achieved for more involved programs. Another characteristic of the analysis that might be an issue when analysing larger programs, is that the binding-time analysis basically deals with boolean binding-times: either a value is instantiated enough with respect to a norm, or it is not. Moreover, the particular norm is fixed throughout analysis and most termination analysers assume the norm given by the user. Even if the norm is provided manually, finding a suitable norm might not be trivial or even impossible – in particular for programs that employ values of different types. These issues are not due to the binding-time analysis itself, but are rather connected with the termination analysis. We expect better (more precise) termination analyses to lead to better (more precise) binding-time analysis.

In this work, we have used termination analysis to ensure – in an off-line setting – *local* termination of the specialisation process. A topic for further research is the use of termination analysis towards *globally* controlling the specialisation process as well. In an off-line setting, global termination has often been sacrificed in favor of computational completeness. Examples are the partial evaluators Similix (Bondorf and Jørgensen 1993), Schism (Consel 1993b) and C-Mix (Andersen 1993). A possible approach towards ensuring (global) termination in off-line partial evaluation of functional programs is presented in (Holst 1991; Andersen and Holst 1996). In the latter work, the output of a termination analysis is used to make enough values *dynamic* such that the program enters – during specialisation – only a finite number of different configurations (where a configuration is defined as a program point together with values for the variables at that program point). If this is the case, the program is said to *quasi-terminate* and termination of partial evaluation is ensured by memoizing the configurations. Note that this approach also solves the problem of (local) non-termination due to the presence of a static loop in the program, as it is present for example in Similix (Bondorf and Jørgensen 1993) and the binding-time analysis for Mercury that was developed in Chapters 3 and 4. A weakness of the approach of (Holst 1991; Andersen and Holst 1996) is that its termination analysis only recognises "in-situ" decreases, i.e. a decrease in the size of a single argument between recursive calls. A more general termination analysis is developed in (Lee, Jones, and Ben-Amram 2001), capable of dealing with indirect function calls and permuted arguments (lifting the in-situ criterion). Developing an analogous analysis for binding-time analysis is mentioned as an important issue in (Lee, Jones, and Ben-Amram 2001).

The notion of quasi-termination for logic programs has also been explored (Decorte, De Schreye, Leuschel, and Martens 1998) in the context of termination analysis of *tabled* logic programs (Chen and Warren 1996; Bol and Degerstedt

1993; Tamaki and Sato 1986). In (Bruynooghe, Leuschel, and Sagonas 1998), it is noted that global termination of the process is ensured if quasi-termination of the program with the residualised predicates tabulated can be established. Precisely how to integrate such a technique with a suitable and refined abstraction mechanism is a interesting topic for further research.

Interludium

Chapter 6

Analysing Dependencies in Typed Logic Programs

Actor: "I'm a smash hit. Why, yesterday during the last act,
I had everyone glued in their seats!"
Oliver Herford: "Wonderful! Wonderful!
Clever of you to think of it!"

In this chapter, we generalise some of the ideas developed in Part I. We formulate an abstract domain, $Pos(\mathcal{T})$, that allows to redevelop *Pos*-based analyses in the presence of type information. Being a more fine-grained domain, $Pos(\mathcal{T})$-analyses can produce more accurate results than their *Pos*-counterparts.

6.1 Introduction

Dependencies play an important role in program analysis. A statement "program variable X has property p" can be represented by the propositional variable x^p and dependencies between properties of program variables can be captured as Boolean functions. For example, the function denoted by $x^p \rightarrow y^p$ specifies that whenever X has property p then so does Y. In many cases, the precision of a data flow analysis for a property p is improved if the underlying analysis domain captures dependencies with respect to that given property.

The analysis of groundness dependencies for logic programs using the class of *positive* Boolean functions is one of the main applications in this area of research.

The analysis aims to identify if program variable X has a unique value which cannot be changed. In logic programming terms this means that X is *ground*, or, contains no variables which can be further instantiated. This is the property presented by the propositional variable x. For dependencies, the choice is the class of *positive* Boolean functions, *Pos*, which consists of the Boolean functions f for which $f(true, \ldots, true) = true$. This restriction is natural as, due to the element of approximation, the result of an analysis is not a "*yes/no*" answer, but rather a "*yes/maybe not*" answer as there is no "negative" information.

One of the key steps in a groundness dependency analysis is to characterise the dependencies imposed by the unifications that could occur during execution. If the program specifies a unification of the form $term_1 = term_2$ and the variables in $term_1$ and $term_2$ are $\{X_1, \ldots, X_m\}$ and $\{Y_1, \ldots, Y_n\}$ respectively, then the corresponding groundness dependency imposed is $(x_1 \wedge \cdots \wedge x_m) \leftrightarrow (y_1 \wedge \cdots \wedge y_n)$ specifying that variables in $term_1$ are (or will become) ground if and only if the variables in $term_2$ are (or do). It is possible to improve the precision of an analysis if additional information about the structure (or patterns) of terms is available. For example, if we know that $term_1$ and $term_2$ are both difference lists of the form $H_1 - T_1$ and $H_2 - T_2$, respectively, then the unification $term_1 = term_2$ imposes the dependency $(h_1 \leftrightarrow h_2) \wedge (t_1 \leftrightarrow t_2)$ which is more precise than $(h_1 \wedge t_1) \leftrightarrow (h_2 \wedge t_2)$ which would be derived without the additional information. This has been the approach in previous works such as (Le Charlier and Van Hentenryck 1994; Mulkers, Simoens, Janssens, and Bruynooghe 1995; Codish, Marriott, and Taboch 2000; Cortesi, Charlier, and Hentenryck 2000; Bagnara, Hill, and Zaffanella 2000) where simple pattern analysis is used to enhance the precision of other analyses.

This chapter presents a technique to improve the precision of dependency analyses in the presence of type information. Given information about the types of the terms $term_1$ and $term_2$ we can specify dependencies regarding the subterms of $term_1$ and $term_2$ which can unify. While the approach is similar to that taken in previous work, type information is more sophisticated than the structure information used in the past and can lead to more precise analysis results. Type information can be derived by analysis, as e.g. in (Janssens and Bruynooghe 1992; Gallagher and de Waal 1994); specified by the user and verified by analysis as possible in Prolog systems such as Ciao (Hermenegildo, Bueno, Puebla, and López 1999); or declared and considered part of the semantics of the program as with strongly typed languages such as Gödel (Hill and Lloyd 1994), Mercury (Somogyi, Henderson, and Conway 1996) and HAL (Demoen, García de la Banda, Harvey, Mariott, and Stuckey 1999).

6.1.1 Groundness Analysis using *Pos*

Program analysis is concerned with the computation of finite approximations of the possibly infinite number of program states that could arise at runtime. In the context of abstract interpretation (Cousot and Cousot 1977; Nielson 1983), approx-

imations are expressed using elements of an abstract domain. Approximations are computed by abstracting a concrete semantics, and the algebraic properties of the abstract domain guarantee that the analysis is terminating and correct. In the framework of Cousot and Cousot (Cousot and Cousot 1977), abstract interpretation is defined in terms of Galois insertions. A Galois insertion is a quadruple $\langle A, \alpha, C, \gamma \rangle$ where $(C, \leq_C)$ and $(A, \leq_A)$ are complete lattices called the *concrete* and *abstract* domain respectively; $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ are monotonic functions called the *abstraction* and *concretisation* function respectively. In general, it is sufficient to specify either α or γ, since in principle a "best possible" α can be determined for a given γ and vice versa. An element $a \in A$ is said to γ-*approximate* (or simply *approximate* if γ is clear from the context) an element $c \in C$, denoted $a \propto_\gamma c$ if and only if $c \leq_C \gamma(a)$. If $\langle A, \alpha, C, \gamma \rangle$ is a Galois insertion, it holds that $\alpha(\gamma(a)) = a$ and $c \leq_C \gamma(\alpha(c))$ for every $a \in A$ and $c \in C$. In other words, an abstract element equals the approximation of its concretisation, and the concretisation of an abstraction of an element approximates the element. The notion of approximation is extended towards monotonic functions (or operations) as follows: given monotonic functions $\mu : C \mapsto C$ and $\mu^A : A \mapsto A$, we say that μ^A γ-approximates μ, denoted $\mu^A \propto_\gamma \mu$, if for all elements $a \in A$ and $c \in C$ holds that $a \propto_\gamma c \Rightarrow \mu^A(a) \propto_\gamma \mu(c)$.

An abstract domain that is often used for (logic) program analysis, is the domain *Pos* that consists of the class of positive Boolean functions together with a bottom element corresponding to the function *false* and ordered by implication. In the analysis of groundness dependencies using *Pos*, a function such as $x \wedge (y \rightarrow z)$ is used to describe a program state in which X is definitely bound to a ground term and in which there exists an instantiation dependency such that whenever Y becomes bound to a ground term then so does Z.

The more formal definition states that a *Pos* function φ describes – or approximates – a substitution θ (a program state) if any set of variables that might become ground by further instantiating θ is a model of φ. Each model m of φ represents the possibility to simultaneously ground the variables of m by further instantiation for a substitution represented by φ. For example, the models of $\varphi = x \wedge (y \rightarrow z)$ are $\{\{X\}, \{X, Z\}, \{X, Y, Z\}\}$. We can see that φ describes $\theta = \{X/a, Y/f(U, V)), Z/g(U)\}$ because under further instantiation X is in all of the models and if Y is in a model (becomes ground) then so is Z. Notice that a substitution such as $\theta = \{X/a\}$ is not described by this set of models. Namely, we can further instantiate θ to $\theta\{Y/a\}$, with the effect that X and Y are ground while Z is not, and $\{X, Y\}$ is not a model of φ. Hence, the abstraction and concretisation functions $\alpha_g : \wp(\mathit{Subst}) \mapsto \mathit{Pos}$ and $\gamma_g : \mathit{Pos} \mapsto \wp(\mathit{Subst})$ are defined as follows. For any substitution θ, let $\mathit{grounds}(\theta)$ be the set of variables of interest that are mapped to a ground term by θ. Then, we have that

$$\alpha_g(\Theta) = \left\{ \mathit{grounds}(\theta\mu) \;\middle|\; \theta \in \Theta, \mu \in \mathit{Subst} \right\}$$
$$\gamma_g(\varphi) = \left\{ \theta \;\middle|\; \alpha_g(\{\theta\}) \subseteq \varphi \right\}$$

It is shown in (Cortesi, Filé, and Winsborough 1991) that γ_g and α_g form a Galois insertion. For more details on the use of *Pos* in abstract interpretation, we refer to (Cortesi, Filé, and Winsborough 1991) and (Marriott and Sndergaard 1993).

An elegant way of implementing a *Pos* based groundness analysis is described in (Codish and Demoen 1995). The main idea of (Codish and Demoen 1995) is to abstract the program that is to be analysed into a propositional logic program that implements the groundness dependencies that exist between the variables of the original program. Consider for example the Prolog program illustrated in Fig. 6.1 which rotates the elements of a list and its corresponding abstraction for groundness dependencies. An atom of the form `iff(`X`,[`$Y_1,\ldots,Y_n$`])` specifies the

Concrete `rotate`	*Abstract* `rotate`

`rotate(`Xs`,`Ys`) :-`
`  append(`As`,`Bs`,`Xs`),`
`  append(`Bs`,`As`,`Ys`).`
	`rotate(`Xs`,`Ys`) :-`
	`  append(`As`,`Bs`,`Xs`),`
	`  append(`Bs`,`As`,`Ys`).`

`append(`Xs`,`Ys`,`Zs`) :-`
$Xs = [\,]$,
$Ys = Zs$.
`append(`Xs`,`Ys`,`Zs`) :-`
$Xs = [X\|Xs1]$,
$Zs = [X\|Zs1]$,
`append(`$Xs1$`,`Ys`,`$Zs1$`).`
	`append(`Xs`,`Ys`,`Zs`) :-`
	`  iff(`Xs`,[])`,
	`  iff(`Ys`,[`Zs`]).`
	`append(`Xs`,`Ys`,`Zs`) :-`
	`  iff(`Xs`,[`X`,`$Xs1$`])`,
	`  iff(`Zs`,[`X`,`$Zs1$`])`,
	`  append(`$Xs1$`,`Ys`,`$Zs1$`).`

Figure 6.1: Corresponding concrete and abstract programs.

formula $x \leftrightarrow (y_1 \wedge \cdots \wedge, y_n)$ with the intended interpretation that X is ground if and only if $Y_1, \ldots, Y_n$ are. Consequently, a unification of the form $X = [Y|Z]$ in the concrete program is replaced by `iff(`$X, [Y, Z]$`)` in the abstract program. Similarly, a unification of the form $X = [\,]$ in the concrete program can simply be replaced by `iff(`X`,[])` which specifies that X is definitely bound to a ground term. The implementation of the `iff/2` predicate is depicted in Fig. 6.2.

```
iff(true,[]).
iff(true,[true|Xs]) :-
  iff(true,Xs).
iff(false,Xs) :-
  member(false,Xs).
```

Figure 6.2: The auxiliary predicate `iff/2`

Now, the result of abstract interpretation of the original program with respect to the domain *Pos* is equivalent with the *concrete* fixed point semantics of the abstract program. For additional details of why this is so, we refer to (Codish and Demoen 1995; Cortesi, Filé, and Winsborough 1996; Marriott and Snder-

gaard 1993). For our purposes, it is sufficient to understand that the problem of performing groundness analysis of the concrete program (on the left part of Figure 6.1) is reduced to the problem of computing the concrete fixed point semantics of the abstract program (in the middle and on the right). The latter is easily computed by simple meta interpreters that implement the T_P-operator such as those described in (Codish 1999; Codish and Demoen 1995). Applying this approach to the abstract rotate program from Figure 6.1 gives the following atoms:

```
rotate(X,X).      append(true,true,true).
                  append(false,Y,false).
                  append(X,false,false).
```

which are interpreted as representing the propositional formula $x_1 \leftrightarrow x_2$ and $(x_1 \wedge x_2) \leftrightarrow x_3$ for the atoms `rotate(`X_1, X_2`)` and `append(`X_1, X_2, X_3`)` respectively. This illustrates a goal-independent analysis. Goal-dependent analyses are supported by applying Magic sets or similar techniques (see e.g. (Codish and Demoen 1995)). The simple, naive scheme described above provides the basis for various more efficient implementations based on semi-naive evaluation, strongly connected components, and other optimisation techniques. For further details and examples of meta-interpreters in Prolog which perform this type of evaluation see (Codish 1999; Codish and Demoen 1995).

6.1.2 About Terms and Types

In what follows, we will reason about the structure of a term, according to the term's type. We assume a standard notion of strong typing as for example in Mercury, which we covered in detail in Section 3.1.1. We define an instantiatedness characterisation of a term by means of an instantiatedness characterisation of several of its subterms. However, for sake of clarity, we first repeat some of the fundamental notations and definitions from Section 3.1.1.

Type Preliminaries

Types, like terms are constructed from (type) variables and (type) symbols. We denote by $\mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$ the set of types constructed from type variables $V_\mathcal{T}$ and type symbols $\Sigma_\mathcal{T}$. A type containing type variables is said to be *polymorphic*, otherwise it is *monomorphic*. Type substitutions are substitutions from type variables to types. The application of a type substitution to a polymorphic type gives a new type which is an instance of the original type.

Function and type symbols are associated with an arity. We write $f/n \in \Sigma$ (or $h/n \in \Sigma_\mathcal{T}$) to specify that f (or h) is an n-ary symbol. We will assume throughout this chapter that the sets of symbols, variables, type symbols and type variables are fixed. Hence we will denote $Term = T(\Sigma, V)$ and $\mathcal{T} = \mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$. Moreover

we will assume that terms and types are constructed from different sets of symbols and different sets of variables, that is $\Sigma \cap \Sigma_{\mathcal{T}} = \emptyset$ and $V \cap V_{\mathcal{T}} = \emptyset$.

We will restrict our attention to *well typed* terms and substitutions. The relation between types and the terms belonging to them is made explicit by a type definition which consists of a finite set of type rules. For each type symbol there is a unique type rule which associates that symbol with a finite set of function symbols.

Definition 6.1 *(repeated from Section 3.1.1) The* type rule *associated to a type constructor $h/n \in \Sigma_{\mathcal{T}}$ is a definition of the form*

$$h(\overline{T}) \rightarrow f_1(\overline{t}_1)\ ;\ \ldots\ ;\ f_k(\overline{t}_k).$$

where $\overline{T}$ is a sequence of n type variables from $V_{\mathcal{T}}$ and for $1 \leq i \leq k$, $f_i/m \in \Sigma$ with $\overline{t}_i$ a sequence of m types from $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ and all of the type variables occurring in the right hand side occur in the left hand side as well. The function symbols $\{f_1, \ldots, f_k\}$ are said to be associated with the type constructor h. A finite set of type rules is called a type definition.

In addition to types defined by the user, we assume also predefined types which are associated with (possibly infinite) sets of terms. For example, the types *int* and *char* may be specified to consist of the integers, respectively characters.

Example 6.1 *Consider the following type rule introduced using the keyword* `type`*:*

```
type  list(T) ---> [] ; [T | list(T)].
```

In the above example, the function symbols [ ] (nil) and | (cons) are associated with the type symbol *list*. The type definition defines also the denotation of each type (the set of terms belonging to the type). For this example, terms of polymorphic type $list(T)$ are either variables, of the form [ ], or of the form $[t_1|t_2]$ with t_1 of type T and t_2 of type $list(T)$. As T is a type variable we cannot determine or commit to its structure under instantiation. For this reason in a program, only a variable can be of type T. Applying the type substitution $\{T/int\}$ on $list(T)$ gives the type $list(int)$. Terms of the form $[t_1|t_2]$ are of type $list(int)$ if t_1 is of type int and t_2 is of type $list(int)$. Type instances can also be polymorphic, e.g. $list(list(T))$.

Characterising the Instantiatedness of a Term

In order to obtain a characterisation of the instantiatedness of a term that is more fine-grained then simply "ground" or "possibly non-ground", we decompose a term in a finite number of (sets of) subterms, and associate an instantiatedness characteristation with each of these. We construct a finite description of the instantiatedness of a term based on the term's type. The next definition specifies

a notion of the *constituents* of a type τ with respect to a type definition ρ. These are the types, in terms of which, a term of type τ may be constructed; or in other words, the possible types of the subterms of a term of type τ.

Definition 6.2 *Let ρ be a type definition. The constituents relation for ρ is the minimal pre-order $\preceq_\rho$: $\mathcal{T} \times \mathcal{T}$ (it is reflexive and transitive) such that if $h(\bar{\tau}) \longrightarrow f_1(\bar{\tau}_1) ; \ldots ; f_k(\bar{\tau}_k)$ is an instance of a type rule in ρ and τ is one of the arguments of $f_i(\bar{\tau}_i)$, then $\tau \preceq_\rho h(\bar{\tau})$. The set of constituents of a type τ is defined as*

$$C_{s\rho}(\tau) = \{\ \tau' \in \mathcal{T} \mid \tau' \preceq_\rho \tau\ \}$$

When ρ is clear from the context we omit it in the notation for $\preceq_\rho$ and $C_{s\rho}$.

Example 6.2 *Given the definition of list/1 from Example 6.1 and the atomic type int, we have:*

$T \preceq T$	$T \preceq list(T)$
$int \preceq int$	$list(T) \preceq list(list(T))$
$list(T) \preceq list(T)$	$T \preceq list(list(T))$
$list(int) \preceq list(int)$	$list(list(T)) \preceq list(list(T))$
$int \preceq list(int)$	$\ldots$

In what follows, we restrict our attention to (polymorphic) types that are not defined in terms of a strict instance of itself. That is, we assume for any type τ that there does not exist a type $\tau' \preceq \tau$ such that $\tau < \tau'$. This is a natural condition and is related to the polymorphism discipline of definitional genericity (Lakshman and Reddy 1991). An important observation is that for any such type τ, the set $C_s(\tau)$ is a finite set of types.

Example 6.3 *Given the definition of list/1 from Example 6.1 and the atomic type int, we have:*

$$C_s(int) = \{int\}$$
$$C_s(T) = \{T\}$$
$$C_s(list(T)) = \{T, list(T)\}$$
$$C_s(list(int)) = \{int, list(int)\}$$

Proposition 6.1 *For any $\tau \in \mathcal{T}$, $C_s(\tau)$ is finite.*

Proof Since ρ is a finite set of type rules, and each type rule introduces only a finite number of constituents, if $C_s(\tau)$ is infinite, it is due to an infinite sequence

$$\tau \succeq \tau_0 \succeq \tau_1 \succeq \tau_2 \succeq \ldots$$

in which $\tau_i \neq \tau_j$ for all i, j. Again, finiteness of ρ implies the existence of an infinite subsequence

$$\tau'_0 \succeq \tau'_1 \succeq \tau'_2 \succeq \ldots$$

which are all instances of a same type τ occurring on the right-hand side of a type rule in ρ (with $\tau'_i \neq \tau'_j$ for all i, j). This contradicts with the fact that $\tau'_i \not< \tau'_j$ for any $i \leq j$.

□

Now, we are in a position to specify an instantiation property on terms depending on their subterms of a given type.

Definition 6.3 *Let τ and τ' be types in a type definition ρ. We say that a term $t : \tau'$ is instantiated with respect to the type τ if there does not exist a well-typed instance $t\sigma : \tau'$ containing a variable of type τ. The predicate $\mu^\rho_\tau(t)$ is true if and only if t is instantiated with respect to type τ defined in ρ.*

Table 6.1 illustrates the values of $\mu^\rho_{list(int)}(s)$ and $\mu^\rho_{int}(s)$ for some terms s of type $list(int)$. For instance, $\mu_{list(int)}([1, X]) = true$ because all $list(int)$-subterms of well-typed instances of $[1, X]$ are instantiated. On the other hand, $\mu_{list(int)}([1|X]) = false$ because the subterm X of type $list(int)$ is a variable. Also $\mu_{int}([1|X]) = false$ as e.g. $[1, Y|Z]$ is an instance with the variable Y of type int.

s	$\mu_{list(int)}(s)$	$\mu_{int}(s)$
$[1, 2]$	*true*	*true*
$[1, X]$	*true*	*false*
$[1\|X]$	*false*	*false*

Table 6.1: Instantiation properties of terms of type $list(int)$.

Note that the classical characterisation of a term being ground can be defined in terms of μ^ρ as follows. If $ground/1$ denotes a predicate such that $ground(t) = true$ if t is a ground term, we have

$$ground(t) \leftrightarrow \bigwedge_{\tau_i \in C_s(\tau)} \mu^\rho_{\tau_i}(t). \tag{6.1}$$

Recall the example depicted in Table 6.1. The term $[1, 2]$ is ground, whereas $[1, X]$ and $[1|X]$ are not.

6.2 $Pos(\mathcal{T})$ in a Monomorphic Setting

In this section, we develop a $Pos(\mathcal{T})$-based groundness analysis for programs employing only monomorphic types. The extensions dealing with polymorphism are presented in Section 6.3. Before presenting the actual analysis, we introduce the following notation. In what follows, we consider definite logic programs in which each atom is either of the form $p(X_1, \ldots, X_n)$, $X = Y$, or $Y = f(X_1, \ldots, X_n)$.

That is, programs are represented in a "flattened" form, much like superhomogeneous form defined for Mercury (see Section 3.1.3): the arguments in an atom of the form $p(X_1, \ldots, X_n)$ are all distinct variables, and a complex unification is broken down into several simpler ones of the form $X = Y$ or $Y = f(Y_1, \ldots, Y_n)$ (with, again, all variables distinct). It is often convenient to denote a variable together with its type. In such cases we write $X : \tau$ to denote that X is a program variable of type τ. For propositional variables we sometimes write x^τ to indicate that x occurs in a *Pos*($\mathcal{T}$) formula about type τ.

6.2.1 Abstraction in *Pos*($\mathcal{T}$)

In *Pos*($\mathcal{T}$) analysis, the truth of propositional variable x^τ expresses the property that program variable X has a value such that no instance contains a subterm of type τ which is a variable. Note that this is a generalisation of *Pos* where x expresses that no instance of the value contains a variable or, in other words, the value is ground. This notion is formalised in the following concretisation function for *Pos*($\mathcal{T}$).

Definition 6.4 *Let V be a set of (typed) variables of interest, τ a type and φ a positive Boolean function over $W = \{X : \tau' \in V \mid \tau \preceq \tau'\}$. The concretisation of φ with respect to τ, denoted $\gamma_\tau(\varphi)$, is the set of well typed substitutions θ such that for all well typed substitutions θ' $\{\ X \in W \mid \mu_\tau(X\theta\theta')\ \}$ is a model of φ.*

A variable $X_i : \tau_i$ is excluded from the domain of φ when τ is not a constituent of τ_i. With s_i the value of such X_i, it is the case that $\mu_\tau(s_i)$ is trivially true, hence instead of excluding X_i, one could state that x_i^τ holds. However, this causes problems for the handling of polymorphism worked out in the next section. Indeed, while τ cannot be a constituent of a polymorphic parameter T, it can be a constituent of an instance of T, hence x_i^τ is not necessarily true for instances.

In a classic groundness analysis, the unification $A = [X|Xs]$ is abstracted as $a \leftrightarrow (x \wedge xs)$. Indeed, we assume that any subterm of the term that A is bound to at runtime could unify with any subterm of the terms bound to X or Xs. In the presence of types we know that A and $[X|Xs]$ are both of the same type (otherwise the program is not well-typed). In addition, we know that all unifications between subterms of (the terms bound to) A and $[X|Xs]$ are between terms corresponding to the same types. So in this example (assuming both terms to be of type $list(int)$), we can specify $a \leftrightarrow xs$ for type $list(int)$ and $a \leftrightarrow (x \wedge xs)$ for type int. It is important to note that the interpretations of the variables in $a \leftrightarrow xs$ and in $a \leftrightarrow (x \wedge xs)$ are different. The former refers to subterms of type $list(int)$ whereas the latter refers to subterms of type int. These intuitions are formalised in the following definitions, that define the abstractions of a unification with respect to a given type.

Definition 6.5 *τ-abstraction I*
Let τ be a type and X, Y program variables of type τ'. The τ-abstraction of $X = Y$ is: if $\tau \notin Constituents(\tau')$ then true else $x^\tau \leftrightarrow y^\tau$.

Definition 6.6 *τ-abstraction II*
Let τ be a type, $X, Y_1, \ldots, Y_n$ variables of types τ_0, $\tau_1, \ldots, \tau_n$ respectively. The τ-abstraction of $X = f(Y_1, \ldots, Y_n)$ is: if $\tau \notin Constituents(\tau_0)$ then true else

$$x^\tau \leftrightarrow \bigwedge_{\substack{1 \leq i \leq n \\ \tau \preceq \tau_i}} y_i^\tau .$$

Note that in the above definition, the τ-abstraction of $X = f(Y_1, \ldots, Y_n)$ reduces to $x \leftrightarrow true$ if τ is not a constituent of any of the types $\tau_1, \ldots, \tau_n$.

Example 6.4 *Let X, Xs and Y denote variables of type $list(int)$ and E a variable of type int. Table 6.4 gives some examples of abstractions.*

unification	$list(int)$-abstraction	int-abstraction	$char$-abstraction
$X = []$	$x \leftrightarrow true$	$x \leftrightarrow true$	$true$
$X = [E\|Xs]$	$x \leftrightarrow xs$	$x \leftrightarrow e \wedge xs$	$true$
$X = Y$	$x \leftrightarrow y$	$x \leftrightarrow y$	$true$

A simple implementation technique, replacing unifications by appropriate calls to `iff/2` (as in Section 6.1.1 for *Pos*) is illustrated for `append/3` in Figure 6.3. We assume that each of `append/3`'s arguments is of type $list(int)$. The least model

Abstraction for type $list(int)$	*Abstraction for type int*

`append_list_int(`Xs`,`Ys`,`Zs`) :-`
`  iff(`Xs`,[]),`
`  iff(`Ys`,[`Zs`]).`
`append_list_int(`Xs`,`Ys`,`Zs`) :-`
`  iff(`Xs`,[`$Xs1$`]),`
`  iff(`Zs`,[`$Zs1$`]),`
`  append_list_int(`$Xs1$`,`Ys`,`$Zs1$`).`
	`append_int(`Xs`,`Ys`,`Zs`) :-`
	`  iff(`Xs`,[]),`
	`  iff(`Ys`,[`Zs`]).`
	`append_int(`Xs`,`Ys`,`Zs`) :-`
	`  iff(`Xs`,[`$X, Xs1$`]),`
	`  iff(`Zs`,[`$X, Zs1$`]),`
	`  append_int(`$Xs1$`,`Ys`,`$Zs1$`).`

Figure 6.3: Abstraction for the types in append

of `append_int/3` expresses the $Pos(int)$-formula $z \leftrightarrow x \wedge y$, i.e. that all subterms of Z of the type int are instantiated iff those of X and Y are. The least model of `append_list_int/3` expresses the $Pos(list(int))$-formula $x \wedge (y \leftrightarrow z)$, i.e. that all subterms of X of type $list(int)$ are instantiated (in other words the backbone of the

list is instantiated when `append/3` succeeds) and that those of Y are instantiated iff those of Z are instantiated. Classical groundness, is obtained by composing the two models, using Equation (6.1) in Section 6.1.2:

```
append(X,Y,Z) :- append_list_int(Xl,Yl,Zl), append_int(Xe,Ye,Ze),
                 iff(X,[Xl,Xe]), iff(Y,[Yl,Ye]), iff(Z,[Zl,Ze]).
```

6.2.2 Correctness

The aim of program analysis is to compute a finite approximation of the program's concrete semantics. This is achieved by replacing the basic operations in the computation of the concrete semantics by corresponding *abstract* operations. To prove correctness of the analysis, we have to prove that these abstract operations approximate the concrete ones. The basic operation involved in computing a definite program's *concrete* semantics is the composition of most general unifiers. The abstract operation corresponding with the composition of most general unifiers is simply $\wedge$-ing the corresponding boolean formulas. Hence, we prove for each kind of unification that $\wedge$-ing its τ-abstraction with a positive boolean formula, say φ, γ_τ-approximates the composition of $\theta \in \gamma_\tau(\varphi)$ and the most general unifier of the unification under θ.

Theorem 6.1 *Correctness.*
Let θ be a substitution on a set of variables V, let φ be a positive Boolean function on V and let $\theta \in \gamma_\tau(\varphi)$. Let $X, Y \in V$ be of type τ' and $\sigma = mgu(X\theta, Y\theta)$. Let φ' be the τ-abstraction of $X = Y$. Then $\theta\sigma \in \gamma_\tau(\varphi \wedge \varphi')$.

Proof First consider the case that $\tau \in C_s(\tau')$, $\gamma_\tau(\varphi \wedge \varphi') = \gamma_\tau(\varphi) \cap \gamma_\tau(\varphi')$ hence it suffices to show that $\theta\sigma \in \gamma_\tau(\varphi)$ and $\theta\sigma \in \gamma_\tau(\varphi')$. The set $\gamma_\tau(\varphi)$ is closed under instantiation hence $\theta\sigma \in \gamma_\tau(\varphi)$. $X\theta\sigma = Y\theta\sigma$ thus $\forall\nu$: $\mu_\tau(X\theta\sigma\nu) = \mu_\tau(Y\theta\sigma\nu)$ and $\theta\sigma \in \gamma_\tau(x \leftrightarrow y) = \gamma_\tau(\varphi')$.

In the other case, $\varphi' = \mathit{true}$ hence $\varphi \wedge \varphi' = \varphi$, moreover, $\gamma_\tau(\varphi)$ is closed under instantiation, so $\theta\sigma \in \gamma_\tau(\varphi \wedge \varphi')$.

□

Theorem 6.2 *Correctness.*
Let θ be a substitution on a set of variables V, let φ be a positive Boolean function on V and let $\theta \in \gamma_\tau(\varphi)$. Let σ be the mgu of $X = f(Y_1, \ldots, Y_n)$ in which all variables are from V and φ' its τ-abstraction. Then $\theta\sigma \in \gamma_\tau(\varphi \wedge \varphi')$.

Proof The set $\gamma_\tau(\varphi)$ is closed under instantiation hence $\theta\sigma \in \gamma_\tau(\varphi)$. $X\theta\sigma = f(Y_1, \ldots, Y_n)\theta\sigma$ hence $\forall\nu$: $\mu_\tau(X\theta\sigma\nu) = \mu_\tau(f(Y_1, \ldots, Y_n)\theta\sigma\nu)$. If $\tau \preceq \tau_i$ then all subterms of type τ occurring in $X\theta\sigma\nu$ occur also in some $Y_i\theta\sigma\nu$ (except $X\theta\sigma\nu$ itself in case $\tau = \tau_0$, but this term is definitely instantiated). Hence all subterms of type τ of $X\theta\sigma\nu$ are instantiated if and only if those

of $Y_i\theta\sigma\nu$ are and $\theta\sigma \in \gamma_\tau(x \leftrightarrow \bigwedge_{\tau \preceq \tau_i} y_i) = \gamma_\tau(\varphi')$.

□

Having shown that the τ-abstraction of unification is correct, we can rely on the results of (Codish and Demoen 1995) for the abstraction of the complete program, for the computation of its least model and for the claim that correct answers to a query $\leftarrow p(X_1, \ldots, X_n)$ belong to the concretisation of the $Pos(\mathcal{T})$-formula of the predicate p/n.

6.3 $Pos(\mathcal{T})$ in a Polymorphic Setting

In this section, we reconsider $Pos(\mathcal{T})$-abstraction in the presence of polymorphism.

6.3.1 $Pos(\mathcal{T})$ in the Presence of Polymorphism

Type polymorphism is an important abstraction tool: a predicate defined with arguments of a polymorphic type can be called with actual arguments of any type that is an instance of the defined type. For example, the `append/3` predicate from Fig. 6.3 usually is defined with respect to a polymorphic type definition, stating that each of its arguments is of type $list(T)$. Abstracting `append/3` for this type definition results in the same abstractions as in Figure 6.3 but with constituent $list(T)$ replacing $list(int)$ and T replacing int.

When abstracting a call to such a predicate, one needs the abstractions with respect to the constituents of the actual types of the call (e.g., $char$ and $list(char)$ in case `append/3` is called with actual types $list(char)$). One possibility to obtain these, is to analyse the definition for each type-instance by which it is called. However, it is much more efficient to analyse the definition once for its given types, and derive the abstractions of a particular call from that result. The need for such an approach is even more urgent when analysing large programs distributed over many modules. It is highly desirable that an analysis does not need the actual code of the predicates it imports (and of the predicates called directly or indirectly by the imported predicates) but only the result of the call independent analysis. Discussions on module based analysis can be found in (Puebla and Hermenegildo 1999; Bueno, de la Banda, Hermenegildo, Marriott, Puebla, and Stuckey 2000) and Sections 4.1.2, 4.2.4 and 4.4.4 in this thesis, the latter discussing modularity in binding-time analysis for Mercury.

The subject of this section is to show how the dependencies of interest (those with respect to the actual constituents) can be obtained using the available dependencies with respect to the constituents in the predicate's definition. Hence, how the dependencies from a call to a polymorphic predicate can be obtained without the need to reanalyse the predicate with respect to the actual types. Consider a predicate `p/2` with both arguments of type $list(list(int))$. The definition of `p/2`

and its abstractions for the constituents *int*, *list*(*int*) and *list*(*list*(*int*)) are depicted in Figure 6.4. Intuitively, it is clear that the constituent *list*(*list*(*int*)) from

Concrete definition	*Abstractions*

`p(X,Y):- append(X,X,Y).`	`p_list_list_int(X,Y) :- append_list_T(X,X,Y).`
	`p_list_int(X,Y) :- append_T(X,X,Y).`
	`p_int(X,Y) :- append_T(X,X,Y).`

Figure 6.4: Concrete and abstract versions of `p/2`

the call to `append/3` corresponds to the constituent *list*(T) in `append/3`'s definition. Hence, instead of computing the *list*(*list*(*int*))-abstraction of `append/3`, we can use the *list*(T)-abstraction of `append/3`. Likewise, the constituents *list*(*int*) and *int* from the call both correspond to the constituent T in the definition, hence also their abstractions.

In what follows, we formally define the relationship between the constituents of the actual types used in a call, and those of the types used in the definition of the called predicate. This relation allows us to redefine the abstraction of a predicate call and to reason about its correctness. To start with, we introduce the notion of a *type assignment*. A type assignment maps program variables to types.

Definition 6.7 *A* type assignment σ *is a mapping from variables to types;* $dom(\sigma)$ *is the set of variables for which the mapping is defined;* $X\sigma$ *denotes the type of* X *under the type assignment* σ*. Given type assignments* σ *and* σ' *over the same domain,* σ' *is an instance of* σ *if , for every* $X \in dom(\sigma)$ *there exists a type substitution* θ *such that* $X\sigma' = X\sigma\theta$*.*

We extend the notion of constituents from a single type to a type assignment in a straightforward way. In a well-typed program, the types of the arguments of a call are always more instantiated than the types of the corresponding arguments in the predicate definition. Hence, if σ' denotes the type assignment associated to the variables of the call, and σ denotes the type assignment associated to the called predicate's head variables, σ' is an instance of σ (modulo the renaming of the variables). Given such σ' instance of σ, the following definition relates the constituents occurring in σ' to the corresponding constituents in σ.

Definition 6.8 *Consider type assignments* σ' *and* σ *such that* σ' *is an instance of* σ*. The* type mapping *between* σ *and* σ' *is the minimal relation* $R^{\sigma}_{\sigma'} : \mathcal{T} \times \mathcal{T}$ *such that:*

- *If* $X/\tau_1 \in \sigma'$ *and* $X/\tau_2 \in \sigma$ *then* $(\tau_1, \tau_2) \in R^{\sigma}_{\sigma'}$*.*
- *If* $(\tau_1, \tau_2) \in R^{\sigma}_{\sigma'}$ *and* $\tau_2 \in V_{\mathcal{T}}$ *(*τ_2 *is a type variable), then* $\forall \tau \in C_s(\tau_1) : (\tau, \tau_2) \in R^{\sigma}_{\sigma'}$*.*

- *If* $(\tau_1, \tau_2) \in R^{\sigma}_{\sigma'}$ *and* $\tau_2 = h(V_1, \ldots, V_n)\theta_2$ *and* $\tau_1 = h(V_1, \ldots, V_n)\theta_1$ *and* $h/n \in \Sigma_{\mathcal{T}}$ *is defined by*

$$h(V_1, \ldots, V_n) \longrightarrow f_1(\tau_{1_1}, \ldots, \tau_{1_{m_1}}) ; \ldots ; f_l(\tau_{l_1}, \ldots, \tau_{l_{m_l}})$$

then, for all τ_{i_j}, $(\tau_{i_j}\theta_1, \tau_{i_j}\theta_2) \in R^{\sigma}_{\sigma'}$.

The type function $\phi^{\sigma}_{\sigma'} : \mathcal{T} \mapsto \wp(\mathcal{T})$ *is defined for constituents of the most instantiated type* σ'*:*

$$\phi^{\sigma}_{\sigma'}(\tau) = \{\tau' | (\tau, \tau') \in R^{\sigma}_{\sigma'}\}.$$

When σ and σ' are obvious from the context, we will simply write $\phi(\tau)$.

Example 6.5 *The type mapping and type function associated to the type assignments* $\sigma = \{X/list(T)\}$ *and* $\sigma' = \{X/list(list(int))\}$ *are*

$$R^{\sigma}_{\sigma'} = \{(list(list(int)), list(T)),\ (list(int), T),\ (int, T)\}$$
$$\phi^{\sigma}_{\sigma'} = \{(list(list(int)), \{list(T)\}),\ (list(int), \{T\}),\ (int, \{T\})\}.$$

If σ' denotes the type assignment associated to the variables of the call, and σ denotes the type assignment associated to the called predicate's head variables, the mapping $\phi^{\sigma}_{\sigma'}$ expresses that the τ-abstraction of a call corresponds to the $\phi^{\sigma}_{\sigma'}(\tau)$-abstraction of the called predicate. (See Fig. 6.4 and $\phi^{\sigma}_{\sigma'}$ from Example 6.5 for an example.) However, in general $\phi^{\sigma}_{\sigma'}(\tau)$ is not a singleton, as a constituent of the call can be mapped to itself *and* to one or more type variables in the polymorphic type definition. Consider the following example:

Example 6.6 *Consider the predicate* `q/4` *and its abstractions for the constituents int and T from its definition in Figure 6.5.*

Concrete	*Abstraction w.r.t. int*	*Abstraction w.r.t.* `T`

`pred q(int,int,T,T).`
`q(X,Y,U,V):- X=Y,U=V.`
`q(X,Y,U,V):- X=0.`
	`q_int(X,Y,U,V):-`
	`  iff(X,[Y]).`
	`q_int(X,Y,U,V):-`
	`  iff(X,[]).`
		`q_T(X,Y,U,V):-`
		`  iff(U,[V]).`
		`q_T(X,Y,U,V).`

Figure 6.5: `q/4` and its *int*- and T-abstractions.

Suppose `q(A,B,C,D)` is a call with A, B, C and D of type *int*. The type substitution on the defined types is $\{T/int\}$, and hence $\phi(int) = \{int, T\}$. A correct *int*-abstraction of the *call* should include both the *int*-abstraction and the T-abstraction from the definition, since in the polymorphic analysis, it was assumed that T does not have *int* as constituent. Hence, the *int*-abstraction of the call becomes

$$q\_int(A, B, C, D) \wedge q\_T(A, B, C, D).$$

The formal definition of the τ-abstraction of a predicate call is then as follows:

Definition 6.9 *Let σ' and σ be the type assignments associated respectively to a predicate call $p(X_1, \ldots, X_n)$ and the head of p's definition. For a type $\tau' \in C_s(\sigma')$, the τ'-abstraction of $p(X_1, \ldots, X_n)$ is defined as*

$$\bigwedge_{\tau \in \phi^{\sigma}_{\sigma'}(\tau')} p\_\tau(X_1, \ldots, X_n)$$

Handling a predicate call in such a way, however, possibly causes some precision loss. Indeed, one can verify that a *monomorphic* analysis of `q(X,Y,U,V)` for the type `q(int,int,int,int)` gives the formula $((x \leftrightarrow y) \wedge (u \leftrightarrow v)) \vee x$ while the conjunction of calls $q\_int(X, Y, U, V), q\_T(X, Y, U, V)$ gives the formula $(x \leftrightarrow y \vee x) \wedge (u \leftrightarrow v \vee \mathit{true})$ which simplifies to $(x \leftrightarrow y \vee x)$ and is less precise. In other words, call-independent analysis only *approximates* the results that can be obtained by a call-dependent (with respect to the type assignment of the call) analysis. We elaborate on this issue in the following section, dealing with the correctness of the analysis in the polymorphic case.

6.3.2 Correctness

In what follows, we deal with the correctness of the polymorphic analysis by relating its result with the results obtainable by a monomorphic analysis. We assume a predicate p that is defined with respect to a (possibly polymorphic) type assignment σ. Furthermore, we assume a type assignment σ' instance of σ, and a constituent $\tau' \in C_s(\sigma')$. We prove the relation between the τ'-abstraction of the predicate and its τ-abstractions for those constituents τ of σ in $\phi^{\sigma}_{\sigma'}(\tau')$. We first investigate this relation for a single unification, showing that its τ'-abstraction equals the $\wedge$ of its corresponding τ-abstractions.

Lemma 6.1 *Let U be a unification and σ and σ' type assignments for the variables of U such that σ' is an instance of σ. For $\tau \in C_s(\sigma)$, let φ^{τ} denote the τ-abstraction of U for the type assignment σ, and similarly for $\tau' \in C_s(\sigma')$, let $\psi^{\tau'}$ denote the τ'-abstraction of U for the type assignment σ'. Then it holds that*

$$\psi^{\tau'} = \bigwedge_{\tau \in \phi^{\sigma}_{\sigma'}(\tau')} \varphi^{\tau}$$

Proof With U of the form $X = \ldots$, $\psi^{\tau'}$ is of the form $x \leftrightarrow y_1 \wedge \ldots \wedge y_n$ with $Y_i : \tau'_{Y_i}$ those variables in the right-hand side of U w.r.t. the type assignment σ' such that $\tau' \in C_s(\tau'_{Y_i})$. Likewise, we denote with τ_{Y_i} the types of these variables in the type assignment σ. Let $\phi(\tau') = \{\tau_1, \ldots, \tau_k\}$. The variables $Y_{i_1}, \ldots, Y_{i_{n_i}}$ such that $\tau_i \in C_s(\tau_{Y_{i_j}})$ are a subset of $\{Y_1, \ldots, Y_n\}$ and $\varphi^{\tau_i} = x \leftrightarrow y_{i_1} \wedge \ldots \wedge y_{i_{n_i}}$. Moreover, by construction of ϕ it holds that $\tau' \in C_s(\tau'_{Y_i})$

if and only if $\exists\, \tau_j \in C_s(\tau_{Y_i})$ and hence $\bigcup_{i=1}^{k}\{Y_{i_1},\ldots,Y_{i_{n_i}}\} = \{Y_1,\ldots,Y_n\}$. Consequently,

$$\begin{array}{rcl}\psi^{\tau'} & = & x \leftrightarrow y_1 \wedge \ldots \wedge y_n \\ & = & (x \leftrightarrow y_{1_1} \wedge \ldots \wedge y_{1_{n_1}}) \wedge \ldots \wedge (x \leftrightarrow y_{k_1} \wedge \ldots \wedge y_{k_{n_k}}) \\ & = & \varphi^{\tau_1} \wedge \ldots \wedge \varphi^{\tau_k}\end{array}$$

which concludes the proof.

□

The following lemma constitutes an important step in the final correctness proof. It states that the τ' abstraction of a predicate is *approximated* by the $\wedge$ of its corresponding τ-abstractions (for $\tau \in \phi^{\sigma}_{\sigma'}(\tau')$), given that the same holds for each of its body atoms.

Lemma 6.2 *Let p be a predicate definition and σ and σ' type assignments for the variables of p such that σ' is an instance of σ. For $\tau \in C_s(\sigma)$, let φ^{τ} and φ_i^{τ} denote, respectively, the τ-abstraction of p and a body atom B_i for the type assignment σ, and similarly for $\tau' \in C_s(\sigma')$, let $\psi^{\tau'}$ and $\psi_i^{\tau'}$ denote respectively the τ'-abstraction of p and the body atom B_i for the type assignment σ'.*

$$\textit{If for each body atom } B_i : \psi_i^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi^{\sigma}_{\sigma'}(\tau')} \varphi_i^{\tau} \textit{ then } \psi^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi^{\sigma}_{\sigma'}(\tau')} \varphi^{\tau}.$$

Proof The definition of p consists of the clauses $C_1,\ldots,C_m$. Each such clause C_i is a conjunction of body atoms $B_{i_1},\ldots,B_{i_{n_i}}$ and hence, if we denote with $\varpi_i^{\tau'}$ the τ'-abstractions of C_i for type assignment σ', it follows from the definition of the τ'-abstraction of a conjunction that

$$\varpi_i^{\tau'} = \psi_{i_1}^{\tau'} \wedge \ldots \wedge \psi_{i_{n_i}}^{\tau'}.$$

Since it holds that $\psi_{i_j}^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi(\tau')} \varphi_{i_j}^{\tau}$ for each j, it follows that

$$\varpi_i^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi(\tau')} \varphi_{i_1}^{\tau} \wedge \ldots \wedge \bigwedge_{\tau \in \phi(\tau')} \varphi_{i_{n_i}}^{\tau}$$

which we can rewrite as

$$\varpi_i^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi(\tau')} (\varphi_{i_1}^{\tau} \wedge \ldots \wedge \varphi_{i_{n_i}}^{\tau}),$$

and, if we denote with χ_i^{τ} the τ-abstraction of the body of clause C_i in type assignment σ, again into

$$\varpi_i^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi(\tau')} \chi_i^{\tau}. \tag{6.2}$$

Now, $\psi^{\tau'}$ is the lub of the *Pos*-formulas for the clauses hence

$$\psi^{\tau'} = \bigvee_{i \in \{1,\ldots m\}} \varpi_i^{\tau'}. \tag{6.3}$$

Combining (6.2) and (6.3) results in the formula

$$\psi^{\tau'} \Rightarrow \bigvee_{i \in \{1,\ldots m\}} \Big(\bigwedge_{\tau \in \phi(\tau')} \chi_i^{\tau} \Big).$$

If we assume that $\phi(\tau') = \{\tau_1, \ldots, \tau_k\}$, the above equals

$$\psi^{\tau'} \Rightarrow (\chi_1^{\tau_1} \wedge \ldots \wedge \chi_1^{\tau_k}) \vee \ldots \vee (\chi_m^{\tau_1} \wedge \ldots \wedge \chi_m^{\tau_k}).$$

Applying distributivity of $\vee$ w.r.t. $\wedge$ results in

$$\psi^{\tau'} \Rightarrow \bigwedge_{\langle \tau_{j_1}, \ldots, \tau_{j_m} \rangle \in S} (\chi_1^{\tau_{j_1}} \vee \ldots \vee \chi_m^{\tau_{j_m}})$$

where S denotes the set of all possible m-tuples of $\{\tau_1, \ldots, \tau_k\}$. Note that the set $\{\langle \tau, \ldots, \tau \rangle \mid \tau \in \phi(\tau')\} \subseteq S$, and hence we can rewrite the above formula into

$$\psi^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi(\tau')} (\chi_1^{\tau} \vee \ldots \vee \chi_m^{\tau}) \wedge \bigwedge_{\langle \tau_{j_1}, \ldots, \tau_{j_m} \rangle \in S \setminus \{\langle \tau, \ldots, \tau \rangle \mid \tau \in \phi(\tau')\}} (\chi_1^{\tau_{j_1}} \vee \ldots \vee \chi_m^{\tau_{j_m}}).$$

Since $\chi_1^{\tau} \vee \ldots \vee \chi_m^{\tau} = \varphi^{\tau}$ and $\psi^{\tau'}$ implies each of the conjuncts separately, we have

$$\psi^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi(\tau')} \varphi^{\tau}$$

which concludes the proof.

□

Before stating the main correctness result about the relation between the τ' and τ-abstractions of a predicate, we derive two corollaries from the above lemma. These allow us, under certain conditions, to prove a stronger correctness result, stating that τ' abstraction of the predicate *equals* – rather than is approximated by – the $\wedge$ of its corresponding τ-abstractions. Or, put otherwise, the result of a call-independent analysis equals the result of a call-dependent analysis. Each of the corollaries treats a particular condition under which such a stronger result can be proven. The first one handles the case that $\phi^{\sigma}_{\sigma'}(\tau') = \{\tau\}$ or, put otherwise, there is a one-to-one correspondence between τ' in σ' and τ in σ.

Corollary 6.1 *With the same notations as in Lemma 6.2, if $\phi(\tau')$ has only one element and*

$$\textit{for each body atom: } \psi_i^{\tau'} = \bigwedge_{\tau \in \phi(\tau')} \varphi_i^{\tau} \textit{ then } \psi^{\tau'} = \bigwedge_{\tau \in \phi(\tau')} \varphi^{\tau}.$$

Proof Immediate from the proof of Lemma 6.2.

□

Note that the set $\phi^{\sigma}_{\sigma'}(\tau')$ is a singleton unless the type instances of two different type variables have a common constituent or the instance of a type variable has a constituent in common with the type of another argument. Even if the latter is the case, this does not necessarily result in a precision loss. Indeed, recall Example 6.6 but without `q/4`'s second clause, the code and relevant abstractions are depicted in Fig. 6.6. Comparing with Example 6.6, we notice that still $\phi^{\sigma}_{\sigma'}(int) = \{int, T\}$ if

Concrete	*Abstraction w.r.t. int*	*Abstraction w.r.t.* `T`
`pred q(int,int,T,T).` `q(X,Y,U,V):- X=Y,U=V.`	`q_int(X,Y,U,V):-` `iff(X,[Y]).`	`q_T(X,Y,U,V):-` `iff(U,[V]).`

Figure 6.6: The reduced predicate `q/4` and its *int*- and T-abstractions.

we consider a type assignment σ' associating all variables with type `int` as before. However, contrary to before, monomorphic analysis of `q(X,Y,U,V)` for the type `p(int,int,int,int)` would now result in the formula $((x \leftrightarrow y) \wedge (u \leftrightarrow v))$ *equaling* the formula obtained from the conjunction $p\_int(X,Y,U,V), p\_T(X,Y,U,V)$. The difference with Example 6.6 consists in the fact that – due to the missing clause – the latter formula no longer introduces the extra disjuncts. This observation is generalised in the following corollary, its condition expressing the second condition under which precision is preserved between a monomorphic analysis and the combination of the results from a polymorphic analysis. The condition expresses that the least upper bound of the τ-abstractions of the individual clauses equals the τ-abstraction of one of the clauses. Indeed, if the latter is the case, upper bounds introduced by the conjunction of different τ-abstractions will not result in less precise formulas than obtained by a monomorphic analysis.

Corollary 6.2 *With the same notations as in Lemma 6.2, if*

$$\psi_i^{\tau'} = \bigwedge_{\tau \in \phi(\tau')} \varphi_i^{\tau},$$

and there exists a clause j such that for all clauses i and for all constituents $\tau \in \phi(\tau')$: $\chi_i^{\tau} \vee \chi_j^{\tau} = \chi_j^{\tau}$ then

$$\psi^{\tau'} = \bigwedge_{\tau \in \phi(\tau')} \varphi^{\tau}.$$

Proof Using the equality instead of the implication in the proof of Lemma 6.2, one obtains:

$$\psi^{\tau'} = (\chi_1^{\tau_1} \wedge \ldots \wedge \chi_1^{\tau_k}) \vee \ldots \vee (\chi_m^{\tau_1} \wedge \ldots \wedge \chi_m^{\tau_k}).$$

Because for all i: $(\chi_i^{\tau_1} \wedge \ldots \wedge \chi_i^{\tau_k}) \vee (\chi_j^{\tau_1} \wedge \ldots \wedge \chi_j^{\tau_k}) = (\chi_j^{\tau_1} \wedge \ldots \wedge \chi_j^{\tau_k})$, it follows that

$$(\chi_1^{\tau_1} \wedge \ldots \wedge \chi_1^{\tau_k}) \vee \ldots \vee (\chi_m^{\tau_1} \wedge \ldots \wedge \chi_m^{\tau_k}) = \chi_j^{\tau_1} \wedge \ldots \wedge \chi_j^{\tau_k}.$$

Hence,

$$\psi^{\tau'} = \chi_j^{\tau_1} \wedge \ldots \wedge \chi_j^{\tau_k} = \bigwedge_{\tau \in \phi(\tau')} \chi_j^{\tau}$$

Using again the assumption about χ_j^{τ}, we obtain:

$$\psi^{\tau'} = \bigwedge_{\tau \in \phi(\tau')} (\chi_1^{\tau} \vee \ldots \vee \chi_m^{\tau}) = \bigwedge_{\tau \in \phi(\tau')} \varphi^{\tau}$$

□

Notice that the condition of Corollary 6.2 could well be often the case in practice because different clauses of a predicate tend to establish the same degree of instantiatedness (or in case of recursion, the *Pos*($\mathcal{T}$)-formula of the base case implies the *Pos*($\mathcal{T}$)-formula of the recursive clause(s)).

The main result, captured in Theorem 6.3, states that analysing a predicate for a type assignment that is an instance of the type assignment it was defined with, possibly results in a more precise result than possible with combining the results from a single polymorphic analysis. Or, put otherwise, call-independent analysis of a polymorphic predicate only approximates the results that can be obtained by a call-dependent analysis (for the type assignment of the call).

Theorem 6.3 *Let p be a predicate definition and σ and σ' type assignments for the variables of p such that σ' is an instance of σ. For $\tau \in C_s(\sigma)$, let φ^{τ} denote the τ-abstraction of p for the type assignment σ, and similarly for $\tau' \in C_s(\sigma')$, let $\psi^{\tau'}$ denote the τ'-abstraction of p for the type assignment σ'. Then it holds that*

$$\psi^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi^{\sigma}_{\sigma'}(\tau')} \varphi^{\tau}.$$

Proof The formula $\psi^{\tau'}$ is computed by a bottom up fixed point operator that, starting from the formula *false* for all predicates, recomputes the *Pos*($\mathcal{T}$)-formula of each predicate until a fixed point is reached. The proof is by induction on the number of iterations. The predicate p is defined by the clauses $C_1, \ldots, C_m$, and a clause C_i is a conjunction of body atoms $B_{i_1}, \ldots, B_{i_{n_i}}$.

- First iteration. Since Lemma 6.1 holds for the unifications, and the predicate calls in the body only introduce the formula *false*, we have for each body atom B_{i_j} that $\psi_{i_j}^{\tau'} = \bigwedge_{\tau \in \phi(\tau')} \varphi_{i_j}^{\tau}$ where $\psi_{i_j}^{\tau'}$ denotes the τ'-abstraction of B_{i_j} for type assignment σ' and $\varphi_{i_j}^{\tau}$ denotes the τ-abstraction of B_{i_j} for type assignment σ. Applying Lemma 6.2 results in
$$\psi^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi(\tau')} \varphi^{\tau}.$$
- Induction step. Assume the formula holds after k iterations. For the abstraction of p as constructed in iteration $k+1$, we have either by Lemma 6.1 (for the unifications) or by the induction hypothesis (for the predicate calls) that for each body atom B_i holds that $\psi_i^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi(\tau')} \varphi_i^{\tau}$ (where $\psi_i^{\tau'}$ denotes the τ'-abstraction of B_i for type assignment σ' and φ_i^{τ} the τ-abstraction of B_i for type assignment σ). Applying Lemma 6.2 provides the result that
$$\psi^{\tau'} \Rightarrow \bigwedge_{\tau \in \phi(\tau')} \varphi^{\tau}.$$
concluding the proof.

□

The following Corollary states the conditions under which precision is preserved by combining the result from a polymorphic analysis.

Corollary 6.3 *With the same notations as in Theorem 6.3, let χ_i^{τ} denote the τ-abstraction of the body of clause i of predicate p's definition. If $\phi(\tau')$ has only one element or there exists a clause j such that for all $\tau \in \phi(\tau')$: $\chi_j^{\tau} = \chi_j^{\tau} \vee \chi_i^{\tau}$, i.e. the τ-abstraction of the body of clause j is the upper bound of the τ-abstraction of all clause bodies, then*
$$\psi^{\tau'} = \bigwedge_{\tau \in \phi_{\sigma'}^{\sigma}(\tau')} \varphi^{\tau}.$$

Proof The proof is analogously to the proof of Theorem 6.3, using Corollaries 6.1 and 6.2 instead of Lemma 6.2.

□

6.4 Discussion and Further Work

At first sight, some similarity exists between the domain $\{static, dynamic\}$ used in binding-time analysis of functional and logic languages and the *Pos* domain.

In chapters 3 and 4, in which we develop a binding-time analysis for the logic programming language Mercury, binding-times are expressed at the level of type nodes in the type graph, resulting in a precise binding-time analysis capable of dealing with partially instantiated structures. This motivated us to explore a similar use of *Pos* for typed logic programs. Rather than associating a single property – represented by a single boolean variable – with each program variable as in traditional *Pos*-based analysis, we associate a number of properties – and hence a number of boolean variables – with each program variable. The set of boolean variables associated to a program variable is finite and constructed from the variable's type. Hence the name $Pos(\mathcal{T})$ for the newly introduced abstract domain. The $Pos(\mathcal{T})$ domain allows to represent more complex abstract values by associating a boolean value (or a property) with a class of subterms of a variable's value. Associating the "ground" (or better "bound") versus "possibly unbound" characteristic to the individual boolean variables in $Pos(\mathcal{T})$, one obtains an abstract domain that approximates a value by its degree of instantiatedness, rather than by "ground" versus "possibly nonground". The obtained analysis is an *instantiatedness analysis* rather than the traditional groundness analysis and approximates values in more detail.

A possible source of precision loss in our analysis, lies in the fact that all subterms of a term that are of the same (sub)type are considered equivalent with regard to the property of interest. Consider for example the polymorphic type definition of a pair where the two elements are of a same type, and a simple predicate that unifies two such pairs.

```
type pair(T) ---> f(T,T).
pred unify(pair(T), pair(T)).
unify(P,Q):- P = Q.
```

When analysing the predicate `unify` with respect to the constituent T, the analysis has no way of expressing that e.g. the first element of P is *ground* while the second is not. Likewise, the T-abstraction of the unification $P = Q$ is the formula $P \leftrightarrow Q$. To obtain more precision, one could distinguish between the two "occurrences" of T in the type $pair(T)$, and consider them separately in the analysis of `unify` with respect to the constituent T. If we denote with $P1$ and $P2$ the first, respectively second occurrence of T in P (and similarly for Q), the abstract code becomes:

```
unify_pair_T(P1,P2):- iff(P1,[P2]).  %for pair(T)
unify_T(P1,P2,Q1,Q2):- iff(P1,[Q1]), iff(P2,[Q2]). %for T
```

Translating the unification $P = Q$ into $P1 \leftrightarrow Q1 \wedge P2 \leftrightarrow Q2$ involves a mapping between the corresponding occurrences of constituents in the participating variables. Also the translation of a predicate call involves a mapping between the occurrences of the constituents used in the call to the occurrences of the constituents used in the definition of the called predicate. Consider the following definition:

```
pred q(int,int).
q(E1,E2) :- P=f(E1,E2), unify(P,Q).
```

In the above code P and Q are of the type $pair(int)$. Since this type is an instance of $pair(T)$ having two occurrences of T, we distinguish between the two occurrences of int in $pair(int)$ (denoted by $P1$ and $Q1$ and $P2$ and $Q2$ respectively). Mapping the first occurrence of constituent `int` to the first occurrence of constituent `T`, and the second occurrence of `int` to the second occurrence of `T`, we obtain the following abstract code:

```
q_int(E1,E2):- iff(P1,[E1]), iff(P2,[E2]), unify_T(P1,P2,Q1,Q2).
```

This extension integrates "pattern" like information in the $Pos(\mathcal{T})$ analysis. It can be constructed systematically as the first and second occurrence of T correspond to distinct type nodes in the type graph of $pair(T)$. It is similar to the distinctions made on the position of a (sub)type in a type graph in the binding time analysis developed in Chapters 3 and 4.

As indicated at the end of Section 6.2, the $Pos(\mathcal{T})$ formulas can be used to derive the Pos formulas, contributing to a more precise groundness analysis. However, the $Pos(\mathcal{T})$ formulas by themselves provide valuable information. Knowing that all subterms of type τ are instantiated allows to optimise compilation. Also termination analysis can exploit such information. Termination analysis for logic programs depends on two types of analysis: (a) a size analysis — to determine that some measure of the data decreases in size over computation; and (b) an instantiation analysis — to determine that the measure which is decreasing is well-founded. We expect that the combination of types and Pos will enable more sophisticated measures to be used in these kinds of analysis. A possible topic for further research is the integration of $Pos(\mathcal{T})$ in the binding-time analysis needed for off-line partial deduction of logic programs. We expect that the use of $Pos(\mathcal{T})$ will lead to a more elegant, more refined and more powerful binding time analysis than the one described in (Bruynooghe, Leuschel, and Sagonas 1998).

Similar work can be found by other authors. Smaus, Hill, and King (2000) define an abstract domain for mode analysis of polymorphically typed logic programs. The domain is based on a polymorphic type relation between a term and its subterms (effectively constituting a type graph). Conditions on the types under consideration (the so-called simple range condition and reflexive condition) guarantee finiteness of the resulting domain over which an abstract unification procedure is defined. The domain is not in particular targeted towards groundness analysis, but the notion of groundness is nevertheless captured by the abstraction. Ridoux and Boizumault (Ridoux, Boizumault, and Malésieux 1999) explicitly combine type information with the abstract domain Pos to obtain groundness (or better instantiatedness) analysis. The program is abstractly compiled and the result of analysis is its concrete semantics over the abstract domain. Finiteness of the analysis is ensured by imposing conditions on the program that is analysed (the condi-

tions are related to the so-called head-condition, implying that all occurrences in clause heads of a predicate have equivalent types, and that all other occurrences have types that are either equivalent to or more instantiated than the types of head occurrences). Our approach differs in that we, like in (Smaus, Hill, and King 2000), impose a condition on the types – rather than the programs. Moreover, we abstractly compile the program once for each node in the relevant type graphs. As such, the (type) structure of the abstract domain is handled during the abstraction, and the concrete semantics of such an abstract program is computed over the standard domain *true*/*false* and no complex values based on (labellings of) type graphs need to be handled during the analysis. This relates our approach strongly with the approach of Lagoon and Stuckey (2001). Also in their work, type information is incorporated in a *Pos*-based analysis, by labelling the nonterminal nodes in the regular tree grammars representing type information and abstracting the program into a set logic program. In this way, the type descriptions are incorporated in the abstract program, while – contrary to our approach – only a single abstraction of a predicate is created. The resulting scheme of (Lagoon and Stuckey 2001) can handle groundness and sharing analysis, but does not deal with parametric polymorphism, which our approach does.

Different incarnations of *Pos* have been developed before, for example (Codish and Demoen 1994); however, that analysis was for untyped programs and each incarnation was developed in a somewhat ad-hoc fashion. In particular, the abstraction of the unifications was done on a case by case basis. Also the model based analysis of (Gallagher, Boulanger, and Saglam 1995) is somewhat related. The authors show how models of the program, based on different pre-interpretations can express different kinds of program properties. But again, the choice of a pre-interpretation is done case by case. It is likely that our work could be formalised in that framework by formulating a type-based pre-interpretation such that the models based on a pre-interpretation systematically derived from type definition τ correspond to the $Pos(\tau)$ formula for the same predicate.

Part II

On-line Program Specialisation Revisited

Chapter 7

Specialising Vanilla: Towards the Limits of Top Down Partial Evaluation

The meta-Turing test counts a thing as intelligent
if it seeks to devise and apply Turing tests
to objects of its own creation.

– Lew Mammel, Jr.

In this chapter, we investigate the partial deduction of a particular class of meta interpreters, showing some limitations of current state-of-the-art on-line partial deduction techniques. We propose a possible extension to a standard unfolding rule, and show that it achieves good specialisation results on the particular class of interpreters, but is too weak to be applicable for meta interpreters in general.

7.1 Meta Interpreters in Logic Programming

Writing meta interpreters is a well-known technique to enhance the expressive power of logic programs. Essentially, a meta interpreter is a program that handles another program as data. Following standard terminology, we refer to the program that is handled as data by the term *object program*, and to the program that is handling (or "interpreting") the object program, by the term *meta program*. Applications of meta programs include interpreters, compilers and various program

analysis tools (Hill and Gallagher 1998). Although meta programs and object programs can in general be written in a variety of languages, a particularly interesting case is when both the meta and object program are written in the same language (Barklund 1995). This technique is often employed in logic programming, in order to alter the execution mechanism of the language, or to add extra functionality to (the execution of) a program. Examples are the ability to control the depth of the SLD-tree, the implementation of an alternative search strategy like breadth-first or bottom-up search (Codish 1999) or to provide information to the user about the refutation built by the evaluation process (returning the proof tree, for example) (Sterling and Shapiro 1986). Meta interpreters are often used to build prototype implementations of program analysis tools like groundness or termination analysers (Codish and Demoen 1995).

An important issue when writing meta interpreters is the representation of the object program. In a logic programming setting, one basically has two options. The most general one is the so-called *ground representation*. The object program is represented by a ground term, that can be handled by the meta program as any regular term can. Even within the ground representation, one has several options when considering an appropriate representation. One – rather extreme – possibility is to represent object program entities as "flat" terms, for example using strings. A flat representation has several disadvantages, the most notably probably being the inefficiency it introduces. For example, implementing a unification algorithm in the meta program that is capable of unifying terms in the object program represented as strings may require exponential time as well as exponential space (Barklund 1995). Therefore, object program entities are more frequently represented as structured terms, in which the variables are represented by a ground term. An advantage of using the ground representation is its generality. Being essentially a regular (ground) data term, the meta program can handle the object program completely in a declarative way, not requiring any extra-logical builtins. Consequently, the object program is unrelated to the meta program or the latter's execution, and the meta program has full control over the execution mechanism under which it executes the object program. This implies, however, that the meta program in general must implement a complete execution mechanism (unification and backtracking for example), usually resulting in some performance loss.

Alternatively, one could use the so-called *nonground representation*. Using this representation, a term of the object program is represented by a term in the meta program having the same syntax. Variables at the object level are represented by variables at the meta level. The nonground representation is often used when writing meta interpreters in Prolog. A classic interpreter for a subset of Prolog can be written in Prolog as follows, and is often denoted by the "vanilla" meta interpreter.

Example 7.1 *Consider the classic three-line Vanilla meta interpreter in Prolog, depicted in the left-hand side of Fig. 7.1, and a predicate implementing the well-*

known `append/3` *predicate as object program depicted on the right-hand side of the figure.*

The Vanilla meta interpreter	`append/3`
`solve(true).`	`cl(app([],L,L),true).`
`solve((A,B)):- solve(A), solve(B).`	`cl(app([E|Es],Y,[E|R]),`
`solve(A):- cl(A,B), solve(B).`	`app(Es,Y,R)).`

Figure 7.1: The Vanilla meta interpreter and a sample object program.

The major advantage of this representation is that SLD-resolution of the object program is immediate, while performing unification and backtracking on the meta program (Barklund 1995). The answer substitutions of the object program are available through the answer substitutions of the meta program. Consequently, meta interpreters using the nonground representation usually require less code to be written (no unification algorithm must be written, for example) and are usually more efficient than their ground representation counterparts, precisely because they employ the execution mechanism that is already present in the language. In spite of these advantages, it is recognised that employing the nonground representation presents a high risk of introducing subtle errors by unintentionally instantiating metavariables that actually represent object variables (Barklund 1995). Moreover, some applications of meta programming, for example partial evaluators and integrity checkers, need to be written using the ground representation in order to be declarative (Leuschel 1997; Leuschel and De Schreye 1998). Nevertheless, the nonground representation is usually favored for rapidly prototyping program analysis tools for Prolog in Prolog like for example in (Codish 1999; Codish and Taboch 1999). In (Sterling and Beer 1989), Sterling and Beer propose a methodology to build knowledge systems by repeatedly enhancing a simple vanilla meta interpreter with extra functionality. Each such enhancement (referred to by the notion of a "flavor" in (Sterling and Beer 1989)) can be written as a separate meta interpreter, but they all can be combined into a single meta interpreter that provides all the enhancements.

It is noteworthy that some logic programming languages, examples being Gödel (Hill and Lloyd 1994) and Mercury (Somogyi, Henderson, and Conway 1996), do not directly support the non-ground representation. Both languages provide, however, mechanisms and tools to deal with object programs written in Gödel, respectively Mercury, using a ground representation. A detailed comparison between the ground- and nonground representation is beyond the scope of this thesis. We refer the interested reader to (Barklund 1995; Hill and Gallagher 1998).

Even with an optimally tuned representation, the execution of an object program through a meta interpreter usually is less efficient (typically an order of magnitude slower) than the execution of an equivalent program in the meta language (Sterling and Beer 1989). Since a single object program is most likely to be

evaluated with respect to a number of object level queries, a natural approach to solving the efficiency problem consists in specialising the interpreter with respect to a given object program (Safra and Shapiro 1986; Gallagher 1986; Takeuchi and Furukawa 1986). As such, the overhead could be removed by performing interpretation of the object program (essentially *parsing* the object program) during specialisation. The idea is that the residual program implements the functionality of the object program, whereas it can be directly executed, without requiring the meta interpreter. Partial deduction of a meta interpreter with respect to an object program has been considered by different authors (among others in (Safra and Shapiro 1986; Gallagher 1986; Takeuchi and Furukawa 1986)) but turns out to be a non-trivial task, requiring knowledge about the interpreter to be explicitly stated by the programmer. For example, Sterling and Beer (Sterling and Beer 1989) declare a number of rules that are sufficient to specialise a certain class of meta interpreters. In (Lakhotia and Sterling 1990), Lakhotia and Sterling follow a somewhat more general approach, defining – independent of a particular interpreter – *what* knowledge should be gathered about an interpreter that is to be specialised and how this knowledge should be coded in unfolding rules. Nevertheless, both techniques result in an ad-hoc specialiser that is constructed by hand.

In what follows, we investigate specialisation of vanilla-like meta interpreters using a *general* and *automatic* top-down partial deduction system. To that extent, we instantiate the general framework of on-line partial deduction from Chapter 2 with respect to concrete strategies for local and global control, that provide good results when partially deducing logic programs in general.

7.2 Specialising Vanilla

In this section, we first formally state the so-called "parsing problem" (Lakhotia and Sterling 1990; Martens 1994), i.e. the problem of removing the overhead associated to the interpretation of the object program by partial deduction. Next, we define a concrete instance of the general framework for automatic, on-line partial deduction and discuss its effect – with respect to the parsing problem – when specialising the simple Vanilla meta interpreter as well as an extended interpreter.

7.2.1 Removing the Parsing Overhead

In order to allow a formal treatment of the Vanilla meta interpreter, let us introduce the following notions from (Martens 1994; Martens and De Schreye 1995).

Definition 7.1 *Suppose $\mathcal{L}$ is a first order language and* Π *its finite set of predicate symbols. We define $\mathcal{F}_\Pi$ to be a* functorisation *of* Π *if and only if $\mathcal{F}_\Pi$ is a set of*

```
(1) solve(true).
(2) solve((A,B)):- solve(A), solve(B).
(3) solve(A):- cl(A,B), solve(B).
(4) cl(app([],L,L),true).
(5) cl(app([E|Es],Y,[E|R]), app(Es,Y,R))
```

Figure 7.2: The vanilla program V_P with P the `append/3` program

function symbols such that there is a one-to-one correspondence between elements of Π *and* $\mathcal{F}_\Pi$ *and the arity of the corresponding elements is equal.*

Following the notation of (Martens 1994), we have the following. Whenever A is an atom in a first order language $\mathcal{L}$ with predicate symbol set Π and a functorisation $\mathcal{F}_\Pi$ of Π is given, A' denotes the term produced by replacing in A the predicate symbol by its corresponding element in $\mathcal{F}_\Pi$. In what follows, let us denote by V the three-line Vanilla meta interpreter as depicted in Fig 7.1 in the previous section. Then we define the *Vanilla meta program associated to* P as follows:

Definition 7.2 *Let P be a definite program. Then V_P, the Vanilla meta program associated to P, is defined as $V \cup F_P$ where*

$$F_P = \begin{array}{c} \left\{\ cl(A', true) \ \middle|\ A \leftarrow\in P\ \right\} \\ \cup \\ \left\{\ cl(A', (B'_1, (\ldots, (B'_n, true))\ldots)) \ \middle|\ A \leftarrow B_1, \ldots, B_n \in P\ \right\} \end{array}$$

Moreover, for reasons of clarity we assume that the predicate and function symbols of a program P under consideration do not interfere with the predicate and function symbols of V, more formally: $\Sigma_P \cap \{solve/1, cl/2, true/0,\ ,/2\} = \emptyset$, $\Pi_P \cap \{solve/1, cl/2, true/0,\ ,/2\} = \emptyset$ and $\Sigma_P \cap \mathcal{F}_{\Pi_P} = \emptyset$. Apart from $\Pi_P \cap \{true/0,\ ,/2\} = \emptyset$, these requirements are not strictly necessary for correctness. They do, however, facilitate the exposition of the control strategy we develop in this section.

Example 7.2 *We reconsider Example 7.1 from the previous section. If P denotes the program comprising the definition of the* `append/3` *predicate, that is*

```
app([],L,L).
app([E|Es],Y,[E|R]):- app(Es,Y,R).
```

then the Vanilla meta program associated to P, V_P is the program defined in Fig. 7.2. The clauses in V_P are numbered for later reference.

In what follows, we call "meta structure" all program structure that may be present in a vanilla meta program associated to a program P, but which can never appear in the predicates of P. In other words, meta structure is the structure that is used solely by the meta interpreter and which has nothing to do with the computations that P performs. More formally:

Definition 7.3 *Assume a definite program P is given. We say that a term t contains* meta structure *(with respect to P) if and only if t contains a functor $f \in \Sigma_V \cup \mathcal{F}_{\Pi_P}$. Likewise, a substitution θ is said to contain meta structure if and only if it contains an element (X, t) such that t contains meta structure. Otherwise the term, respectively substitution is called* meta structure free.

Recall that the basic motivation to partially deduce a meta interpreter is to remove the interpretation overhead introduced by the interpreter. In the context of the vanilla interpreter, this corresponds with performing all the unification operations that involve meta structure during partial deduction. Note that when partially deducing more involved meta interpreters, in particular interpreters that communicate information about the object program's behaviour to the user, it may be necessary to retain some of the meta structure in the specialised program. This, however, is not necessary when specialising the vanilla meta interpreter, and in order to characterise a partial deduction that does not contain any unifications involving meta structure, we introduce the following definition, stating when a partial deduction of a meta program is meta structure free.

Definition 7.4 *Let P be a definite program, Q a definite query and $\mathcal{A}$ an independent set of atoms. Let V'_P be a partial deduction of $\mathcal{A}$ in V_P. We say that V'_P is meta structure free if and only if for each atom B in $V'_P \cup \{Q\}$, there exist an atom $A \in \mathcal{A}$ such that $B = A\theta$ and θ is meta structure free with respect to P.*

Note that the condition under which V'_P is meta structure free is a refinement of the closedness condition (see Definition 2.13). Also note that Definition 7.4 does not require the absence of meta structure in the residual program. It does, however, requires the absence of meta structure in the substitutions between the atoms of the residual program combined with the query and the atoms of $\mathcal{A}$. The intuition is as follows: if the latter substitutions are meta structure free, the arguments of the body atoms of the $\mathcal{A}$-*filtered* partial deduction (see Definition 2.14) will also be meta structure free. Indeed, let $V_{P_f} \cup \{Q_f\}$ denote the $\mathcal{A}$-filtering of $V'_P \cup \{Q\}$. Any call occurring during a derivation for $V_{P_f} \cup \{Q_f\}$ is an instance of a body atom of V_{P_f}, and since no atom from Q_f contains meta structure, no terms containing meta structure are propagated from the query to the program clauses, and hence none of the calls spawned by the clause contain meta structure.

Example 7.3 *Let P denote the* `append/3` *program as in Example 7.2. Let V'_P be the following program:*

```
solve(app([],L,L)).
solve(app([E|Es],Y,[Z|R])):- solve(app(Es,Y,R)).
```

then V'_P is a partial deduction of $\mathcal{A} = \{solve(app(X, Y, Z))\}$ in V_P and V'_P is meta structure free with respect to P. Indeed, for any query of which the atoms are instances of $solve(app(X, Y, Z))$, e.g. $Q = solve(app([a, b], [c, d], Z))$, the $\mathcal{A}$-filtering

of $V'_P \cup \{Q\}$ *is the program*

```
solve_app([],L,L).
solve_app([E|Es],Y,[Z|R]):- solve_app(Es,Y,R).
```

and the query `solve_app([a,b],[c,d],Z).`

Note that the removal of meta structure is effectively done by the structure filtering transformation, not by the partial deduction process an sich. Nevertheless, it is the partial deduction process that transforms the vanilla meta program into a program from which the meta structure *can* be removed by the structure filtering. The removal of (meta) structure by filtering has also been referred to as "pushing down meta arguments" (Sterling and Beer 1989) or by its abbreviation "PDMA" (Owen 1989).

In this paragraph, we have defined a condition on the partial deduction of a vanilla meta program that is sufficient to guarantee that the residual program (after a filtering transformation) does not perform any unifications involving meta structure. Hence, we can rephrase the *parsing problem* as follows: for a vanilla program V_P and query Q, construct – if possible – a suitable set of atoms $\mathcal{A}$ and a partial deduction V'_P of $\mathcal{A}$ in V_P such that V'_P is meta structure free according to Definition 7.4. This is the subject of the following paragraph, that deals with a concrete control strategy.

7.2.2 A Sophisticated Control Strategy

Recall from Section 2.2.2 that controlling the partial deduction process is performed at two levels. The so-called *local* control (see Section 2.2.2) comprises the definition of an *unfolding rule* which specialises a given atom by building a finite partial SLD-tree for it. The *global control* (see Section 2.2.2) on the other hand, controls the set of atoms that are specialised by performing appropriate abstractions among them. In this section, we instantiate the generic partial deduction algorithm from Section 2.2.2 with concretisations of the local- and global control components that are considered to provide good control of the partial deduction process in general.

In what follows, we devise a concrete partial deduction algorithm in which the termination control at both levels is based upon a well-quasi order relation, namely the homeomorphic embedding relation, $\trianglelefteq$ (see Section 2.2.2). The homeomorphic embedding relation is used as a means of ensuring termination in a lot of recent work (Sørensen and Glück 1995; Leuschel, Martens, and De Schreye 1998; Glück, Jørgensen, Martens, and Sørensen 1996; Jørgensen, Leuschel, and Martens 1997; Alpuente, Falschi, Julián, and Vidal 1997; Lafave and Gallagher 1998; Albert, Alpuente, Falaschi, and Julian 1998; Alpuente, Falaschi, and Vidal 1998; De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999). Reasons for its popularity (Leuschel and Bruynooghe 2001) are both the simplicity of the

approach and its power. See (Leuschel 1999; Leuschel 1998a) for an assessment of the power of well-quasi orders (in particular the homeomorphic embedding relation) and its use for termination control of partial deduction. For a survey of other means of controlling the partial deduction process, we refer to (Leuschel and Bruynooghe 2001).

We start by defining an unfolding rule, $U_{\trianglelefteq}$, that implements local control using the homeomorphic embedding relation. This unfolding rule can be found in (Glück, Jørgensen, Martens, and Sørensen 1996), and was adapted from (Sørensen and Glück 1995; Leuschel and Martens 1996). We start by the following definition, that defines a relation between the atoms in an SLD-tree:

Definition 7.5 *Given an SLD-tree τ. Let $\leftarrow A_1, \ldots, A_k, \ldots, A_n$ be a goal in τ, A_k the selected atom, and $H \leftarrow B_1, \ldots, B_m$ a clause in P such that θ is the most general unifier of A_k and H. Then, in the derived goal*

$$\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n)\theta$$

we have the following:

- *for each $i \in \{1, \ldots, m\}$, $B_i\theta$* descends from *A_k, and*
- *for each $i \in \{1, \ldots, k-1\} \cup \{k+1, \ldots, n\}$, $A_i\theta$* descends from *A_i.*

The descends *relation is transitive: if A' descends from A and A'' descends from A', then A'' also descends from A.*

Since the homeomorphic embedding relation is a well-quasi order relation, finiteness of the constructed tree is guaranteed if the unfolding rule does not create a branch in which a (selected) atom A descends from a (selected) atom A' such that $A' \trianglelefteq A$. This is formally defined by the following definitions (from (Glück, Jørgensen, Martens, and Sørensen 1996)):

Definition 7.6 *An atom A in a goal at the leaf of an SLD-tree is* selectable *unless it descends from a selected atom A' with $A' \trianglelefteq A$.*

Definition 7.7 *The* unfolding rule *$U_{\trianglelefteq}$ unfolds the leftmost selectable atom in each goal of the SLD-tree under construction. If no atom is selectable, no further unfolding is performed.*

The intuition behind the unfolding rule $U_{\trianglelefteq}$ is that every atom in a goal of the SLD-tree under construction will be unfolded (in left-to-right order), unless it descends from an atom which it embeds.

Example 7.4 *Reconsider the vanilla meta program V_P of Fig. 7.2. The partial SLD-tree associated to the atom* `solve(app(X,Y,Z))` *by the unfolding rule $U_{\trianglelefteq}$ is the tree depicted in Fig. 7.3. In each goal of the tree, the leftmost selectable atom*

Figure 7.3: An SLD-tree created by $U_{\trianglelefteq}$.

(according to Definition 7.6) is underlined. Branches in the tree are decorated with the substitutions created by the SLD resolution and a number denoting the clause in V_P that was used for the unfolding. Failing branches are depicted by a dashed line. The atom `solve(app(Es,Y,R))` *is not selectable since it descends from the root atom* `solve(app(X,Y,Z))` *and* `solve(app(X,Y,Z))`$\trianglelefteq$`solve(app(Es,Y,R))`.

Concerning the global control component, we adopt the *characteristic atom* approach (Leuschel 1995a; Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998). Recall that a characteristic tree is an abstraction of an SLD-tree that registers which atoms have been selected for unfolding and which clauses were used for resolution when building the SLD-tree (Leuschel 1995a). As such, a characteristic tree provides a characterisation of the computation- or specialisation behaviour of the atom in the root of the corresponding SLD-tree. The combination of an atom and a characteristic tree for the atom is called a characteristic atom:

Definition 7.8 *(From (Leuschel and Martens 1996)) A* characteristic atom *is a couple $(A, \overline{\tau})$ where A is an atom and $\overline{\tau}$ the characteristic tree associated to a partial SLD-tree τ for A.*

The general idea is then to label the global tree with *characteristic atoms* and to define a well-quasi order relation and abstraction operator on characteristic atoms instead of atoms. Consequently, one is able to take the specialisation behaviour of the atoms into account upon deciding when to abstract and retain some of this behaviour in the abstraction. Such a computation-based approach avoids over-generalisation in some cases, and ensures a much better control of polyvariance than purely syntax-based global control techniques (Leuschel and Bruynooghe 2001; Leuschel 1995a; Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998).

Next, we extend the homeomorphic embedding relation to characteristic atoms. This can be achieved by defining a term representation of a characteristic tree and subsequently using $\trianglelefteq$ with this term representation (Leuschel and Martens 1996): To that extent, a total mapping, denoted by $\lceil . \rceil$ from characteristic trees to terms (expressible in some finite alphabet) can be defined and informally, $\lceil \overline{\tau}_{A_1} \rceil \trianglelefteq \lceil \overline{\tau}_{A_2} \rceil$ means that $\overline{\tau}_{A_1}$ can be obtained from $\overline{\tau}_{A_2}$ by "wiping out" some sub-branches of $\overline{\tau}_{A_2}$. Also, we say that a characteristic tree $\overline{\tau}_{A_1}$ is more general than a characteristic tree $\overline{\tau}_{A_2}$, denoted by $\overline{\tau}_{A_1} \leq \overline{\tau}_{A_2}$ if and only if $\overline{\tau}_{A_2}$ can be obtained by attaching subtrees to the leaves of $\overline{\tau}_{A_1}$. The precise definitions of the above concepts are less important in our present setting and we refer the interested reader to (Leuschel 1995a; Leuschel, Martens, and De Schreye 1998). We do, however, consider the following definition that states that a characteristic atom is homeomorphically embedded in another characteristic atom if the atom parts as well as the term representations of their associated characteristic trees are homeomorphically embedded.

Definition 7.9 *Let* $(A_1, \overline{\tau}_{A_1})$, $(A_2, \overline{\tau}_{A_2})$ *be characteristic atoms. We say that* $(A_2, \overline{\tau}_{A_2})$ homeomorphically embeds $(A_1, \overline{\tau}_{A_1})$, *denoted by* $(A_1, \overline{\tau}_{A_1}) \trianglelefteq (A_2, \overline{\tau}_{A_2})$ *if and only if* $A_1 \trianglelefteq A_2$ *and* $\lceil \overline{\tau}_{A_1} \rceil \trianglelefteq \lceil \overline{\tau}_{A_2} \rceil$.

Given a global tree γ in which the nodes are labelled with characteristic atoms, the generic partial deduction algorithm of Section 2.2.2 is instantiated with the unfolding rule $U_{\trianglelefteq}$ and the following instantiations for global control component:

- *covered*$((A, \overline{\tau_A}), \gamma)$ returns *true* if there exists a node labelled $(B, \overline{\tau_B})$ in γ and $B \leq A$ and $\overline{\tau}_B \leq \overline{\tau}_A$.
- *whistle*$((A, \overline{\tau}_A), \gamma)$ returns $(B, \overline{\tau}_B)$ if the node $(B, \overline{\tau}_B)$ is an ancestor of $(A, \overline{\tau}_A)$ in γ such that $(B, \overline{\tau}_B) \trianglelefteq (A, \overline{\tau}_A)$. Otherwise, it returns *fail*.
- *abstract*$((A, \overline{\tau}_A), (B, \overline{\tau}_B))$ is defined as the node $(msg(\{A, B\}), \overline{\tau})$, where $\overline{\tau}$ denotes the characteristic tree abstracting the SLD-tree constructed for $msg(\{A, B\})$ by the unfolding rule under consideration.

The resulting algorithm is a typical algorithm as found in a lot of recent work on partial deduction. Details on its termination behaviour and correctness results can be found in (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998). A wealth of experiments using the partial deduction system ECCE (Leuschel 1996) that implements, among others, this control scheme, can be found in (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998; Glück, Jørgensen, Martens, and Sørensen 1996; Leuschel 1997). Note that the characteristic atom resulting from an abstraction operation can be refined by imposing the greatest common initial subtree of the involved characteristic trees upon the generalisation. We return to this issue in Section 7.4. In what follows, we denote with $PD_{\trianglelefteq}(P, Q)$ the set of *atoms* resulting from running the concrete algorithm with respect to a

program P and a partial deduction query Q. Also, if we consider *the* partial deduction of $PD_{\trianglelefteq}(P,Q)$, we consider the partial deduction originating from the SLD-trees for $PD_{\trianglelefteq}(P,Q)$ constructed by $U_{\trianglelefteq}$.

7.2.3 Automatically Removing the Parsing Overhead

We now return to the subject of removing the parsing overhead of a vanilla meta program V_P using the concrete algorithm for partial deduction obtained by instantiating the generic algorithm with the operations defined above. The basic question we need to resolve is as follows: if we denote with V'_P the partial deduction of $PD_{\trianglelefteq}(V_P,Q)$, under what conditions is $PD_{\trianglelefteq}(V_P,Q)$ such that V'_P satisfies the conditions of Definition 7.4 or, in other words, under what conditions will the residual program be meta structure free.

In what follows, we assume the partial deduction query Q to be of the form $solve(t)$ where t is a term that can not, due to resolution in V_P, become instantiated with meta structure.

Definition 7.10 *Let $\mathcal{H}_V$ denote the extended (non-ground) Herbrand Base associated to a definite program P. An atom A is called* meta instantiated enough *with respect to P if and only if A is such that for all $A\theta \in \mathcal{H}_V$, θ is meta structure free.*

Requiring that the partial deduction query Q is meta instantiated enough is a very natural assumption if the resulting partial deduction of a vanilla meta program is to be meta structure free. Indeed, if the query can become instantiated with meta structure due to resolution, there simply is no way to remove all the meta structure by partial deduction. Indeed, if the partial deduction query is, for example $solve(X)$, any partial deduction of V_P constructed will contain meta structure, in order to parse (instances of) the top level query $solve(X)$.

A first observation is the following: if no atoms are abstracted during the construction of $PD_{\trianglelefteq}(P,Q)$ and Q is meta instantiated enough, then the associated partial deduction V'_P is meta structure free. In order to see this, let us first have a look at how the atoms in $PD_{\trianglelefteq}(P,Q)$ look like in case no abstractions occur:

Proposition 7.1 *Let P be a definite program, and Q of the form $solve(t)$. If Q is meta instantiated enough and if no atoms are abstracted during the construction of $PD_{\trianglelefteq}(P,Q)$, then for each atom $A \in PD_{\trianglelefteq}(P,Q)$ it holds that A is of the form $solve(t')$ and meta instantiated enough.*

Proof First, we prove that if A is of the form $solve(t)$ and meta instantiated enough, and if τ represents the partial SLD-tree for $P \cup \{\leftarrow A\}$ constructed by $U_{\trianglelefteq}$ then $\forall A' \in leaves(\tau)$ holds that A' is of the form $solve(t')$ and meta instantiated enough. Indeed, $U_{\trianglelefteq}$ unfolds all non-recursive calls, thus if a call is not unfolded by $U_{\trianglelefteq}$, it is of the form $solve(t')$. Moreover, such a $solve(t')$ is meta instantiated enough, since it is derived by SLD-resolution

from $solve(t)$. Now, the result follows since the process is started from Q, where Q is of the form $solve(t)$ and meta instantiated enough, and each atom that is brought on the global level is of this form and no atoms are abstracted.

□

The property expressed by Proposition 7.1 is sufficient to guarantee meta freeness of the residual program. Indeed, the closedness condition states that for any atom $B \in V'_P \cup \{Q\}$, there exists an atom $A \in PD_{\trianglelefteq}(V_P, Q)$ such that $B = A\theta$ for some θ. If each such A is meta instantiated enough, by definition, θ is meta structure free and the condition of Definition 7.4 is satisfied.

Example 7.5 *Reconsider the vanilla meta program V_P with P the* `append/3` *program (see Fig. 7.2). For a partial deduction query $Q = solve(app(X, Y, Z))$, it holds that $PD_{\trianglelefteq}(P, Q) = \{solve(app(X, Y, Z))\}$. Indeed, the SLD-tree for the initial query Q constructed by $U_{\trianglelefteq}$ (see Fig. 7.3) contains only a single leaf with the atom $solve(app(Es, Y, R))$, which – being a variant of the original query – is not put in the global tree.*

However, a problem rises in case atoms in $PD_{\trianglelefteq}(P, Q)$ are the result of an abstraction, in what case the abstracted atom possibly is not meta instantiated enough and the resulting residual program possibly is not meta structure free.

Example 7.6 *Consider the following vanilla meta program, which is the vanilla encoding of the* `reverse/3` *with accumulating parameter program, enhanced with a type check on the accumulator, depicted in Fig. 7.4. The example is from (Leuschel and Martens 1996). The SLD-trees constructed by $U_{\trianglelefteq}$ for $PD_{\trianglelefteq}(P, Q)$ are depicted*

```
(1) solve(true).
(2) solve((A,B)):- solve(A), solve(B).
(3) solve(A):- cl(A,B), solve(B).
(4) cl(rev([],L,L), true).
(5) cl(rev([E|Es],L,R), (ls(L), rev(Es,[E|L],R))).
(6) cl(ls([]),true).
(7) cl(ls([X|Xs]), ls(Xs)).
```

Figure 7.4: The vanilla program V_P with P the `reverse/3` program

in Fig. 7.5. The original query $solve(rev(X, Y, Z))$ is homeomorphically embedded in the leaf $solve((ls(L), rev(Es, [E|L], R)))$ of its SLD-tree. Hence, instead of entering the latter in the global tree, the atom

$$solve(X) = msg(\{solve(rev(X, Y, Z)), solve((ls(L), rev(Es, [E|L], R)))\})$$

is added. Now, the atoms in the leafs of the SLD-tree for this abstraction (see Figure 7.5) are all instances of the abstracted atom, hence they are not processed

Figure 7.5: The SLD-trees constructed by $U_{\trianglelefteq}$ for $PD_{\trianglelefteq}(P, Q)$.

any further, and the residual program associated to $PD_{\trianglelefteq}(V_P, Q)$ *is as follows, from which no structure can be filtered:*

```
solve(true).
solve((A,B)):-solve(A), solve(B).
solve(rev([],L,L)).
solve(rev([E|Es],L,R)):- solve((ls(L), rev(Es,[E|L],R))).
solve(ls([])).
solve(ls([X|Xs])):- solve(ls(Xs)).
```

In general, the unfolding rule $U_{\trianglelefteq}$ (and similar unfolding rules (Martens and De Schreye 1996)) performs well when predicate arguments either shrink or grow throughout the unfolding process. Unfolding is allowed as long as information is consumed and appropriately halted when such is no longer the case. Problems however arise for predicates handling *fluctuating structure(s)*: structures that can grow, but also shrink between successive recursive calls. Although a lot of natural logic programs do not contain such predicates, meta interpreters are a typical example of programs that do. This is one of the main reasons why it is notoriously difficult to handle them well in automatic, general partial deduction. Reconsider Example 7.6; the basic problem is that we have at the global level the situation

$$solve(rev(X, Y, Z)) \trianglelefteq solve((ls(Y), rev(Es, [E|Y], Z)))$$

from which we obtain, after abstraction, the atom $solve(X)$. The problem is that the $rev/3$ functor and all structure surrounding the embedded term are lost in

the abstraction. Hence, the residual predicate according to the SLD-tree built for $solve(X)$ will handle (parse) this meta structure.

Now, if $solve((ls(Y), rev(Es, [E|Y], Z)))$ was unfolded (at the local level) by $U_{\trianglelefteq}$ one step further, the atoms $solve(ls(Y))$ and $solve(rev(Es, [E|Y], Z))$ would be brought onto the global level. Still, we would have that

$$[solve(rev(X, Y, Z)) \trianglelefteq solve(rev(Es, [E|Y], Z))$$

but now less structure – and no meta structure at all – is lost by the abstraction, leading to

$$solve(rev(X, Y, Z)) = msg(\{solve(rev(X, Y, Z)), solve(rev(Es, [E|Y], Z))\}$$

and leaving the other atom, $solve(ls(Y))$ as it is. So, the observation is that by the unfolding of the call $solve((ls(Y), rev(Es, [E|Y], Z)))$, the meta level goal is "parsed" as being a conjunction which is split, resulting in two new calls to the meta interpreter. Hence, the SLD-tree now ends in a leaf of which the atoms are better suited to be put on the global level from a viewpoint of creating a meta structure free residual program.

In (Lakhotia and Sterling 1990), it was noticed that *always* unfolding such parsing calls seems a good idea, although no indication was given how to incorporate this idea in a general, automatic partial deduction technique. In subsection 7.2.4, we argue that, if we aim at obtaining a high degree of specialisation, it is often a better idea *not* to unfold parsing calls, at the cost of obtaining what we will call *specialised* parsing. We will therefore aim at further unfolding parsing calls only if they lead to generalisation. To that end, we modify our unfolding rule such that it takes global information into account and tries to avoid generalisation (or minimise the structure lost by it) when this can be safely achieved through some extra unfolding.

First, we introduce an unfolding rule that will unfold an atomic query $\leftarrow A$ deterministically (using $\trianglelefteq$ for termination control). Moreover, it will not further unfold an atom B if it holds that $B \triangleleft A$, since in this case – due to the definition of $\triangleleft$ – the root atom (a predicate call) resolves to another call to the same predicate with less structure in its argument(s). We define this unfolding rule, denoted U^+ in terms of the following notion:

Definition 7.11 *Let Q_0 be an atomic query. An atom A in a leaf of a partial SLD-tree is* +-selectable with respect to Q_0 *if*

- *A does not descend from a selected atom A' with $A' \trianglelefteq A$,*
- *A has a single resolvent (that is, unfolding the atom – possibly using a lookahead – is deterministic, and*
- *$A \not\triangleleft Q_0$*

Now, we are able to define the unfolding rule U^+:

Definition 7.12 *The unfolding rule U^+ creates a partial SLD-tree for an atomic goal Q_0 by repeatedly unfolding the leftmost atom that is +-selectable with respect to Q_0 in the SLD-tree under construction. If no atom is +-selectable, no further unfolding is performed.*

Example 7.7 *Reconsider the vanilla program from Example 7.6 – representing the* `reverse` *object program with a type check on the accumulator. Unfolding the atoms $solve((ls(Y), rev(Es, [E|Y], Z))$ and $solve(ls(Y))$ using U^+ results in the SLD-trees depicted in Fig. 7.6. The atoms in the leaf of the leftmost tree are not*

Figure 7.6: SLD-trees created by U^+.

+-selectable since they are embedded in the tree's root; the atoms in the leaf of the rightmost tree are not +-selectable since they can not be unfolded deterministically.

Since the unfolding rule U^+ creates – due to its deterministic nature – an SLD-tree containing a single (non-failing) branch, we will denote the result of U^+ for an atomic query Q_0 as an SLD-derivation $Q_0, Q_1, \ldots, Q_n$ with computed answers $\theta_1, \ldots, \theta_n$, or simply

$$U^+(Q_0) = Q_0 \overset{\theta}{\leadsto}_P A_1, \ldots, A_m$$

if $Q_n = A_1, \ldots, A_m$ and $\theta = \theta_1 \ldots \theta_n$. The SLD-derivation constructed by U^+ is finite:

Lemma 7.1 *For any atomic goal Q_0, $U^+(Q_0)$ is a finite SLD-derivation.*

Proof Immediate from Definition 7.11 and $\trianglelefteq$ being a well-quasi order relation. □

We will try to take fluctuating structure into account by merging $U_{\trianglelefteq}$ and U^+into a single unfolding rule: an SLD-tree τ is built as usual, using $U_{\trianglelefteq}$, but when introducing an atom $A \in \mathit{leaves}(\tau)$ would cause abstraction to take place in the global tree, the atom A is further unfolded using U^+, but only if such unfolding results in new calls to the same predicate with less structure in them. This is for example the case in the leftmost SLD-tree depicted in Fig. 7.6, but not in the same figure's other SLD-tree. Note that such an unfolding rule implements *local* control based on information (whether or not a generalisation is about to occur) from the *global* level. We define the combined unfolding rule, denoted by $U^+_{\trianglelefteq}$, by means of the following algorithm:

Algorithm 7.1

Input: *a definite program P, a global tree γ, with the atom A a leaf of γ*
Output: *a partial SLD-tree τ for $P \cup \{\leftarrow A\}$*
Initialise: *Let $i = 0$, $\tau_0 = \emptyset$, and τ_1 be the partial SLD-tree constructed for $P \cup \{\leftarrow A\}$ by $U_{\trianglelefteq}$. Moreover, let $Anc(A)$ denote the set of ancestors of A in γ (including A).*

while $\tau_{i+1} \neq \tau_i$ **do**
$i = i + 1$
if there exists *a leaf $\leftarrow B_1, \ldots, B_n$ in τ_i such that for some $k \in \{1, \ldots, n\}$:*
- *$A' \trianglelefteq B_k$ for some $A' \in Anc$,*
- *$U^+(B_k) = B_k \overset{\theta}{\rightsquigarrow}_P A_1, \ldots, A_m$ such that $A_i \triangleleft B_k$ for all i, and*
- *$dom(\theta) \cap \mathcal{V}(B_1, \ldots, B_{k-1}, B_{k+1}, \ldots, B_n) = \emptyset$*

then *τ_{i+1} is the SLD-tree obtained by replacing the leaf $\leftarrow B_1, \ldots, B_n$ with the SLD-derivation*

$$\delta = \begin{array}{l} \leftarrow B_1, \ldots, B_{k-1}, B_k, B_{k+1}, \ldots, B_n \\ \leftarrow B_1, \ldots, B_{k-1}, Q_1, B_{k+1}, \ldots, B_n \\ \vdots \\ \leftarrow B_1, \ldots, B_{k-1}, Q_l, B_{k+1}, \ldots, B_n \\ \leftarrow B_1, \ldots, B_{k-1}, A_1, \ldots, A_m, B_{k+1}, \ldots, B_n \end{array}$$

with B_k, $Q_1, \ldots, Q_l$ and $A_1, \ldots, A_m$ the subsequent goals in $U^+(B_k) = B_k \rightsquigarrow_P A_1, \ldots, A_m$.

else $\tau_{i+1} = \tau_i$
end while
$\tau = \tau_i$

The following proposition states that the result of combining both unfolding rules is a finite, partial SLD-tree:

Proposition 7.2 *Algorithm 7.1 terminates and its output, τ, is a partial SLD-tree for $P \cup \{\leftarrow A\}$.*

Proof First, we prove by induction that each constructed τ_i ($i \geq 1$) is a finite partial SLD-tree. By definition, this holds for τ_1 being the result of $U_{\trianglelefteq}$. Next, assume τ_i is a finite partial SLD-tree with a leaf $B_1, \ldots, B_n$ for which the condition in the algorithm holds. By Lemma 7.1, $U^+(B_k)$ is a finite SLD-derivation $B_k \overset{\theta}{\rightsquigarrow}_P A_1, \ldots A_m$. Since

$$dom(\theta) \cap \mathcal{V}(B_1, \ldots, B_{k-1}, B_{k+1}, \ldots, B_n) = \emptyset,$$

the sequence of goals

$$\delta = \begin{array}{c} \leftarrow B_1, \ldots, B_{k-1}, B_k, B_{k+1}, \ldots, B_n \\ \leftarrow B_1, \ldots, B_{k-1}, Q_1, B_{k+1}, \ldots, B_n \\ \vdots \\ \leftarrow B_1, \ldots, B_{k-1}, Q_l, B_{k+1}, \ldots, B_n \\ \leftarrow B_1, \ldots, B_{k-1}, A_1, \ldots, A_m, B_{k+1}, \ldots, B_n \end{array}$$

with B_k, Q_1,...,Q_l and $A_1, \ldots, A_m$ the subsequent goals in $U^+(B_k)$, is a finite SLD-derivation. Consequently, replacing the leaf $B_1, \ldots, B_n$ in τ_i with δ results in a finite, partial SLD-tree τ_{i+1}.

Now, we prove that the algorithm terminates. Since τ_1 is a finite, possibly incomplete SLD-tree, it has a finite number of branches. In each round of the algorithm, one such branch is selected and lengthened by an SLD-derivation, without further branches being created in the tree. Therefore, it suffices to concentrate on one such branch β in τ_1 and prove that it is not being lengthened an infinite number of times by repeated applications of U^+. Let β' denote a branch obtained by lengthening β a number of times. We consider a subset of the nodes in β', namely those nodes $S_1, S_2, \ldots$ that were, during construction of β', a leaf of the SLD-tree under construction, that is:

$S_1 \in \beta'$ and S_1 a leaf of τ_1, and
$\forall i > 1 : S_i$ the end point of the SLD-derivation $U^+(S_{i-1})$

Suppose an infinite sequence of such nodes $S_1, S_2, \ldots$ exists. Since $\trianglelefteq$ is a well-quasi order relation, then there exist i, j with $i < j$ such that $s(S_i) \trianglelefteq s(S_j)$ where $s(S)$ denotes the selected atom in S. By construction in Algorithm 7.1, it holds that $s(S_j) \triangleleft s(S_{j-1})$. Now, replacing S_{j-1} by the derivation δ does not instantiate atoms in S_{j-1} other than $s(S_{j-1})$. Hence, by transitivity of $\triangleleft$, it holds that $s(S_j) \triangleleft s(S_i)$. This contradicts the fact that $s(S_i) \trianglelefteq s(S_j)$, and proves that no branch β can be lengthened an infinite number of times and hence termination of the algorithm.

□

If we denote with $PD^+_{\trianglelefteq}$ the set of atoms as produced by the instance of the generic partial deduction algorithm in which the unfolding rule is $U^+_{\trianglelefteq}$, and the *covered*, *whistle* and *abstract* functions remain as before, we can formulate the following important result that allows to conclude that the result of partially deducing any vanilla meta program is meta structure free.

Theorem 7.1 *Let P be a definite program, and Q a partial deduction query of the form solve(t). If Q is meta instantiated enough, then for each atom $A \in PD^+_{\trianglelefteq}(V_P, Q)$ it holds that A is of the form solve(t') and meta instantiated enough.*

Proof First, we prove that for an atom of the form $solve(t)$ that is meta instantiated enough, the SLD-tree constructed by $U^+_{\trianglelefteq}$, say τ, is such that $\forall A \in leaves(\tau)$, A is of the form $solve(t')$, A is meta instantiated enough and $A \neq solve(true)$. The proof is by induction:

- Let us first consider the SLD-tree τ_1 constructed by $U_{\trianglelefteq}$. We have proven (see the proof of Proposition 7.1) that $\forall A \in leaves(\tau_1)$ holds that A is of the form $solve(t)$ and meta instantiated enough. Moreover, since $true \notin \mathcal{F}_{\Pi_P}$, an atom $solve(true)$ can never descend from another atom $solve(true)$ and consequently (by definition of $\trianglelefteq$), any occurrence of $solve(true)$ in the SLD-tree under construction will be unfolded by $U_{\trianglelefteq}$.
- Assume the same holds for any $A \in leaves(\tau_i)$. We prove that it also holds for any $A \in leaves(\tau_{i+1})$. The only difference between τ_i and τ_{i+1} is that one such leaf atom, $solve(t) \in leaves(\tau_i)$, is no longer a leaf atom of τ_{i+1} since the leaf to which it belongs is replaced by an SLD-derivation δ constructed from $solve(t) \rightsquigarrow A_1, \ldots, A_m$. By construction of U^+, each of the atoms $A_1, \ldots, A_m$ have `solve/1` as predicate symbol (since $A_i \lhd solve(t)$). Again, by similar reasoning as in the proof of Proposition 7.1, we conclude that if $solve(t)$ is meta instantiated enough, so will be each A_i. By similar reasoning as in the case for τ_1, $A_i \neq solve(true)$.

Now, let us denote with γ the global tree under construction, and with τ an SLD-tree, freshly constructed by $U^+_{\trianglelefteq}$. Since we start from a single atom $solve(t)$ that is meta instantiated enough and every atom in a leaf of an SLD-tree constructed by $U^+_{\trianglelefteq}$ for such an atom is of the same form (and not equal to $solve(true)$), all that remains to be proven is that if such a leaf atom of τ is generalised before entering in γ, the generalisation is meta instantiated enough. Assume $solve(t) \in leaves(\tau)$, and assume there exist $solve(t') \in Anc(solve(t))$ in γ such that $solve(t') \trianglelefteq solve(t)$. We have two possibilities:

- Either $solve(t)$ is of the form $solve(p(t_1, \ldots, t_n))$ (with $p \in \mathcal{F}_{\Pi_P}$) in what case also $solve(t')$ is of the form $solve(p(t'_1, \ldots, t'_n))$ (by definition of $\trianglelefteq$) and, by definition of the most specific generalisation, we have that
$$msg(solve(p(t'_1, \ldots, t'_n)), solve(p(t_1, \ldots, t_n))) = solve(p(msg(t'_1, t_1), \ldots, msg(t'_n, t_n)))$$
Since only object level terms are generalised, the resulting atom is meta instantiated enough.

- Or $solve(t)$ is of the form $solve((a, c))$ (with (a, c) representing an object level conjunction). However, $solve((a, c))$ can be deterministically (using a look-ahead and assuming that $\mathcal{F}_{\Pi_P} \cap \{true/0, \ , /2\} = \emptyset$) unfolded by using clause (2) of V, resulting in the conjunction $solve(a), solve(c)$. Since during this unfolding no variables are instantiated in a nor c, we have that $solve(a) \lhd solve((a, c))$ and $solve(c) \lhd solve((a, c))$. Moreover, neither $solve(a)$ nor $solve(c)$ are selectable by Definition 7.11, and hence (by Definition 7.12),

$$leaves(U^+(solve((a, c)))) = \{solve(a), solve(c)\}$$

which contradicts with the fact that $solve((a, c)) \in leaves(\tau)$.

Hence, if an atom is generalised, it is of the form $solve(p(t_1, \ldots, t_n))$ and the resulting generalisation is meta instantiated enough.

□

Consequently, the conditions of Definition 7.4 are satisfied and we have the following:

Corollary 7.1 *Let P be a definite program, and Q a partial deduction query of the form $solve(t)$. If V'_P denotes the partial deduction of $PD^+_{\trianglelefteq}(V_P, Q)$ in V_P, V'_P is meta structure free.*

Corollary 7.1 represents an important result: it states that the overhead due to the handling of meta structure can be removed from any vanilla meta program using an automatic and general partial deduction technique.

Example 7.8 *Reconsider the vanilla meta program V_P from Example 7.6 with the partial deduction query $Q = solve(rev(X, Y, Z))$. We have that*

$$PD^+_{\trianglelefteq}(V_P, Q) = \{solve(rev(X, Y, Z)), solve(ls(Y))\}$$

and the corresponding partial deduction

```
solve(rev([],Y,Y)).
solve(rev([E|Es],Y,Z)):- solve(ls(Y)), solve(rev(Es,[E|Y],Z)).
solve(ls([])).
solve(ls([X|Xs])):- solve(ls(Xs)).
```

from which all meta structure can be filtered.

7.2.4 Specialised Parsing

In the previous section, we have defined an instance of the generic partial deduction algorithm, and have proven that the partial deduction it creates of a vanilla meta

program is (under certain conditions) meta structure free. The algorithm achieves this by altering a standard unfolding rule in such a way that it continues the unfolding of an atom in a leaf of an SLD-tree under construction if doing so results in a number of atoms which are all embedded in the former atom, a situation that occurs frequently in the case of vanilla programs derived from the standard three-line vanilla meta interpreter. More precisely, the extension of the unfolding rule, U^+, ensures that calls to the meta interpreter with an object conjunction as argument are unfolded, since doing so – parsing the object conjunction – results in two new calls to the meta interpreter, each with a "smaller" object goal.

In previous work on partial deduction of meta interpreters, Lakhotia and Sterling (Lakhotia and Sterling 1990) argued that such (parsing) calls should *always* be unfolded. This is indeed the case, if one aims at obtaining a partial deduction that is syntactically equivalent with the original object program. Our extended unfolding rule, $U^+_{\trianglelefteq}$ will often stop unfolding at such parsing calls, putting the atom on the global level such that a new predicate will be generated for it. Consider the following example:

Example 7.9 *Let P denote the following definite program:*

```
p([]).                          b(cat, mammal).
p([E|Es]):- a(E), p(Es).        b(dog, mammal).
a(mammal).                      b(eagle, bird).
a(E):- b(E,Y), a(Y).
```

Figure 7.7 shows the first two SLD-trees constructed during generation of the set $PD^+_{\trianglelefteq}(V_P, Q)$ for the partial deduction query $solve(p(X))$. The left-hand side of

Figure 7.7: First two constructed SLD-trees for the query $solve(p(X))$.

Fig. 7.7 shows the SLD-tree built for the query. Using the unfolding rule $U_{\trianglelefteq}$, unfolding stops at the leaf node $solve((a(E), p(Es)))$. However, using Algorithm 7.1, it is detected that bringing this atom in the global tree, would result in an abstraction. Indeed, we have that

$$solve(p(X)) \trianglelefteq solve((a(E), p(Es)))$$

and it can be verified that also their characteristic trees are embedded. The node $solve((a(E), p(Es)))$ can however be unfolded one step further, using U^+ into the conjunction $solve(a(E))$ and $solve(p(Es))$ (both atoms being embedded in the former leaf node). The latter atoms are brought on the global level without generalisation taking place. The right-hand side of Fig. 7.7 shows the SLD-tree built for the atom $solve(a(E))$. Again, unfolding using $U_{\trianglelefteq}$ stops at the leaf $solve((b(E,Y), a(Y))$. Although we have an atom $solve(a(E))$ in the global tree, and it holds that

$$solve(a(E)) \trianglelefteq solve((b(E,Y), a(Y)),$$

a closer look at their associated characteristic trees reveals that they are not embedded, and hence no generalisation will occur when the latter atom is put on the global level. The reason why the corresponding characteristic trees are not embedded can be seen from Fig. 7.8, showing the SLD-tree for the atom $solve((b(E,Y), a(Y))$. The computation associated to the atom $solve(a(E))$ does *not* occur in the latter tree. Indeed, only the more specific computations associated to $solve(a(mammal))$ and $solve(a(bird))$ are present, neither of them embedding the former more general computation. Hence, the atom $solve((b(E,Y), a(Y)))$ is not further unfolded, and hence the object level conjunction $b(E,Y), a(Y)$ is not parsed, but rather brought on the global level, and a new SLD-tree (the one in Fig. 7.8) is built. The residual predicates for the SLD-trees in Fig. 7.7 and 7.8 are, after structure filtering, as follows:

```
d1([]).                          d3(cat).
d1([E|Es]):-d2(E), d2(Es).       d3(dog).
d2(mammal).
d2(E):-d3(E).
```

In the above example, the residual program resulting from the partial deduction is meta structure free. It does not, however, correspond syntactically with the original program. Indeed, the partial deduction algorithm, being a general technique, has performed several unfoldings at the level of the *object program*. Hence, not only the overhead introduced by the meta interpretation has been removed, but also some inefficiencies of the object program itself. This was possible precisely due to the fact that the object level conjunction $b(E,Y), a(Y)$ was *not* parsed locally, but rather brought on the global level in a single call $solve((b(E,Y), a(Y)))$, such that in the construction of the SLD-tree for the atom, the bindings produced by the call $b(E,Y)$ are propagated to the atom $a(Y)$. Following the strategy proposed in (Lakhotia and Sterling 1990), and unfolding the parsing call $solve((b(E,Y), a(Y)))$ would have resulted in two SLD-trees, one for $solve(b(E,Y))$ and another for $solve(a(Y))$. In this case, the bindings produced by unfolding $solve(b(E,Y))$ are not propagated to the atom $solve(a(Y))$, and the residual predicates associated to these trees would be syntactically equivalent to the object level predicates `b/2` and `a/1`. So, contrary to what is generally believed, unfolding parsing calls during

Figure 7.8: The SLD-tree for $solve((b(E,Y),a(Y))$.

partial deduction does not always lead to optimal results.

On the other hand, the above example again shows the necessity of being able to handle fluctuating structure in a sophisticated way. If, in the SLD-tree built for the original query $solve(p(X))$ (see Fig. 7.7), the leaf atom $solve((a(E),p(Es)))$ was not further unfolded, a generalisation at the global level would have resulted into the atom

$$solve(X) = msg(\{solve(p(X)), solve((a(E),p(Es)))\}$$

being brought on the global level, resulting in a residual predicate parsing the meta structure terms that can be bound to X.

Within the context of *conjunctive partial deduction* (Leuschel, De Schreye, and de Waal 1996; Glück, Jørgensen, Martens, and Sørensen 1996), a less sophisticated local control strategy may be sufficient to obtain the information propagation between object level atoms that is obtained when the object level conjunction is not unfolded by $U^+_\trianglelefteq$ (resulting in what we called *specialised parsing*). Conjunctive partial deduction extends the basic partial deduction framework of Lloyd and Shepherdson (1991) by using conjunctions of atoms rather than atoms as the basic specialisation entity and incorporating renaming at the level of these conjunctions.

Consequently, even if a conjunction of object level atoms gets unfolded into two or more calls to the `solve/1` predicate, the resulting atoms can – under certain conditions – be kept together in a conjunction at the global level. When the *conjunction* of atoms is unfolded rather than each of the atoms in isolation, the information propagation between the atoms is retained, which is not the case in the standard partial deduction framework. Moreover, conjunctions of atoms can be split and/or reordered in the conjunctive partial deduction framework, providing more control opportunities to ensure termination and diminishing, in general, the need for a sophisticated local control strategy. Note however that one of the basic issues when specialising the vanilla meta interpreter – ensuring that no meta structure is lost by generalisation of `solve/1` atoms at the global level – remains to be taken care of, also in the context of conjunctive partial deduction.

7.3 Specialising an Extended Meta Interpreter

In this section, we discuss the applicability of the general partial deduction algorithm on a more involved Vanilla-like meta interpreter. The meta interpreter, based on the interpreter defined by Brogi and Contiero in (Brogi and Contiero 1997; Brogi and Contiero 1996), extends logic programming in the sense that it deals with *compositions* of object programs. The implementation of the interpreter is found in Fig. 7.9. Queries to the meta interpreter are of the form $demo(E, G)$,

```
demo(E,true).
demo(E,(A,B)):- demo(E,A), demo(E, B).
demo(E,H):- cl(E,H,B), demo(E, B).
cl(union(E1,E2),H,B):- cl(E1,H,B).
cl(union(E1,E2),H,B):- cl(E2,H,B).
cl(inters(E1,E2),H,B):- cl(E1,H,B), cl(E2,H,B).
cl(enc(E),H,true):- demo(E,H).
cl(pr(P),H,B):- statement(P,H,B).
```

Figure 7.9: The extended Vanilla Meta Interpreter.

where E is a *program expression* (i.e. a meta level composition of object programs) and G is an *object level query*. The `demo/2` predicate plays the role of the `solve/1` query from the simple Vanilla meta interpreter: it *parses* an object level goal, the difference being that it considers the goal with respect to a particular program expression. The object level goal *true* is true in any program expression, and an object level conjunction (A, B) is true in a program expression E if both conjuncts are true in the program expression E. The predicate `cl/3` relates an object level clause (expressed by a head H and a body B) with a particular program expression: an object level clause $H \leftarrow B$ belongs to the *union* of two program expressions $E1$ and $E2$, if it belongs either to $E1$ or to $E2$. A clause belongs to the

intersection of two program expressions if it belongs to both expressions independently. A unit clause $H \leftarrow$ belongs to the *encapsulation* of a program expression, if H succeeds with respect to the program expression. The last clause in Fig. 7.9 states that an object level clause belongs to a particular object program named P, if it is represented by a `statement/3` fact for the particular object program. A more elaborated discussion of these composition operators can be found in (Brogi, Mancarella, Pedreschi, and Turini 1990; Brogi, Mancarella, Pedreschi, and Turini 1992; Brogi and Contiero 1994; Brogi and Contiero 1997). Object level programs are thus represented using the `statement/3` predicate, as the following example, adapted from (Brogi and Contiero 1996), shows.

Example 7.10 *Consider the object level programs named* `train`, `ic` *and* `path`, *in Fig. 7.10. The object program* `train` *defines a small database of train connections.*

```
statement(path, path(C1, C2), train(C1, C2, T)).
statement(path, path(C1, C2), (train(C1, C3, T), path(C3, C2))).

statement(train, train(florence, pisa, reg), true).
statement(train, train(florence, rome, int), true).
statement(train, train(pisa, genova, nat), true).
statement(train, train(pisa, rome, nat), true).
statement(train, train(milan, florence, int), true).
statement(train, train(milan, pisa, nat), true).

statement(ic, train(X, Y, int), true).
```

Figure 7.10: Three sample object programs.

Each connection is given by an object level unit clause of the form `train(C1,C2,T)` *expressing that there is a direct train connection from city* $C1$ *to* $C2$ *where* T *is the type of the connecting train. The object program* `path` *defines when there is a (non direct) train connection between two cities. The object program* `ic` *defines a constraint on possible train connections.*

Given the object programs defined in Example 7.10, the query

```
demo(union(pr(path),pr(train)), path(florence, C))
```

can be used to find all the cities that can be reached by train from Florence. Likewise, the query

```
demo(union(pr(path), inters(pr(train),pr(ic))), path(florence, C))
```

can be used to find all such cities that are connect by a train of type `int`. Implementing program compositions by the above meta interpreter introduces some overhead during the evaluation of a query (Brogi and Contiero 1997; Brogi and Contiero 1996):

- the overhead associated to the handling of the program compositions, and
- the overhead introduced by interpreting the object program

In (Brogi and Contiero 1997; Brogi and Contiero 1996), it is recognised that (part of) the overhead can be removed by program specialisation, and the design and implementation of such a specialiser is given. The specialiser is ad-hoc, in the sense that it is especially targeted towards specialising this class of meta interpreters. Being ad-hoc, the described specialiser relies heavily on knowledge about the interpreter. A thorough discussion of the specialisation method can be found in (Brogi and Contiero 1996). The results obtainable by it can be summarised as follows: If the interpreter is specialised with respect to a program composition that does not employ an encapsulation operation, the resulting program does not manipulate the program expression any more, hence the overhead introduced by the handling of the program compositions is completely removed. It is also pointed out (Brogi and Contiero 1996) that the resulting program – basically resembling a standard vanilla meta interpreter with respect to a single object program – *can* be transformed into an equivalent object program. No indication, however, is given how to automate this transformation. In case the program expression contains an encapsulation operation, all explicit handling of the program expression is removed, but the resulting program is essentially a set of `solve` vanilla meta interpreters, each with a particular associated data base of object level clauses. Thus, no overhead due to the interpretation of a single object program is removed. Also, since the object level goal is not taken into account by the method, no specialisation at the object level can be achieved.

In what follows, we investigate the partial deduction of the extended meta interpreter defined in Fig. 7.9 using the partial deduction algorithm described earlier in this chapter. Let us first extend some notions of Section 7.2 appropriately in the context of the extended interpreter. In what follows, we denote with D the extended vanilla meta interpreter depicted in Fig. 7.9. We define the *vanilla program associated to a composition of object programs* as follows:

Definition 7.13 *Let $P_1, \ldots, P_n$ be definite programs. Then the vanilla program associated to $\{P_1, \ldots, P_n\}$, denoted by $D_{\{P_1,\ldots,P_n\}}$, is defined as*

$$D \cup \bigcup_{P \in \{P_1,\ldots,P_n\}} F_P$$

where

$$F_P = \begin{array}{c} \left\{ \; statement(P_i, A', true) \;\middle|\; A \leftarrow\in P_i \; \right\} \\ \cup \\ \left\{ \; statement(P_i, A', (B'_1, (\ldots, (B'_n, true))\ldots)) \;\middle|\; A \leftarrow B_1, \ldots, B_n \in P_i \; \right\} \end{array}$$

in which A' denotes the term produced by replacing in an atom A the predicate symbol by its corresponding element in $\mathcal{F}_\Pi$.

The definition of meta structure is adapted as follows (see Definition 7.3): a term t contains meta structure (with respect to $\{P_1, \ldots, P_n\}$) if and only if t contains a functor

$$f \in \Sigma_D \cup \bigcup_{P \in \{P_1,\ldots,P_n\}} \mathcal{F}_{\Pi_P}.$$

An first important result is the following, showing meta freeness of a partial deduction in case no encapsulation operations are used.

Theorem 7.2 *Let $P_1, \ldots, P_n$ be definite programs, and Q a partial deduction query. If $Q = demo(e, t)$ is meta instantiated enough, and the program expression e does not contain an encapsulation operation, then for each atom $A \in PD^+_{\trianglelefteq}(D_{P_1,\ldots,P_n}, Q)$ it holds that A is of the form $demo(e, t')$ and meta instantiated enough.*

Proof We concentrate on the first part of the proof, and show that each atom $A \in PD^+_{\trianglelefteq}(D_{P_1,\ldots,P_n}, Q)$ is of the form $demo(e, t')$ (with e the same program expression as in the partial deduction query). First of all, we show that if τ represents the SLD-tree constructed by $U^+_{\trianglelefteq}$ for an atom $demo(e, t')$, then each $A \in leaves(\tau)$ is of the form $demo(e, t'')$. Consider the interpreter's call dependency graph depicted in the left-hand side of Fig. 7.11. In case the program expression e does not contain an *enc* operation, the call dependency graph contains two recursive predicates, which are not mutually recursive. We observe the following:

- Every call to `cl/3` will be unfolded by $U^+_{\trianglelefteq}$. Indeed, since the first argument in a call to `cl/3` – the program expression – is instantiated, every subsequent call to `cl/3` has a strictly smaller first argument. Hence, none of the calls to `cl/3` will embed one of its ancestors and consequently each of them will be unfolded.
- Since the only other recursive predicate is `demo/2`, each $A \in leaves(\tau)$ will have `demo/2` as predicate. Moreover, every such atom will be of the form $demo(e, t'')$ – thus having the same program expression as τ's root. Indeed, in case the program expression does not contain an *enc*-operation, each of the calls to `demo/2` resulting from a call $demo(e, t')$ has an unchanged first argument.

Moreover, if two such atoms $demo(e, t)$ and $demo(e, t')$ are generalised, we have that

$$msg(demo(e, t), demo(e, t')) = demo(e, msg(t, t')).$$

Since the partial deduction query is of the form $demo(e, t)$, and every atom in the leaf of an SLD-tree constructed by $U^+_{\trianglelefteq}$ is of the form $demo(e, t')$ and generalisation in γ preserves the program expression e, we conclude that

every atom in the global tree γ (and hence in $PD^{+}_{\trianglelefteq}(D_{P_1,\ldots,P_n}, Q)$) is of the form $demo(e, t'')$. To prove that each of these atoms is meta instantiated enough, it suffices to concentrate on their second argument representing the object level goal, which renders the proof analogously to the proof of Theorem 7.1.

□

Figure 7.11: The call dependency graphs of $D_{\{P_1,\ldots,P_n\}}$.

Example 7.11 *Reconsider the programs* `path`, `train` *and* `ic` *from Example 7.10. If* D_{pti} *denotes the associated vanilla meta program and* Q *denotes the partial deduction query*

$$Q = demo(union(pr(path), inters(pr(train), pr(ic))), path(X1, X2)),$$

the filtered partial deduction of $PD^{+}_{\trianglelefteq}(D_{pti}, Q)$ *is the query* `demo1(X1,X2)` *together with the residual program*

```
demo1(florence,rome).
demo1(milan,florence).
demo1(X1,X2):- demo2(X1,X3), demo1(X3,X2).
demo2(florence,rome).
demo2(milan,florence).
```

In the above example, the overhead due to the handling of program compositions as well as the overhead due to the meta interpretation of the object program has been removed. The `demo1` predicate in the resulting program is essentially equivalent with the original, object level, `path` predicate. This result is equivalent with the result obtained by the ad-hoc approach of (Brogi and Contiero 1997; Brogi and Contiero 1996). A second, weaker result deals with the situation where the program expression of the partial deduction query possibly contains an encapsulation operation. While the following result makes no claim about the removal of meta interpretation overhead in general, it does imply that all program expressions occurring in the residual program are the same, and can hence be filtered away. As such, the program resulting from partial deduction does not incorporate any overhead due to the handling of the program expression.

Theorem 7.3 *Let $P_1, \ldots, P_n$ be definite programs, and Q a partial deduction query. If $Q = demo(e, t)$ is meta instantiated enough, then for each atom $A \in PD^{+}_{\trianglelefteq}(D_{P_1,\ldots,P_n}, Q)$ it holds that A is of the form $demo(e, t')$.*

Proof As in the case of Theorem 7.2, if the partial deduction query Q is of the form $demo(e, t)$ and meta instantiated enough, we can prove that each atom $A \in PD^{+}_{\trianglelefteq}(D_{P_1,\ldots,P_n}, Q)$ has `demo/2` as predicate. Indeed, $U^{+}_{\trianglelefteq}$ will unfold every call to `cl/3`: the first such call definitely will and since the first argument – the program expression – is instantiated, every subsequent call to `cl/3` – either via direct recursion or indirectly via a call to `demo/2` – has a strictly smaller first argument. Hence, none of the calls to `cl/3` will embed one of its ancestors and consequently every such call will be unfolded.

Contrary to the case of Theorem 7.2, the atoms in the global tree γ – although each one being a call to `demo/2` – do not necessary have the same program expression as first argument. However, if $demo(e_1, t_1) \in \gamma$ and $demo(e_2, t_2) \in \gamma$ such that $demo(e_1, t_1)$ is an ancestor of $demo(e_2, t_2)$ in γ, it holds that if $demo(e_1, t_1) \trianglelefteq demo(e_2, t_2)$, then $e_1 = e_2$. Indeed, assume that $e_1 \neq e_2$. Since $e_1 \trianglelefteq e_2$, e_2 must be strictly greater than e_1. But then, by definition of $D_{P_1,\ldots,P_n}$, $demo(e_2, t_2)$ can not descend from $demo(e_1, t_1)$ via resolution. Consequently, since only atoms that embed one of their ancestors are generalised and these atoms have the same program expression as first argument, we conclude that generalisation of an atom $demo(e, t)$ does not generalises the program expression e. Hence, since the partial deduction query $demo(e, t)$ is meta instantiated enough, and the program expressions occurring in atoms in γ are all subterms of e, each of the atoms $demo(e', t') \in \gamma$ (and hence in $PD^{+}_{\trianglelefteq}(D_{P_1,\ldots,P_n}, Q)$ is meta instantiated enough with respect to its first argument.

□

Corollary 7.2 *If D' denotes the partial deduction of $PD^{+}_{\trianglelefteq}(D_{P_1,\ldots,P_n}, Q)$ with $P_1, \ldots, P_n$ and Q as in Theorem 7.3, then D' does not contain any meta structure from the program expression e.*

While this result does not guarantee that the overhead due to the meta interpretation of a single object program is removed, it does not exclude the possibility. In fact, due to the general nature of the employed partial deduction method, this overhead often will be removed. To illustrate this, let us partially deduce the program from Example 7.10 with respect to the following query:

$$Q = demo(union(pr(path), enc(inters(pr(train), pr(ic)))), X).$$

Note that the absence of an object level query implies that the resulting program *needs* to handle meta structure in order to parse at least the top-level query. Partial

deduction with the general technique results in the program depicted in Fig. 7.12. Besides the necessary top level parsing (`demo1`), no meta structure is present: all

```
demo1(true).
demo1(','(X1,X2)) :- demo1(X1), demo1(X2).
demo1(path(X1,X2)) :- demo2(X1,X2,X3).
demo1(path(X1,X2)) :- demo3(X1,X3,X4,X2).
demo1(','(X1,X2)) :- demo4(X1), demo4(X2).
demo1(train(florence,rome,int)).
demo1(train(milan,florence,int)).

demo2(florence,rome,int).
demo2(milan,florence,int).

demo3(milan,florence,int,rome).

demo4(true).
demo4(','(X1,X2)) :- demo4(X1), demo4(X2).
demo4(train(florence,rome,int)).
demo4(train(milan,florence,int)).
```

Figure 7.12: Specialisation by the general technique.

manipulations concerning the program compositions have been specialised away, as well as all manipulations concerning parsing of the single object programs. Moreover, a certain degree of object level specialisation has been achieved, by precomputing the path between Milan and Rome (`demo3`). Compare this with the obtained result of the dedicated technique, as reported in (Brogi and Contiero 1996) depicted in Fig. 7.13. As a final remark, note that the duplication of

```
demo(union(pr(path), enc(inters(pr(train),pr(ic)))), X):- demo0(X).
demo0(E, true).
demo0(E, (H,T)):-demo0(E, H),demo0(E,T).
demo0(E, H):-clause0(E, H, B), demo0(E, B).
clause0(path(C1, C2), train(C1, C2, T)).
clause0(path(C1, C2), (train(C1, C3,T), path(C3, C2, T))).
clause0(X, true):- demo1(X).
demo1(E, true).
demo1(E, (H,T)):-demo1(E, H),demo1(E,T).
demo1(E, H):-clause1(E, H, B), demo1(E, B).
clause1(train(florence, rome, int), true).
clause1(train(milan, florence, int), true).
```

Figure 7.13: Specialisation by the dedicated technique.

the parsing clause `demo(','(A,B))` is due to some strange behaviour of the *enc*

composition operator: the goal `demo(enc(e),(A,B))` will unify with the second clause of the interpreter (and a derivation will be started for `demo(enc(e),A)`, `demo(enc(e),B)`) as well as with the interpreter's third clause (starting a derivation for `demo(e,(A,B))`, such that the object atoms A and B are evaluated once more. Of course, this behaviour is reflected in the specialised program.

7.4 Discussion

In this chapter, we have investigated the effect of a general, automatic partial deduction method both on the classical, three-line vanilla meta interpreter, and on an extension of this interpreter. Our starting point was the instantiation of the generic partial deduction algorithm of Chapter 2 with particular local and global control strategies. The resulting concrete algorithm is quite powerful: the use of the homeomorphic embedding relation on the local as well as the global level gave satisfying results in recent work on partial deduction (Leuschel, Martens, and De Schreye 1998; Glück, Jørgensen, Martens, and Sørensen 1996) and was proven to be more powerful than other control strategies (Leuschel 1998a). The use of characteristic atoms on the global level provides in most cases a satisfying control of polyvariance (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998).

Although the resulting concrete algorithm is very powerful, it is not in general capable of producing a partial deduction of a vanilla meta program that is meta structure free. The basic problem being that this kind of meta interpreter (as well as other kinds) handles fluctuating structures: the structures they parse can either shrink or grow over recursive calls, something which the traditional use of the homeomorphic embedding relation is unable to deal with appropriately. We have extended the local control component in such a way that it takes information from the global control level into account during unfolding: if an atom would be brought on the global level and result in a generalisation, the atom is – under certain conditions – further unfolded. The control extension we have proposed is a general extension (it does not assume anything about the program that is partially deduced), but is nevertheless inspired by the characteristics of the vanilla meta interpreter. Our extended control strategy is capable of producing a meta structure free partial deduction of any vanilla meta program.

To show the applicability of the proposed control extension to a somewhat more involved meta interpreter, we have investigated its effect on a meta interpreter that deals with compositions of logic programs, as described in (Brogi, Mancarella, Pedreschi, and Turini 1990; Brogi, Mancarella, Pedreschi, and Turini 1992). We have shown that our general and automatic control strategy is able to remove the overhead introduced by the handling of the program compositions, and – under certain conditions – also the overhead introduced by meta interpretation of the object program(s); as such equaling (and even surpassing) the results obtained by

an ad-hoc specialiser, especially developed for partially deducing this kind of meta interpreters (Brogi and Contiero 1997; Brogi and Contiero 1996). Being a general technique, our control strategy treats all program structure as equal; something that enables it to remove some overhead present in the object program.

Example 7.12 *Reconsider Example 7.10, where we add an extra clause to the* `path` *program:*

```
statement(path, torome(X), path(X,rome)).
```

Partially deducing the query

$$Q = demo(union(pr(path), pr(train)), torome(X))$$

results in the filtered query $demo(X)$ *and the filtered program*

```
demo(X):- demo1(X).
demo1(florence).
demo1(pisa).
demo1(milan).
```

This is typically something ad-hoc specialisers like in (Brogi and Contiero 1997; Brogi and Contiero 1996) are unable to do, since they focus their attention towards dealing with the particular *meta* structure alone.

The use of a sophisticated global control component, employing characteristic trees to detect embedding is a key factor in obtaining specialisation at the level of the object program. Recall Example 7.9, where the use of characteristic trees allowed to detect that, although the atom $solve(a(X))$ is syntactically embedded in $solve((b(X,Y), a(Y)))$, the former's computation – being more general than the computations associated to the instances of $a(Y)$ in the latter's SLD-tree – is *not* embedded. Consequently, the use of characteristic trees allowed to keep the object level conjunction of atoms together at the global level instead of unfolding (using $U^{+}_{\trianglelefteq}$) the parsing call thereby splitting the conjunction, or even performing an unnecessary generalisation (in case $U_{\trianglelefteq}$ was used). Sophisticated techniques like (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998) use characteristic trees also to improve the precision of a generalisation: when two atoms are generalised, the common initial subtree of the involved characteristic trees is *imposed* on the generalisation. Doing so in general results in a gain of precision (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998) since by this method, no branches are created in the SLD-tree of the abstracted atom that are not needed in the residual program. However, in the context of partially deducing vanilla meta programs, this common initial subtree often ends on atoms that would (and should) be unfolded, thereby spoiling careful unfolding by a sophisticated local control component.

Example 7.13 *Consider the following vanilla meta program, that implements an object level predicate* `ls/1` *that checks whether its argument is a list.*

```
(1) solve(true).
(2) solve((A,B)):- solve(A), solve(B).
(3) solve(A):- cl(A,B), solve(B).
(4) cl(ls([]), true).
(5) cl(ls([X|Xs]), ls(Xs)).
```

Consider the atoms `solve(ls([]))` and `solve(ls([A]))` together with their associated SLD-trees (τ_1 and τ_2) depicted in Fig. 7.14. Since the selected atom with

Figure 7.14: SLD-trees of `solve(ls([]))` (τ_1) and `solve(ls([A]))` (τ_2).

predicate `cl/2` unifies in τ_1 and τ_2 with different clauses of the program (clause (4) and (5) respectively), the common initial subtree of τ_1 and τ_2 ends after one unfolding step, at the selection of the atom with predicate `cl/2`. When the two atoms are abstracted into `solve(ls(X))`, the method of (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998) imposes this common initial subtree on the generalised atom. As a result, the SLD-tree for `solve(ls(X))` (depicted in Fig. 7.15) has a leaf with a call to `cl/2`. When this atom is put on the global

Figure 7.15: The common initial subtree of τ_1 and τ_2 imposed on the generalised atom `solve(ls(X))`.

level, a new SLD-tree is built for it, and the database of object level clauses will be present (and used) in the specialised program, something we surely want to avoid when specialising meta interpreters. This suggests that the way in which a characteristic tree is imposed upon a generalisation (Leuschel and Martens 1996; Leuschel, Martens, and De Schreye 1998) should be refined, again incorporating extra information (from either local or global level).

In (Owen 1989), Owen argued that if partial evaluation has to be able to specialise meta interpreters as effectively as specialisation by hand, numerous en-

hancements should be incorporated into a basic partial evaluation system. Among these are automatic control of termination and the ability to handle more transformations (such as folding). The most vital feature of a partial evaluation system (Owen 1989) however seems to be its flexibility in deciding where to apply which particular transformation. In this work, we have made automatic partial deduction more "flexible" in this sense, by devising a local control technique that incorporates information about the possible danger of generalisation from the global level. We have demonstrated that such flexibility indeed allows an automatic system to fruitfully specialise the vanilla meta interpreter. Other approaches to specialising this kind of meta interpreters rely on additional information about which atoms should be unfolded (Lakhotia and Sterling 1990; Sterling and Beer 1989). Since this information has to be provided by the programmer, automation of the proposed techniques is problematic. In (Martens 1994), Martens investigates automatic specialisation of meta interpreters, based on the notion of well-founded unfolding. Although good results are obtained on several examples, some issues in making it a generally applicable technique are still open. The loop prevention mechanism used in the partial evaluator Mixtus (Sahlin 1993) is based on comparing terms by comparing their outermost functors up to a certain (ad hoc) depth and subsequently using the *termsize* function to determine whether there is a danger of looping. While this mechanism performs well on most simple examples, setting the system variables like *max_depth* may be non trivial in the case of specialising more complex meta interpreters. Jones (Jones 1996) uses the specialisation of meta interpreters as a tool to describe a number of general phenomena that occur frequently during specialisation, and advocates the importance of learning and communicating a program style that leads to good specialisation (of meta interpreters), obtaining high speedups without code explosion.

The effect of always unfolding parsing calls, as described by Lakhotia and Sterling (Lakhotia and Sterling 1990), can be achieved by altering $U^+_{\trianglelefteq}$ such that it applies U^+ on *every* atom that would be brought in the global tree. Doing so again results in a less flexible system, since it no longer uses information from the global level to decide whether or not to apply U^+. On the other hand, we have demonstrated that the ability to keep the object atoms together in one *solve* atom can cause more specialisation to be achieved. Keeping object atoms together in one *solve* atom mimics to some extent the behaviour of *conjunctive* partial deduction (Leuschel, De Schreye, and de Waal 1996; Glück, Jørgensen, Martens, and Sørensen 1996) on the object program and often results in what we called *specialised* parsing. The question whether this is good or bad is not easily answered, since besides 'parsing', unifications on different head arguments are often involved, indicating that the performance of the specialised program can be system dependent.

As several experiments showed, generalising $solve(C)$ with C a meta representation of an object conjunction, and further processing the generalised atom often

does result in a node G on which $solve(C)$ can be mapped during code generation *without* loss of meta structure. This indicates that $U^+_{\trianglelefteq}$ is perhaps too rash, splitting *solve* atoms when it is not necessary – thereby possibly loosing opportunities for object level specialisation – and suggests that the decision for which atoms residual code will be generated, should be based on even more information, including information from the global level. This may require to (partially) redo some local unfoldings when more information gets available. However, this is a topic of further research. Due to the restrictions in the definition of $U^+_{\trianglelefteq}$, it has an effect only in a well-defined number of cases. A predicate call at the global level can be replaced by a set of calls, but only if each of the resulting calls has the same predicate symbol than the call that was removed and carries strictly less structure in its arguments. Moreover, such a call is only replaced at the global level in case it would have been generalised by global control. Consequently, if a generalisation occurs with one of the newly introduced atoms, it is likely to remove less structure than in case the original atom was generalised – in general allowing more information to be exploited during a subsequent unfolding of the generalised atom. As such we expect the use of $U^+_{\trianglelefteq}$ (in particular when combined with conjunctive partial deduction that may keep the newly introduced atoms together in a conjunction) not to worsen – and in some cases to improve – the specialisation that would be obtained by a more traditional unfolding rule like $U_{\trianglelefteq}$. This was illustrated using the specialisation of the vanilla meta interpreter: the resulting partial deduction method deals very well with this meta interpreter and is, to the best of our knowledge, the first fully automatic, non ad hoc method to do so. The method comprising the unfolding rules U^+ and $U^+_{\trianglelefteq}$ as introduced in this work is nevertheless a general method and does not require any information about what precisely is "meta" structure. In further work, its performance in other contexts, including the specialisation of other, more involved meta interpreters should be investigated.

Chapter 8

Specialising the Other Way Round

If we don't change direction soon,
we'll end up where we're going.

– Prof. Irwin Corey.

In this chapter, we present a novel partial deduction technique, capable of specialising a logic program with respect to a set of unit clauses, rather than a query. We define a framework of what we call *bottom-up partial deduction*, and prove the transformation correct with respect to the fixed point semantics. We instantiate the framework with an example specialisation strategy, and point out how partial deduction in general might benefit from a combination of the classical top-down partial deduction and the novel bottom-up partial deduction.

8.1 Introduction and Motivation

In recent years, a lot of enhancements and extensions to the basic partial deduction framework have been formulated. These enhancements mainly concern the practical aspects of the partial deduction process: how to control the process in such a way that the resulting partial deduction is not only correct, but rather satisfying with respect to the obtained degree of specialisation. A lot of research has focussed on designing sophisticated local as well as global control techniques; the most sophisticated global techniques taking information from the local level

into account. See Chapter 2 for an overview of commonly used control techniques and references. Our work described in Chapter 7 follows this line of research, by incorporating global information into the local control strategy. It is recognised (Leuschel and Bruynooghe 2001) and again demonstrated in Chapter 7, that the interaction between the two control levels can be very subtle and determines – at least in the context of the partial deduction of meta interpreters – how much of the program's *internal* structure handling is removed by the partial deduction process. Apart from the construction of refined control strategies, research on partial deduction has resulted in a conceptual extension of the framework towards building partial deductions for conjunctions of atoms, rather than atoms (Leuschel, De Schreye, and de Waal 1996). This extension basically allows – under some conditions – to preserve the data flow between the atoms in a leaf of a constructed SLD-tree, again showing the interactions between the local and the global control level to be far from trivial (Glück, Jørgensen, Martens, and Sørensen 1996; De Schreye, Glück, Jørgensen, Leuschel, Martens, and Sørensen 1999; Jørgensen, Leuschel, and Martens 1997; Leuschel 1997). Occasionally, partial deduction has been combined with abstract interpretation, in order to improve precision in the residual program. In (Leuschel and Schreye 1996), for example, partial deduction is combined with a *more specific program* transformation (Marriott, Naish, and Lassez 1988) based on abstract interpretation: after unfolding, a non ground version of the T_P operator is used to derive success information which is then imposed on the residual predicates. Other approaches combining partial deduction and abstract interpretation include the Ciao system in which assertions about the program code can be stated by the user or derived by analysis and subsequently used for optimisation purposes (Hermenegildo, Bueno, Puebla, and López 1999).

Most of these approaches towards enhancing and extending (the control of) partial deduction have one thing in common: they assume, exploit and rely on the basic framework for partial deduction (Lloyd and Shepherdson 1991; Komorowski 1992; Gallagher 1993) that was formulated as an extension of SLD-resolution (Lloyd 1987b; Apt 1990). Consequently, partial deduction heirs the top-down nature of the SLD resolution mechanism: the information with respect to which one wants to specialise a program is represented by a partially instantiated query, and is propagated downwards in the program by repeatedly performing resolution steps. Such an approach is feasible for those applications in which the information one wants to exploit during specialisation is representable by a partially instantiated query. Although this is likely to be the case for most logic programs, applications do exist in which one wants to specialise a logic program with respect to information, other than that representable by a query. Consider the implementation of the `append/3` predicate in Example 8.1. The predicate in Example 8.1 can be seen as a general program for list concatenation. It does not manipulate any terms representing lists directly, but rather performs its operations through an abstract data type that provides all the functionality necessary for dealing with

Example 8.1

```
append(X,Y,Y):-list_nil?(X).
append(X,Y,Z):-list_not_nil?(X),
               list_head(X,H), list_tail(X,T),
               append(T,Y,R), list_cons(H,R,Z).
```

lists. Its implementation can be seen as "open", or incomplete: evaluating a call to `append/3` requires a definition of the predicates constituting the abstract data type. Note that the predicate is very general in the sense that it can be run with respect to a variety of different implementations for the `list` abstract data type. Such a possible implementation is as follows, the actual representation for lists being the classical one.

Example 8.2

```
list_nil?([]).          list_head([X|_], X).     list_cons(X,Xs, [X|Xs]).
list_not_nil?([_|_]).   list_tail([_|Xs], Xs).
```

Abstracting such representation information through several layers makes sense from a software engineering point of view: concrete representations can be altered without much effort. On the other hand, every layer of abstraction decreases efficiency. Therefore, it makes sense to propagate the concrete information up into the program, to the places where it is really used. In the above example, propagating the concrete list representation into the program can eliminate all calls to the abstract data type, removing a layer of overhead, and resulting in the classical definition for the `append/3` predicate:

```
append([],Y,Y).
append([X|Xs],Y,[X|R]):- append(Xs,Y,R).
```

While this version of `append/3` answers exactly the same queries as the abstract version (Example 8.1) with respect to the given abstract data type (Example 8.2) – and thus is not *specialised* in the classical sense – it is specialised in that it is no longer "open" and can hence no longer be run with respect to different representations for lists. This kind of specialisation becomes more important as software tends to be developed using *general* components in a *specific* context (see Section 1.2.2).

Another example is the specialisation of a meta interpreter with respect to an object program. Recall from Chapter 7 the definition of the classic vanilla meta interpreter:

```
solve(true).
solve((A,B)):- solve(A), solve(B).
solve(A):- cl(A,B), solve(B).
```

Also this program can be seen as "open", and evaluation requires a definition of the `cl/2` predicate representing the object program we want to evaluate. As we have argued in Chapter 7, it makes sense to specialise the interpreter with respect to a particular object program.

The envisioned kind of specialisation *can* be achieved by existing, top-down, partial deduction methodologies. Particularly the partial deduction of meta interpreters with respect to an object program has been the subject of a vast amount of research; see Chapter 7 for an overview. However, as we have illustrated in the same chapter, obtaining this kind of specialisation of meta interpreters in a *general* and *completely automatic* way is far from trivial, and requires a sophisticated control of the partial deduction process. The main problem posed by this kind of examples is that the control information, needed by the specialiser to decide whether or not to continue the specialisation process often flows (also) in a bottom-up fashion, considerably complicating a top-down approach towards partial deduction. This is often reflected in the query provided to the top-down partial deduction system that, being as general as possible, does not contain any information at all except for the predicate that is called. Reconsider Example 8.1. When the aim is to partially deduce the definition of `append/3` with respect to a particular abstract data type, the partial deduction query will likely be `append(X,Y,Z)`. The fact that all arguments of the query are uninstantiated indicates that the information with respect to which one wants to compute a partial deduction is supplied elsewhere. Assume that we extend the example with a second predicate definition, say `reverse/3` that also employs the abstract data type for list representation. Together, both predicates can be seen as constituting a "library" of predicates for list manipulation and it makes sense to specialise the library with respect to a particular list abstract data type. In this case, a single query does not even suffice to partially deduce the "library". Each of the library's predicates of interest needs to be partially deduced in isolation, starting from a particular query.

But even in case the query *does* contain the information to be exploited by partial deduction, a top-down system might be hampered by some bottom-up information flow. Consider the following example: The predicate `make_list(T,I,R)` can be used to create a list of fixed length (represented by T) , in which each element is initialised with a particular value (I). The resulting list is returned in R.

Example 8.3

```
type(list1, [X]).          fill_list(L,T,I,L):- type(T,L).
type(list3, [X1,X2,X3]).   fill_list(L,T,I,R):- fill_list([I|L],T,I,R).
                           make_list(T,I,R):- fill_list([],T,I,R).
```

Example 8.3 is a representative of a class of recursive predicates that build up some structure between calls before a base clause of the predicate – ending the recursion – is reached, depending on the particular structure built. All top-down

specialisers are based on unfolding techniques, which are known to have problems with these: Often, it can not be derived from the SLD-tree built so far whether or not unfolding such a recursive call will eventually terminate. Consider for example the partial SLD-tree depicted on the left-hand side of Fig. 8.1 for the atom `make_list(list3,a,R)`. The intention of the atom is to create a list of three elements, all initialised to the constant `a`. In the constructed SLD-tree, the only

Figure 8.1: A partial SLD-tree for `make_list(list3,a,R)`.

argument in which the atom `fill_list([a],list3,a,R)` is differing from its predecessor `fill_list([],a,list3,a,R)` is growing with respect to the latter atom and any sensible unfolding rule (based on the syntactical structure of the atoms in the tree) will conclude possible non-termination. Nevertheless, as depicted on the right-hand side of Fig. 8.1, further unfolding will – despite the growing structure – eventually lead to a successful derivation with R instantiated to $[a, a, a]$. If this recursive predicate is handled in a bottom-up fashion, structure is shrinking between recursive calls and the following facts are derived:

```
make_list(list1,I,[I]).
make_list(list3,I,[I,I,I]).
```

which is precisely the kind of specialisation of the `make_list/3` predicate we are interested in. So at least for this example, a bottom-up approach seems a more preferable way to obtain the desired specialisation.

In this chapter, we develop a partial deduction scheme that concentrates on the bottom-up information flow in the program to be partially deduced. The result is a partial deduction scheme in which a program is partially deduced with respect to the information residing in its unit clauses (instead of in a query).

8.2 A Framework for Bottom-up Partial Deduction

Recall from Chapter 2 that the key concept of partial deduction is to construct a number of *partial derivations*, the result of which is recorded in the residual program. In the classical top-down framework, partial derivations are created by SLD-resolution: A set of partial SLD-derivations $\{A_1 \overset{\theta_1}{\leadsto}_P Q_1, \ldots, A_n \overset{\theta_n}{\leadsto}_P Q_n\}$ is built that together cover the complete computation of an initial (atomic) goal in the program P under consideration. Next, each of the partial SLD-derivations $A_i \overset{\theta_i}{\leadsto}_P Q_i$ from this set is contracted into a single residual clause $A_i\theta_i \leftarrow Q_i$ and together these clauses constitute the residual program P'. The effect of partial deduction is that a partial derivation $A' \rightsquigarrow Q'$ in the original program is replaced by a single resolution step from A' to Q' in the residual one. In what follows, we define a framework in which the partial derivations are built the other way round. To that extent, we define a bottom-up inference operator that computes – given a set of partial derivations – a new set of partial derivations, and later on introduce the notion of an abstraction mechanism to guarantee that a *finite* set of such derivations can be computed. In the present section we focus on the conceptual issues concerning what we will call *bottom-up partial deduction*. We postpone the discussion on how to control the construction of a bottom-up partial deduction to a later section.

The result of a partial derivation, whether computed top-down or bottom-up, can be represented by a (residual) clause. The difference between a residual clause computed top-down or bottom-up consists in the origin of the information that resides in it. If the residual clause was created top-down, it resembles a derivation that was constructed by propagating information from the clause's head downwards while performing resolution steps. If, on the other hand, the residual clause was constructed by bottom-up inference, it resembles a derivation constructed by propagating information from the clause's body atoms upwards while performing derivation steps. Hence, the main difference between a top-down and a bottom-up partial deduction process will be the information that it incorporates in the process. A top-down partial deduction process propagates the information from the query downwards, whereas a bottom-up partial deduction process propagates the information residing in the program's unit clauses upwards.

In what follows, we denote with $\mathcal{HC}$ the domain of Horn clauses modulo variance. A bottom-up inference operator on the semantic domain of clauses exist, and is defined in compositional semantics (Bossi, Gabbrielli, Levi, and Meo 1994).

Definition 8.1 *Let P be a definite program. Let S be a set of Horn clauses (modulo variance), that is $S \subseteq \mathcal{HC}$. Then we define the* compositional T_P- operator,

T_P^c as follows:

$$T_P^c(S) = S \cup \left\{ (H \leftarrow \overline{L}_1, \ldots, \overline{L}_n)\theta \;\middle|\; \begin{array}{l} H \leftarrow B_1, \ldots, B_n \in P,\ n \geq 0, \\ A_1 \leftarrow \overline{L}_1, \ldots, A_n \leftarrow \overline{L}_n \textit{ renamed} \\ \textit{apart variants of clauses in } S, \\ \theta = \textit{mgu}((B_1, \ldots, B_n), (A_1, \ldots, A_n)) \end{array} \right\}$$

Note that T_P^c constructs a new clause by unifying the body atoms of a clause in P with the heads of clauses in S. The conjunction of the (appropriately instantiated) body atoms of the clauses in S becomes the new clause's body, whereas the (appropriately instantiated) head of the clause in P becomes the new clause's head.

Proposition 8.1 *The operator T_P^c is monotonic and continuous on the complete lattice $(\wp(\mathcal{H}C), \subseteq)$.*

Proof Let us first proof monotonicity of T_P^c. Let $S, S' \in \wp(\mathcal{H}C)$ such that $S \subseteq S'$. For any clause $C \in T_P^c(S)$ we have that either $C \in S$ and consequently $C \in S'$ and $C \in T_P^c(S')$, or C is a clause of the form $(H \leftarrow \overline{L}_1, \ldots, \overline{L}_n)\theta$ where $H \leftarrow B_1, \ldots, B_n \in P$, $\theta = \text{mgu}((B_1, \ldots, B_n), (A_1, \ldots, A_n))$ and $A_1 \leftarrow \overline{L}_1, \ldots, A \leftarrow \overline{L}_n \in S$. Since $S \subseteq S'$, also $(H \leftarrow \overline{L}_1, \ldots, \overline{L}_n)\theta \in T_P^c(S')$. This proves the monotonicity of T_P^c.

To prove continuity of T_P^c, we must prove that for any directed subset $\mathcal{S} \subseteq \wp(\mathcal{H}C)$ holds that

$$T_P^c(\bigcup_{S \in \mathcal{S}} S) = \bigcup_{S \in \mathcal{S}} T_P^c(S).$$

First, assume that $C \in \bigcup_{S \in \mathcal{S}} T_P^c(S)$. Then there must exist some $S_0 \in \mathcal{S}$ such that $C \in T_P^c(S_0)$. Since $S_0 \subseteq \bigcup_{S \in \mathcal{S}} S$, monotonicity of T_P^c implies that $C \in T_P^c(\bigcup_{S \in \mathcal{S}} S)$. Next, assume that $C \in T_P^c(\bigcup_{S \in \mathcal{S}} S)$, then C is of the form $(H \leftarrow \overline{L}_1, \ldots, \overline{L}_n)\theta$ where $H \leftarrow B_1, \ldots, B_n \in P$, $\theta = \text{mgu}((B_1, \ldots, B_n), (A_1, \ldots, A_n))$ and for each $i \in \{1, \ldots, n\}$, $A_i \leftarrow \overline{L}_i \in \bigcup_{S \in \mathcal{S}} S$. For each such i, let S_i be an element of $\mathcal{S}$ such that $A_i \leftarrow \overline{L}_i \in S_i$. Since $\mathcal{S}$ is directed, and $\{S_1, \ldots, S_n\}$ is a finite subset of $\mathcal{S}$, $\mathcal{S}$ contains an element S_M such that $S_i \subseteq S_M$ for each $i \in \{1, \ldots, n\}$. Therefore, $C \in T_P^c(S_M)$ and also $C \in \bigcup_{S \in \mathcal{S}} T_P^c(S)$.

□

Since T_P^c is monotonic on the complete lattice $(\wp(\mathcal{H}C), \subseteq)$, we can define the ordinal powers of T_P^c as usual:

$$\begin{array}{l} T_P^c \uparrow 0 = \emptyset \\ T_P^c \uparrow i = T_P^c(T_P^c \uparrow i - 1), \text{ for all } i > 0 \end{array}$$

Moreover, the least fixed point of the operator exist, and since T_P^c is continuous the following corollary is immediate (Lloyd 1987b).

Corollary 8.1 *The least fixed point of T_P^c exists and $lfp(T_P^c) = T_P^c \uparrow \omega$.*

The following example illustrates the relation between $T_P^c \uparrow \omega$ and the fixed point semantics of a program.

Example 8.4 *Let P denote the following definite program.*

```
p(X):- animal(X,Y), is_mammal(Y).      animal(cat, mammal).
is_mammal(mammal).                     animal(dog, mammal).
                                       animal(eagle, bird).
```

Then, we have that

$$T_P^c \uparrow \omega = T_P^c \uparrow 2 = \left\{ \begin{array}{l} p(cat). \\ p(dog). \\ animal(cat, mammal). \\ animal(dog, mammal). \\ animal(eagle, bird). \\ is\_mammal(mammal). \end{array} \right\}$$

Example 8.4 shows that $T_P^c \uparrow \omega$ essentially constructs the fixed point semantics of a definite program P, $\mathcal{F}(P)$ as a set of unit clauses rather than atoms. Intuitively, if $T_P^c \uparrow \omega$ is a *finite* set of clauses, this is precisely what we would like to achieve with a (bottom-up) partial deduction. Indeed, if $T_P^c \uparrow \omega$ is finite, it represents a program in which information is maximally propagated, since every atomic query either fails or succeeds in $T_P^c \uparrow \omega$ with an SLD-derivation of length 1.

In general, however, the fixed point semantics of a program P is infinite. The aim of our bottom-up partial deduction is to compute a *finite* set of clauses, say P', that essentially has the same denotation as the original program P, but possibly requires – due to the bottom-up propagation of information – less derivation steps to compute. The T_P^c operator, performing the bottom-up propagation, can serve as a basis to obtain this. However, to construct a *finite* set of clauses, we incorporate an abstraction operation on a set of Horn clauses in the following way:

Definition 8.2 *An abstraction function $\mathcal{A}_b$ is an idempotent function $\wp(\mathcal{H}C) \mapsto \wp(\mathcal{H}C)$ that is defined as follows:*

$$\forall S \subseteq \mathcal{H}C, \mathcal{A}_b(S) = \bigcup_{C \in S} A'(C)$$

where $A' : \mathcal{H}C \mapsto \mathcal{H}C$ is a function such that $\forall A \leftarrow \overline{B} \in \mathcal{H}C$, either $A'(A \leftarrow \overline{B}) = A \leftarrow \overline{B})$ or $A'(A \leftarrow \overline{B}) = H \leftarrow H$ where $H \leq A$. The set $\mathcal{A}_b(S)$ is called an abstraction *of S.*

The difference between a set $S \subseteq \mathcal{H}C$ and the set $\mathcal{A}_b(S)$ is that the latter possibly contains a number of tautologies of the form $H \leftarrow H$, replacing a number of clauses in the former set S of the form $A \leftarrow \overline{B}$ of which the head is an instance of H.

Example 8.5 *Consider the function $A' : \mathcal{H}C \mapsto \mathcal{H}C$ defined for a clause C as follows:*

$$A'(C) = \begin{cases} r(X) \leftarrow r(X) & \text{if } C \text{ is the clause } r(a) \leftarrow \\ r(X) \leftarrow r(X) & \text{if } C \text{ is the clause } r(f(X)) \leftarrow r(X) \\ C & \text{otherwise} \end{cases}$$

then, the function $A : \wp(\mathcal{H}C) \mapsto \wp(\mathcal{H}C)$ defined as

$$A(S) = \bigcup_{C \in S} A'(C)$$

is an abstraction function.

The general idea behind combining bottom-up inference with an abstraction function is that the latter replaces a clause $A \leftarrow \overline{B}$ by $H \leftarrow H$ with $H \leq A$ if the clause $A \leftarrow \overline{B}$ represents a partial (bottom-up) derivation that we wish to residualise. That is, we do not want to extend the derivation $A \leftarrow \overline{B}$ any further using the head atom A, but rather use an abstraction H of A to start the construction of new bottom-up derivations. The reason that abstraction introduces a tautology $H \leftarrow H$ rather than a unit clause H is to ensure that the residualised partial derivations, constituting the clauses of the residual program P', can be appropriately glued together to construct refutations in P'. The task of a *concrete* bottom-up partial deduction strategy will then be to create a suitable abstraction function such that a finite number of bottom-up derivations can be constructed that together "cover" the (possibly infinite) set of all possible bottom-up derivations in the program, much in the same way as the partial SLD-derivations (or trees) representing a top-down partial deduction cover the complete (but possibly infinite) SLD-tree. In our framework, the process of bottom-up partial deduction can be expressed as the repetitive application of an *abstracting T_P^c*-operator.

Definition 8.3 *The* abstracting *T_P^c-operator, A_P^c, associated to a program P and an abstraction function $\mathcal{A}_b$ is defined as $A_P^c = \mathcal{A}_b \circ T_P^c$.*

Example 8.6 *Consider the following definite program P:*

```
p(a,a).              r(a).
q(X,Y):- p(X,Y).     r(f(X)):- r(X).
```

and the abstraction function A defined in Example 8.5. Repeatedly applying the A^c_P-operator associated to P and A results in the following sets of clauses:

$$I_0 = \emptyset$$

$$I_1 = A^c_P(I_0) = A\left(\left\{\begin{array}{l} p(a,a). \\ r(a). \end{array}\right\}\right) = \left\{\begin{array}{l} p(a,a). \\ r(X) \leftarrow r(X) \end{array}\right\}$$

$$I_2 = A^c_P(I_1) = A\left(\left\{\begin{array}{l} p(a,a). \\ r(a). \\ q(a,a). \\ r(f(X)) \leftarrow r(X). \\ r(X) \leftarrow r(X). \end{array}\right\}\right) = \left\{\begin{array}{l} p(a,a). \\ q(a,a). \\ r(X) \leftarrow r(X) \end{array}\right\}$$

It can be verified that I_2 represents the fixed point of the A^c_P-operator defined in Example 8.6. In general, A^c_P is not a monotonic operator, since the employed abstraction function may remove clauses due to the presence (or introduction by) of tautology clauses. However, the following proposition holds:

Proposition 8.2 *For any $S \subseteq \mathcal{H}C$ such that $\mathcal{A}_b(S) = S$, we have that $S \subseteq A^c_P(S)$.*

Proof By definition, $A^c_P(S) = \mathcal{A}_b(T^c_P(S))$. The definition of T^c_P implies the existence of a set $S' \subseteq \mathcal{H}C$ such that $T^c_P(S) = S \cup S'$. By definition of $\mathcal{A}_b$ and its being idempotent, we have $A^c_P(S) = \mathcal{A}_b(S \cup S') = S \cup \mathcal{A}_b(S')$. Hence, $S \subseteq A^c_P(S)$.

□

Corollary 8.2 *Let $S_0 = \emptyset \subset \mathcal{H}C$. If $S_i = A^c_P(S_{i-1})$ for all $i > 0$, then $S_0 \subseteq S_1 \subseteq \ldots \subseteq S_n \subseteq \ldots$, that is, $S_0, S_1, \ldots, S_n, \ldots$ is a monotonic sequence.*

Proof By induction. First, $\emptyset \subseteq A^c_P(\emptyset)$, hence $S_0 \subseteq S_1$. Now, for any $k > 0$ it holds that $\mathcal{A}_b(S_k) = \mathcal{A}_b(\mathcal{A}_b(T^c_P(S_{k-1}))) = \mathcal{A}_b(T^c_P(S_{k-1})) = S_k$ by idempotency of $\mathcal{A}_b$ and the definition of A^c_P. From $\mathcal{A}_b(S_k) = S_k$, it follows from Proposition 8.2 that $S_k \subseteq A^c_P(S_k) = S_{k+1}$.

□

Thus, if we start off from an initial set $S_0 = \emptyset$, monotonicity of a sequence of A^c_P-applications is ensured. Therefore, we can define the ordinal powers of A^c_P as follows:

Definition 8.4

$$\begin{array}{lcl} A^c_P \uparrow 0 & = & \emptyset \\ A^c_P \uparrow i & = & A^c_P(A^c_P \uparrow i-1) \text{ for all } i > 0 \\ A^c_P \uparrow \omega & = & \bigcup_{n \in \mathbb{N}} A^c_P \uparrow n \end{array}$$

As long as no abstraction occurs, subsequent A^c_P-applications starting from an initial set $\emptyset$ derive sets of unit clauses. Abstraction introduces clauses in a set $A^c_P \uparrow i$ that are no longer unit clauses, but clauses capable of generating (among others) the same atoms as the abstracted ones, *provided the originals are present.* To that extent, we define two abbreviations, $\boldsymbol{\mathcal{R}es}_i$ and $\boldsymbol{\mathcal{G}en}_i$ that denote, respectively, the set of clauses $A \leftarrow \overline{B}$ that where removed during the computation of $A^c_P \uparrow i$ due to abstraction, and $\boldsymbol{\mathcal{G}en}_i$ comprising the abstractions $H \leftarrow H$ themselves. Formally:

Definition 8.5 *Given an abstract T^c_P-operator A^c_P for a definite program P and an abstraction function $\mathcal{A}_b$, define*

$$\boldsymbol{\mathcal{R}es}_0 = \boldsymbol{\mathcal{G}en}_0 = \emptyset$$

and

$$\begin{array}{rcl} \boldsymbol{\mathcal{R}es}_i & = & \boldsymbol{\mathcal{R}es}_{i-1} \cup (T^c_P(A^c_P \uparrow i-1) \setminus (A^c_P \uparrow i)) \\ \boldsymbol{\mathcal{G}en}_i & = & \boldsymbol{\mathcal{G}en}_{i-1} \cup ((A^c_P \uparrow i) \setminus T^c_P(A^c_P \uparrow i-1)) \end{array}$$

for all $i > 0$.

Obviously, all clauses in any $\boldsymbol{\mathcal{G}en}_i$ are of the form $H \leftarrow H$. Also note that $\boldsymbol{\mathcal{G}en}_i \subseteq A^c_P \uparrow i$ whereas $(\boldsymbol{\mathcal{R}es}_i \cap A^c_P \uparrow i) = \emptyset$.

Example 8.7 *Reconsider the definite program P and abstraction function A from Example 8.6. Then, we have that*

$$\begin{array}{ll} \boldsymbol{\mathcal{G}en}_0 = \emptyset & \boldsymbol{\mathcal{R}es}_0 = \emptyset \\ \boldsymbol{\mathcal{G}en}_1 = \{r(X) \leftarrow r(X)\} & \boldsymbol{\mathcal{R}es}_1 = \{r(a)\} \\ \boldsymbol{\mathcal{G}en}_2 = \{r(X) \leftarrow r(X)\} & \boldsymbol{\mathcal{R}es}_2 = \{r(a),\ r(f(X)) \leftarrow r(X)\} \end{array}$$

In order to be of practical use, the abstraction function $\mathcal{A}_b$ needs to be defined such that A^c_P is finitary. In other words, abstraction must guarantee that the least fixed point is reached after a finite number of A^c_P-applications: $A^c_P \uparrow \omega = A^c_P \uparrow n$ for some $n \in \mathbb{N}$. If A^c_P is indeed finitary and reaches its least fixed point in $n \in \mathbb{N}$, $\boldsymbol{\mathcal{R}es}_n$ and $\boldsymbol{\mathcal{G}en}_n$ are necessarily finite sets, since all sets $A^c_P \uparrow i$, $i \leq n$, are finite. We are now in a position to define the notion of a bottom-up partial deduction:

Definition 8.6 *Given a definite program P and an abstraction function $\mathcal{A}_b$, giving rise to a finitary abstracting T^c_P-operator, A^c_P, with least fixed point $A^c_P \uparrow n$, the* bottom-up partial deduction *of P with respect to $\mathcal{A}_b$ is the program*

$$A^c_P \uparrow n \setminus \boldsymbol{\mathcal{G}en}_n \cup \boldsymbol{\mathcal{R}es}_n$$

Example 8.8 *Reconsider the program P, abstraction function A and associated A^c_P-operator from Example 8.6. The sets $\boldsymbol{\mathcal{G}en}_i$ and $\boldsymbol{\mathcal{R}es}_i$ associated with subsequent applications of this A^c_P operator are found in Example 8.7. Since $A^c_P \uparrow 2$ is the operator's fixed point, the bottom-up partial deduction of P with respect to A*

is the program

```
p(a,a).     r(a).
q(a,a).     r(f(X)):- r(X).
```

In this section, we have defined the notion of a bottom-up partial deduction of a program P. In the following section, we investigate the semantical correctness of the transformation.

8.3 Correctness

If A^c_P is finitary, the bottom-up partial deduction of a definite program P is a finite set of clauses constituting the residual program P'. Since P' is built using an abstraction of the immediate consequence operator T^π_P (see Chapter 2), it is natural to express the correctness of the transformation with respect to the fixed point semantics. Correctness of the transformation with respect to the fixed point semantics implies correctness with respect to the operational semantics, due to their equivalence, expressed by Theorem 2.3. In order to formally state our correctness result, let us introduce the notions of *soundness* and *completeness* with respect to the fixed point semantics:

Definition 8.7 *Let P' denote a bottom-up partial deduction of a definite program P. Then, P' is* sound *with respect to P if $\mathcal{F}(P') \subseteq \mathcal{F}(P)$ and P' is* complete *with respect to P if $\mathcal{F}(P) \subseteq \mathcal{F}(P')$.*

Operationally, soundness states that P' does not compute answers that are not also computed by P whereas completeness states that every answer that is computed by the original program P is also computed by the transformed program P'. In what follows, we will show that a bottom-up partial deduction as defined in Definition 8.6 is complete. Without further measures, however, the transformation is strictly speaking not *sound*, since due to the bottom-up propagation, the residual program P' can compute answers that are *instances* of answers computed by P. Let us demonstrate this with the following example.

Example 8.9 *Let P be the following definite program:*

```
p(a,a).
p(X,c).
q(X,Y):- p(X,Y).
```

The fixed point semantics of the program defined in Example 8.9 is

$$\mathcal{F}(P) = \left\{\ p(a,a), p(X,c), q(a,a), q(X,c)\ \right\}.$$

Consider an abstraction function $\mathcal{A}_b$ that replaces the unit clause $p(a,a)$ by the tautology $p(a,Y) \leftarrow p(a,Y)$ and leaves all other clauses unchanged. Then,

$$A_P^c \uparrow \omega = A_P^c \uparrow 2 = \left\{ \begin{array}{l} p(a,Y) \leftarrow p(a,Y) \\ p(X,c) \\ q(a,Y) \leftarrow p(a,Y) \\ q(X,c) \end{array} \right\}$$

Since $\boldsymbol{\mathcal{R}es}_2 = \{p(a,a)\}$ and $\boldsymbol{\mathcal{G}en}_2 = \{p(a,Y) \leftarrow p(a,Y)\}$, the bottom-up partial deduction of P with respect to $\mathcal{A}_b$ is the following program, denoted by P':

```
p(a,a).      q(a,Y):- p(a,Y).
p(X,c).      q(X,c).
```

and we have that

$$\mathcal{F}(P') = \left\{ \; p(a,a), p(X,c), q(a,a), q(X,c), q(a,c) \; \right\}.$$

Note the presence of $q(a,c) \in \mathcal{F}(P')$, while $q(a,c) \notin \mathcal{F}(P)$. This atom is due to the presence of the clause $q(a,Y) \leftarrow p(a,Y)$ in the residual program of which the body atom unifies with the atom $p(X,c) \in \mathcal{F}(P')$ and imposing the most general unifier on the clause's head results in $q(a,c)$. The problem can be traced back to the abstraction $p(a,Y) \leftarrow p(a,Y)$ that is created for the atom $p(a,a)$. Combining this clause with the answers for p in $\mathcal{F}(P)$, that is $\{p(a,a), p(X,c)\}$, does *not* result in the same set of answers, but contains the more instantiated answer $p(a,c)$; effectively combining the information from two different answers. Strict soundness can be obtained by imposing an extra condition on a bottom-up partial deduction, ensuring that if $\mathcal{A}_b$ introduces a clause of the form $H \leftarrow H$, any atom $A \in \mathcal{F}(P')$ that unifies with H, is also an instance of H; that is $H \leq A$ for all such A. Indeed, if this is the case, a situation like in Example 8.9 can not occur, since it ensures that *if* the body atom of a generalisation unifies with an atom in $\mathcal{F}(P')$, it does not instantiate the atom any further.

Definition 8.8 *Let S denote a set of first-order formulas and $\mathcal{H}$ a finite set of tautologies. Then S is* inside closed *with respect to $\mathcal{H}$ if and only if for all $A \leftarrow \overline{B} \in S$ holds: if $H \leftarrow H \in \mathcal{H}$ such that $\theta = mgu(A,H)$ then $H\theta = A$.*

Reconsider the program from Example 8.9, and an abstraction function that now replaces the unit clause $p(a,a)$ by the tautology $p(X,Y) \leftarrow p(X,Y)$. Again $A_P^c \uparrow 2$ is the least fixed point of the associated A_P^c-operator, now defined as

$$A_P^c \uparrow 2 = \left\{ \begin{array}{l} p(X,Y) \leftarrow p(X,Y) \\ p(X,c) \\ q(X,Y) \leftarrow p(X,Y) \\ q(X,c) \end{array} \right\},$$

the associated bottom-up partial deduction being the program P' defined as follows:

```
p(a,a).      q(X,Y):- p(X,Y).
p(X,c).      q(X,c).
```

The bottom-up partial deduction P' is inside-closed with respect to the singleton set $\{p(X,Y) \leftarrow p(X,Y)\}$, and we have that

$$\mathcal{F}(P') = \mathcal{F}(P) = \left\{\ p(a,a), p(X,c), q(a,a), q(X,c)\ \right\}.$$

We will prove soundness and completeness of the bottom-up partial deduction transformation shortly, but first introduce a lemma that was proven in (Denis and Delahaye 1991) in the context of the relation between the procedural and the fixed point semantics. The lemma states the associativity of two subsequent resolution steps.

Lemma 8.1 *(From (Denis and Delahaye 1991)). Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be a set of positive literals, and $\mathcal{B}$ and $\mathcal{C}$ sets of definite clauses as follows:*

$$\mathcal{B} = \left\{ \begin{array}{l} B_1 \leftarrow L_{1,1}, \ldots, L_{1,m_1} \\ \vdots \\ B_n \leftarrow L_{n,1}, \ldots, L_{n,m_n} \end{array} \right\} \quad \mathcal{C} = \left\{ \begin{array}{l} C_{1,1} \leftarrow \overline{K}_{1,1} \\ \vdots \\ C_{1,m_1} \leftarrow \overline{K}_{1,m_1} \\ \vdots \\ C_{n,1} \leftarrow \overline{K}_{n,1} \\ \vdots \\ C_{n,m_n} \leftarrow \overline{K}_{n,m_n} \end{array} \right\}$$

Let us further assume that $\mathcal{V}(\mathcal{A})$, $\mathcal{V}(\mathcal{B})$ and $\mathcal{V}(\mathcal{C})$ are disjoint sets. Then, there exist substitutions α and β such that

$$(I) \left\{ \begin{array}{l} \alpha = mgu(\langle A_1, \ldots, A_n \rangle, \langle B_1, \ldots, B_n \rangle) \\ \beta = mgu(\langle L_{1,1}, \ldots, L_{n,m_n} \rangle\alpha, \langle C_{1,1}, \ldots, C_{n,m_n} \rangle) \end{array} \right.$$

if and only if there exist substitutions γ and δ such that

$$(II) \left\{ \begin{array}{l} \gamma = mgu(\langle L_{1,1}, \ldots, L_{n,m_n} \rangle, \langle C_{1,1}, \ldots, C_{n,m_n} \rangle) \\ \delta = mgu(\langle A_1, \ldots, A_n \rangle, \langle B_1, \ldots, B_n \rangle\gamma) \end{array} \right.$$

Moreover, if (I) or (II) holds then $\alpha\beta = \gamma\delta$ (modulo variable renaming).

Apart from using the above lemma directly, we also need the following corollary. It states that if the most general unifier of two sequences of atoms, $\langle A_1, \ldots, A_n \rangle$ and $\langle B_1, \ldots, B_n \rangle$ exist, so does the most general unifier of $\langle H_1, \ldots, H_n \rangle$ and $\langle B_1, \ldots, B_n \rangle$ if $\langle H_1, \ldots, H_n \rangle\alpha = \langle A_1, \ldots, A_n \rangle$ and the involved sequences have no variables in common. Formally:

Corollary 8.3 *Let* $\{A_1, \ldots, A_n\}$, $\{B_1, \ldots, B_n\}$ *and* $\{H_1, \ldots, H_n\}$ *be sets of atoms with* $\mathcal{V}(\{A_1, \ldots, A_n\})$, $\mathcal{V}(\{B_1, \ldots, B_n\})$ *and* $\mathcal{V}(\{H_1, \ldots, H_n\})$ *disjoint sets such that*

$$\langle A_1, \ldots, A_n \rangle = \langle H_1, \ldots, H_n \rangle \alpha$$

for some substitution α. *Then, if mgu*$(\langle A_1, \ldots, A_n \rangle, \langle B_1, \ldots, B_n \rangle)$ *exists, so does mgu*$(\langle H_1, \ldots, H_n \rangle, \langle B_1, \ldots, B_n \rangle)$.

Proof Assume $\beta = \text{mgu}(\langle A_1, \ldots, A_n \rangle, \langle B_1, \ldots, B_n \rangle)$. Since $\langle A_1, \ldots, A_n \rangle = \langle H_1, \ldots, H_n \rangle \alpha$, we have that $\alpha = \text{mgu}(\langle A_1, \ldots, A_n \rangle, \langle H_1, \ldots, H_n \rangle)$ and $\beta = \text{mgu}(\langle H_1, \ldots, H_n \rangle \alpha, \langle B_1, \ldots, B_n \rangle)$. If we denote with $\mathcal{A} = \{A_1, \ldots, A_n\}$, and

$$\mathcal{B} = \left\{ \begin{array}{l} H_1 \leftarrow H_1 \\ \vdots \\ H_n \leftarrow H_n \end{array} \right\} \quad \mathcal{C} = \left\{ \begin{array}{l} B_1 \leftarrow \\ \vdots \\ B_n \leftarrow \end{array} \right\}$$

the conditions of Lemma 8.1 are satisfied, and the existence of a substitution $\gamma = \text{mgu}(\langle H_1, \ldots, H_n \rangle, \langle B_1, \ldots, B_n \rangle)$ is immediate.

□

The following theorem formally states soundness of the transformation, given the inside closedness condition holds:

Theorem 8.1 *Let* P *be a definite program and* $\mathcal{A}_b$ *an abstraction function giving rise to a finitary abstracting* T_P^c*-operator* A_P^c *with least fixed point* $A_P^c \uparrow n$. *If* P' *denotes the bottom-up partial deduction of* P *with respect to* $\mathcal{A}_b$, *and if* P' *is inside closed with respect to* $\boldsymbol{Gen}_n$, *then*

$$\mathcal{F}(P') \subseteq \mathcal{F}(P).$$

In other words, P' *is sound with respect to* P.

Proof First, note that every iteration in the computation of $A_P^c \uparrow n$ represents a Horn clause program. Hence, we consider the bottom-up partial deduction process as the construction of a sequence of a programs $P_0, P_1, \ldots, P_n = P'$ which we define as follows:

$$\begin{array}{l} P_0 = \emptyset \\ P_i = A_P^c \uparrow i \setminus \boldsymbol{Gen}_i \cup \boldsymbol{Res}_i \end{array}$$

Since $A_P^c \uparrow n$ denotes the least fixed point of A_P^c, the residual program $P' = P_n$ and the result follows if we prove that $\mathcal{F}(P_k) \subseteq \mathcal{F}(P)$ for all $0 \leq k \leq n$. First, observe that $\boldsymbol{Res}_0, \boldsymbol{Res}_1, \ldots, \boldsymbol{Res}_n$ and $\boldsymbol{Gen}_0, \boldsymbol{Gen}_1, \ldots, \boldsymbol{Gen}_n$ are

monotonically increasing sequences. That is, for any $0 < k \leq n$, we have that $\boldsymbol{\mathcal{R}es}_{k-1} \subseteq \boldsymbol{\mathcal{R}es}_k$ and $\boldsymbol{\mathcal{G}en}_{k-1} \subseteq \boldsymbol{\mathcal{G}en}_k$. Hence, it follows that

$$\begin{array}{l} P_{k-1} \cup T_P^c(A_P^c \uparrow k-1) \\ = (A_P^c \uparrow k-1) \setminus \boldsymbol{\mathcal{G}en}_{k-1} \cup \boldsymbol{\mathcal{R}es}_{k-1} \cup T_P^c(A_P^c \uparrow k-1) \\ = \boldsymbol{\mathcal{R}es}_{k-1} \cup T_P^c(A_P^c \uparrow k-1) \\ = \boldsymbol{\mathcal{R}es}_{k-1} \cup A_P^c \uparrow k \setminus \boldsymbol{\mathcal{G}en}_k \cup (\boldsymbol{\mathcal{R}es}_k \setminus \boldsymbol{\mathcal{R}es}_{k-1}) \\ = A_P^c \uparrow k \setminus \boldsymbol{\mathcal{G}en}_k \cup \boldsymbol{\mathcal{R}es}_k \\ = P_k \end{array}$$

Thus, any intermediate program P_k (with $k > 0$) can be partitioned into P_{k-1} and clauses not in P_{k-1}, obtained by a T_P^c-application on elements of $A_P^c \uparrow k-1$. Now, denote $\mathbf{F} = \mathcal{F}(P) \cap \mathcal{F}(P')$ and consider the following construction:

$$\begin{array}{ll} T_{P_k}^F \uparrow 0 & = \mathbf{F} \\ T_{P_k}^F \uparrow n & = T_{P_k}^\pi(T_{P_k}^F \uparrow (n-1)) \cup (T_{P_k}^F \uparrow (n-1)) \\ T_{P_k}^F \uparrow \omega & = \bigcup_n (T_{P_k}^F \uparrow n) \end{array}$$

Observe that the constructed sequence is monotonic in n, so that $\mathbf{F} \subseteq T_{P_k}^F \uparrow n$, for all k and n. Also $\mathcal{F}(P_k) = T_{P_k}^\pi \uparrow \omega \subseteq T_{P_k}^F \uparrow \omega$, for all k, since their construction is the same, but starting from a larger initial set in the case of $T_{P_k}^F \uparrow \omega$. We show that for all k and n: $T_{P_k}^F \uparrow n \subseteq \mathbf{F}$ (or, equivalently: $T_{P_k}^F \uparrow n = \mathbf{F}$). As a consequence, we get: $\mathcal{F}(P_k) \subseteq T_{P_k}^F \uparrow \omega = \mathbf{F} = \mathcal{F}(P) \cap \mathcal{F}(P')$, so that $\mathcal{F}(P_k) \subseteq \mathcal{F}(P)$ follows. The proof of $T_{P_k}^F \uparrow n = \mathbf{F}$ is by double induction on k and n.

First, consider the case for $k = 0$. Then, $P_0 = \emptyset$ and for all $n > 0$:

$$\begin{array}{ll} T_{P_0}^F \uparrow n & = T_{P_0}^\pi(T_{P_0}^F \uparrow (n-1)) \cup T_{P_0}^F \uparrow (n-1) \\ & = \emptyset \cup T_{P_0}^F \uparrow (n-1) \end{array}$$

Also, $T_{P_0}^F \uparrow 0 = \mathbf{F}$. Thus, for all $n : T_{P_0}^F \uparrow n = \mathbf{F}$.

Next, assume that for some $k-1$, the hypothesis $T_{P_{k-1}}^F \uparrow n = \mathbf{F}$ holds for all n. We prove that also $T_{P_k}^F \uparrow n = \mathbf{F}$, for all n. Now, $T_{P_k}^F \uparrow 0 = \mathbf{F}$ by definition, which proves the base case. Next, assume that $T_{P_k}^F \uparrow (n-1) = \mathbf{F}$ holds as a hypothesis. We prove it for k and n. We have: $T_{P_k}^F \uparrow n = T_{P_k}^\pi(T_{P_k}^F \uparrow (n-1)) \cup T_{P_k}^F \uparrow (n-1)$. By the induction hypothesis on n, $T_{P_k}^F \uparrow (n-1) = \mathbf{F}$. Remains to be proven that $T_{P_k}^\pi(\mathbf{F}) \subseteq \mathbf{F}$. To prove this, recall that the clauses of P_k can be partitioned into

1. those clauses already obtained in P_{k-1}
2. clauses not in P_{k-1} that are obtained by a T_P^c application on elements of $A_P^c \uparrow k-1$.

If in $T^{\pi}_{P_k}(\mathbf{F})$ we use a clause of the first type, then by the induction hypothesis on k ($T^{F}_{P_{k-1}} \uparrow n = \mathbf{F}$ for all n, thus $T^{\pi}_{P_{k-1}}(\mathbf{F}) \subseteq \mathbf{F}$), the result is in $\mathbf{F}$. Remains the use of freshly introduced clauses in P_k. Any such clause, C, is of the form $(H \leftarrow \overline{L}_1, \ldots, \overline{L}_n)\alpha$, where there exists a clause $H \leftarrow B_1, \ldots, B_n$ in P, and clauses $H_1 \leftarrow \overline{L}_1, \ldots, H_n \leftarrow \overline{L}_n$ in $A^c_P \uparrow k-1$, and

$$\alpha = \mathrm{mgu}(\langle B_1, \ldots, B_n\rangle, \langle H_1, \ldots, H_n\rangle).$$

Now, let $\overline{D}$ be a sequence of atoms in $\mathbf{F}$ such that $\beta = \mathrm{mgu}(\overline{D}, \langle \overline{L}_1, \ldots, \overline{L}_n\rangle\alpha)$ exists. We need to prove that the corresponding element $H\alpha\beta$, obtained in $T^{\pi}_{P_k}(\mathbf{F})$ using this clause, is again an element of $\mathbf{F} = \mathcal{F}(P) \cap \mathcal{F}(P')$.

First, it is clear that $H\alpha\beta \in \mathcal{F}(P')$. To see this, note that $C \in P_k \subseteq P'$. Also, $T^{\pi}_{P'}(\mathcal{F}(P')) = \mathcal{F}(P')$. Thus, applying C to any element of $\mathcal{F}(P')$ gives an element of $\mathcal{F}(P')$.

Proving that $H\alpha\beta \in \mathcal{F}(P)$ is more complicated. Intuitively, we decompose the application of C to $\overline{D}$ into the application of the clauses of $A^c_P \uparrow k-1$ and the application of the clause of P. For the clauses of $A^c_P \uparrow k-1$, we use the induction hypothesis on $T^{F}_{P_{k-1}} \uparrow n$ together with the inside closedness. For the use of the clause of P, we use that $T^{\pi}_P(\mathcal{F}(P)) \subseteq \mathcal{F}(P)$. More specifically, using Lemma 8.1, we know that there exist substitutions γ and δ, such that $\gamma = \mathrm{mgu}(\overline{D}, \langle \overline{L}_1, \ldots, \overline{L}_n\rangle)$ and $\delta = \mathrm{mgu}(\langle H_1\gamma, \ldots, H_n\gamma\rangle, \langle B_1, \ldots, B_n\rangle)$. Moreover, $H\alpha\beta = H\gamma\delta = H\delta$. First, we show that $H_1\gamma, \ldots, H_n\gamma$ are in $\mathcal{F}(P)$. For those clauses $H_1 \leftarrow \overline{L}_1, \ldots, H_n \leftarrow \overline{L}_n$ which are in P_{k-1}, this follows immediately from the induction hypothesis on k, $T^{F}_{P_{k-1}} \uparrow n = \mathbf{F}$, for all n. For the clauses $H_i \leftarrow \overline{L}_i$ which are in $A^c_P \uparrow k-1 \setminus P_{k-1}$, we know that they are in $\boldsymbol{Gen}_{k-1}$. So, they are of the form $H_i \leftarrow H_i$. For such i, the corresponding $\overline{D}_i$ must be a sequence of just one atom, say E_i. Since $E_i \in \mathbf{F} = \mathcal{F}(P) \cap \mathcal{F}(P')$, it is an instance of the head of a clause in P'. It is also given that it unifies with the generalisation clause $H_i \leftarrow H_i$. Thus, by the inside closedness condition, E_i must be an instance of H_i. Therefore $H_i\gamma = E_i \in \mathbf{F}$. Remains to show that $H\delta$ is in $\mathcal{F}(P)$. This is obvious, since we apply a clause from P, $H_1\gamma, \ldots, H_n\gamma$ are in $\mathcal{F}(P)$ and $T^{\pi}_P(\mathcal{F}(P)) = \mathcal{F}(P)$.

□

The second correctness result states completeness of the residual program. Note that the condition of Theorem 8.2 is more general than the condition under which soundness was proven, since the proof of completeness does not require the bottom-up partial deduction to be inside closed.

Theorem 8.2 *Let P be a definite program and $\mathcal{A}_b$ an abstraction function giving rise to a finitary abstracting T^c_P-operator A^c_P with least fixed point $A^c_P \uparrow n$. If P'*

denotes the bottom-up partial deduction of P with respect to $\mathcal{A}_b$ then

$$\mathcal{F}(P) \subseteq \mathcal{F}(P').$$

In other words, P' is complete with respect to P.

Proof In order to prove that $\mathcal{F}(P) \subseteq \mathcal{F}(P')$, we prove that $T_P^\pi \uparrow k \subseteq \mathcal{F}(P')$ for every $k \in \mathbb{N}$. The result follows since $\mathcal{F}(P) = T_P^\pi \uparrow \omega$. The proof is by induction. Obviously, for $k = 0$ we have $T_P^\pi \uparrow 0 = \emptyset \subseteq \mathcal{F}(P')$. As induction hypothesis, we assume that $T_P^\pi \uparrow k \subseteq \mathcal{F}(P')$ for some k and prove that $T_P^\pi \uparrow k+1 \subseteq \mathcal{F}(P')$. In the proof of the induction step, due to the monotonicity of T_P^π, we need to consider only those atoms that are added in the $(k+1)$'th application of T_P^π; that is the set

$$(T_P^\pi \uparrow k+1) \setminus T_P^\pi \uparrow k = \left\{ H\delta \;\middle|\; \begin{array}{l} H \leftarrow B_1, \ldots, B_n \in P, \\ A_1, \ldots, A_n \in T_P^\pi \uparrow k, \\ \delta = \mathrm{mgu}(\langle B_1, \ldots, B_n \rangle, \langle A_1 \ldots, A_n \rangle), \\ H\delta \notin T_P^\pi \uparrow k \end{array} \right\}.$$

We prove that each such $H\delta \in \mathcal{F}(P')$. First, since $A_1, \ldots, A_n \in T_P^\pi \uparrow k$, we have by the induction hypothesis that $A_1, \ldots, A_n \in \mathcal{F}(P')$. This implies the existence of clauses $H_1 \leftarrow \overline{L}_1, \ldots, H_n \leftarrow \overline{L}_n \in P'$ and conjunctions of atoms $\overline{D}_1, \ldots, \overline{D}_n \in \mathcal{F}(P')$ such that

$$\gamma = \mathrm{mgu}(\langle \overline{D}_1, \ldots, \overline{D}_n \rangle, \langle \overline{L}_1, \ldots, \overline{L}_n \rangle) \tag{8.1}$$

and $\langle H_1, \ldots, H_n \rangle \gamma = \langle A_1, \ldots, A_n \rangle$. Now, since $H_1 \leftarrow \overline{L}_1, \ldots, H_n \leftarrow \overline{L}_n$ are clauses of P', there exist an $m \in \mathbb{N}$ such that some of these clauses are in $A_P^c \uparrow m$ and the others in $\boldsymbol{\mathcal{R}es}_m$. Let us assume that – possibly after renumbering – $H_1 \leftarrow \overline{L}_1, \ldots, H_k \leftarrow \overline{L}_k \in A_P^c \uparrow m$ and $H_{k+1} \leftarrow \overline{L}_{k+1}, \ldots, H_n \leftarrow \overline{L}_n \in \boldsymbol{\mathcal{R}es}_m$. Since the latter clauses were abstracted in the process of computing $A_P^c \uparrow m$, there also exist clauses $H'_{k+1} \leftarrow H'_{k+1}, \ldots, H'_n \leftarrow H'_n$ in $A_P^c \uparrow m$ and a substitution η such that

$$\langle H'_{k+1}, \ldots, H'_n \rangle \eta = H_{k+1}, \ldots, H_n. \tag{8.2}$$

Since $\delta = \mathrm{mgu}(\langle B_1, \ldots, B_n \rangle, \langle A_1, \ldots, A_n \rangle)$ and the latter sequence of atoms $\langle A_1, \ldots, A_n \rangle = \langle H_1, \ldots, H_n \rangle \gamma$, we have – due to Corollary 8.3 – that also $\mathrm{mgu}(\langle B_1, \ldots, B_n \rangle, \langle H_1, \ldots, H_n \rangle)$ exists and, again by Corollary 8.3, that also

$$\theta = \mathrm{mgu}(\langle B_1, \ldots, B_n \rangle, \langle H_1, \ldots, H_k, H'_{k+1}, \ldots, H'_n \rangle) \tag{8.3}$$

exists since $\langle H_1, \ldots, H_k, H'_{k+1}, \ldots, H'_n\rangle\eta = \langle H_1, \ldots, H_n\rangle$. Recall that

$$H \leftarrow B_1, \ldots, B_n \in P \text{ and } \left\{ \begin{array}{l} H_1 \leftarrow \overline{L}_1 \\ \vdots \\ H_k \leftarrow \overline{L}_k \\ H'_{k+1} \leftarrow H'_{k+1} \\ \vdots \\ H'_n \leftarrow H'_n \end{array} \right\} \subseteq A_P^c \uparrow m. \tag{8.4}$$

Together, (8.3) and (8.4) imply that $H\theta \leftarrow (\overline{L}_1, \ldots, \overline{L}_k, H'_{k+1}, \ldots, H'_{k+1})\theta \in T_P^c(A_P^c \uparrow m)$ and hence is a clause of P'. Now, from (8.2) follows the existence of

$$\eta' = \text{mgu}(\langle H'_{k+1}, \ldots, H'_n\rangle, \langle H_{k+1}\gamma, \ldots, H_n\gamma\rangle)$$

for the substitution γ from (8.1). Moreover, (8.1) implies the existence of

$$\gamma' = \text{mgu}(\langle \overline{L}_1, \ldots, \overline{L}_k\rangle, \langle \overline{D}_1, \ldots, \overline{D}_k\rangle).$$

Since $\mathcal{V}(H'_{k+1}, \ldots, H'_n)$, $\mathcal{V}(H_{k_1}\gamma, \ldots, H_n\gamma)$, $\mathcal{V}(\overline{L}_1, \ldots, \overline{L}_k)$, $\mathcal{V}(\overline{D}_1, \ldots, \overline{D}_k)$ are disjoint sets, η' and γ' can be composed into $\eta'\gamma'$ and

$$\eta'\gamma' = \text{mgu}(\langle \overline{L}_1, \ldots, \overline{L}_k, H'_{k+1}, \ldots, H'_n\rangle, \langle \overline{D}_1, \ldots, \overline{D}_k, H_{k+1}\gamma, \ldots, H_n\gamma\rangle).$$

From the fact that

$$\begin{array}{rl} \langle H_1, \ldots, H_k, H'_{k+1}, \ldots, H'_n\rangle\eta'\gamma' = & \langle H_1\gamma', \ldots, H_k\gamma', H_{k+1}\gamma\gamma', \ldots, H_n\gamma\gamma'\rangle \\ & = \langle H_1\gamma, \ldots, H_k\gamma, H_{k+1}\gamma, \ldots, H_n\gamma\rangle \\ & = \langle A_1, \ldots, A_n\rangle \end{array}$$

and $\delta = \text{mgu}(\langle B_1, \ldots, B_n\rangle, \langle A_1, \ldots, A_n\rangle)$ it follows that

$$\delta = \text{mgu}(\langle B_1, \ldots, B_n\rangle, \langle H_1, \ldots, H_k, H'_{k+1}, \ldots, H'_n\rangle\eta'\gamma').$$

Using the if-part of Lemma 8.1 with $\mathcal{A} = \{B_1, \ldots, B_n\}$, $\mathcal{B}$ being the the $A_P^c \uparrow m$-clauses of (8.4), and $\mathcal{C}$ being the set of unit clauses constructed from the atoms in

$$\{\overline{D}_1, \ldots, \overline{D}_k, H_{k+1}\gamma, \ldots, H_n\gamma\},$$

the existence of $\eta'\gamma'$ and δ implies the existence of θ – as in (8.3) – and

$$\sigma = \text{mgu}(\langle \overline{L}_1, \ldots, \overline{L}_k, H'_{k+1}, \ldots, H'_n\rangle\theta, \langle \overline{D}_1, \ldots, \overline{D}_k, H_{k+1}\gamma, \ldots, H_n\gamma\rangle).$$

Now, $\overline{D}_1, \ldots, \overline{D}_k \in \mathcal{F}(P')$ and $H_{k+1}\gamma, \ldots, H_n\gamma = A_{k+1}, \ldots, A_n \in \mathcal{F}(P')$ and

$$H\theta \leftarrow (\overline{L}_1, \ldots, \overline{L}_k, H'_{k+1}, \ldots, H'_n)\theta$$

is a clause of P'. Hence, it follows that $H\theta\sigma \in \mathcal{F}(P')$. Moreover, $H\theta\sigma = H\eta'\gamma'\delta$ by Lemma 8.1 and $H\eta'\gamma'\delta = H\delta$ since $H\eta'\gamma' = H$. Therefore, we have shown that $H\delta \in \mathcal{F}(P')$ which concludes the proof.

□

Combining the proofs of Theorems 8.1 and 8.2 immediately result in the following, stating that a bottom-up partial deduction preserves the fixed point semantics of a program, given the inside closedness condition hold.

Corollary 8.4 *Let P be a definite program and $\mathcal{A}_b$ an abstraction function giving rise to a finitary abstracting T_P^c-operator A_P^c with least fixed point $A_P^c \uparrow n$. If P' denotes the bottom-up partial deduction of P with respect to $\mathcal{A}_b$, and if P' is inside closed with respect to $\boldsymbol{Gen}_n$, then*

$$\mathcal{F}(P) = \mathcal{F}(P').$$

In other words, P' is sound and complete with respect to P.

Note that the combination of Corollary 8.4 with Theorem 2.3 – stating the equivalence between the fixed point semantics and operational semantics of a program – guarantees the preservation of computed answer substitutions. Apart from being important in its own right, this result enables the integration with a classical top-down partial deduction framework, since the latter's correctness is traditionally expressed in terms of preservation of computed answer substitutions.

Note how the inside closedness condition, required for soundness of bottom-up partial deduction, is related with the closedness condition as well as the independence condition in classical top-down partial deduction. Indeed, for a top-down partial deduction of a set of atoms $\mathcal{A}$ to be sound, each atom in the leaf of a partial SLD-tree must be an instance (closedness) of a single (independence) atom in $\mathcal{A}$, being the root point of a new partial SLD-tree. Together, closedness and independence ensure that if such an atom in the leaf of a partial SLD-tree unifies with the root of another SLD-tree, the former is an instance of the latter, which matches exactly the inside-closedness condition in a bottom-up context.

However, the combination of closedness and independence is stronger than the inside closedness condition, as they imply that the end point of a partial derivation unifies with exactly one starting point of a partial derivation, as such avoiding the duplication of answers. We can formally define the notion of *independence* in a bottom-up context as follows:

Definition 8.9 *A set S of tautologies is* independent *if and only if no elements of the set have a common instance.*

Imposing the independence condition on the set of tautologies with respect to which a bottom-up partial deduction is computed allows, by appropriate renaming,

to avoid the explicit inclusion of the tautologies during bottom-up computation altogether. Indeed, let $\mathcal{H}$ denote the set of tautologies associated to an abstraction function $\mathcal{A}_b$. For a set of clauses S, let S be partitioned into sets R and T such that T is the largest set $T \subseteq S$ for which $\mathcal{A}_b(T) = T$. Then we define the *renaming abstraction* of S as follows:

$$\mathcal{A}_b^r(T \cup R) = T \cup \left\{ \begin{array}{l|l} p(\overline{s}) \leftarrow p'(\overline{s}), & p(\overline{t}) \leftarrow \overline{B} \in R \text{ and} \\ p'(\overline{t}) \leftarrow \overline{B} & p(\overline{s}) \leftarrow p(\overline{s}) \in \mathcal{H} \text{ and } p(\overline{s}) \leq p(\overline{t}) \end{array} \right\}$$

where p' is a fresh predicate symbol with the same arity as p and uniquely associated to the clause $p(\overline{s}) \leftarrow p(\overline{s}) \in \mathcal{H}$. The renaming clauses $p(\overline{s}) \leftarrow p'(\overline{s})$ have the same functionality as the clauses from $\mathcal{H}$, with the exception that they do not introduce loops. The bottom-up partial deduction transformation can now be simulated by using $\mathcal{A}_b^r \circ T_P^c$ instead of $\mathcal{A}_b \circ T_P^c$. The main difference is that the abstractions of the type $p(\overline{s}) \leftarrow p'(\overline{s})$ do not need to be removed at the end of the transformation, nor do the abstracted clauses need to be added at the end.

Example 8.10 *Reconsider the program from Example 8.6 and the renaming abstraction $\mathcal{A}_b^r(S) = \bigcup_{C \in S} A^r(C)$ where A^r is defined as follows:*

$$A^r(r(a)) = \{\ r_1(a), r(X) \leftarrow r_1(X)\ \}$$

$$A^r(r(f(X)) \leftarrow r_1(X)) = \{\ r_1(f(X)) \leftarrow r_1(X), r(X) \leftarrow r_1(X)\ \}$$

Redoing the bottom-up partial deduction process from Example 8.6, now with the operator $(\mathcal{A}_b^r \circ T_P^c)$ result in the following sets:

$$I_0 = \emptyset$$

$$I_1 = (\mathcal{A}_b^r \circ T_P^c)(I_0) = \left\{ \begin{array}{l} p(a,a) \\ r(X) \leftarrow r_1(X) \\ r_1(a) \end{array} \right\}$$

$$I_2 = (\mathcal{A}_b^r \circ T_P^c)(I_1) = \left\{ \begin{array}{l} p(a,a) \\ q(a,a) \\ r(X) \leftarrow r_1(X) \\ r_1(a) \\ r_1(f(X) \leftarrow r_1(X)) \end{array} \right\}$$

and it can again be verified that I_2 is the least fixed point. The difference between I_2 and the program's bottom-up partial deduction as defined in Example 8.8 is that I_2 contains the extra (renaming) clause $r(X) \leftarrow r_1(X)$ linking the original predicate to the renamed one.

In this section, we have proven that a bottom-up partial deduction preserves the S-Herbrand semantics of the original program. This is an important result,

as it basically states that the transformed program computes the same answers as the original program. Another interesting property regarding correctness is the preservation of finite failure. In other words, will a query that finitely fails in the original program still finitely fail in the transformed program, or may it fail infinitely instead – and vice versa. Since our transformation is defined as an abstraction of the S-semantics, it seems natural to investigate the property of finite failure with respect to this semantics. However, we note that – to the best of our knowledge – a correct and fully abstract generalisation of the S-semantics modeling finite failure does not yet exist (Bossi, Gabbrielli, Levi, and Martelli 1994; Falaschi, Levi, Martelli, and Palamidessi 1989). We nevertheless conjecture the following:

Conjecture 8.1 *Let P be a definite program and $\mathcal{A}_b$ an abstraction function giving rise to a finitary abstracting T_P^c-operator A_P^c with least fixed point $A_P^c \uparrow n$. Let P' denotes the bottom-up partial deduction of P with respect to $\mathcal{A}_b$. If P' is inside closed with respect to $\boldsymbol{Gen}_n$ and $\boldsymbol{Gen}_n$ is independent, we have for any query Q that if $P \cup \{Q\}$ fails finitely, so does $P' \cup \{Q\}$.*

Motivation Independence of $\boldsymbol{Gen}$ makes that we can describe the bottom-up partial deduction process by an equivalent transformation $(\mathcal{A}_b^r \circ T_P^c) \uparrow n$. The difference between the latter and P' is that $(\mathcal{A}_b^r \circ T_P^c) \uparrow n$ contains clauses of the form $p(\overline{s}) \leftarrow p'(\overline{s})$ that link the original predicates to the renamed ones. This involves isolated one-step inferences in computations, which do not affect the termination of derivations.

Studying the preservation of finite failure for $(\mathcal{A}_b^r \circ T_P^c) \uparrow n$ is easier than for P', because it allows to reason over the individual T_P^c and $\mathcal{A}_b^r$ operations. Note that this is not that easy for P', because of the global operations of adding $\boldsymbol{Res}$ and removing $\boldsymbol{Gen}$ at the end of the transformation. Consider the set of all derivations for Q in P. T_P^c-application converts, at a finite number of points in the derivations, a sequence S of 2 derivation steps into a single one, say S'. Similarly, $\mathcal{A}_b^r$-application adds 1 resolution step to a subderivation S, resulting in S', at a finite number of points in the derivations. Note that abstraction may introduce a loop in the residual program, but, by construction, only a loop that is present in the original program as well. Hence, if S is a subsequence of a finitely failing derivation for Q in P, so is S' in P'. Note however that different selection rules may be needed in order to obtain S and S' in P, respectively P'.

Note that the converse of the above conjecture does not hold. Indeed, bottom-up partial deduction may convert *infinite* failure into *finite* failure, as the following example shows.

Example 8.11 *Consider a definite program P and a bottom-up partial deduction, P' of P:*

P: `q(a).` `q(X):- q(X).` P': `q(a).`

The query $q(b)$ fails infinitely in P, whereas it fails finitely in the transformed program P'.

Hence, correctness with respect to finite failure is weaker for the bottom-up transformation than for the top-down partial deduction transformation since in the later case finite failure is preserved either way (see Theorem 2.4). However, the conversion of infinite failure into finite failure – a natural effect of a transformation's bottom-up nature – is generally considered advantageous and also obtainable by other bottom-up based transformations like for example the most specific program transformation of (Leuschel and Schreye 1996).

8.4 Controlling Bottom-up Partial Deduction

In the previous sections, we have introduced the notion of a bottom-up partial deduction, and have proven the transformation to be correct: the fixed point semantics of the bottom-up partial deduction of a program equals the original program's fixed point semantics. In what follows, we devise a concrete algorithm to compute, given a program, such a bottom-up partial deduction. Although some design choices are hardwired in the algorithm, it is still a generic algorithm in the sense that, like the algorithm for top-down partial deduction presented in Chapter 2, it can be instantiated with a particular implementation of abstraction.

In top-down partial deduction, the constructed partial derivations (in the form of partial SLD-trees) are most commonly ordered in a tree structure rather than in a set (Martens and Gallagher 1995; Sørensen and Glück 1995; Leuschel, Martens, and De Schreye 1998). The tree structure makes the relation explicit that exist between the partial derivations, which can be employed during abstraction – in general resulting in much more precision than in set-based approaches like (Gallagher and Bruynooghe 1991; Gallagher 1993). To enable the formulation of a sufficiently precise abstraction function, we will also make the relations that exist between the partial bottom-up derivations explicit by ordering them in a structure. Contrary to the top-down setting, where the end point of a partial derivation can only be the starting point of a single new partial derivation, the end point of a partial bottom-up derivation can be used to create several new derivations. Hence, we order the partial derivations in a directed acyclic graph (DAG). We will denote such a graph by a pair (V, E) where V denotes the set of nodes of the graph, and $E \subseteq V \times V$ the set of edges between the graph's nodes. Given a directed graph (V, E), we use the notation $a \rightarrow b$ to denote an edge in E between $a, b \in V$, where

the arrow shows the direction associated to the edge. We define the notion of a *path* in such a graph as usual: given a directed acyclic graph (V, E) and nodes $v, v' \in V$, we say that there is a path from v to v' in (V, E) if $v \to v' \in E$ or there exist a node $w \in V$ such that $v \to w \in E$ and there is a path from w to v' in (V, E). An important operation on a directed acyclic graph (V, E) is the addition of a new node v together with a number of edges that connect elements of V with the new node v:

Definition 8.10 *Given a directed graph (V, E), a node $v \notin V$ and a set of edges $E' \subseteq V \times \{v\}$, the* update *of (V, E) with respect to v and E' is denoted by $(V, E) \cup (v, E')$ and defined as*

$$(V, E) \cup (v, E') = (V \cup \{v\}, E \cup E').$$

Since the update does only add edges connecting nodes of the existing graph with the newly added node, the resulting graph remains directed and acyclic:

Proposition 8.3 *Given a directed acyclic graph (V, E), a node $v \notin V$ and a set of edges $E' \subseteq V \times \{v\}$, $(V, E) \cup (v, E')$ is a directed acyclic graph.*

Note that if $\{v_1, \ldots, v_n\}$ is a set of nodes such that $v_i \neq v_j$ for any i and j, $i \neq j$ and if for each such v_i, $E_i \subseteq V \times \{v_i\}$, the order in which the updates $(v_1, E_1), \ldots, (v_n, E_n)$ are performed upon a graph (V, E) – with $\{v_1, \ldots, v_n\} \cap V = \emptyset$ is irrelevant. Hence we extend the notion of an update to a set of such nodes with corresponding sets of edges in a straightforward way:

$$(V, E) \cup \{(v_1, E_1), \ldots, (v_n, E_n)\} = (V \cup \{v_1, \ldots, v_n\}, E \cup E_1 \cup \ldots \cup E_n)$$

which is, again, a directed acyclic graph.

The general idea is then to construct, during bottom-up partial deduction, such a directed acyclic graph in which the nodes are labelled with the clauses that are derived by the process. In what follows we denote with *DHC* the set of such labelled directed acyclic graphs. Given such a labelled graph $(V, E) \in \mathit{DHC}$ and a node $v \in V$, we denote with $\mathit{label}(v)$ the clause associated to the node v. The set of clauses that are present in the graph is denoted by $\mathit{Cl}((V, E))$ and defined as

$$\mathit{Cl}((V, E)) = \{\mathit{label}(v) \mid v \in V\}.$$

Note that there is not necessarily a one-to-one correspondence between the set of nodes V and the set of clauses $\mathit{Cl}((V, E))$ since different nodes in V might be labelled with the same clause (modulo variance). That is, there may exist $v, v' \in V$, $v \neq v'$ such that $\mathit{label}(v) \approx \mathit{label}(v')$. Construction of a such a labelled graph can be described by a variant of the T_P^c-operator: it derives a new clause exactly in the same way as T_P^c does, the only difference being that the newly derived clause is entered in the graph as the label of a newly introduced node v, together with an edge $v_i \to v$ if the clause labelling v_i was used in the derivation.

Definition 8.11 *Let P be a definite program and (V, E) a labelled directed acyclic graph, that is $(V, E) \in DHC$. Then we define $T_P^d : DHC \mapsto DHC$ as follows:*

$$T_P^d((V,E)) = (V,E) \cup \left\{ (v, E') \;\middle|\; \begin{array}{l} v_1, \ldots, v_n \in V, H \leftarrow B_1, \ldots, B_n \in P \\ A_1 \leftarrow \overline{L}_1, \ldots, A_n \leftarrow \overline{L}_n \text{ renamed apart} \\ \text{variants of } label(v_1), \ldots, label(v_n) \\ \theta = mgu((B_1, \ldots, B_n), (A_1, \ldots, A_n)) \\ v \text{ a new node} \notin V \text{ with} \\ label(v) = (H \leftarrow \overline{L}_1, \ldots, \overline{L}_n)\theta \\ \text{and } E' = \{v_1 \rightarrow v, \ldots, v_n \rightarrow v\} \end{array} \right\}$$

The relation between T_P^d and T_P^c is obvious from the definition of T_P^d and formally stated in the following proposition:

Proposition 8.4 *for any labelled graph (V, E), we have*

$$Cl(T_P^d((V,E))) = T_P^c(Cl((V,E))).$$

If we order labelled graphs according to the $\subset$ relation on the sets of their labels, that is $(V_1, E_1) \subset (V_2, E_2)$ if and only if $Cl((V_1, E_2)) \subset Cl((V_2, E_2))$, monotonicity and continuity of T_P^d is immediate given the relation with T_P^c. Hence, we define its ordinal powers as usual, and we have existence of its least fixed point.

Example 8.12 *Consider for example an extension of the abstract data type example from the introduction (Example 8.1). Let P denote the following program, comprising the definitions of* `append/3` *and* `reverse/3` *employing an abstract data type for list representation, together with the definition of the abstract data type.*

```
append(X,Y,Y):-list_nil?(X).
append(X,Y,Z):-list_not_nil?(X),
               list_head(X,H), list_tail(X,T),
               append(T,Y,R), list_cons(H,R,Z).
reverse(L,A,A):-list_nil?(L).
reverse(L,A,R):-list_not_nil?(L),
                list_head(L,H), list_tail(L,T),
                list_cons(H,A,NA), reverse(T,NA,R).
list_nil?([]).        list_head([X|_], X).    list_cons(X,Xs, [X|Xs]).
list_not_nil?([_|_]). list_tail([_|Xs], Xs).
```

The first two labelled graphs constructed by T_P^d are depicted in Fig. 8.2, the third one in Fig. 8.3. In order to obtain an algorithm that computes a bottom-up partial deduction of a program, the immediate consequence operator T_P^d needs to be combined with a suitable abstraction function. Operationally, the task of abstraction is to create a *finite* labelled graph. In order to obtain an algorithm that is – to some extent – generic and comparable with the generic algorithm for

Figure 8.2: The first two DAGs constructed: $T_P^d \uparrow 1$ and $T_P^d \uparrow 2$.

top-down partial deduction from Chapter 2, we parametrise the algorithm with respect to the abstraction function and represent the latter by means of the *covered*, *whistle* and *abstract* operations. Note that in the present setting of bottom-up partial deduction, these operations are defined on (the nodes of) a labelled graph, rather than a labelled tree. Their meanings, however, remain much the same. The task of the predicate $covered(v, (V, E))$ is to check whether the set of clauses associated to the nodes in V are an abstraction of the singleton set $\{label(v)\}$. If this is the case, it is not necessary to use $label(v)$ for subsequent bottom-up derivations, since either a variant of $label(v)$ is already present in the graph, or a clause with a more general head is labelling a node in V. Note that the *covered* predicate plays the same role as its top-down counterpart: it prevents the creation of a new partial derivation using the head atom of $label(v)$, since this derivation is covered by a more general one already present in the graph. Recall that the role of abstraction is to keep the graph under construction finite. The task of the *whistle* function is to detect whether further processing a node in the graph may eventually lead to an infinite path in the graph. To that extent, $whistle(v, (V, E))$ must be defined such that it returns a node $w \in V$ if there is a path from w to v in (V, E) and w is not admissible as a predecessor of v if the graph must be guaranteed to be finite. Otherwise, $whistle(v, (V, E))$ returns *fail*. Note again the similarity with the top-down setting, where the *whistle* function "sounds the alarm" in case adding a node to a branch of the labelled tree no longer guarantees that a finite branch of the tree is constructed. Like in the top-down case, the *whistle*

Figure 8.3: The third DAG constructed: $T_P^d \uparrow 3$.

function is used to indicate what (end points of) derivations will be abstracted into more general starting points for new derivations. To that end, the operation $abstract(v, w)$ computes an abstraction of $label(v)$ and $label(w)$ (v and w being nodes of V) which is used in our algorithm to replace the label of the node v if $whistle(v, (V, E)) = w$. We are now in a position to define the generic algorithm for bottom-up partial deduction. We will later on derive a particular instance of this algorithm by providing concrete instantiations of the above mentioned operations and prove termination of the particular algorithm.

Algorithm 8.1
Input: *A definite program* P.
Output: *Finite sets of clauses,* $\mathcal{H}$ *and* $\mathcal{R}$*, and the residual program* P'.
Initialise: *A* DAG $(V_0, E_0) = (\emptyset, \emptyset)$, $\mathcal{H} = \emptyset$, $\mathcal{R} = \emptyset$ *and* $i = 0$
  repeat
    $(V', E') = T_P^d((V_i, E_i))$, *the nodes* $(V' \setminus V_i)$ *are marked* unprocessed
    repeat
      pick *a node* $v \in V'$ *marked* unprocessed
        if $covered(v, (V_i, E_i))$
        then
          $(V', E') = (V' \setminus \{v\}, E' \setminus \{w \rightarrow v \mid w \in V_i\})$
          $\mathcal{R} = \mathcal{R} \cup \{label(v)\}$
        else
          $w = whistle(v, (V', E'))$
          if $w \neq fail$
            then
              $\mathcal{R} = \mathcal{R} \cup \{label(v)\}$
              label(v):=abstract(w,v)
              $\mathcal{H} = \mathcal{H} \cup \{label(v)\}$
            else *mark* v *as* processed
    until *there are no* unprocessed *nodes*
    $(V_{i+1}, E_{i+1}) = (V', E')$
    $i = i + 1$
  until $(V_i, E_i) = (V_{i-1}, E_{i-1})$
Output: $\mathcal{H}$, $\mathcal{R}$ *and* $P' = Cl((V_i, E_i)) \setminus \mathcal{H} \cup \mathcal{R}$

Given the operations *covered*, *whistle* and *abstract*, the implementation of the algorithm is straightforward. In each round, a new DAG is computed, starting from the DAG computed during the previous iteration. The sets $\mathcal{R}$ and $\mathcal{H}$ contain respectively the clauses that are removed due to the presence – or introduction – of a generalisation and the generalisations themselves. First, the old DAG is extended with nodes labelled by the clauses derived by T_P^d. Each of these nodes is marked as *unprocessed*. Then, as long as there are unprocessed nodes, the algorithm selects one such node, v, and performs the following operations: if the clause associated to the node is covered by a generalisation in the old graph, the node is removed

(and put in $\mathcal{R}$). If it is not covered, but there is a predecessor node w in the graph such that extending the path between w and v may possibly lead to an infinite path (indicated by the *whistle* function), the label of the node v is replaced by a generalisation. If the node v has no such predecessor, it remains in the newly constructed graph and is marked as *processed*. If all nodes are marked *processed*, the algorithm starts over with a new iteration round, until two subsequent rounds result in the same graph. Hence, the algorithm terminates if every newly by T_P^d added node carries a label that is either already present in the graph, or covered by an earlier generalisation. Algorithm 8.1 constructs a sequence of labelled graphs $(V_0, E_0), (V_1, E_1), \ldots$ The link between two such subsequent graphs is formalised in the following proposition.

Proposition 8.5 *If $(V_0, E_0), (V_1, E_1), \ldots$ denotes a sequence of labelled graphs as constructed by Algorithm 8.1, then we have for each $i \geq 0$ that*

$$Cl((V_{i+1}, E_{i+1})) \text{ is an abstraction of } Cl(T_P^d((V_i, E_i)))$$

and that $Cl((V_i, E_i)) \subseteq Cl((V_{i+1}, E_{i+1}))$.

Proof Let us define, like in Algorithm 8.1, $(V', E') = T_P^d((V_i, E_i))$ (for some $i > 0$). First, note that $V_i \subseteq V'$ and since V_{i+1} is constructed from V' only by manipulating the nodes $V' \setminus V_i$, leaving the other nodes intact, we have that $Cl((V_i, E_i)) \subseteq Cl((V_{i+1}, E_{i+1}))$.

Now, let v be a node of V'. We have to prove that there exist a node $w \in V_{i+1}$ such that $label(w) = label(v)$ or $label(w) = H \leftarrow H$ and $H \leq A$ if $label(v) = A \leftarrow \overline{B}$. If $v \in V_i \subseteq V'$, take for w the node v itself, and we have that $label(w) = label(v)$. Remains to consider how the nodes of $(V' \setminus V_i)$ are manipulated by the algorithm. One possibility is that such a node v is simply marked *processed*, in what case $v \in V_{i+1}$ and the proposition holds. A second possibility is that such a node v is removed due to the fact that $covered(v, (V_i, E_i))$. But by definition of the *covered* predicate, this implies the existence of a node $w \in V_i$ (and hence $w \in V_{i+1}$) such that either $label(w) = label(v)$ or $label(w) = H \leftarrow H$ and $H \leq A$ if $label(v) = A \leftarrow B$. The last possibility we need to consider is that the node v is assigned another label. But by definition of the *abstract* operation, the new label is a generalisation, its head being more general than the head of $label(v)$ and the proposition holds.

□

The following result formally states that the residual program computed by Algorithm 8.1 is a bottom-up partial deduction.

Theorem 8.3 *Assume suitable concretisations of the covered, whistle and abstract operations given. Then, if Algorithm 8.1 terminates with a definite program P as input, its output P' is a bottom-up partial deduction of P.*

Proof If Algorithm 8.1 terminates, it constructs a finite sequence of directed acyclic labelled graphs $(V_0, E_0), (V_1, E_1), \ldots, (V_n, E_n)$ such that $(V_n, E_n) = (V_{n-1}, E_{n-1})$. Now, $Cl((V_0, E_0)) = \emptyset$, and for $0 < i \leq n$ we have that $Cl((V_{i-1}, E_{i-1})) \subseteq Cl((V_i, E_i))$ by Proposition 8.5. Moreover, by the same proposition, the set $Cl((V_i, E_i))$ is an abstraction of $Cl(T_P^d((V_{i-1}, E_{i-1}))) = T_P^c(Cl((V_{i-1}, E_{i-1}))$ – the latter due to Proposition 8.4. Since a clause $label(v)$ is only considered for abstraction if $\neg covered(v, (V_i, E_i))$, variants of a single clause are never abstracted differently and consequently, the sequence $Cl((V_0, E_0)), Cl((V_1, E_1)), \ldots, Cl((V_n, E_n))$ represents the result of repeatedly applying an A_P^c operator starting from an empty set with $Cl((V_n, E_n))$ being its fixed point. Now, since $\mathcal{H} = \boldsymbol{\mathcal{G}en}_n$ and $\mathcal{R} = \boldsymbol{\mathcal{R}es}_n$ by definition, we have that $P' = Cl((V_n, E_n)) \setminus \mathcal{H} \cup \mathcal{R}$ is a bottom-up partial deduction of P.

□

Algorithm 8.1 represents a generic algorithm to compute a bottom-up partial deduction of a program. Concretisations of the algorithm are obtained by instantiating the *covered*, *whistle* and *abstract* operations with appropriate concrete operations such that the algorithm terminates. One such concretisation is the following, which we denote with $\mathcal{B}_{\trianglelefteq}$ and implements a control strategy based on the homeomorphic embedding relation, much similar to a popular control scheme used for top-down partial deduction. Formally, $\mathcal{B}_{\trianglelefteq}$ is defined as follows:

Definition 8.12 *Let $\mathcal{B}_{\trianglelefteq}$ denote an instance of Algorithm 8.1, instantiated with the following operations:*

- *Given a node v with $label(v) = A \leftarrow \overline{B}$ and a labelled graph (V, E), define covered such that $covered(v, (V, E)) = true$ if there exist a node $w \in V$ with $label(w) = label(v)$ or $label(w) = H \leftarrow H$ and $H \leq A$. Otherwise, $covered(v, (V, E)) = false$.*
- *Given a node v with $label(v) = A \leftarrow \overline{B}$ and a labelled graph (V, E) with $v \in V$, define whistle such that $whistle(v, (V, E)) = w$ if there exist a path from w to v in (V, E), $label(w) = A' \leftarrow \overline{B}'$ and $A' \trianglelefteq A$. Otherwise, $whistle(v, (V, E)) = fail$.*
- *Given two nodes v and w such that $label(v) = A \leftarrow \overline{B}$ and $label(w) = A' \leftarrow \overline{B}'$, define abstract such that $abstract(w, v) = H \leftarrow H$ with $H = msg(A, A')$.*

Note that $\mathcal{B}_{\trianglelefteq}$ implements a control scheme that is inspired by a popular scheme to control the partial deduction process in a top-down setting (see Chapter 2). In a bottom-up context, the labelled graph is kept finite by ensuring that the sequence of labels associated to the nodes on each (acyclic) path in the graph are well-quasi ordered. If a path is extended by a node, the label of which destroys the well-quasi

orderedness of the path, the node is replaced by a generalisation such that the path again becomes well-quasi ordered. The generalisation is implemented through the *most specific generalisation* of the (head atoms) of the embedded clauses.

Example 8.13 *Reconsider the program P defined in Example 8.12. Computing a bottom-up partial deduction of P by the algorithm $\mathcal{B}_{\trianglelefteq}$ proceeds as follows. No abstraction occurs during the first two iteration rounds, and hence the first two constructed graphs simply equal those constructed by the T_P^d-operator which were depicted in Fig. 8.2. Applying the T_P^d-operator for the third time again results in the graph that was depicted in Fig. 8.3, but now we have that*

$$append([], Y, Y) \trianglelefteq append([H], Y, [H|Y])$$
$$reverse([], A, A) \trianglelefteq reverse([H], A, [H|A])$$

Consequently, the whistle blows on each of the unit clauses added by this T_P^d-application and each of the clauses is replaced by a tautology that is constructed from the msg of the involved unit clauses' heads.

$$msg(append([], Y, Y), append([H], Y, [H|Y])) = append(X, Y, Z)$$
$$msg(reverse([], A, A), reverse([H], A, [H|A]) = reverse(X, Y, Z)$$

The resulting graph is depicted in Fig. 8.4. During the fourth iteration of $\mathcal{B}_{\trianglelefteq}$, the

Figure 8.4: The graph constructed in the third iteration round of $\mathcal{B}_{\trianglelefteq}$.

application of the T_P^d-operator results in several new clauses that are added to the graph, which is depicted in Fig. 8.5. Only the interesting subparts of the graph are shown. Each of the newly constructed clauses (underlined in Fig. 8.5) is covered by the earlier generalisation (indicated by the dashed arrows), so each of the clauses that were added in this round is again removed (or rather moved to the set $\mathcal{R}$). The net result is a graph that is equal to the graph depicted in Fig. 8.4 and a fixed point is reached by the algorithm. The bottom-up partial deduction of P computed by $\mathcal{B}_{\trianglelefteq}$ is the following program of which only the predicates of interest – `append/3` *and* `reverse/3` *– are given.*

Figure 8.5: The graph constructed by T_P^d during the fourth iteration of $\mathcal{B}_{\trianglelefteq}$.

```
append([],Y,Y).
append([H],Y,[H|Y]).
append([H|T],Y,[H|R]):- append(T,Y,R).

reverse([],A,A).
reverse([H],A,[H|A]).
reverse([H|T],A,R):- reverse(T,[H|A],R).
```

We conclude this section on concrete control for bottom-up partial deduction by proving that the algorithm $\mathcal{B}_{\trianglelefteq}$ terminates for any definite program P.

Theorem 8.4 *Let P be a definite program. The algorithm $\mathcal{B}_{\trianglelefteq}$ terminates with P as input.*

Proof Let us first concentrate on the algorithm's outermost loop. Assume that the algorithm $\mathcal{B}_{\trianglelefteq}$ constructs the sequence $(V_0, E_0), (V_1, E_1), (V_2, E_2), \ldots$ of labelled graphs but does not terminate because there does not exist $n \in \mathbb{N}$ such that $(V_n, E_n) = (V_{n-1}, E_{n-1})$. First of all, observe that this sequence is monotonically increasing, since $Cl((V_0, E_0)) \subseteq Cl((V_1, E_1)) \subseteq Cl((V_2, E_2)) \subseteq \ldots$ due to Proposition 8.5. Consequently, if there does not exist such $n \in \mathbb{N}$, it is because the algorithm constructs an infinite graph, say (V, E). Since a finite number of nodes is added to the graph in each round, the graph (V, E) can become infinite only through an infinite path. In what follows we show that in each iteration of the algorithm's outermost loop, the innermost loop terminates and constructs a graph in which the sequence of labels associated to each path is well-quasi ordered. The fact that (the sequence of labels associated to) each such path is well-quasi ordered contradicts the existence of an infinite path in a graph constructed by the algorithm, and hence implies the existence of $n \in \mathbb{N}$ such that $(V_n, E_n) = (V_{n-1}, E_{n-1})$. To ease formulation of the proof, we say that a DAG is well-quasi ordered if each of its paths is well-quasi ordered, and that such a path is well-quasi ordered if the sequence of labels associated to its nodes is well-quasi ordered.

We prove well-quasi orderedness of the constructed graphs by induction on the number of graphs that is created. The base case, (V_0, E_0) is proven immediately, since $(V_0, E_0) = (\emptyset, \emptyset)$ is well-quasi ordered by definition. Next, assume that (V_i, E_i) is well-quasi ordered. We prove that the construction of (V_{i+1}, E_{i+1}) terminates and the resulting graph is well-quasi ordered. Construction of (V_{i+1}, E_{i+1}) starts from a graph (V', E') in which only a finite number of nodes is marked *unprocessed*. In each round of the innermost loop, one of this *unprocessed* nodes is either removed, marked *processed*, or its label is replaced by a generalisation. Removing a node or marking it *processed* results in one less (*unprocessed*) node to consider in the next iteration – a necessarily terminating process. Hence, since a node remains marked *unprocessed* only in case its label is abstracted, the only possible source of nontermination of the loop is the occurrence of an infinite sequence of consecutive generalisations. Assume there is a node v such that $w = \mathit{whistle}(v, (V', E'))$ and $\mathit{label}(v) = A \leftarrow \overline{B}$ and $\mathit{label}(w) = A' \leftarrow \overline{B}'$. By definition of the algorithm, if $\mathit{label}(v)$ is abstracted, we have that $\mathit{covered}(v, (V_i, E_i)) = \mathit{false}$ and $w = \mathit{whistle}(v, (V', E'))$ implies that $A' \lhd A$. Hence, if abstraction is performed, it is strict in the sense that $\mathit{msg}(A', A) < A$. Since strict abstraction is well-founded, no infinite sequence of consecutive generalisations can exist. This proves termination of the innermost loop, resulting in a graph (V_{i+1}, E_{i+1}) in which all nodes are marked *processed*. Now, note that (V_{i+1}, E_{i+1}) is an update of (V_i, E_i); it is constructed by adding a number of nodes to (V_i, E_i) and extending some of the paths in (V_i, E_i) by an edge connecting a node in V_i with a newly added node. The nodes in $V_{i+1} \setminus V_i$ are those marked *processed* by the innermost loop of the algorithm. By definition, such a node v is marked *processed* only if $\mathit{whistle}(v, (V', E')) = \mathit{fail}$ which is, by definition of $\mathcal{B}_{\trianglelefteq}$, only the case if each path in (V_{i+1}, E_{i+1}) ending in v is well-quasi ordered. Hence, the resulting graph is well-quasi ordered.

□

8.5 Discussion

We conclude this chapter with a fairly elaborated discussion. In a first part, we discuss in detail the impact of abstraction on the information propagation achieved by bottom-up partial deduction. As a running example, we reconsider the partial deduction of the vanilla meta interpreter from Chapter 7 and we demonstrate that the combination of bottom-up partial deduction with top-down partial deduction (both using straightforward control mechanisms) looks promising as it is capable of removing the meta interpretation overhead from the vanilla interpreter. The resulting blend of top-down and bottom-up partial deduction is conceptually cleaner and requires a much less sophisticated control of the process(es) than a uniquely

top-down approach as was illustrated in Chapter 7. In the second part of the discussion, we briefly return to some issues regarding the proposed framework and control of partial deduction. We discuss its relation with other work and touch some outstanding problems for future research.

8.5.1 Mixing Bottom-up and Top-down Control

In the previous sections, we have presented a novel technique to specialise a logic program. In traditional top-down partial deduction, a logic program is specialised with respect to a query using top-down SLD-resolution. Our framework allows specialising a logic program the other way round, using bottom-up resolution to obtain information propagation. Such a technique is useful when one does not want to incorporate information from a query into the program, but rather propagate information upwards, starting from the unit clauses. An example is the specialisation of a library with respect to an abstract data type, as in Example 8.12. Propagating the concrete list representation into the library's predicates can also be obtained using top-down partial deduction, but requires in general several partial deductions to be computed: one for every predicate p with respect to a most general query $p(\overline{X})$.

Apart from being an alternative to top-down partial deduction when no query at all (or several queries not containing any information at all) is provided, bottom-up partial deduction can achieve information propagation that is hard to achieve with a general and automatic top-down partial deducer. In order to avoid non-termination, a partial deducer (whether top-down or bottom-up) generally halts the creation of a partial derivation when structure is growing between some successive derivation steps. The fact whether the derivation is created top-down or bottom-up determines the kind of "problematic" structure handling. Indeed, structure that is growing between successive top-down derivation steps – likely causing a top-down partial deducer to halt the derivation – is shrinking between successive bottom-up derivation steps – enabling the bottom-up partial deducer to continue the derivation. Consequently, in those cases in which a bounded growing of structure causes a top-down specialiser to preliminary abort the creation of a partial derivation, thereby loosing specialisation opportunities, a bottom-up partial deducer might be capable of creating a longer derivation, since the particular structure – which is shrinking from a bottom-up viewpoint – does not cause any problems. Reconsider Example 8.3 from this chapter's introduction. Performing bottom-up partial deduction on this example using $\mathcal{B}_{\trianglelefteq}$ results in the graphs depicted in Fig. 8.6. No structure growing at all is detected (using $\trianglelefteq$) during the creation of the derivations, and hence the bottom-up partial deduction effectively computes the program's fixed point semantics and the predicate `make_list/3` is specialised with respect to the two types of lists:

Figure 8.6: The graphs constructed by $\mathcal{B}_{\trianglelefteq}$ for Example 8.3

```
make_list(list1,I,[I]).
make_list(list3,I,[I,I,I]).
```

However, the reverse observation also holds: structure that is shrinking between successive derivation steps during top-down specialisation, is growing between successive bottom-up derivation steps and is hence problematic to handle by a bottom-up partial deducer. An example is the parsing clause of the vanilla meta interpreter (see Chapter 7).

$$solve((A, B)) \leftarrow solve(A), solve(B).$$

When this kind of clause is used the other way round during a bottom-up derivation, the argument in the newly derived instance of the head is always growing with respect to the arguments of both atoms unifying with the clause body, meaning that – in order to guarantee termination – the former most likely can not be used for further derivations.

As we have discussed in Chapter 7, some programs contain so-called fluctuating structure – structure that can grow or shrink between successive derivation steps. Meta interpreters like the vanilla interpreter are known to show this characteristic. In the previously mentioned chapter, we have investigated the effect of (automatic) top-down partial deduction of the vanilla meta interpreter, and have shown that it can handle the interpreter at the cost of incorporating quite a sophisticated control mechanism. However, the vanilla meta interpreter is a program in which some of the information flows bottom-up, and hence is a natural candidate for bottom-up partial deduction.

Example 8.14 *Consider the vanilla meta interpreter with the following object program. The object program does not perform any useful computations, but is provided only as a means to illustrate the behaviour of $\mathcal{B}_{\trianglelefteq}$ on the meta interpreter.*

```
solve(true).
solve((A,B)):- solve(A), solve(B).
solve(A):- cl(A,B), solve(B).
cl(a,true).
cl(a,a).
cl(b,(a,b)).
```

Computing the bottom-up partial deduction of the program in Example 8.14 using $\mathcal{B}_{\trianglelefteq}$ results in the graph depicted in Fig. 8.7 from which the following residual

Figure 8.7: The graph computed by $\mathcal{B}_{\trianglelefteq}$ for the program of Example 8.14.

program is extracted:

```
solve(true).                          cl(a,true).
solve((A,B)):- solve(A), solve(B).    cl(a,a).
solve(a):- solve(true).               cl(b,(a,b)).
solve(a):- solve(a).
solve(b):- solve((a,b)).
```

The graph in Fig. 8.7 contains a single tautology $solve(X) \leftarrow solve(X)$, due to the fact that $solve(true) \trianglelefteq solve((true, true))$ resulting in the latter's generalisation. The clause heads derived from this generalisation are instances of $solve(X)$, so they are not used for any further bottom-up derivations. Let us have a closer look at the residual clauses. Recall that each such clause represents a partial derivation that was created by a bottom-up derivation process. The head of a residual clause is derived from the head of one of the original program's clauses, possibly instantiated with information that was propagated upwards. If the residual clause has one or more body atoms, these are derived from some generalised clauses' body atoms, possibly instantiated with information that was propagated downwards during the bottom-up derivation starting from the generalisation. Indeed, abstraction introduces a barrier in the propagation process: since the head atom of a generalisation is an abstraction of another derivation's head, using the generalised head instead possibly introduces some information loss (the information that was removed by the generalisation is no longer propagated upwards). If

during the subsequent bottom-up derivation some of the generalised information is unified with some structure, these unifications are naturally residualised in the body atoms of the residual clause, and not propagated further downwards (where the unification could effectively be performed). This is illustrated in the above example by the clauses

$$solve(a) \leftarrow solve(true).$$
$$solve(a) \leftarrow solve(a).$$
$$solve(b) \leftarrow solve((a, b)).$$

Each of these clauses' body atoms is constructed by instantiating the generalised atom $solve(X)$ with information that was propagated downwards during a bottom-up derivation starting from (among others) the clause $solve(X) \leftarrow solve(X)$. While the barrier in the bottom-up information propagation process was introduced for a particular – and probably good – reason (ensuring termination), it also halts the *top-down* information flow. In some cases, it may be worthwhile to break the barrier on the latter kind of information flow, and propagate the information from the residual clauses' body atoms further downwards by partially deducing them in the classical top-down way. Example 8.15 shows the result of partially deducing the above program's body atoms top-down, building partial SLD-trees for them en recording the resulting resultants in a residual program.

Example 8.15 *Figure 8.8 shows the partial SLD-trees created for the residual program's body atoms by a most simple partial deduction strategy: determinate unfolding. The figure shows only the non-trivial SLD-trees. The residual program*

Figure 8.8: The SLD-trees created for the body atoms of Example 8.14's residual clauses.

is the following, when `solve`$_1$ *and* `solve`$_2$ *denote the residual predicates created for the SLD-trees in Fig. 8.8.*

```
solve(true).
solve((A,B)):- solve(A), solve(B).
solve(a):- solve1.
solve(a):- solve(a).
solve(b):- solve2.
```

```
solve1.
solve2:- solve(a), solve(b).
```

Another bottom-up partial deduction of the program depicted in Example 8.15 propagates the definition of `solve`$_1$ and `solve`$_2$ into the program, resulting in a program from which, when provided with an initial query of the form `solve(a)` or

`solve(b)`, all meta structure can be filtered and hence is meta structure free. As we showed in Chapter 7, the same result can be achieved by a top-down system alone, but only at the cost of sophisticated control mechanism. Using a combination of bottom-up and top-down partial deduction on the other hand results in a conceptually cleaner approach towards handling the combination of top-down and bottom-up data flow. Moreover, in case of the vanilla interpreter, it requires two less sophisticated control mechanisms, each one concentrating on a particular kind of data flow. Example 8.14 illustrates this, since excellent specialisation results are achieved using a traditional control strategy (based on well-quasi orders) for the bottom-up component, and an almost trivial control mechanism (determinate unfolding) for the top-down component. The resulting program in Example 8.15 suggests that the two transformation phases can be alternated to obtain programs in which information is maximally propagated. The above examples suggest that such a combined approach in which each of the partial deduction schemes is equipped with a sufficiently powerful control mechanism, is capable to propagate information beyond barriers imposed by a top-down or bottom-up scheme alone, possibly paving the way for automatic partial deduction of more involved meta interpreters. However, fully automatic control – and thus termination – of such an alternation of bottom-up and top-down partial deduction is not straightforward: if a program P'' is derived from a program P', the transformation cycle can be continued as long as the "quality" of P'' is significantly better than that of P'. While (automatically) determining a program's quality with respect to partial deduction is still an open problem, some promising directions for future research into this area exist. Cost analysis (Debray and Lin 1993) for example, defines a method to compute the worst-case cost of a particular class of logic programs, mainly based on the number of evaluation steps as a criterion to determine the "cost" of evaluation of a logic program. Another technique, primarily targeted towards measuring the effectiveness of partial evaluation, is described in (Albert, Antoy, and Vidal 2000) and formally defines the cost of evaluating an expression in terms of the number of evaluation steps, function applications and the complexity of pattern-matching or unification.

8.5.2 General Discussion

In Section 8.2, we have developed a theoretical framework in which bottom-up partial deduction can be achieved. The partial deduction process is essentially described as a fixed point computation of a bottom-up resolution operator, combined with abstraction. The bottom-up operator on the semantic domain of clauses was originally introduced by Bossi and others (Bossi, Gabbrielli, Levi, and Meo 1994) as T_P^Ω, in order to extend the S-semantics in the context of "open" logic programs. Informally, the set Ω represents a set of so-called open predicates which are considered, during bottom-up computation of a program's fixed point semantics, as being defined by $p(\overline{X}) \leftarrow p(\overline{X})$. As noted in (Bossi, Gabbrielli, Levi, and Meo

1994; Bossi, Gabbrielli, Levi, and Martelli 1994), $T_P^\Omega \uparrow \omega$ is in essence a set of resultants which can be seen as the result of a top-down partial deduction of P with respect to a set of atomic goals of the form $p(\overline{X})$. As a consequence of abstraction, this will not be the case with our transformation. Consider the following program, being slightly different from Example 8.9.

Example 8.16 *Let P denote the following definite program:*

```
p(a,a).
p(X,c).
r(Y):- p(X,Y).
```

If the abstraction function $\mathcal{A}_b$ is defined as in Example 8.9, that is the unit clause $p(a,a)$ is abstracted into $p(a,Y) \leftarrow p(a,Y)$ while all other clauses remain unchanged, the fixed point of the induced A_P^c operator, $A_P^c \uparrow 2$ contains the clause $r(Y) \leftarrow p(a,Y)$. Although the latter clause was produced by T_P^c starting from an abstraction, it can not be derived by a top-down partial deduction of $r(Y)$, since its body contains the constant a as the result of an earlier bottom-up propagation.

The aim of bottom-up partial deduction is to maximize the propagation of information in the program, while preserving its least S-Herbrand model. Without abstraction, the process reduces to bottom-up evaluation and the bottom-up partial deduction $P' = \mathcal{F}(P)$. Enriching T_P^c with abstraction introduces several new complications. While in general the resulting operator, A_P^c, is not continuous, it can be defined finitary. This is mandatory if we plan to use it for program transformation, since the transformation as well as the resulting program must be finite. We have given the necessary conditions for the resulting bottom-up partial deduction to be sound and complete with respect to the S-Herbrand (or fixed point) semantics. These proven correctness results together with the equivalence between the S-Herbrand semantics and the operational semantics allow our transformation to be applied in isolation, as well as in combination with top-down partial deduction.

Bottom-up specialisation has only been occasionally considered before, notably in the context of meta interpreter specialisation. For example (Cosmadopoulos, Sergot, and Southwick 1991) proposes to push the unit clauses defining an object program, into the meta interpreter. While (Cosmadopoulos, Sergot, and Southwick 1991) presents an ad-hoc approach to the specialisation of the vanilla meta interpreter, our work is, to the best of our knowledge, the first attempt to achieve this bottom-up propagation in a completely general way. Somewhat related to our work is (Leuschel and Schreye 1996), in which Leuschel and De Schreye combine top-down partial deduction with a *more specific program* transformation (Marriott, Naish, and Lassez 1988) based on abstract interpretation: after unfolding, a nonground version of T_P is used to derive success information which is then imposed on the residual predicates. Consider for example the following predicate:

```
q(a).
q(X):- q(X).
```

The technique of (Leuschel and Schreye 1996) will correctly deduce that the only answer substitution for $q(X)$ is $\{X = a\}$. However, to compute success information, bottom-up resolution is combined with a rather general abstract domain: in each bottom-up step, the predicate-wise msg of every derived consequence is computed. If the above example is extended with another base clause, for example $q(b)$, the algorithm will not be able to derive $\{q(a), q(b)\}$ as the only possible answers. Our approach is quite different: bottom-up inference is not used to gather information (expressed in an abstract domain) over the resulting clauses, but rather to derive the resulting clauses directly. Generalisation between head atoms of the residual clauses is only performed if a control strategy decides to do so in order to ensure finiteness of the process. Using a refined control strategy – one that does not perform predicate-wise generalisations, but rather only generalises "related" clauses – more precise success information than in (Leuschel and Schreye 1996) can be achieved. The $\mathcal{B}_{\trianglelefteq}$ algorithm, for example, deduces $\{q(a), q(b)\}$ as the possible answers for the example above. The technique of (Leuschel and Schreye 1996) is generalised in (Leuschel 1998b), integrating (top-down) partial deduction and abstract interpretation. Of course, for examples like above, when more precise abstractions are employed, such an integration can also obtain more refined results than (Leuschel and Schreye 1996). Moreover, Leuschel (1998b) presents a generic top-down/bottom-up abstract interpretation framework in which (conjunctive) partial deduction can be cast, providing as such an alternative view on the partial deduction framework of Lloyd and Shepherdson (1991). Possibly also the current work on bottom-up partial deduction can be cast into a general (bottom-up) abstract interpretation framework which may result – when combined with (Leuschel 1998b) – in a generalised view on top-down as well as bottom-up partial deduction, as such paving the way for a thorough comparison and possibly a complete integration of the two techniques.

Our algorithm for controlling bottom-up partial deduction neatly fits in the general framework defined in Section 8.2. Nevertheless, other control schemes might exist that are practically applicable, while deviating slightly from the proposed framework. Our generic algorithm, for example, employs a generalisation scheme that is called *child generalisation* in top-down partial deduction terminology. The notion of child generalisation refers to the fact that, when two partial derivations are abstracted, the most recently constructed derivation is replaced by the generalisation. In our setting, if $w = \mathit{whistle}(v, (V, E))$ for nodes v, w in a DAG (V, E), it is $\mathit{label}(v)$ that is replaced by the generalisation. A possible effect of child generalisation is the presence in the residual program of a number of extra base cases in the definition of a recursive predicate. Reconsider Example 8.13, in which child generalisation leads to the residual clauses

```
append([],Y,Y).
append([H],Y,[H|Y]).
append([H|T],Y,[H|R]):- append(T,Y,R).
```

The presence of the unit clause `append([H],Y,[H|Y])` could be avoided if so-called *parent* generalisation was employed. Parent generalisation refers to the fact that, when two partial derivations are abstracted, the least recently constructed derivation is replaced by the generalisation. In our setting, if $w = whistle(v, (V, E))$, parent generalisation involves removing the subgraph originating in w and replacing w by the generalisation. Parent generalisation is often employed in top-down partial deduction, in general resulting in less residual code. In the above example, the classic definition of `append/3` – having the single unit clause `append([],Y,Y)` – would be derived instead. Parent generalisation, however, does not directly fit into the framework since it destroys monotonicity of the corresponding A^c_P operator. However, we expect that it is still possible to define a finitary A^c_P operator employing parent generalisation due to the presence of a "sufficiently large" monotonic subsequence in the sequence of A^c_P-applications. Exactly how to incorporate parent generalisation in the framework is an interesting topic for further research.

Concerning the control of bottom-up partial deduction, we have provided an algorithm that is generic in the sense that it is parametrised with respect to the abstraction function. The abstraction function – and hence the concrete control mechanism – is represented by a number of abstract operations. The algorithm is very similar to the generic algorithm for top-down partial deduction, of which the *global* control component is expressed in terms of the same abstract operations. Indeed, contrary to the bottom-up algorithm which employs a single notion of control, its top-down counterpart divides the control in two levels: a local and a global component. When partially deducing a program top-down, it is assumed that all queries with respect to which the partial deduction will be evaluated are instances of the partial deduction query. Hence, when initiating a partial deduction comprising the latter's atoms, it suffices to ensure that the end points of the partial derivations (the atoms in the leafs of the constructed SLD-trees) are covered by a starting point of other partial derivations (the root of an other SLD-tree). The residual program must not cover any other atoms than (instances of) those comprising the set of partially deduced atoms – constructed by global control. When computing a bottom-up partial deduction on the other hand, no knowledge is assumed about the queries with respect to which the bottom-up partial deduction will be evaluated. Hence, the computation of *every* atom in the original program's denotation must be covered by the residual program. In other words, not only the end points of the partial derivations must be accounted for in the residual program, but also the intermediate derivations. This explains why we have chosen to handle all control decisions at single (global) level. Although most recent techniques for top-down partial deduction employ dual-level control, practical control schemes for *supercompilation* – a much related top-down tech-

nique in the context of functional program specialisation – also employs a single level of control (Sørensen and Glück 1995). See Chapter 2 for a more elaborated discussion on these control levels in a top-down setting.

Apart from the basic idea (Cosmadopoulos, Sergot, and Southwick 1991) of propagating object clauses bottom-up into meta interpreters, little is known concerning the use of bottom-up evaluation in program specialisation and its relation to standard unfolding techniques. On the other hand, abstractions of bottom-up evaluation have been extensively used in program analysis – examples being groundness and termination analysis – where the analysis computes an abstraction of a program's S-Herbrand semantics (Codish, Debray, and Giacobazzi 1993; Codish and Demoen 1995; Codish and Taboch 1997; Codish, Marriott, and Taboch 2000).

Bottom-up evaluation of logic programs, as an alternative to top-down evaluation, has perhaps been mostly studied in the field of deductive databases (Bancilhon and Ramakrishnan 1986; Ullman 1989b). One reason explaining its popularity in the latter field, is the fact that bottom-up evaluation guarantees completeness and termination for datalog programs. This is not the case with top-down evaluation. While in general top-down evaluation is considered to be more efficient and seems to be the preferred evaluation technique for logic programs, the evaluation does not terminate for a lot of programs. For some kinds of programs and queries, bottom-up evaluation equals – and sometimes even outperforms – top-down evaluation (Ullman 1989a). Different approaches exist towards controlling the search in a bottom-up evaluation, guiding the evaluation towards a query. Magic templates (Mumick, Finkelstein, Pirahesh, and Ramakrishnan 1990; Beeri and Ramakrishnan 1991) and Alexander Templates (Seki 1989) constitute a transformation technique by which a program is transformed in such a way that when the transformed program is evaluated bottom-up, it simulates top-down evaluation as such restricting the search space to the query of interest. A generalisation of these methods in the context of databases is presented in (Bry 1990). Codish's generalised magic sets technique (Codish 1999) produces the same (bottom-up) computation, but avoids the need to effectively transform the program. Bottom-up partial deduction is interesting in its own right, as it enables to partially deduce a program with respect to other information than the information residing in a partial deduction query – in fact not requiring such a query at all. Still, the technique might profit from a partial deduction query when such is available, in particular when the technique is combined with a more traditional top-down approach. Like with pure evaluation, the availability of a partial deduction query enables to guide the bottom-up search towards "interesting" atoms, rendering the need to cover *all* possible bottom-up derivations unnecessary. How to integrate such a query-guided bottom-up search, for example based on magic sets, into bottom-up partial deduction is an interesting topic for further research and might lead to a complete integration between bottom-up and top-down partial deduction.

Chapter 9

Conclusions

"Is that all?" asked Alice.
"That is all." said Humpty Dumpty.
"Goodbye."

– Lewis Carrol, Through the Looking Glass

In the final chapter of this thesis, we discuss some of our achievements and indicate some directions for future research. The main contributions of this thesis are threefold. Firstly, we have concentrated on an off-line approach towards specialisation of logic programs. Contrary to its on-line counterpart, off-line specialisation has been considered only occasionally before in a logic programming setting. We have developed a binding-time analysis – being the most important ingredient of an off-line specialiser – for Mercury, a strongly moded logic programming language, and for a pure subset of Prolog. Mercury is an excellent candidate to devise a binding-time analysis for: its execution model exposes enough similarities with the execution model of functional languages to allow to start from known techniques for binding-time analysis in the latter field, while the fact that it is a logic programming language specifically tuned towards use in large scale applications creates new challenges and adds an extra dimension to the analysis. Nevertheless, the resulting analysis relies heavily on the fact that Mercury is a strongly moded language which makes the approach hard to devise for logic programming in general. Other logic languages, in particular Prolog, are not strongly moded and in a second part of our work on off-line partial deduction, we have developed a binding-time analysis for (a pure subset of) Prolog using a substantially different approach based on termination analysis.

Secondly, we have generalised some of the ideas underlying the type-based domain of binding-times and demonstrated how a traditional *Pos*-based analysis can be refined in the presence of type information. We have shown how, by imposing the type information on a *Pos*-based groundness analysis, one obtains a more general instantiatedness analysis. While the approach is somewhat similar to earlier approaches in which structural information from the involved terms was used to enhance such analysis, the use of type information is more sophisticated, can be derived by analysis and can lead to more precise results.

Finally, we have devised a novel technique, which we have called bottom-up partial deduction, by which a logic program can be specialised with respect to a set of unit clauses rather than with respect to a query. The development of the technique is inspired by the problems encountered by traditional top-down partial deduction techniques, in particular with the (automatic) partial deduction of the vanilla meta interpreter. We have constructed a framework for bottom-up partial deduction, and have developed a concrete instance of the framework that controls the construction of a bottom-up partial deduction in an on-line way, similar in behaviour to the way classical top-down partial deduction is controlled. Apart from being interesting in its own right, we conjecture that a combination of bottom-up and top-down partial deduction, each of the strategies focusing on a particular direction in the program's data flow, can obtain the same (or even a better) degree of specialisation while requiring two less sophisticated control strategies than is the case with a single strategy, be it top-down or bottom-up. We have illustrated this by the partial deduction of the vanilla meta interpreter.

In the following two sections, we discuss our contributions to the off- and on-line approaches to partial deduction into somewhat more detail. A first section deals with the developed binding-time analyses while a second one elaborates on bottom-up partial deduction.

9.1 Binding-time Analysis

In order to approximate a value at specialisation-time in a sufficiently precise way, we have developed a domain of binding-times in which the binding-time of a value basically represents the value's degree of instantiatedness. The basic idea is to use the type of a value – which is provided by Mercury – to identify a (finite) number of descriptions of the instantiatedness of a value and use such a description to approximate a specialisation-time value during binding-time analysis. We have approached the problem of computing binding-time approximations from two different angles. First, we have developed a binding-time analysis by abstract interpretation over the domain of binding-times. The analysis is inspired on work in the field of binding-time analysis for functional languages, but is more involved, most notably by the fact that the analysis has to deal with the determinism- and success and failure characteristics of the involved subgoals, complicating the notion

of dynamic control and hence the specialisation strategy. Inspired by Gomard's work regarding the correctness of a partial evaluator for the Lambda Calculus, we have proven the result of the Mercury binding-time analysis to be correct, in the sense that a specialiser that follows the annotations satisfies the partial evaluation equation and hence preserves the semantics of the original program. Inspired by the need to perform the analysis (at least partly) in a modular way – as such scaling the analysis to multi-module programs – we have explored a second approach towards binding-time analysis for Mercury and have reformulated the analysis as a two-phased process. In a first phase, binding-times and the existing relations between them are represented symbolically by means of a constraint system over the domain of binding-times. Actual binding-times are derived in a second phase, by computing minimal solutions of the constraint systems. Correctness of the constraint approach is induced by its equivalence with respect to the former approach of binding-time analysis by abstract interpretation. We have shown how to derive a closure analysis, needed for a precise higher-order binding-time analysis, by adding closures to the domain of binding-times and creating a number of additional constraints. The resulting binding-time analysis for Mercury is polyvariant, deals with partially instantiated structures, modules and higher-order control flow. Few binding-time analyses have been reported upon that expose the combination of all these characteristics.

A prototype implementation of the analysis has been created and some experiments have shown the technique to work for Mercury. However, in order to perform some more involved experiments, the implementation – in particular the constraint normalisation phase – must be optimised, and the analysis must be extended in order to deal with some more aspects of the richness of the Mercury language, for example type classes. Moreover, in order to remain modular while obtaining closure propagation over module boundaries and handling circularities in the module hierarchy, the analysis must be performed iteratively. Naturally, to exploit the result of analysis, it must be coupled with a partial deducer for Mercury. An interesting experiment would be to investigate which of the existing heuristics-based optimisations in the Melbourne Mercury compiler can be subsumed – and to what extent – by a generalised partial deduction approach based on binding-time analysis.

In a second part of our research on binding-time analysis, we have developed a binding-time analysis for an untyped and unmoded logic programming language, in casu a pure subset of Prolog. Our analysis is monovariant, and uses a simple binding-time characterisation: either a value is definitely instantiated enough with respect to a given norm, or it is not. The analysis works in a way that is substantially different from earlier approaches. An annotated version of a program is produced by repeatedly performing termination analysis in combination with the annotation of a problematic call as dynamic until the analysis can prove that reducing the program according to the annotations terminates. Compared with

earlier approaches, our analysis produces annotations that allow more liberal unfolding of the program. In particular for some of the benchmarks that were used in experimentation with the cogen-approach for logic programming, our analysis created annotated programs that match the annotations created by hand. Interesting issues in this area are to make the analysis polyvariant, and how to incorporate the use or even the derivation of more involved norm(s).

9.2 Bottom-up Partial Deduction

Before defining the novel, bottom-up approach to partial deduction, we have investigated the problems that arise during (top-down) partial deduction, in particular with partial deduction of the vanilla meta interpreter. We have taken a popular well-quasi order, the homeomorphic embedding relation, and studied how it guides specialisation of the interpreter. We have focussed on the so-called parsing problem – removing all parsing overhead from the program, and have demonstrated that further refinements in the control of partial deduction are necessary to properly deal with it. In particular, we have modified local control on the basis of information imported from the global level. The resulting strategy, while remaining fully general, leads to excellent specialisation of vanilla-like meta programs. We have subjected an extended vanilla meta interpreter capable of dealing with compositions of programs to our techniques, and have showed that we equal or surpass results obtained through a more ad hoc approach. Although the resulting technique is general, it is nevertheless inspired on dealing with the problem of fluctuating structure in case of the vanilla meta interpreter. Consequently, it is likely to be unable to deal in a satisfactorily way with other programs exposing a complicated (fluctuating) data flow. Moreover, extending the local control strategy even further to incorporate more of the bottom-up data flow seems nontrivial and may lead to complicated unfolding rules which are hard to reason about. In the remaining part of the thesis, we have taken a rather different approach, and have devised a novel partial deduction technique that concentrates, in contrast with the classical partial deduction, on the program's bottom-up data flow. We have developed a solid theoretical foundation for such bottom-up partial deduction by defining an operator that combines a well-known immediate consequence operator with abstraction. We have proven the transformation to be correct with respect to the S-semantics and have designed, within the framework, a concrete control strategy. The transformation can be used as a stand-alone specialisation technique, useful when a program needs to be specialised with respect to its internal structure instead of a single query, or it can be combined with a more traditional top-down partial deduction strategy. An important topic of further research is the integration in the framework of query-guided bottom-up resolution. Apart from a possible improvement in the efficiency of the resulting strategy, this will enable the computation of much more practically feasible bottom-up partial deductions from

which the need to cover every possible query is removed. Moreover, the fact that not every intermediate partial derivation needs to be residualised creates opportunities to divide the control of bottom-up partial deduction in a local and global control level, leading the way to a full comparison and integration of bottom-up and top-down partial deduction.

Appendix A

Benchmarks from Chapter 5

This appendix contains the annotated versions computed by the binding-time analysis of Chapter 5. The benchmarks are selected from the DPPD Library (Leuschel 1996). Atoms that are annotated to be residualised (i.e. the dynamic atoms) are underlined.

A.1 ex_depth

```
solve([],Depth,Depth).
solve([Head|Tail],DepthSoFar,Res) :-
     claus(Head,Body),
     solve(Body,s(DepthSoFar),IntDepth),
     solve(Tail,IntDepth,Res).

claus(member(X,[X|T]),[]).
claus(member(X,[Y|T]),[member(X,T)]).

claus(inboth(X,L1,L2),[member(X,L1),member(X,L2)]).

claus(app([],L,L),[]).
claus(app([H|X],Y,[H|Z]),[app(X,Y,Z)]).

claus(delete(X,[X|T],T),[]).
claus(delete(X,[Y|T],[Y|D]),[delete(X,T,D)]).

claus(test(A,L1,L2,Res),[inboth(A,L1,L2),delete(A,L1,D1),app(D1,L2,Res)]).
```

A.2 match

```
match(Pat,T) :- match1(Pat,T,Pat,T).

match1([],Ts,P,T).
match1([A|Ps],[B|Ts],P,[X|T]) :-
     A=B,
     match1(P,T,P,T).
match1([A|Ps],[A|Ts],P,T) :-
     match1(Ps,Ts,P,T).
```

A.3 map.rev/reduce

```
map(P,[],[]).
map(P,[H|T],[PH|PT]) :-
     Call =..  [P,H,PH],
     call(Call),
     map(P,T,PT).

reduce(Func,Base,[],Base).
reduce(Func,Base,[H|T],Res) :-
     reduce(Func,Base,T,TRes),
     Call =..  [Func,H,TRes,Res],
     call(Call).
```

A.4 parser

```
nont(X,T,R):-t(a,T,V), nont(X,V,R).
nont(X,T,R):-t(X,T,R).

t(X,[X|Es],Es).
```

A.5 regexp1-2

```
generate(empty,T,T).

generate(char(X),[X|T],T).

generate(or(X,Y),H,T) :- generate(X,H,T).
generate(or(X,Y),H,T) :- generate(Y,H,T).

generate(cat(X,Y),H,T) :- generate(X,H,T1), generate(Y,T1,T).

generate(star(X),T,T).
generate(star(X),H,T) :- generate(X,H,T1), generate(star(X),T1,T).
```

A.6 transpose

```
transpose(Xs,[]) :-
     nullrows(Xs).
transpose(Xs,[Y|Ys]) :-
     makerow(Xs,Y,Zs),
     transpose(Zs,Ys).

makerow([],[],[]).
makerow([[X|Xs]|Ys],[X|Xs1],[Xs|Zs]) :-
     makerow(Ys,Xs1,Zs).

nullrows([]).
nullrows([[]|Ns]) :-
     nullrows(Ns).
```

A.7 Benchmarks from Chapter 4 for the module `list`

Procedure	Turns	Total time	Constraints
append/3 : 0	3	0.12	34
append/3 : 1	3	0.11	35
append/3 : 2	3	0.25	44
append/3 : 3	3	0.15	50
append/3 : 4	3	0.12	36
remov_suffix/3 : 0	2	0.12	49
merge/3 : 0	3	1.79	164
merge_and_remove_dups/3 : 0	3	1.39	179
remove_adjacent_dups/2 : 0	2	0.05	25
remove_dups/2 : 0	2	0.02	15
member/2 : 0	3	0.03	19
member/2 : 1	3	0.18	22
member/3 : 0	2	0.02	18
length/3 : 0	2	0.02	12
same_length/2 : 0	3	0.09	29
same_length/2 : 1	3	0.09	29
same_length/2 : 2	3	0.09	36
split_list/4 : 0	3	1.23	101
take/3 : 0	3	0.84	81
take_upto/3 : 0	2	0.03	32
drop/3 : 0	3	0.46	66
insert/3 : 0	2	0.00	5
insert/3 : 1	2	0.00	7
insert/3 : 2	2	0.00	9
insert/3 : 3	2	0.00	8
delete/3 : 0	3	0.31	56
delete/3 : 1	3	0.14	42
delete/3 : 2	3	0.16	42
delete/3 : 3	3	0.23	33
delete_first/3 : 0	3	0.13	46
delete_all/3 : 0	3	0.38	57
delete_all/3 : 1	3	0.25	57
delete_elems/3 : 0	4	0.30	41
replace/4 : 0	3	0.23	75
replace/4 : 1	3	0.29	45
replace_first/4 : 0	3	0.14	49
replace_all/4 : 0	3	0.47	74
replace_nth/4 : 0	3	0.03	23
replace_nth_det/4 : 0	2	0.14	75

Procedure	Turns	Total time	Constraints
filter/3 : 0	2	0.01	10
sort_and_remove_dups/2 : 0	2	0.02	16
sort/2 : 0	2	0.01	7
reverse/2 : 0	2	0.01	14
perm/2 : 0	3	0.12	32
nth_member_search/3 : 0	2	0.32	65
index0/3 : 0	3	0.45	61
index1/3 : 0	2	0.04	22
index0_det/3 : 0	2	0.05	37
index1_det/3 : 0	2	0.04	20
zip/3 : 0	3	0.13	48
duplicate/3 : 0	3	0.29	55
condense/2 : 0	3	0.27	42
chunk/3 : 0	2	0.03	20
sublist/2 : 0	3	0.33	112
all_same/1 : 0	2	0.03	19
last/2 : 0	3	0.05	23
map/3 : 0	3	0.14	39
map/3 : 1	3	0.14	39
map/3 : 2	3	0.28	39
map/3 : 3	3	0.13	39
map/3 : 4	3	0.13	43
foldl/4 : 0	3	0.22	32
foldl/4 : 1	3	0.09	31
foldl/4 : 2	3	0.08	31
foldl/4 : 3	3	0.08	31
foldr/4 : 0	3	0.23	31
foldr/4 : 1	3	0.09	31
foldr/4 : 2	3	0.10	31
foldl2/6 : 0	3	0.34	41
foldl2/6 : 1	3	0.28	41
foldl2/6 : 2	3	0.12	41
foldl2/6 : 3	3	0.12	42
foldl2/6 : 4	3	0.27	42
foldl2/6 : 5	3	0.13	43
map_foldl/5 : 0	3	0.19	50
map_foldl/5 : 1	3	0.32	49
map_foldl/5 : 2	3	0.18	49
map_foldl/5 : 3	3	0.33	49

Procedure	Turns	Total time	Constraints
filter/4 : 0	3	0.66	75
filter_map/3 : 0	3	0.38	45
filter_map/4 : 0	3	0.68	73
takewhile/4 : 0	3	0.46	65
sort/3 : 0	2	0.43	70
sort_and_remove_dups/3 : 0	2	0.02	16
merge/4 : 0	3	3.51	176
merge_and_remove_dups/4 : 0	3	2.75	160
replace_nth_2/4 : 0	3	0.76	75
length_2/3 : 0	3	0.31	45
reverse_2/3 : 0	3	0.10	35
qsort/3 : 0	4	0.60	53
partition/4 : 0	3	1.43	83
merge_sort/2 : 0	3	5.44	150
remove_dups_2/3 : 0	3	0.44	74
remove_adjacent_dups_2/3 : 0	3	0.27	66
zip2/3 : 0	3	0.30	48
chunk_2/5 : 0	5	7.18	223
all_same_2/2 : 0	3	0.19	26
hosort/5 : 0	3	8.51	272
det_head/1 : 0	2	0.02	22
det_tail/1 : 0	2	0.04	22

Bibliography

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: principles, techniques, tools*. Addison-Wesley.

Albert, E., M. Alpuente, M. Falaschi, and P. Julian (1998). Improving control in functional logic program specialization. In *Proceedings of SAS'98*, Number 1503 in Lecture Notes in Computer Science, pp. 262–277. Springer-Verlag.

Albert, E., M. Alpuente, M. Hanus, and G. Vidal (1999). A partial evaluation framework for curry programs. In H. Ganzinger, D. McAllester, and A. Voronkov (Eds.), *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR-99)*, Volume 1705 of *LNAI*, Berlin, pp. 376–395. Springer.

Albert, E., S. Antoy, and G. Vidal (2000). Measuring the effectiveness of partial evaluation. In K. Lau (Ed.), *Pre-Proceedings of LOPSTR2000*, pp. 72 –79. Technical Report Report Series, Dept. of Computer Science, University of Manchester.

Albert, E., M. Hanus, and G. Vidal (2000). Using an abstract representation to specialize functional logic programs. In *Proceedings of LPAR'2000*, pp. 381 – 398.

Albert, E., M. Hanus, and G. Vidal (2001). A practical partial evaluator for a multi-paradigm declarative language. In *Proceedings of FLOPS'2001*, pp. 326 – 342.

Allen, F. E. (1969). Program optimization. In M. I. Halpern and C. J. Shaw (Eds.), *Annual Review in Automatic Programming*, Volume 5, pp. 239–307. Oxford: Pergamon Press.

Allen, F. E. and J. Cocke (1972). *A catalogue of optimizing transformations*, pp. 1–30. Prentice-Hall, Englewood Cliffs, NJ.

Alpuente, M., M. Falaschi, and G. Vidal (1998). Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems 20*(4), 768–844.

Alpuente, M., M. Falschi, P. Julián, and G. Vidal (1997). Specialization of lazy functional logic programs. In *Proceedings of PEPM'97*, pp. 151–162. ACM.

Andersen, L. (1993). Binding-time analysis and the taming of C pointers. In *PEPM93*, pp. 47–58. ACM.

Andersen, P. (1995). Partial evaluation applied to ray tracing. DIKU Research Report 95/2, DIKU.

Andersen, P. H. and C. K. Holst (1996). Termination analysis for offline partial evaluation of a higher order functional language. In *Proceedings of the Third International Static Analysis Symposium (SAS)*, pp. 67 – 82.

Andersson, E., E. Housos, N. Kohl, and D. Wedelin (1997). Crew pairing optimization. In *OR in Airline Industry*. Kluwer Academic Press.

Andersson, J., S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. Sjöland, and J. Widn (1993). SICStus prolog user's manual. Technical Report T93-01, SICS.

Apt, K. R. (1990). Logic programming. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pp. 493–574. Elsevier Science Publishers B.V.

Augustsson, L. (1997). Partial evaluation in aircraft crew planning. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, Volume 32, 12 of *ACM SIGPLAN Notices*, New York, pp. 127–136. ACM Press.

Auslander, J., M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad (1996). Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania, pp. 149–159.

Bacon, D. F., J.-H. Chow, D. ching R. Ju, K. Muthukumar, and V. Sarkar (1994). A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON'94 – Integrated Solutions*, Toronto, Ontario, pp. 270–282. IBM Centre for Advanced Studies.

Bacon, D. F., S. L. Graham, and O. J. Sharp (1994). Compiler transformations for high-performance computing. *ACM Computing Surveys 26*(4), 345–420.

Bagnara, R., P. M. Hill, and E. Zaffanella (2000). Efficient structural information analaysis for real CLP languages. In M. Parigot and A. Voronkov (Eds.), *Proceedings LPAR2000*, Volume 1955 of *Lecture Notes in Computer Science*, pp. 189 –206.

Ball, J. E. (1979). Predicting the effects of optimization on a procedure body (program compilers). *ACM SIGPLAN Notices 14*(8), 214–220.

Bancilhon, F. and R. Ramakrishnan (1986). An amateur's introduction to recursive query processing strategies. In C. Zaniolo (Ed.), *Proceedings of the 1986*

ACM SIGMOD International Conference on Management of Data, Washington, D.C., pp. 16–52.

Barklund, J. (1995). Metaprogramming in logic. In A. Kent and J. Williams (Eds.), *Encyclopedia of Computer Science and Technology, Vol. 33*, pp. 205–227. Marcel Dekker, Inc., New York.

Beckman, L. et al. (1976). A partial evaluator, and its use as a programming tool. *Artificial Intelligence 7*(4), 319–357.

Beeri, C. and R. Ramakrishnan (1991). On the power of magic. *Journal of Logic Programming 10*, 255 – 299.

Benkerimi, K. and P. M. Hill (1993). Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation 3*(5), 469–486.

Benkerimi, K. and J. W. Lloyd (1990). A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo (Eds.), *Proceedings NACLP'90*, Austin, Texas, pp. 343–358. MIT Press.

Benoy, F. and A. King (1997). Inferring argument size relationships with CLP(R). In *Proceedings of LOPSTR'96*, Volume 1207 of *Lecture Notes in Computer Science*, pp. 204–223.

Berlin, A. (1990). Partial evaluation applied to numerical computation. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pp. 139–150. ACM.

Berlin, A. and R. Surati (1994). Partial evaluation for scientific computing: The supercomputer toolkit experience. In *PEPM94*, pp. 133–141.

Bernstein, R. L. (1986). Multiplication by integer constants. *Software Practice and Experience 16*(7), 641–652.

Bezem, M. (1990). Characterizing termination of logic programs with level mappings. In E. L. Lusk and R. A. Overbeek (Eds.), *Proceedings of the North American Conference on Logic Programming*, pp. 69–80. MIT Press.

Bohlin, T. (1990). The Carmen rule language. Technical report, Carmen Systems AB.

Bol, R. (1993). Loop checking in partial deduction. *Journal of Logic Programming 16*(1&2), 25–46.

Bol, R. and L. Degerstedt (1993). The underlying search for magic templates and tabulation. In D. S. Warren (Ed.), *Proceedings of the Tenth International Conference on Logic Programming*, pp. 793–811. The MIT Press.

Bondorf, A. (1990). Automatic autoprojection of higher order recursive equations. In N. Jones (Ed.), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990 (Lecture Notes in Computer Science, vol. 432)*, pp. 70–87. Springer-Verlag.

Bondorf, A. (1991). Automatic autoprojection of higher order recursive equations. *Science of Computer Programming 17*, 3–34.

Bondorf, A. and J. Jørgensen (1993). Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming 3*(3), 315–346.

Bossi, A., N. Cocco, and M. Fabris (1994). Norms on terms and their use in proving universal termination of a logic program. *Theoretical Computer Science 124*(2), 297–328.

Bossi, A., M. Gabbrielli, G. Levi, and M. Martelli (1994). The S-semantics approach: Theory and applications. *Journal of Logic Programming 19/20*, 149–197.

Bossi, A., M. Gabbrielli, G. Levi, and M. C. Meo (1994). A compositional semantics for logic programs. *Theoretical Computer Science 122*(1-2), 3–47.

Brodsky, A. and Y. Sagiv (1989). Inference of monotonicity constraints in datalog programs. In S. ACM SIGACT, SIGMOD (Ed.), *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '89)*, Philadelphia, PA, USA, pp. 190–199. ACM Press.

Brogi, A. and S. Contiero (1994). Gödel as a meta language for composing logic programs. In A. Turini (Ed.), *Proceedings Meta'94*, pp. 377–394. University of Pisa.

Brogi, A. and S. Contiero (1996). A program specialiser for meta-level compositions of logic programs. Technical Report TR-96-20, Dipartimento di Informatica, Università di Pisa, Pisa, Italy.

Brogi, A. and S. Contiero (1997). Specialising meta-level compositions of logic programs. In J. Gallagher (Ed.), *Proceedings LOPSTR'96*, Stockholm, pp. 275–294. Springer-Verlag, LNCS 1207.

Brogi, A., P. Mancarella, D. Pedreschi, and F. Turini (1990). Composition operators for logic theories. In J. W. Lloyd (Ed.), *Proceedings of the Esprit Symposium on Computational Logic*, pp. 117–134. Springer-Verlag.

Brogi, A., P. Mancarella, D. Pedreschi, and F. Turini (1992). Meta for modularising logic programming. In A. Pettorossi (Ed.), *Proceedings Meta'92*, pp. 105–119. Springer-Verlag, LNCS 649.

Bruynooghe, M., D. De Schreye, and B. Martens (1992). A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing 11*(1), 47–79.

Bruynooghe, M., M. Leuschel, and K. Sagonas (1998). A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin (Ed.), *Programming Languages and Systems, Proc. of ESOP'98, part of ETAPS'98*, Lisbon, Portugal, pp. 27–41. Springer-Verlag. LNCS 1381.

Bruynooghe, M., W. Vanhoof, and M. Codish (2001). Pos(T) : Analyzing dependencies in typed logic programs. In *Proceedings of Andrei Ershov Fourth International Conference "Perspectives of System Informatics"*. Accepted.

Bry, F. (1990). Query evaluation in recursive databases: Bottom-up and top-down reconciled. *Data & Knowledge Engineering 5*(4), 289–312.

Buda, A. O., A. A. Granovsky, and A. P. Ershov (1975). Implementation of the ALPHA-6 programming system. *ACM SIGPLAN Notices 10*(6), 371–381.

Bueno, F., M. de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey (2000). A model for inter-module analysis and optimizing compilation. In K. Lau (Ed.), *Preproceedings of LOPSTR 2000*, pp. 64–71.

Burnett, G. J. and E. G. Coffman, Jr. (1970). A study of interleaved memory systems. In *Proc., AFIPS 1970 Spring Joint Computer Conf.*, Volume 36, pp. 467–474. Montvale, NJ: AFIPS Press.

Burstall, R. and J. Darlington (1977). A transformation system for developing recursive programs. *Journal of the ACM 24*(1), 44–67.

Callahan, D., K. D. Cooper, K. Kennedy, and L. Torczon (1986). Interprocedural constant propagation. In *ACM SIGPLAN 86 Symposium on Compiler Construction*, pp. 152–161. ACM.

Cavedon, L. (1989). Continuity, consistency, and completeness properties for logic programs. In M. Levi, Giorgio; Martelli (Ed.), *Proceedings of the 6th International Conference on Logic Programming (ICLP '89)*, Lisbon, Portugal, pp. 571–584. MIT Press.

Chaitin, G. J. (1982). Register allocation & spilling via graph coloring. In *SIGPLAN '82 Symposium on Compiler Construction*, pp. 98–105.

Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein (1981). Register allocation via coloring. *Computer Languages 6*(1), 47–57.

Chen, W. and D. S. Warren (1996). Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43*(1), 20–74.

Chow, F. C. and J. L. Hennessy (1990). The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems 12*(4), 501–536.

Cocke, J. (1970). Global common subexpression elimination. *SIGPLAN Notices 5*(7), 20–24.

Cocke, J. and J. Schwartz (1970). *Programming Languages and Their Compilers*. NYU, Courant Inst., TR., Second Revised Version.

Codish, M. (1999). Efficient goal directed bottom-up evaluation of logic programs. *The Journal of Logic Programming 38*(3), 354–370.

Codish, M., S. K. Debray, and R. Giacobazzi (1993). Compositional analysis of modular logic programs. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, pp. 451–464. ACM Press.

Codish, M. and B. Demoen (1994). Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. In *Proc. SAS'94*, Volume 864 of *LNCS*, pp. 281–296. Springer.

Codish, M. and B. Demoen (1995). Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *Journal of Logic Programming 25*(3), 249–274.

Codish, M., K. Marriott, and C. Taboch (2000). Improving program analyses by structure untupling,. *Journal of Logic Programming 43*(3), 251 – 263.

Codish, M. and C. Taboch (1997). A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In *Proceedings of ALP'97*, Volume 1298 of *Lecture Notes in Computer Science*, pp. 31–45. Springer-Verlag.

Codish, M. and C. Taboch (1999). A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming 41*(1), 103–123.

Codognet, P. and D. Diaz (1995). WAMCC: Compiling Prolog to C. In L. Sterling (Ed.), *Proceedings of the 12th International Conference on Logic Programming*, Cambridge, pp. 317–332. MIT Press.

Consel, C. (1990). Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pp. 264–272. ACM.

Consel, C. (1993a). Polyvariant binding-time analysis for applicative languages. In *PEPM93*, pp. 66–77. ACM.

Consel, C. (1993b). A tour of schism: A partial evaluation system for higher-order applicative languages. In *PEPM93*, pp. 145–154. ACM.

Consel, C. et al. (1996). A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann (Eds.), *Partial Evaluation*, Volume 1110 of *Lecture Notes in Computer Science*, pp. 54–72. Springer-Verlag.

Consel, C. and O. Danvy (1990). From interpreting to compiling binding times. In N. Jones (Ed.), *ESOP '90. 3rd European Symposium on Programming*, Number 432 in Lecture Notes in Computer Science, pp. 88–105. Springer-Verlag.

Consel, C. and F. Noël (1996). A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23rd *ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, pp. 145–156.

Conway, T., F. Henderson, and Z. Somogyi (1995). Code generation for Mercury. In J. Lloyd (Ed.), *Proceedings of the International Symposium on Logic Programming*, Cambridge, pp. 242–256. MIT Press.

Cortesi, A., B. L. Charlier, and P. V. Hentenryck (2000). Combinations of abstract domains for logic programming: open product and generic pattern construction. *Science of Computer Programming 38*(1–3), 27–71.

Cortesi, A., G. Filé, and W. Winsborough (1991). Prop revisited: propositional formula as abstract domain for groundness analysis. In *Proceedings of LICS'91*, pp. 322–327. IEEE Press.

Cortesi, A., G. Filé, and W. H. Winsborough (1996). Optimal groundness analysis using propositional logic. *The Journal of Logic Programming 27*(2), 137–167.

Cosmadopoulos, Y., M. Sergot, and R. W. Southwick (1991). Data-driven transformation of meta-interpreters: A sketch. In H. Boley and M. M. Richter (Eds.), *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, Volume 567 of *LNAI*, pp. 301–308. Springer Verlag.

Cousot, P. and R. Cousot (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pp. 238–252.

Cousot, P. and N. Halbwachs (1978). Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth annual ACM Symposium on Principles of Programming Languages*, pp. 84–96. ACM: ACM.

De Schreye, D. and S. Decorte (1994). Termination of logic programs: the never-ending story. *Journal of Logic Programming 19/20*, 199–260.

De Schreye, D., R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen (1999). Conjunctive partial deduction: Foundations, control, algorithms and experiments. *Journal of Logic Programming 41*(2-3), 231 – 277.

Dean, J., C. Chambers, and D. Grove (1994). Identifying profitable specialization in object-oriented languages. In *PEPM94*, pp. 85–96.

Dean, J., C. Chambers, and D. Grove (1995). Selective specialization for object-oriented languages. In D. W. Wall (Ed.), *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, pp. 93 – 102. ACM Press.

Debray, S. and N.-W. Lin (1993). Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems 15*(5), 826–875.

Decorte, S., D. de Schreye, and M. Fabris (1993). Automatic inference of norms: A missing link in automatic termination analysis. In D. Miller (Ed.), *Logic Programming - Proceedings of the 1993 International Symposium*, Vancouver, Canada, pp. 420–436. The MIT Press.

Decorte, S., D. De Schreye, M. Leuschel, and B. Martens (1998). Termination analysis for tabled logic programming. In N. Fuchs (Ed.), *Proceedings of LOPSTR'97*, Number 1463 in Lecture Notes in Computer Science, pp. 107–123.

Decorte, S., D. De Schreye, and H. Vandecasteele (1999). Constraint based automatic termination analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems 21*(6), 1137–1195.

Decorte, S. and D. D. Schreye (1998). Termination analysis: Some practical properties of the norm and level mapping space. In J. Jaffar (Ed.), *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP-98)*, Cambridge, pp. 235–249. MIT Press.

Demoen, B., M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey (1999). An overview of HAL. In J. Jaffar (Ed.), *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Volume 1713 of *LNCS*, pp. 174–188. Springer.

Denis, F. and J. Delahaye (1991). Unfolding, procedural and fixpoint semantics of logic programs. In *8th Annual Symposium on Theoretical Aspects of Computer Science*, Volume 480 of *LNCS*, pp. 511–522. Springer.

Dershowitz, N. and Z. Manna (1979). Proving termination with multiset orderings. *Communications of the ACM 22*(8), 465–476.

Dongarra, J. J. and A. R. Hinds (1979). Unrolling loops in FORTRAN. *Software Practice and Experience 9*(3), 219–226.

Dowd, T., Z. Somogyi, F. Henderson, T. Conway, and D. Jeffery (1999). Run time type information in Mercury. In *Proceedings of the International Conference on the Principles and Practice of Declarative Programming*, Volume 1702 of *Lecture Notes in Computer Science*, pp. 224–243. Springer-Verlag.

Drabent, W. (1987). Do logic programs resemble programs in conventional languages? In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, San Francisco, pp. 389–397. IEEE: Computer Society Press.

Dussart, D., R. Heldal, and J. Hughes (1997). Module-sensitive program specialisation. In *SIGPLAN '97 Conference on Programming Language Design and Implementation, June 1997, Las Vegas*, pp. 206–214. ACM.

Engler, D. R., W. C. Hsieh, and M. F. Kaashoek (1996). 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, pp. 131–144.

Falaschi, M., G. Levi, M. Martelli, and C. Palamidessi (1989). Declarative modeling of the operational behaviour of logic programs. *Theoretical Computer Science 69*, 289–318.

Ferrari, D. (1976). Improvement of program behavior. *Computer 9*(11), 39–47.

Fujinami, N. (1997). Automatic run-time code generation in C++. In *Proceedings of ISCOPE 1997*, Number 1343 in Lecture Notes in Computer Science, pp. 9–16. Springer-Verlag.

Fuller, D. A. and S. Abramsky (1988). Mixed computation of Prolog programs. *New Generation Computing 6*(2&3), 119–141.

Futamura, Y. (1971). Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls 2*(5), 45–50.

Gallagher, J. (1986). Transforming logic programs by specialising interpreters. In *Proceedings ECAI'86*, pp. 109–122.

Gallagher, J. (1993). Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, pp. 88–98. ACM Press.

Gallagher, J., D. Boulanger, and H. Saglam (1995). Practical model-based static analysis for definite logic programs. In *Proc. ILPS'95*, pp. 351–365. MIT Press.

Gallagher, J. and M. Bruynooghe (1990). Some low-level source transformations for logic programs. In M. Bruynooghe (Ed.), *Proceedings Meta'90*, Leuven, pp. 229–244.

Gallagher, J. and M. Bruynooghe (1991). The derivation of an algorithm for program specialisation. *New Generation Computing 9*(3&4), 305–333.

Gallagher, J. and D. A. de Waal (1994). Fast and precise regular approximations of logic programs. In *Proc. ICLP'94*, pp. 599–613.

Glück, R. and J. Jørgensen (1997). An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation 10*, 113–158.

Glück, R., J. Jørgensen, B. Martens, and M. H. Sørensen (1996). Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra (Eds.), *Proceedings PLILP'96*, Aachen, Germany, pp. 152–166. Springer-Verlag, LNCS 1140.

Glück, R., R. Nakashige, and R. Zöchling (1995). Binding-time analysis applied to mathematical algorithms. In J. Dolevzal and J. Fidler (Eds.), *System Modelling and Optimization*, pp. 137–146. Chapman and Hall.

Glück, R. and M. H. Sørensen (1994). Driving and partial deduction are equivalent. In M. Hermenegildo and J. Penjam (Eds.), *Proceedings PLILP'94*, Madrid, Spain, pp. 165–181. Springer-Verlag, LNCS 844.

Glück, R. and M. H. Sørensen (1996). A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, and P. Thiemann (Eds.), *Proceedings Dagstuhl Seminar on Partial Evaluation*, Number 1110 in Lecture Notes in Computer Science, Schloss Dagstuhl, Germany, pp. 137 – 160. Springer-Verlag.

Goad, C. (1982). Automatic construction of special purpose programs. In D. Loveland (Ed.), *6th Conference on Automated Deduction, New York, USA (Lecture Notes in Computer Science, vol. 138)*, pp. 194–208. Springer-Verlag.

Gomard, C. (1992). A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems 14*(2), 147–172.

Gomard, C. and N. Jones (1991). A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming 1*(1), 21–69.

Gurr, C. A. (1994a). *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. Ph. D. thesis, Department of Computer Science, University of Bristol, U.K.

Gurr, C. A. (1994b). Specialising the ground representation for the logic programming language Gödel. In Y. Deville (Ed.), *Proceedings LOPSTR'93*, Louvain-la-Neuve, Belgium, pp. 124–140. Springer-Verlag, Workshops in Computing Series.

Hanus, M. (1994). The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming 19,20*, 583–628.

Hanus, M. (1997). A unified computation model for functional and logic programming. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, pp. 80–93.

Hausman, B. (1993). Turbo Erlang. In D. Miller (Ed.), *Logic Programming - Proceedings of the 1993 International Symposium*, Vancouver, Canada, pp. 662. The MIT Press.

Heintze, N. (1994). Set-based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pp. 306–317.

Henderson, F., T. Conway, and Z. Somogyi (1995). Compiling logic programs to C using GNU C as a portable assembler. In *ILPS'95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming*, Portland, Oregon, pp. 1–15.

Henglein, F. (1991). Efficient type inference for higher-order binding-time analysis. In J. Hughes (Ed.), *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pp. 448–472. ACM: Springer-Verlag.

Henglein, F. (1992). Simple closure analysis. Technical Report D-193, DIKU Semantics Report.

Henglein, F. and C. Mossin (1994). Polymorphic binding-time analysis. In D. Sannella (Ed.), *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pp. 287–301. Springer-Verlag.

Hermenegildo, M. (1994a). Some methodological issues in the design of CIAO, a generic, parallel concurrent constraint logic programming system. In *Principles and Practice of Constraint Programming*, Number 874 in Lecture Notes in Computer Science, pp. 123–133.

Hermenegildo, M. (1994b). Towards CIAO-Prolog – A parallel concurrent constraint system. In A. Borning (Ed.), *Principles and Practice of Constraint Programming*, Volume 874 of *Lecture Notes in Computer Science*. Springer. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).

Hermenegildo, M., F. Bueno, G. Puebla, and P. López (1999). Debugging, and optimization using the ciao system preprocessor. In D. D. Schreye (Ed.), *Proceedings of ICLP'99*, pp. 52–66.

Hill, P. and J. Lloyd (1994). *The Gödel Programming Language*. MIT Press.

Hill, P. M. and J. Gallagher (1998). Meta-programming in logic programming. In G. D. M. and H. C. J. (Eds.), *Volume V of the Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press.

Holst, C. (1991). Finiteness analysis. In J. Hughes (Ed.), *Functional Programming Languages and Computer Architecture*, Number 523 in Lecture Notes in Computer Science, pp. 473–495. Springer-Verlag.

Hornof, L. and J. Noyé (1997). Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *PEPM97*, pp. 63–73. ACM.

Hwu, W. W. and P. P. Chang (1989). Achieving high instruction cache performance with an optimizing compiler. In M. Yoeli and G. Silberman (Eds.), *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, pp. 242–251. IEEE Computer Society Press.

Jacobsen, H. F., C. K. Gomard, and P. Sestoft (1993). Speeding up back propagation by partial evaluation. In N. C. S. R. F. Albrecht, C. R. Reeves (Ed.), *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, Innsbruck, Austria, pp. 63–66. Springer.

Jaffar, J. and M. J. Maher (1994). Constraint logic programming: A survey. *The Journal of Logic Programming 19 & 20*, 503–582.

Janssens, G. and M. Bruynooghe (1992). Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic programming 13*(2&3), 205–258.

Jones, N., P. Sestoft, and H. Søndergaard (1985). An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud (Ed.), *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pp. 124–140. Springer-Verlag.

Jones, N. D. (1996). What not to do when writing an interpreter for specialisation. In O. Danvy, R. Glück, and P. Thiemann (Eds.), *Partial Evaluation*, Volume 1110 of *Lecture Notes in Computer Science*, pp. 216–237. Springer-Verlag.

Jones, N. D., C. K. Gomard, and P. Sestoft (1993). *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

Jørgensen, J. and M. Leuschel (1996). Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann (Eds.), *Proceedings Dagstuhl Seminar on Partial Evaluation*, Schloss Dagstuhl, Germany, pp. 238–262. Springer-Verlag, LNCS 1110. Extended version as Technical Report CW 221, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

Jørgensen, J., M. Leuschel, and B. Martens (1997). Conjunctive partial deduction in practice. In J. Gallagher (Ed.), *Proceedings LOPSTR'96*, Stockholm, pp. 59–82. Springer-Verlag, LNCS 1207. Shorter and preliminary version in Proceedings of Benelog'96.

Karr, M. (1976). On affine relationships among variables of a program. *Acta Informatica 6*(2), 133–151.

Keppel, D., S. J. Eggers, and R. R. Henry (1991). A case for runtime code generation. Technical Report TR-91-11-04, University of Washington, Department of Computer Science and Engineering.

Keppel, D., S. J. Eggers, and R. R. Henry (1993). Evaluating runtime-compiled value-specific optimisations. Technical Report UW-CSE-93-11-02, Department of Computer Science and Engineering, University of Washington.

Kildall, G. A. (1973). A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pp. 194–206. ACM SIGACT and SIGPLAN: ACM Press.

Komorowski, J. (1992). An introduction to partial deduction. In A. Pettorossi (Ed.), *Proceedings Meta'92*, pp. 49–69. Springer-Verlag, LNCS 649.

Kuck, D. J., R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe (1981). Dependence graphs and compiler optimization. In *Proceedings of the Eighth Symposium on the Principles of Programming Languages (8th POPL'81), SIGPLAN Notices*, pp. 207–218.

Lafave, L. and J. P. Gallagher (1998). Constraint-based partial evaluation of rewriting-based functional logic programs. In *Program Synthesis and Transformation. 7th International Workshop, LOPSTR'97, Leuven, Belgium, July 1997, editor N. Fuchs.*, pp. 168–188. Springer-Verlag (LNCS 1463).

Lagoon, V. and P. J. Stuckey (2001). A framework for analysis of typed logic programs. In *Proceedings of FLOPS 2001*, Number 2024 in Lecture Notes in Computer Science, pp. 296–310. Springer-Verlag.

Lakhotia, A. and L. Sterling (1990). How to control unfolding when specializing interpreters. *New Generation Computing 8*(1), 61–70.

Lakshman, T. K. and U. S. Reddy (1991). Typed prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In K. Saraswat, Vijay; Ueda (Ed.), *Proceedings of the 1991 International Symposium on Logic Programming (ISLP'91)*, San Diego, CA, pp. 202–220. MIT Press.

Lassez, J.-L., M. J. Maher, and K. Marriott (1988). Unification revisited. In J. Minker (Ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 587–625. Morgan-Kaufmann.

Launchbury, J. (1990). Dependent sums express separation of binding times. In K. Davis and J. Hughes (Eds.), *Functional Programming, Glasgow, Scotland, 1989*, pp. 238–253. Springer-Verlag.

Lawall, J. (1999). Faster Fourier transforms via automatic program specialization. In J. Hatcliff, T. Mogensen, and P. Thiemann (Eds.), *Partial Evaluation: Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, Volume 1706 of *Lecture Notes in Computer Science*, pp. 338–355. Springer-Verlag.

Le Charlier, B. and P. Van Hentenryck (1994). Experimental evaluation of a generic abstract interpretation algorithm for PROLOG. *ACM Transactions on Programming Languages and Systems 16*(1), 35–101.

Lee, C. S., N. D. Jones, and A. M. Ben-Amram (2001). The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, Volume 28, pp. 81–92. ACM press.

Lee, P. and M. Leone (1996). Optimizing ML with run-time code generation. *ACM SIGPLAN Notices 31*(5), 137–148.

Leone, M. and P. Lee (1994). Lightweight run-time code generation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne.

Leuschel, M. (1995a). Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti (Ed.), Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, Utrecht, The Netherlands, pp. 1–16. Springer-Verlag.

Leuschel, M. (1995b). Partial evaluation of the "real thing". In L. Fribourg and F. Turini (Eds.), *Proceedings LOPSTR'94 and META'94*, pp. 122–137. Springer-Verlag, LNCS 883.

Leuschel, M. (1996). The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~lpai`.

Leuschel, M. (1997). *Advanced Techniques for Logic Program Specialisation*. Ph. D. thesis, K.U.Leuven.

Leuschel, M. (1998a). On the power of homeomorphic embedding for online termination. In G. Levi (Ed.), *Static Analysis. Proceedings*, Volume 1503 of *Lecture Notes in Computer Science*, Pisa, Italy, pp. 230–245. Springer-Verlag.

Leuschel, M. (1998b). Program specialisation and abstract interpretation reconciled. In J. Jaffar (Ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, Manchester, UK, pp. 220 – 234. MIT Press.

Leuschel, M. (1999). Improving homeomorphic embedding for online termination. In *Proceedings of LOPSTR'98*, Number 1559 in Lecture Notes in Computer Science, pp. 199–218.

Leuschel, M. and M. Bruynooghe (2001). Control issues in partial deduction: The ever ending story. Draft.

Leuschel, M. and D. De Schreye (1995). Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, La Jolla, California, pp. 253–263. ACM Press.

Leuschel, M. and D. De Schreye (1998). Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing 16*, 283 – 342.

Leuschel, M. and D. De Schreye (1998). Creating specialised integrity checks through partial evaluation of meta-interpreters. *Journal of Logic Programming 36*(2), 149 – 193.

Leuschel, M., D. De Schreye, and A. de Waal (1996). A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher (Ed.), *Proceedings JICSLP'96*, Bonn, Germany, pp. 319–332. MIT Press.

Leuschel, M., J. Jørgensen, W. Vanhoof, and M. Bruynooghe (2001). Offline specialisation in Prolog using a hand-written compiler generator. Submitted for publication.

Leuschel, M. and B. Martens (1996). Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann (Eds.), *Proceedings Dagstuhl Seminar on Partial Evaluation*, Schloss Dagstuhl, Germany, pp. 263–283. Springer-Verlag, LNCS 1110.

Leuschel, M., B. Martens, and D. De Schreye (1998). Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems 20*(1), 208 – 258.

Leuschel, M. and D. D. Schreye (1996). Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra (Eds.), *Proceedings of the International Symposium on Programming H Languages, Implementations, Logics and Programs (PLILP'96)*, Aachen, Germany, pp. 137–151. Springer-Verlag. LNCS 1140.

Leuschel, M. and M. H. Sørensen (1996). Redundant argument filtering of logic programs. In J. Gallager (Ed.), *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, Stockholm, Sweden. Springer-Verlag.

Levi, G. and G. Sardu (1988). Partial evaluation of metaprograms in a multiple worlds logic language. *New Generation Computing 6*(2&3), 227–247.

Lindenstrauss, N. and Y. Sagiv (1997). Automatic termination analysis of logic programs. In L. Naish (Ed.), *Proceedings of the 14th International Conference on Logic Programming*, Cambridge, pp. 63–77. MIT Press.

Lindenstrauss, N., Y. Sagiv, and A. Serebrenik (1997). TermiLog: A system for checking termination of queries to logic programs. In *Proc. 9th International Computer Aided Verification Conference*, pp. 444–447.

Lloyd, J. W. (1987a). Declarative error diagnosis. *New Generation Computing 5*, 133–154.

Lloyd, J. W. (1987b). *Foundations of Logic Programming*. Springer-Verlag.

Lloyd, J. W. and J. C. Shepherdson (1991). Partial evaluation in logic programming. *Journal of Logic Programming 11*(3&4), 217–242.

Marlet, R., C. Consel, and P. Boinot (1999). Efficient incremental run-time specialization for free. *ACM SIGPLAN Notices 34*(5), 281–292.

Marriott, K., L. Naish, and J.-L. Lassez (1988). Most specific logic programs. In R. A. Kowalski and K. A. Bowen (Eds.), *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, pp. 909–923. ALP, IEEE: The MIT Press.

Marriott, K. and H. Sndergaard (1993). Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems 2*(1-4), 181–196.

Martens, B. (1994). *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. Ph. D. thesis, Departement Computerwetenschappen, K.U.Leuven, Belgium.

Martens, B. and D. De Schreye (1995). Why untyped non-ground metaprogramming is not (much of) a problem. *Journal of Logic Programming 22*(1), 47–99.

Martens, B. and D. De Schreye (1996). Automatic finite unfolding using well-founded measures. *Journal of Logic Programming 28*(2), 89–146.

Martens, B., D. De Schreye, and T. Horváth (1994). Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science 122*(1-2), 97–117.

Martens, B. and J. Gallagher (1995). Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling (Ed.), *ICLP'95, Twelfth International Conference on Logic Programming, Tokyo, Japan, June 1995*, pp. 597–611. MIT Press.

Mesnard, F. (1996). Inferring left-terminating classes of queries for constraint logic programs. In M. Maher (Ed.), *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, Cambridge, pp. 7–21. MIT Press.

Mogensen, T. (1986). The application of partial evaluation to ray-tracing. Master's thesis, DIKU, University of Copenhagen, Denmark.

Mogensen, T. (1989). Binding Time Analysis for Polymorphically Typed Higher Order Languages. In J. Diaz and F. Orejas (Eds.), *TAPSOFT'89, Barcelona, Spain*, Volume 352 of *LNCS*, pp. 298–312. Springer-Verlag.

Mogensen, T. (1999). Partial evaluation: concepts and applications. In J. Hatcliff, T. Mogensen, and P. Thiemann (Eds.), *Partial Evaluation: Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, Volume 1706 of *Lecture Notes in Computer Science*, pp. 1–19. Springer-Verlag.

Mogensen, T. and A. Bondorf (1993). Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement (Eds.), *Proceedings LOPSTR'92*, pp. 214–227. Springer-Verlag, Workshops in Computing Series.

Mulkers, A., W. Simoens, G. Janssens, and M. Bruynooghe (1995). On the practicality of abstract equation systems. In *Proc. ICLP'95*, pp. 781–795. MIT Press.

Mumick, I. S., S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan (1990). Magic is relevant. *SIGMOD Record (ACM Special Interest Group on Management of Data) 19*(2), 247–258.

Mycroft, A. and R. A. O'Keefe (1984). A polymorphic type system for PROLOG. *Artificial Intelligence 23*(3), 295–307.

Nielson, F. (1983). A denotational framework for data flow analysis. *Acta Informatica 18*, 265–288.

Owen, S. (1989). Issues in the partial evaluation of meta-interpreters. In H. D. Abramson and M. H. Rogers (Eds.), *Proceedings Meta'88*, pp. 319–339. MIT Press.

Pagan, F. (1990). Comparative efficiency of general and residual parsers. *SIGPLAN Notices 25*(4), 59–65.

Palsberg, J. (1995). Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems 17*(1), 47–62.

Patterson, J. R. C. (1995). Accurate static branch prediction by value range propagation. *ACM SIGPLAN Notices 30*(6), 67–78.

Penello, T. (1986). Very fast LR parsing. In *SIGPLAN'86 Conference on Compiler Construction*, pp. 145–151. ACM.

Pettorossi, A. and M. Proietti (1994). Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming 19/20*, 261–320.

Pfenning, F. (Ed.) (1992). *Types in Logic Programming*. MIT Press.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992). *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.)*. Cambridge: Cambridge University Press. ISBN 0-521-43108-5.

Prestwich, S. (1993). Online partial deduction of large programs. In *Proceedings PEPM'93*, Copenhagen, Denmark, pp. 111–118. ACM.

Proietti, M. and A. Pettorossi (1993). The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming 16*(1&2), 123–161.

Puebla, G. and M. Hermenegildo (1999). Some issues in analysis and specialization of modular Ciao-Prolog programs. In M. Leuschel (Ed.), *Proceedings of the Workshop on Optimization and Implementation of Declarative Languages*, Las Cruces. In Electronic Notes in Theoretical Computer Science, Volume 30 Issue No.2, Elsevier Science.

Reddy, U. S. (1985). Narrowing as the operational semantics of functional languages. In *IEEE Symposium on Logic Programming*, pp. 138–151. New York, NY: IEEE Computer Society Press.

Ridoux, O., P. Boizumault, and F. Malésieux (1999). Typed static analysis: Application to groundness analysis of Prolog and lambda-Prolog. In *Proceedings of FLOPS'99*, Number 1722 in Lecture Notes in Computer Science, pp. 267–283. Springer-Verlag.

Rytz, B. and M. Gengler (1992). A polyvariant binding time analysis. In *PEPM92*, pp. 21–28. Yale.

Safra, S. and E. Shapiro (1986). Meta interpreters for real. In H.-J. Kugler (Ed.), *Information Processing 86*, pp. 271–278.

Sahlin, D. (1993). Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing 12*(1), 7–51.

Saraswat, V. A., K. Kahn, and J. Levy (1990). Janus: A step towards distributed constraint programming. In M. Hermenegildo and S. Debray (Eds.), *Proceedings of the 1990 North American Conference on Logic Programming*, Austin, TX, pp. 421–438. MIT Press.

Scheifler, R. W. (1977). An analysis of inline substitution for a structured programming language. *Communications of the ACM 20*(9), 647–654.

Seki, H. (1989). On the power of alexander templates. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '89)*, pp. 150–159. ACM Press.

Sestoft, P. (1986). The structure of a self-applicable partial evaluator. In H. Ganzinger and N. Jones (Eds.), *Programs as Data Objects 1985*, Number 217 in Lecture Notes in Computer Science, pp. 236–256. Copenhagen, Denmark: Springer-Verlag.

Sestoft, P. (1988). Automatic call unfolding in a partial evaluator. In D. Bjørner, A. Ershov, and N. Jones (Eds.), *Partial Evaluation and Mixed Computation*, pp. 485–506. North-Holland.

Singh, S. and N. McKay (1999). Partial evaluation of hardware. In J. Hatcliff, T. Mogensen, and P. Thiemann (Eds.), *Partial Evaluation: Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, Volume 1706 of *Lecture Notes in Computer Science*, pp. 221 – 230. Springer-Verlag.

Smaus, J.-G., P. Hill, and A. King (2000). Mode analysis domains for typed logic programs. In A. Bossi (Ed.), *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation*, pp. 82 – 101. Springer-Verlag.

Somogyi, Z. et al. The Melbourne Mercury compiler, release 0.9.

Somogyi, Z., F. Henderson, and T. Conway (1994). The implementation of Mercury, an efficient purely declarative logic programming language. In *Proceedings of the ILPS'94 Postconference Workshop on Implementation Techniques for Logic Programming Languages*.

Somogyi, Z., F. Henderson, and T. Conway (1995). Logic programming for the real world. In *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming*.

Somogyi, Z., F. Henderson, and T. Conway (1996). The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29*(1–3), 17–64.

Somogyi, Z., F. Henderson, T. Conway, A. Bromage, T. Dowd, D. Jeffery, P. Ross, P. Schachte, and S. Taylor (1996). Status of the Mercury system. In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*.

Sørensen, M. (1994). Turchin's supercompiler revisited. Master's thesis, DIKU. DIKU Research Report 94/9.

Sørensen, M., R. Glück, and N. Jones (1994). Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella (Ed.), *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pp. 485–500. Springer-Verlag.

Sørensen, M. H. and R. Glück (1995). An algorithm of generalization in positive supercompilation. In J. Lloyd (Ed.), *Proceedings ILPS'95*, Portland, Oregon, pp. 465–479. MIT Press.

Sørensen, M. H., R. Glück, and N. D. Jones (1996). A positive supercompiler. *Journal of Functional Programming 6*(6), 811–838.

Sterling, L. and R. D. Beer (1989). Meta interpreters for expert system construction. *Journal of Logic Programming 6*(1&2), 163–178.

Sterling, L. and E. Shapiro (1986). *The Art of Prolog*. MIT Press.

Takeuchi, A. and K. Furukawa (1986). Partial evaluation of Prolog programs and its application to metaprogramming. In H.-J. Kugler (Ed.), *Information Processing 86*, pp. 415–420.

Tamaki, H. and T. Sato (1986). OLD resolution with tabulation. In E. Shapiro (Ed.), *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, London, pp. 84–98. Springer-Verlag.

Thompson, S. (1996). *Haskell: The Craft of Functional Programming*. Harlow, England: Addison-Wesley.

Turchin, V. F. (1986). The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems 8*(3), 292–325.

Ullman, J. D. (1989a). Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '89)*, Philadelphia, PA, USA, pp. 140–149. ACM Press.

Ullman, J. D. (1989b). *Database and Knowledge-Base Systems, Volume II*. Computer Science Press.

van Harmelen, F. (1989). The limitations of partial evaluation. In P. Jackson, H. Reichgelt, and F. van Harmelen (Eds.), *Logic-Based Knowledge Representation*, pp. 87–111. MIT Press.

Vanhoof, W. (1997). Bottom up information propagation for partial deduction. In M. Leuschel (Ed.), *Proceedings of the International Workshop on Specialization of Declarative Programs and its Applications*, Port Jefferson, Long Island N.Y. (USA), pp. 73 – 82. Proceedings available as report CW255, Dept. of Computer Science, K.U.Leuven, Belgium.

Vanhoof, W. (2000). Binding-time analysis by constraint solving: a modular and higher-order approach for Mercury. In M. Parigot and A. Voronkov (Eds.), *Logic for Programming and Automated Reasoning, 7th Intl. Conference, LPAR2000*, Volume 1955 of *Lecture Notes in Artificial Intelligence*, Reunion Island, France, pp. 399 – 416.

Vanhoof, W. and M. Bruynooghe (1999a). Binding-time analysis for Mercury. In *Pre-proceedings of LOPSTR'99*, Venice, Italy. Technical Report, Univeristy of Venice.

Vanhoof, W. and M. Bruynooghe (1999b). Binding-time analysis for Mercury. In D. D. Schreye (Ed.), *16th International Conference on Logic Programming*, pp. 500 – 514. MIT Press.

Vanhoof, W. and M. Bruynooghe (1999c). Towards modular binding-time analysis for first-order Mercury. In M. Leuschel (Ed.), *Proceedings of the Workshop on Optimization and Implementation of Declarative Languages*, Las Cruces. In Electronic Notes in Theoretical Computer Science, Volume 30 Issue No.2, Elsevier Science.

Vanhoof, W. and M. Bruynooghe (2000). Towards a modular binding-time analysis for higher-order Mercury. In K. K. Lau (Ed.), *Pre-proceedings of LOPSTR 2000*, London, United Kingdom.

Vanhoof, W. and M. Bruynooghe (2001). Binding-time annotations without binding-time analysis (extended abstract). In N. Dershowitz (Ed.), *Extended Abstracts of the Fifth International Workshop on Termination (WST'01)*, Utrecht, The Netherlands, pp. 52 – 54.

Vanhoof, W., B. Martens, D. D. Schreye, and K. D. Vlaminck (1998). Specialising the other way around. In J. Jaffar (Ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Manchester, United Kingdom, pp. 279–293. MIT-Press.

Vanhoof, W., D. D. Schreye, and B. Martens (1998a). Bottom up specialisation of logic programs (abstract). In P. Flener (Ed.), *Proceedings of LOPSTR'98*, Volume 1559 of *Lecture Notes In Computer Science*, Manchester, UK, pp. 325 – 327. Springer-Verlag.

Vanhoof, W., D. D. Schreye, and B. Martens (1998b). A framework for bottom up specialisation of logic programs. In C. Palamidessi, H. Glaser, and K. Meinke (Eds.), *Proceedings of the Joint International Symposia PLILP/ALP 1998*, Volume 1490 of *Lecture Notes In Computer Science*, Pisa, Italy, pp. 54–72. Springer-Verlag.

Vanhoof, W., D. D. Schreye, and B. Martens (1998c). A framework for bottom up specialisation of logic programs (abstract). In H. L. Poutré and J. van den Herik (Eds.), *Proceedings of the 10th Netherlands/Belgium Conference on Artificial Intelligence (NAIC'98)*, pp. 277 – 278.

Vanhoof, W., D. D. Schreye, and B. Martens (1999). Bottom-up partial deduction of logic programs. *The Journal of Functional and Logic Programming 1999*, 1–33.

Venken, R. (1984). A Prolog meta interpreter for partial evaluation and its application to source to source transformation and query optimization. In T. O'Shea (Ed.), *Advances in Artificial Intelligence, Proceedings ECAI'84*, pp. 347–356. North-Holland.

Verschaetse, K. and D. De Schreye (1991). Deriving Termination Proofs for Logic Programs, using Abstract Procedures. In K. Furukawa (Ed.), *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France, pp. 301–315. The MIT Press.

Volanschi, E. N., C. Consel, and C. Cowan (1997). Declarative specialization of object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, Volume 32, 10 of *ACM SIGPLAN Notices*, New York, pp. 286–300. ACM Press.

Wegman, M. N. and F. K. Zadeck (1991). Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*(2), 181–210.

Weise, D., R. Conybeare, E. Ruf, and S. Seligman (1991). Automatic online partial evaluation. In J. Hughes (Ed.), *Proceedings of the 3rd ACM Conference on Functional Programming Languages and Computer Architecture*, Cambridge, Massachusets, USA, pp. 165–191. Springer-Verlag, LNCS 523.

Technieken voor On- en Off-line Specialisatie van Logische Programma's

Nederlandse Samenvatting

Inhoudsopgave

In deze Nederlandse samenvatting pogen we op een bondige, en eerder informele manier de belangrijkste lijnen van de thesis weer te geven. De structuur van deze samenvatting volgt grotendeels de structuur van de thesis zelf. In een eerste paragraaf geven we een algemene inleiding betreffende het fenomeen specialisatie, meer bepaald in de context van logisch programmeren. Paragrafen 2 tot en met 5 schetsen dan de pijlers van het eigenlijke onderzoek waarover in de thesis gerapporteerd wordt. Ze behandelen op hoog niveau de thema's die in hoofdstukken 3 tot en met 8 van de thesistekst gedetailleerd behandeld worden. Paragraaf 2 behelst de ontwikkeling van een bindingstijd-analyse voor Mercury, beschreven in hoofdstukken 3 en 4 van de thesis. Vervolgens schetst Paragraaf 3 een verschillende aanpak van het probleem, nodig om een bindingstijd-analyse voor Prolog te kunnen ontwikkelen, en komt overeen met hoofdstuk 5 in de thesis. Paragraaf 4 behandelt dan het werk beschreven in hoofdstuk 6 van de Engelse tekst, en verhaalt ons werk rond het uitbreiden van een klassieke *Pos*-analyse met type informatie. Vervolgens schetst Paragraaf 5 het werk rond de ontwikkeling van een nieuwe specialisatietechniek, bottom-up specialisatie, en beschrijft de resultaten uit hoofdstukken 7 en 8 van de thesistekst. Tenslotte herhalen we in een zesde paragraaf de belangrijkste verwezenlijkingen van dit werk.

1 Inleiding

Programma-uitvoering wordt typisch gezien als een éénstaps proces, waarin een programma P simpelweg uitgevoerd wordt met betrekking tot zijn volledige invoer, wat resulteert in een bepaalde uitvoer. Soms kan het echter gebeuren dat verschillende delen van de invoer op verschillende tijdstippen bekend worden, en kan het de moeite lonen – in termen van efficiëntie – om het programma in verschillende stappen uit te voeren: wanneer een *gedeelte* van de invoer bekend is, kan het programma *gespecialiseerd* worden met betrekking tot dit gedeelte van de invoer. Dit specialisatieproces houdt in dat de berekeningen in P die enkel afhangen van het gekende gedeelte van de invoer effectief uitgevoerd worden, terwijl het resultaat van deze berekeningen geregistreerd wordt in een nieuw programma, samen met de resterende programmacode van de berekeningen die nog niet konden uitgevoerd worden. Het resultaat van de specialisatie is een nieuw programma P' dat, wanneer het uitgevoerd wordt met betrekking tot het resterende gedeelte van de invoer *hetzelfde* resultaat berekent als P. In dit werk volgen we de traditionele benadering van programmaspecialisatie, en behandelen we programma-uitvoering als een tweestaps proces: de invoer van een programma wordt opgesplitst in een *statisch* en een *dynamisch* gedeelte. In een eerste fase wordt het programma gespecialiseerd met betrekking tot het statische gedeelte van de invoer. Het resulterende programma wordt vaak aangeduid als het *residuele* programma, omdat het samengesteld is uit de resterende code van het originele programma die niet uitgevoerd kon worden tijdens de specialisatie. De tweede fase bestaat dan uit de uitvoering

van het residuele programma met betrekking tot de dynamische invoer.

Het optimaliseren van programmacode (met betrekking tot de efficiëntie) wordt vaak gezien als de belangrijkste toepassing van programmaspecialisatie. Inderdaad, wanneer een programma bijvoorbeeld een aantal keer uitgevoerd moet worden met een deel van zijn invoer onveranderlijk, dan kan het de moeite lonen om het programma te specialiseren met betrekking tot deze onveranderlijke invoer, en nadien telkens het residuele programma uit te voeren met betrekking tot de wisselende invoer. Aangezien de uitvoering van het residuele programma minder bewerkingen omvat, is het in het algemeen veilig te stellen dat deze uitvoering sneller zal zijn dan de uitvoering van het originele programma. Ook indien de invoer van het programma niet op verschillende tijdstippen bekend wordt, kan het toch de moeite lonen om het programma (of delen ervan) te specialiseren. Vaak wordt immers bij het schrijven van software gebruik gemaakt van algemene software componenten die in een aantal, mogelijk verschillende, contexten gebruikt worden. Programmaspecialisatie is in staat om zo'n algemene component te specialiseren, eens voor elke context waarin hij gebruikt wordt. Omdat de resulterende componenten minder algemeen zijn, is de uitvoering ervan vaak efficiënter dan de uitvoering van de originele component. Afgezien van de snelheidswinsten die met programmaspecialisatie gehaald kunnen worden, speelt de techniek een belangrijke rol in het onderzoek rond programmageneratie, en maakt het mogelijk om vertalers en vertaler generatoren af te leiden uit vertolkers. Voor een overzicht van zowel de gebruikte technieken als mogelijke toepassingen van programmaspecialisatie verwijzen we naar (Jones, Gomard, and Sestoft 1993).

Programmaspecialisatie kan gezien worden als een combinatie van (partiële) uitvoering en codegeneratie, en het proces wordt vaak beschreven als *partiële evaluatie*. Het basisprobleem waar elke specialisator mee te kampen krijgt, is te beslissen *welke* opdrachten tijdens de specialisatie uitgevoerd moeten worden. Dit probleem wordt in de literatuur benaderd vanuit een aantal verschillende uitvalshoeken, maar profileert zich meestal als een trade-off tussen terminatie van het specialisatieproces en de kwaliteit van de behaalde specialisatie. In de literatuur wordt voornamelijk een onderscheid gemaakt tussen zogenaamde *on-line* en *off-line* technieken.

1.1 On- en Off-line Programmaspecialisatie

On-line programmaspecialisatie kan gezien worden als een speciale vorm van uitvoering van het programma, waar het programma uitgevoerd wordt met concrete maar partiële invoer onder de supervisie van een controlesysteem. Zo'n controlesysteem bekijkt voortdurend het specialisatieproces, en kan op elk punt beslissen om de uitvoering te stoppen, code te genereren, en de uitvoering op een andere plaats in het programma te hervatten. On-line specialisatoren houden gewoonlijk de volledige uitvoeringsgeschiedenis van het programma bij (bijvoorbeeld onder de vorm van de opeenvolgende recursieve oproepen die tijdens de uitvoering ge-

maakt werden) en gebruiken deze informatie bij het maken van hun controlebeslissingen. Samen met het feit dat een on-line specialisator werkt met *concrete* (maar partiële) invoer maakt deze vorm van controle on-line specialisatie tot een krachtige techniek waarmee gewoonlijk een hogere specialisatiegraad gehaald kan worden dan met andere, zogenaamde *off-line* specialisatoren. De te betalen prijs is de efficiëntie van de specialisator zelf: het continu bekijken van de uitvoeringsgeschiedenis van het programma en voor elke opdracht tijdens het proces beslissen of ze al dan niet geëvalueerd moet worden maakt de specialisator inherent trager dan een competitieve off-line specialisator.

In tegenstelling tot een on-line specialisator, werkt een *off-line* specialisator in twee fasen. Eerst wordt het te specialiseren programma geanalyseerd door een zogenaamde *bindingstijd-analyse*. Zo'n analyse bekijkt niet de concrete invoer van het programma, maar eerder een *beschrijving* van deze invoer in termen van welke gedeelten ervan bekend zullen zijn tijdens de specialisatie, en welke gedeelten ervan slechts beschikbaar worden voor de uiteindelijke uitvoering van het gespecialiseerde programma. De meest gebruikte beschrijving is die waar ieder deel van de invoer gekarakteriseerd wordt als zijnde *static* dan wel *dynamic*. Deze beschrijving noemt men de *bindingstijd* van het invoergedeelte, een begrip dat verwijst naar de fase van het uitvoeringsproces waarin het invoergedeelte gekend wordt: *static* beschrijft invoer die gekend is tijdens de specialisatie van het programma, terwijl *dynamic* invoer beschrijft die mogelijk nog niet gekend is tijdens de specialisatie, maar pas bekend wordt tijdens de uitvoering van het gespecialiseerde programma. De taak van de bindingstijd-analyse is dan om voor elke variabele in het programma zo'n bindingstijd karakterisering af te leiden die beschrijft of de waarde van de betreffende variabele al dan niet gekend zal zijn tijdens specialisatie. Bovendien associeert de bindingstijd-analyse aan elke opdracht in het programma een *annotatie* die als het ware een instructie voor de specialisator is die zegt of de opdracht al dan niet uitgevoerd moet worden tijdens specialisatie. Deze annotaties implementeren als dusdanig de controle van het off-line specialisatieproces, welke uitsluitend gebaseerd is op de bindingstijden van de betrokken variabelen, *niet* op de concrete invoer van de opdracht, noch op enige uitvoeringsgeschiedenis zoals het geval is bij on-line specialisatoren. De uiteindelijke specialisatie van het programma wordt bereikt in een tweede fase, door díe opdrachten uit te voeren die als dusdanig geannoteerd werden door de bindingstijd-analyse.

Zowel on- als off-line programmaspecialisatie is bestudeerd voor tal van paradigma's, onder meer in de context van imperatieve talen, bijvoorbeeld (Weise, Conybeare, Ruf, and Seligman 1991; Andersen 1993; Consel and Noël 1996), functionele talen, zoals in (Sørensen and Glück 1995; Jones, Gomard, and Sestoft 1993) en logische talen (Gallagher 1993; Leuschel, Martens, and De Schreye 1998). In deze thesis onderzoeken we programmaspecialisatie in de context van logische programmeertalen, ook *partiële deductie* genoemd. In een volgende paragraaf van deze inleiding gaan we dieper in op deze logische programmeertalen en de specialisatie

van logische programma's.

1.2 Specialisatie van Logische Programma's

In deze paragraaf geven we een bondige inleiding tot logisch programmeren en de specialisatie van logische programma's. Voor een gedetailleerde introductie tot logisch programmeren verwijzen we naar (Lloyd 1987; Apt 1990), en naar (Gallagher 1993) voor meer details omtrent de specialisatie van logische programma's (partiële deductie).

1.2.1 Een Beknopte Inleiding tot Logisch Programmeren

Beschouwen we een eerste orde taal met Π de verzameling predikaatsymbolen, Σ de verzameling functiesymbolen en $\mathcal{V}$ de verzameling variabelen. Aan een predikaat- of functiesymbool f is een ariteit n geassocieerd – wat we noteren als f/n – die het aantal argumenten van het symbool definieert. Een term opgebouwd door elementen uit Σ en $\mathcal{V}$ is ofwel een variabele (van $\mathcal{V}$) of een functiesymbool $f/n \in \Sigma$ toegepast op een sequentie van n termen. De verzameling van al zo'n termen wordt aangeduid met $\mathcal{T}(\mathcal{V}, \Sigma)$. Voor een term t duiden we met $\mathcal{V}(t)$ de verzameling variabelen die voorkomen in t aan. Een term t is *gegrond* (of *ground*) wanneer $\mathcal{V}(t) = \emptyset$. Een atoom is een predikaatsymbool $p/n \in \Pi$ toegepast op een sequentie van n termen. In wat volgt beperken we ons tot de zogenaamde klasse van *positieve* programma's (*definite programs*). Een clause is een formule van de vorm

$$H \leftarrow B_1, \ldots, B_n \text{ met } n \geq 0$$

waarin H en $B_1, \ldots, B_n$ atomen voorstellen. De formule dient gelezen te worden als een logische implicatie waarvan het voorwaardelijk gedeelte, $B_1, \ldots, B_n$ een conjunctie van atomen is en het gevolg van de implicatie een atoom. Het atoom H wordt ook het hoofd van de clause genoemd, de conjunctie $B_1, \ldots, B_n$ het lichaam. Alle variabelen die voorkomen in het lichaam van een clause zijn universeel gekwantificeerd over het lichaam. Een (positief) logisch programma is dan een verzameling clauses. Een clause met een leeg hoofd en een niet-leeg lichaam, $\leftarrow B_1, \ldots, B_n$ $(n \geq 1)$ wordt een *query* of *doel* genoemd. Een substitutie is gedefinieerd als een eindige afbeelding $\mathcal{V} \mapsto \mathcal{T}(\Sigma, \mathcal{V})$ van verschillende variabelen naar termen. Indien E een expressie is (een term, een atoom, een clause of een query) en θ een substitutie, dan duidt $E\theta$ het resultaat aan van het toepassen van de substitutie θ op de expressie E en is gedefinieerd als de expressie die verkregen wordt door alle variabelen uit het domein van θ simultaan te vervangen in E door de overeenkomstige term uit θ. Een belangrijke operatie op termen en atomen is *unificatie*:

Definitie 1 *Laat S een verzameling termen of atomen zijn. Een substitutie θ is een* unificator *voor S indien $S\theta$ een singleton is. Een unificator θ voor S wordt*

een meest algemene unificator *(mgu) voor S genoemd indien voor elke unificator σ voor S geldt dat er een andere substitutie γ bestaat zodat $\sigma = \theta\gamma$.*

Het uitvoeren van een logisch programma wordt verwezenlijkt door SLD-resolutie. Exacte definities kunnen weerom gevonden worden in (Lloyd 1987; Apt 1990).

Definitie 2 *Laat Q de query $\leftarrow A_1, \ldots, A_k, \ldots, A_n$ zijn en C de clause $A \leftarrow B_1, \ldots, B_q$. Dan zeggen we dat de query Q'* afgeleid *is van Q en C via de meest algemene unificator θ indien aan de volgende voorwaarden voldaan is:*

- *A_k is een atoom, en wordt het* geselecteerde *atoom in Q genoemd*
- *θ is een meest algemene unificator voor A_k en A*
- *Q' is de query $\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_q, A_{k+1}, \ldots, A_n)\theta$.*

Indien P een programma en Q_0 een query is, dan definiëren we een SLD-afleiding van $P \cup \{Q_0\}$ als een mogelijk oneindige sequentie van queries $Q_0, Q_1, Q_2, \ldots$, een sequentie van varianten van programma clauses $C_1, C_2, \ldots$ en een sequentie $\theta_1, \theta_2, \ldots$ van meest algemene unificatoren zodat elke Q_{i+1} afgeleid is van Q_i en C_{i+1} gebruik makend van θ_{i+1}. Een SLD-afleiding kan eindig of oneindig zijn. Een SLD-afleiding die eindigt in de lege query wordt een succesvolle afleiding genoemd, alsook een SLD-*refutatie*. Een SLD-afleiding die eindigt in een query waarvan het geselecteerde atoom niet unificeert met het hoofd van een programma clause wordt een falende SLD-afleiding genoemd.

Het basis uitvoeringsmechanisme van logische programma's bestaat uit de constructie van SLD-afleidingen voor een query en een programma. Indien een succesvolle afleiding geconstrueerd kan worden, is men geïnteresseerd in wat er effectief "berekend" wordt door de afleiding. Dit wordt een *berekend antwoord* genoemd, en is gedefinieerd als de samenstelling $\theta_1\theta_2 \ldots \theta_n$ van de meest algemene unificatoren uit de succesvolle afleiding. Het construeren van SLD-afleidingen voor een query en een programma vergt een bepaalde strategie. Inderdaad, tijdens het bouwen van een afleiding moet een bepaald atoom geselecteerd worden en, indien dit atoom unificeert met het hoofd van meerdere clauses in het programma, dient één van de clauses geselecteerd te worden om de afleiding verder te zetten. Een *selectieregel* bepaalt, voor een willekeurige query, welk atoom in de query geselecteerd wordt voor verdere afleiding. Gegeven een selectieregel kan een SLD-*boom* geconstrueerd worden.

Definitie 3 *Gegeven een programma P en een query Q. Een SLD-*boom *voor $P \cup \{Q\}$ is een boom waarin elke knoop een – mogelijk lege – query is, de wortel van de boom de query Q is, en voor elke knoop $\leftarrow A_1, \ldots, A_k, \ldots, A_n$ (met $n \geq 1$) in de boom hebben we het volgende: indien A_k het geselecteerde atoom is, dan heeft de knoop een kind knoop van de vorm*

$$\leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_m, A_{k+1}, \ldots, A_n)\theta$$

voor een variant $A \leftarrow B_1, \ldots, B_m$ *van een clause in* P *met* θ *een meest algemene unificator van* A_k *en* A.

Elke tak in een SLD-boom is een SLD-afleiding. Indien minstens één van de takken een oneindige afleiding is, dan wordt de boom oneindig genoemd. Anders is het een eindige boom. Indien alle takken van de boom falende afleidingen zijn, wordt de boom eindig falend genoemd.

1.2.2 Van Uitvoering naar Specialisatie

Het uitvoeren van een logisch programma bestaat uit het evalueren van een query Q met betrekking tot een programma P door het bouwen van een SLD-boom voor $P \cup \{Q\}$. De invoer voor het programma is vervat in de atomen en termen in Q. *Partiële* invoer van een programma wordt voorgesteld door een minder geïnstantieerde query Q'. In wat volgt zullen we verwijzen naar een *partiële deductie query* als een generalisatie van de queries waarin we geïnteresseerd zijn om het programma voor uit te voeren. Door het karakter van logisch programmeren, kunnen we eveneens een SLD-boom bouwen voor zo'n meer algemene query en lijkt het proces van "partiële" deductie neer te komen op de gewone deductie. Hoewel, in het algemeen zal de resulterende SLD-boom oneindig zijn en de constructie ervan een niet-eindigend proces. Om zo'n mogelijk oneindige constructie te benaderen met een eindige (en bijgevolg in de praktijk toepasbare) constructie, bestaat partiële deductie erin van een *eindig* aantal *partiële* SLD-bomen te bouwen, die samen de volledige – en mogelijk oneindige – SLD-boom "bedekken". Het raamwerk dat partiële deductie definieert werd ontwikkeld door Lloyd en Shepherdson (Lloyd and Shepherdson 1991). We raken hier enkel de meest belangrijke kernbegrippen aan, en verwijzen naar (Lloyd and Shepherdson 1991) voor meer formele en gedetailleerde definities.

De basisidee omvat de constructie van een *partiële* SLD-boom. Dit is een variant van een SLD-boom waarin de SLD-afleidingen niet alleen succesvol of falend kunnen zijn, maar tevens kunnen eindigen in een query waarin geen enkel atoom geselecteerd is. Het selecteren van een atoom in een blad van een SLD-boom en het toevoegen van alle kinderen wordt *ontvouwen* genoemd. Bijgevolg kan een partiële SLD-boom verkregen worden door het herhaaldelijk ontvouwen van een initiële query – de wortel van de boom. In het standaard raamwerk van partiële deductie wordt de initiële query verondersteld een atoom te zijn. Met elke SLD-derivatie uit een partiële SLD-boom is een clause geassocieerd op de volgende manier:

Definitie 4 *Indien* P *een programma is,* A *een atoom, en* $A \overset{\theta}{\rightsquigarrow}_P B_1, \ldots, B_n$ *een SLD-afleiding voor* $P \cup \{\leftarrow A\}$*, dan is de* resultante *van de afleiding gedefinieerd als de clause*

$$A\theta \leftarrow B_1, \ldots, B_n.$$

Indien τ een partiële SLD-boom is voor $P \cup \{\leftarrow A\}$, wordt de verzameling resultanten die geassocieerd is aan de niet-falende takken van τ een *partiële deductie van A in P* genoemd. Indien $\mathcal{A} = \{A_1, \ldots, A_n\}$ een eindige verzameling atomen is, dan is de *partiële deductie van $\mathcal{A}$ in P* gedefinieerd als de unie van de partiële deducties van $A_1, \ldots, A_n$ in P. In (Lloyd and Shepherdson 1991) worden de nodige voorwaarden gedefinieerd waaronder een partiële deductie correct is. Het standaard raamwerk voor partiële deductie wordt uitgebreid tot partiële deductie van *conjuncties* van atomen in (De Schreye et al. 1999).

Het raamwerk voor partiële deductie (Lloyd and Shepherdson 1991) definieert wat een partiële deductie is, en onder welke voorwaarden de transformatie correct is. Het effectief construeren van een bruikbare partiële deductie van een verzameling atomen $\mathcal{A}$ in een programma P vergt een *controlestrategie*, die rekening moet houden met de volgende aspecten:

- Het construeren van een *eindige* en niet-triviale partiële SLD-boom voor $P \cup \{\leftarrow A\}$ met A een atoom in $\mathcal{A}$. De vorm van de SLD-boom bepaalt de structuur van een predikaat in het gespecialiseerde programma aangezien de clauses van dit predikaat geconstrueerd worden vanuit de resultanten van de SLD-boom.

- De verzameling atomen $\mathcal{A}$ moet zodanig gekozen worden dat aan de voorwaarden voor correctheid van (Lloyd and Shepherdson 1991) voldaan is met betrekking tot de partiële deductie query Q.

De constructie van een eindige SLD-boom wordt vaak de *locale* controle genoemd, terwijl naar het bepalen van de verzameling $\mathcal{A}$ vaak verwezen wordt onder de benaming *globale* controle. Tal van controlestrategieën zijn voorgesteld in de literatuur. De overgrote meerderheid van deze, zowel locale als globale strategieën is on-line aangezien ze de inhoud (en soms de vorm) van de geconstrueerde SLD-bomen in rekening brengen, enerzijds om te bepalen waar constructie van zo'n SLD-boom moet stoppen (locale controle) als voor welke atomen een SLD-boom geconstrueerd moet worden (globale controle). Voor een recent overzicht van controlestrategieën verwijzen we naar het overzichtsartikel van Leuschel en Bruynooghe (Leuschel and Bruynooghe 2001).

2 Bindingstijd-analyse voor Mercury

We schrijven oktober 1993 wanneer onderzoekers van de Melbourne University beginnen met de ontwikkeling van een nieuwe logische programmeertaal, genaamd Mercury. Hoewel logische programmeertalen reeds geruime tijd voorhanden waren, leek geen van de bestaande talen de voordelen die zo'n hoog-niveau taal zou bieden ten overstaan van meer klassieke, imperatieve talen, ten volle te realiseren. De voordelen van zo'n logische taal worden in (Somogyi, Henderson, and

Conway 1995) als volgt samengevat: een hogere graad van expressiviteit (waarin de programmeur eerder kan uitdrukken *wat* het probleem is dat moet opgelost worden dan *hoe* het probleem opgelost moet worden), het bestaan van een nuttige formele semantiek van de taal – een semantiek die onafhankelijk is van volgorde waarin de opdrachten in het programma worden uitgevoerd – en de mogelijkheden voor zogenaamde *declaratieve* debugging. Het belangrijkste uitgangspunt bij de ontwikkeling van Mercury was het creëren van een logische taal die *zuiver* zou zijn, en geschikt voor de ontwikkeling van grote, reële-wereld toepassingen. De belangrijkste pijlers waarop de ontwikkeling van Mercury steunt, kunnen als volgt samengevat worden: *Ondersteuning bij het ontwikkelen van betrouwbare programma's.* Dit behelst een taal die toestaat om bepaalde categorieën van fouten op te sporen tijdens de compilatie van het programma. *Ondersteuning voor het ontwikkelen van software in teams.* De taal moet dus de mogelijkheid bieden om één toepassing te creëren door het samenvoegen van verschillende onderdelen die ontwikkeld worden door verschillende programmeurs – vaak in isolatie. Deze laatste twee pijlers vormen een belangrijke conceptuele scheiding met eerdere logische talen zoals Prolog, die toendertijd zo goed als geen ondersteuning boden voor het creëren van grote toepassingen in teamverband. De andere pijlers waarop de ontwikkeling van Mercury berust zijn terug te vinden in (Somogyi, Henderson, and Conway 1995): een efficiënt uitvoeringsmechanisme, ondersteuning voor het onderhoud van programmatuur en de mogelijkheid tot het koppelen met gegevensbanken.

Om dit doel te bereiken werd Mercury uitgerust met een systeem van type-, mode- en determinisme declaraties. Deze declaraties vormen uitstekende commentaar over hoe de gegevens die in een predikaat gebruikt worden eruit zullen zien en over hoe de code verondersteld wordt gebruikt te worden. Bovendien vormen deze declaraties de basis van een efficiënt uitvoeringsmechanisme. Verder biedt Mercury een modern module systeem aan waarmee zowel gegevens als code ingekapseld kunnen worden en waarmee de ondersteuning voor het creëren van grote programma's door verschillende programmeurs verzekerd wordt.

Een eerste bijdrage van dit werk situeert zich in de ontwikkeling van een bindingstijd-analyse voor Mercury. In deze samenvatting staan we kort stil bij de belangrijkste pijlers van deze bijdrage. De details van dit werk zijn terug te vinden in hoofdstukken 3 en 4 van de thesis.

2.1 Basisingrediënten van Bindingstijd-analyse

Zoals het geval is in de meeste declaratieve talen, worden gegevens in Mercury meestal voorgesteld als gestructureerde termen. Gegevens die slechts gedeeltelijk gekend zijn bestaan bijgevolg vaak uit een partieel geïnstantieerde term. Een eerste vereiste om een nauwkeurige, en in de praktijk nuttige bindingstijd-analyse te verkrijgen is dan ook de mogelijkheid om de *bindingstijd* van een waarde – die een karakterisering is van "hoeveel" van een waarde gekend is tijdens de specialisatie

– op een voldoende precieze manier te kunnen benaderen. De meest eenvoudige karakterisering zegt van een waarde of ze *static* dan wel *dynamic* is. Static wil in deze context zeggen dat de waarde als geheel gekend is tijdens de specialisatie (en dus een gegronde term zal zijn), terwijl *dynamic* zegt dat de waarde mogelijk niet ground is, en bijgevolg variabelen kan bevatten. Aangezien op deze manier elke niet volledig geïnstantieerde term gekarakteriseerd zal worden als *dynamic* wordt een bindingstijd-analyse die gebruik maakt van dit domein in het algemeen als te zwak ervaren voor een taal als Mercury.

In de thesis introduceren we een domein van bindingstijden dat een meer precieze karakterisering van een waarde toelaat. De basisidee is om het type van de waarde – dat gekend is in Mercury – te gebruiken om eindig aantal karakteriseringen van de instantiatiegraad van de waarde af te leiden. De waarde die een variabele tijdens specialisatie zal hebben kan dan gekarakteriseerd worden door zo'n instantiatiegraad. Beschouwen we als voorbeeld de volgende type definitie:

Voorbeeld 1 `:- type list(T) ---> [] ; [ T | list(T) ].`

In het bovenstaande voorbeeld wordt een type `list(T)` gedefinieerd: waarden van dit type zijn termen van de vorm [] (de lege lijst), of van de vorm $[A|B]$ waarin A een term is van type T en B een term is van type $list(T)$. Merk op dat de definitie polymorf is, en geïnstantieerd kan worden met eender welk type voor T. In Mercury wordt elk type, uitgezonderd een aantal ingebouwde types, gedefinieerd door een regel zoals in Voorbeeld 1. Een type kan op de klassieke manier voorgesteld worden door een zogenaamde *typeboom*, die oneindig is in het geval van een recursieve type definitie. De typeboom voor $list(T)$ is weergegeven aan de linkerzijde van Figuur 1. Door bepaalde voorkomens van typeknopen in de

Figuur 1: Typeboom en typegrafe voor $list(T)$.

boom terug te vouwen op eerdere typeknopen kan men een oneindige typeboom omvormen naar een eindige typegrafe. De rechterzijde van Figuur 1 geeft zo'n typegrafe voor $list(T)$. Aangezien zo'n typegrafe een abstractie is van alle mogelijke

waarden van dat type, identificeert een typeknoop in de grafe een verzameling van deeltermen van een term van het betreffende type. Beschouwen we als voorbeeld een willekeurige term van het eerder gedefinieerde type $list(T)$. Elke deelterm van deze term is ofwel van type T (wanneer het gaat om een element van de lijst) of van het type $list(T)$ (wanneer het gaat over een deellijst van de oorspronkelijke lijst). De eerstgenoemde klasse van deeltermen (de elementen van de lijst) worden in de typegrafe geïdentificeerd door de knoop T, terwijl de laatstgenoemde klasse van deeltermen (de deellijsten) geïdentificeerd wordt door de knoop $list(T)$. Vervolgens kunnen we een bindingstijd definiëren als een typegrafe waarin elke typeknoop voorzien is van een label *static* of *dynamic*. Zo'n bindingstijd voor een bepaald type karakteriseert de instantiatiegraad van een waarde van dat type als volgt: indien aan een bepaalde knoop het label *static* is toegekend, betekent dit dat elke deelterm in de waarde die geïdentificeerd wordt door de knoop in de typegrafe tijdens specialisatie gebonden is aan een functor. Indien het label *dynamic* geassocieerd is aan een knoop betekent dit dat er minstens één deelterm is die geïdentificeerd wordt door die knoop en die mogelijk niet gebonden is. Figuur 2 toont de drie mogelijke bindingstijden die verkregen worden door labels uit de verzameling $\{static, dynamic\}$ toe te kennen aan de typeknopen in de typegrafe voor $list(T)$. Alle bindingstijden uit Figuur 2 karakteriseren een waarde van het type

Figuur 2: Mogelijke bindingstijden voor $list(T)$.

$list(T)$. De eerste bindingstijd, β_s karakteriseert een lijst waarvan alle deeltermen (de elementen zowel als de deellijsten) gebonden zijn aan een functor. Zo'n lijst is met andere woorden volledig ground, bijvoorbeeld $[1, 2, 3]$. De tweede bindingstijd, β_{ls} karakteriseert een lijst waarin de deellijsten tenminste gebonden zijn aan een functor, maar de elementen mogelijk niet. Zo'n lijst is met andere woorden minstens geïnstantieerd met een skelet van een lijst, bijvoorbeeld $[1, X, 3]$. De derde bindingstijd tenslotte, β_d karakteriseert een lijst waarvan zowel het skelet als de elementen mogelijk niet gebonden zijn, bijvoorbeeld $[1|Y]$. Merk op dat de vierde mogelijkheid, waarin het label *dynamic* toegekend wordt aan de knoop $list(T)$ en het label *static* aan de knoop T, niet voorkomt. Het is geen geldige karakterisering aangezien we geen lijst kunnen construeren waarvan de elementen geïnstantieerd zijn maar het skelet niet.

Een formele definitie van een bindingstijd is terug te vinden in Hoofdstuk 3 van de thesis. De verzameling mogelijke bindingstijden die als dusdanig aan een type geassocieerd kunnen worden is eindig, en bovendien geordend door de *covers* relatie. Intuïtief wordt een bindingstijd β_1 als "groter" dan een bindingstijd β_2 gezien wanneer β_1 minstens even *dynamic* is als β_2 in de zin dat elke knoop die in β_2 het label *dynamic* draagt ook in β_1 het label *dynamic* moet dragen.

2.2 Berekenen van Bindingstijden

Samen met de *covers* relatie vormt de verzameling bindingstijden, in de tekst aangeduid met $(\mathcal{BT}^+, \succeq)$, een domein dat geschikt is voor abstracte interpretatie. Het basisidee van abstracte interpretatie (Cousot and Cousot 1977) kan als volgt samengevat worden. In plaats van de operaties van het programma uit te voeren over een domein van concrete waarden, worden (abstracte versies van) de operaties uitgevoerd over een abstract domein. Indien de abstracte interpretatie correct is, kan men stellen dat het resultaat van een concrete operatie benaderd wordt door het resultaat van de overeenkomstige abstracte operatie wanneer deze laatste uitgevoerd wordt met betrekking tot abstracte waarden die de concrete invoerwaarden benaderen.

In Hoofdstuk 3 van de thesis wordt een bindingstijd-analyse via abstracte interpretatie uitgewerkt. Daartoe wordt eerst een semantische functie voor Mercury gedefinieerd:

$$\mathcal{S} : \mathit{Goal} \times \mathit{Subst} \mapsto \wp(\mathit{Subst}).$$

De gegevens waarmee een Mercury programma werkt worden voorgesteld als een substitutie die gezien kan worden als een omgeving waarin een variabele aan een waarde gebonden wordt. De semantische denotatie van een doel G, $\mathcal{S}[\![G]\!]$ is dan gedefinieerd als een functie die zo'n substitutie afbeeldt op een verzameling van zulke substituties. Deze verzameling substituties modelleert het mogelijk niet deterministische gedrag van Mercury: indien $\mathcal{S}[\![G]\!]\theta = \emptyset$ betekent dit dat evaluatie van het doel G faalt onder de omgeving θ. Indien $\mathcal{S}[\![G]\!]\theta = \{\theta_1, \ldots, \theta_n\}$ betekent dit dat het doel G n keer slaagt, resulterend in de omgevingen (of berekende antwoorden) $\theta_1, \ldots, \theta_n$.

In de thesis worden een aantal functies gedefinieerd die samen de semantische functie $\mathcal{S}$ abstraheren over het domein van bindingstijden. Het resultaat van de analyse is een zogenaamde programma-omgeving die, voor elk predikaat, een sequentie van bindingstijden voor de invoerargumenten van het predikaat afbeeldt op een sequentie van uitvoerargumenten. Zo'n omgeving modelleert het gegevensverloop (of de data-flow) in het programma zoals die zich voordoet tijdens de specialisatie. De geconstrueerde omgeving moet aan een aantal vereisten voldoen, waaronder *congruentie*. Intuïtief gezien garandeert congruentie dat er nooit een als *static* gekarakteriseerde waarde geconstrueerd wordt door gebruik te maken van een als *dynamic* gekarakteriseerde waarde. Congruentie is een gekende en veel ge-

bruikte eigenschap bij de ontwikkeling van bindingstijd-analyses (Jones, Gomard, and Sestoft 1993). De door de analyse geconstrueerde programma-omgeving karakteriseert het gegevensverloop binnen het programma zoals dat zich zal voordoen tijdens specialisatie. Bijgevolg dient er tijdens de abstracte interpretatie rekening gehouden te worden met de controlestrategie die de specialisator wordt opgelegd: indien de bindingstijd-analyse beslist dat een bepaald doel niet geëvalueerd zal worden tijdens de specialisatie, mogen er ook geen bindingstijden geconstrueerd worden voor de uitvoerargumenten van dat doel. Inderdaad, aangezien een bindingstijd de effectieve waarde van zo'n argument tijdens specialisatie benadert, moet een uitvoerargument van het doel als volledig *dynamic* gekarakteriseerd worden wanneer het doel niet geëvalueerd zal worden.

Het vervolg van Hoofdstuk 3 definieert hoe de geconstrueerde programma-omgeving omgezet kan worden in een goed geannoteerd programma. De annotaties zijn "goed" (Jones, Gomard, and Sestoft 1993) in de zin dat ze een specialisator alleen maar correcte instructies geven: indien de annotaties impliceren dat (een deel van) een term tijdens specialisatie gebruikt wordt, zal de term tijdens specialisatie voldoende geïnstantieerd zijn om correct gebruikt te kunnen worden. De instructies voor de specialisator kunnen weergegeven worden door van elke constructie die in een Mercury doel gebruikt kan worden twee varianten te voorzien: een statische variant, die aangeeft dat het doel tijdens specialisatie geëvalueerd moet worden, en een dynamische variant, die aangeeft dat het doel geresidualiseerd moet worden. Wanneer we de verzameling van alle dusdanig geannoteerde doelen aanduiden met $2Goal$, kunnen we het specialisatieproces van een geannoteerd doel formeel definieren als een functie $\mathcal{T}$:

$$\mathcal{T} : 2Goal \times Subst \mapsto \wp(2Goal \times Subst).$$

De functie $\mathcal{T}$ is een veralgemening van de semantische functie: statisch geannoteerde doelen worden uitgevoerd (zoals door $\mathcal{S}$) terwijl dynamisch geannoteerde doelen geresidualiseerd worden. Invoer van de specialisator is (net zoals bij $\mathcal{S}$) een omgeving die een deel van de invoervariabelen van het doel bindt aan mogelijk partiële waarden. Het resultaat van de specialisatie is weerom ofwel de lege verzameling, die een falende afleiding aangeeft, of – bij succes – een verzameling resulterende omgevingen samen met een residueel doel. Zo'n resulterende omgeving bevat de bindingen die geconstrueerd werden tijdens de specialisatie, het residuele doel is de code die in het gespecialiseerde programma terecht komt.

In de thesis tonen we de correctheid van de specialisator aan, en we bewijzen de equivalentie van een tweestaps uitvoering met de corresponderende éénstaps uitvoering door formeel aan te tonen dat aan de partiële evaluatie vergelijking (Jones, Gomard, and Sestoft 1993) voldaan is. In onze context ziet deze vergelijking er als volgt uit:

$$\mathcal{S}[\![G]\!]\theta = \mathcal{S}[\![\phi(\mathcal{T}[\![G_a]\!]\theta_s)]\!]\theta_d.$$

Ze stelt het volgende. Beschouwen we uitvoering van een doel G onder een om-

geving θ die opgesplitst kan worden in een statisch gedeelte θ_s en een dynamisch gedeelte θ_d. Indien G_a een versie van dit doel voorstelt die goed geannoteerd is met betrekking tot een veilige benadering van θ_s, en indien $\phi(\mathcal{T}[\![G_a]\!]\theta_s)$ het resultaat voorstelt van het specialiseren van G_a met betrekking tot θ_s, dan hebben we dat het uitvoeren van dit resultaat met betrekking tot het resterende gedeelte van de invoer, θ_d, equivalent is met het uitvoeren van het oorspronkelijke doel G met betrekking tot de volledige invoer θ.

Onze bindingstijd-analyse is geïnspireerd op gelijksoortig werk binnen het onderzoeksdomein van functionele talen. Een belangrijk verschil is dat onze analyse overweg moet kunnen met de determinisme eigenschappen van de betrokken doelen, aangezien deze de specialisatiestrategie compliceren. We hebben bewezen dat specialisatie volgens de annotaties die berekend worden door onze analyse correct is, in de zin dat de semantiek van het oorspronkelijke programma bewaard blijft.

2.3 Symbolische Bindingstijd-analyse

De eerder geschetste bindingstijd-analyse is *doelgericht* of *oproep-afhankelijk*: een predikaat wordt geanalyseerd met betrekking tot de bindingstijden van de argumenten (het oproeppatroon) waarmee het predikaat opgeroepen wordt zodat verschillende oproepen naar een predikaat mogelijk tot verschillende analyseresultaten leiden. Doelgerichtheid is een belangrijk nadeel van een analyse wanneer deze toegepast moet worden op grote Mercury programma's, die traditioneel uit verschillende modules opgebouwd zijn. De verschillende modules waaruit een programma is opgebouwd zijn geordend in een zogenaamde module-hiërarchie. Figuur 3 schetst zo'n mogelijke hiërarchie. De modules zijn aangegeven door een vierkant, terwijl een pijl van een module M naar een module M' wil zeggen dat M gegevens uit M' importeert. Een doelgerichte analyse van de module M_1 in

Figuur 3: Een voorbeeld module hiërarchie.

de bovenstaande figuur vereist de analyse van bepaalde predikaten in de modules

M_2, M_3 en M_5 aangezien deze laatste modules rechtstreeks gebruikt worden in de module M_1. Deze analyses vereisen op hun beurt de analyse van predikaten in de modules M_4 en M_5. Wanneer nu dezelfde predikaten van een bepaalde module in verschillende contexten gebruikt worden, zullen deze predikaten herhaaldelijk opnieuw geanalyseerd moeten worden, mogelijk voor eenzelfde oproeppatroon. In de bovenstaande figuur is M_4 zo'n module: wanneer dezelfde predikaten van M_4 zowel in M_2 als M_3 gebruikt worden zullen ze herhaaldelijk opnieuw geanalyseerd worden tijdens een analyse van M_1. Aangezien er een algemene tendens heerst om software te ontwikkelen door gebruik te maken van verschillende, algemene componenten die geschikt zijn om in verschillende toepassingen gebruikt te worden, is dit probleem niet te onderschatten.

Bovengenoemd probleem stelt zich niet wanneer de analyse *niet-doelgericht* (of *oproep-onafhankelijk*) kan gebeuren. Bij zo'n niet-doelgerichte analyse is het immers mogelijk om de modules één voor één te analyseren, van beneden naar boven in de module-hiërarchie. In het voorbeeld van Figuur 3 worden dan eerst de modules M_4 en M_5 geanalyseerd, en het resultaat van deze analyse kan vervolgens gebruikt worden om M_2 en M_3 te analyseren waarna uiteindelijk de module M_1 kan geanalyseerd worden. Eens een module M geanalyseerd is, kan het resultaat van deze analyse gebruikt worden tijdens de analyse van eender welke andere module die M gebruikt, zonder dat de noodzaak zich voordoet om opnieuw M te analyseren. Hoewel bindingstijd-analyse een van nature doelgericht proces is, kunnen we het proces als volgt verdelen over twee fasen. In een eerste fase worden de bindingstijden en de relaties die bestaan tussen deze bindingstijden (gebaseerd op de congruentievereisten en de controlestrategie van de specialisator) *symbolisch* voorgesteld. Het resultaat van de analyse is, per predikaat, een systeem van symbolische beperkingen (constraints). Deze beperkingen relateren de bindingstijden van de relevante variabelen aan de bindingstijden van de (invoer-)argumenten van het predikaat. Het annoteren van de code aan de hand van een concreet oproeppatroon van bindingstijden gebeurt dan door het berekenen van een minimale oplossing van het systeem van beperkingen met betrekking tot het specifieke oproeppatroon.

Wanneer we alleen Mercury programma's beschouwen waarin geen hogere orde constructies voorkomen, is het mogelijk om de eerste fase in het analyseproces – het berekenen van de systemen van symbolische beperkingen – op een niet-doelgerichte manier (en dus modulair) uit te voeren. Het grote voordeel van deze manier van werken is dat alleen het annoteren van de code (door het berekenen van minimale oplossingen) doelgericht moet gebeuren, wat rekenkundig gezien een veel minder zware taak is dan het berekenen van de systemen van beperkingen. Wanneer we het objectief van de analyse verschuiven en ook Mercury programma's met hogere orde constructies toelaten, is een zuiver modulaire aanpak niet meer mogelijk. De reden hiervoor is dat het berekenen van de systemen van beperkingen gebeurt aan de hand van het controleverloop in een predikaat, terwijl dit controleverloop in een

hogere orde context kan bepaald worden door het oproeppatroon. Een oplossing voor dit probleem is het berekenen van de systemen van beperkingen gedeeltelijk doelgericht te maken, met betrekking tot de hogere orde informatie uit het oproeppatroon. Het resultaat is dat er, per predikaat, mogelijk een *aantal* systemen van beperkingen gecreëerd worden die gespecialiseerd zijn met betrekking tot de hogere orde informatie uit het oproeppatroon.

In Hoofdstuk 4 van de thesis werken we een bindingstijd-analyse met beperkingen uit voor Mercury. Wanneer we ons beperken tot Mercury programma's zonder hogere orde constructies, is deze analyse equivalent met de analyse uit Hoofdstuk 3 en we bewijzen deze equivalentie. Het belangrijkste verschil met de analyse van Hoofdstuk 3 is dat de nieuwe analyse op een modulaire basis uitgevoerd kan worden. Een prototype implementatie van deze analyse illustreert de toepasbaarheid ervan. We geven tevens aan hoe het berekenen van de systemen van beperkingen gemodelleerd kan worden als een proces dat doelgericht is met betrekking tot de hogere orde informatie uit het oproeppatroon en breiden als dusdanig de analyse uit tot hogere orde Mercury programma's.

3 Bindingstijd-analyse voor Pure Prolog

De bindingstijd-analyse die we ontwikkeld hebben voor Mercury steunt op het feit dat Mercury een sterk getypeerde en sterk gemodeerde taal is. De eerste eigenschap maakt het creëren van een precies domein van bindingstijden mogelijk, terwijl de tweede eigenschap toestaat om bindingstijden op een correcte en precieze manier doorheen het programma te propageren. Hoewel deze eigenschappen niet beperkt zijn tot de taal Mercury, maken ze geenszins deel uit van logisch programmeren als paradigma. Prolog programma's, bijvoorbeeld, dragen in se geen type- of mode informatie. In Hoofdstuk 5 van de thesis ontwikkelen we dan ook een bindingstijd-analyse die steunt op andere principes, en die wel toepasbaar is op (een zuivere deelverzameling) van Prolog.

Off-line specialisatie van zuivere logische talen is in het verleden slechts sporadisch bekeken geweest. In (Mogensen and Bondorf 1993) wordt een off-line specialisator voor Prolog ontwikkeld, maar een bindingstijd-analyse – nodig om de specialisatie te kunnen automatiseren – ontbreekt. De taak van bindingstijd-analyse voor zuivere logische programma's kan als volgt samengevat worden: annoteer als zijnde ontvouwbaar zoveel predikaatoproepen in het programma als mogelijk, terwijl gegarandeerd is dat de constructie van een partiële SLD-boom volgens de annotaties een eindig proces is. Voor zover we weten is de enige concrete aanzet tot een automatische bindingstijd-analyse voor Prolog het werk van Bruynooghe, Leuschel, and Sagonas (1998). Om eindigheid van het ontvouwen te garanderen steunt dit werk op het gebruik van zogenaamde eindigheidsvoorwaarden. Via een eindigheidsanalyse van een predikaat (hetzij handmatig, hetzij automatisch) worden een aantal voorwaarden opgesteld – uitgedrukt in functie van de instantiatiegraad van

de argumenten van het predikaat – waaronder een oproep naar het predikaat gegarandeerd eindigt. Vervolgens wordt een klassieke groundness analyse (bijvoorbeeld (Marriott and Sndergaard 1993)) uitgevoerd op het programma om de instantiatiegraad van de variabelen uit een predikaatoproep te bepalen. Deze resultaten worden dan vervolgens gecombineerd met de eindigheidsvoorwaarden om te beslissen of een oproep al dan niet als ontvouwbaar gekenmerkt zal worden. Een belangrijk probleem bij zo'n aanpak is dat *runtime*-eindigheid gebruikt wordt als ontvouwconditie, wat aanzienlijke beperkingen op de ontvouwmogelijkheden induceert. Het feit dat een predikaatoproep alleen als ontvouwbaar gekarakteriseerd wordt wanneer de oproep eindigt onder normale evaluatie maakt dat er alleen oproepen ontvouwen zullen worden die *volledig* te ontvouwen zijn tot succes of falen. Met andere woorden, een specialisator die steunt op zo'n bindingstijd-analyse is alleen in staat om *volledige* SLD-bomen te bouwen tijdens de specialisatie, in plaats van *partiële* SLD-bomen. Dit is een serieuze beperking, die we illustreren aan de hand van het volgende voorbeeld:

Voorbeeld 2 *Beschouwen we een klassieke* Vanilla *metavertolker (Sterling and Shapiro 1986) met twee object predikaten* `mem/2` *en* `app/3` *die de klassieke* member *en* append *predikaten implementeren.*

```
solve([]).
solve([A|Gs]):- solve_atom(A), solve(Gs).

solve_atom(A):- cl(A,Body), solve(Body).

cl(mem(X,Xs), [app(_,[X|_],_Xs)]).
cl(app([],L,L), []).
cl(app([X|Xs],Y,[Z|Zs]),[app(Xs,Y,Zs)]).
```

Figuur 4: De Vanilla metavertolker.

Stel nu dat we de query `solve([mem(X,Xs)])` willen ontvouwen met betrekking tot het programma van Voorbeeld 2. Wanneer we *runtime*-eindigheid als voorwaarde voor het ontvouwen nemen, zal geen enkele oproep naar het `solve/1` predikaat ontvouwen worden, en zal er bijgevolg zeer weinig specialisatie verkregen worden. Inderdaad, elke oproep naar het *solve/1* predikaat is van de vorm `solve([mem(X,Xs)])` of `solve([app(A,[X|_],Xs)])` en geen van deze oproepen is geïnstantieerd genoeg om eindig te zijn.

De algemene idee achter de bindingstijd-analyse die we in Hoofdstuk 5 van de thesis uitwerken bestaat erin van stapsgewijs een geannoteerd programma te construeren waarbij we een (automatische) eindigheidsanalyse gebruiken om te bewijzen dat het construeren van een *partiële* SLD-boom voor een initieel doel met betrekking tot het geannoteerde programma *onder constructie* eindigt.

Voorbeeld 3 *Laten we de Vanilla metavertolker van Figuur 4 opnieuw bekijken. Intuïtief is het duidelijk dat het ontvouwen van alle oproepen naar het* `solve/1` *predikaat eindigt zolang de tussenliggende oproepen naar het* `solve_atom/1` *predikaat niet ontvouwd worden. De idee is dat het* `solve/1` *predikaat in weze zorgt voor het ontleden van een object doel (dit is de deconstructie van een lijst van object atomen), wat in Voorbeeld 2 eindigt. Indien we dan de oproepen naar* `solve_atom/1` *residualiseren en de anderen ontvouwen krijgen we het gespecialiseerde programma dat afgebeeld wordt in Figuur 5. Het gespecialiseerde programma komt overeen met*

```
solve([mem(X,Xs)]):- solve_atom(mem(X,Xs)).

solve_atom(mem(X,Xs)):-
     solve_atom(app(A,[X|_], Xs)).

solve_atom(app([],[X|B],[X|B])).
solve_atom(app([E|Es],[X|B],[Z|Zs])):-
     solve_atom(app(Es, [X|B], Zs)).
```

Figuur 5: De gespecialiseerde Vanilla metavertolker.

de standaard definities van de `append/3` *en* `member/2` *predikaten – modulo het filteren van de resterende structuur (Gallagher and Bruynooghe 1990). Bijgevolg is al de overhead ten gevolge van de meta interpretatie verdwenen, en is de specialisatie in dit opzicht optimaal.*

De sleutelobservatie om zo'n bindingstijd-analyse, die een geannoteerde versie P_a construeert van een origineel programma P is dat het eindigheidsgedrag van het bouwen van een partiële SLD-boom voor een doel Q aan de hand van P_a equivalent is met het eindigheidsgedrag van het bouwen van een volledige SLD-boom voor een programma dat afgeleid is van P door het *verwijderen* van de oproepen in P die in P_a geannoteerd zijn als te residualiseren, en alleen die oproepen over te houden die als ontvouwbaar gekenmerkt zijn in P_a. Onze bindingstijd-analyse gaat dan als volgt. Veronderstel dat we een programma P moeten annoteren met betrekking tot een initieel doel Q. Indien met een automatische eindigheidsanalyse kan bewezen worden dat Q eindigt met betrekking tot P, kunnen alle predikaatoproepen in P als ontvouwbaar geannoteerd worden. Het resultaat hiervan is dat tijdens specialisatie van Q alle oproepen ontvouwd zullen worden, en dat specialisatie van Q neerkomt op volledige *evaluatie* van Q, en bijgevolg het bouwen van een volledige SLD-boom voor $P \cup \{Q\}$. Indien, anderzijds, de eindigheid van Q met betrekking tot P niet bewezen kan worden ten gevolge van de aanwezigheid van een "verdachte" (mogelijk niet eindigende) predikaatoproep, wordt een nieuw programma afgeleid van P door de betreffende oproep te verwijderen. Vervolgens wordt opnieuw een eindigheidsanalyse uitgevoerd op het getransformeerde programma, en dit proces wordt herhaald tot voldoende oproepen uit het oorspron-

kelijke programma verwijderd zijn zodat kan bewezen worden dat het programma eindigt. Het verwijderen van een oproep uit het programma komt overeen met het markeren van die bewuste oproep als te residualiseren in het geannoteerde programma: die oproep zal niet ontvouwen worden tijdens de specialisatie, en er worden geen bindingen gecreëerd door de oproep.

Om de bewuste bindingstijd-analyse te implementeren vertrekken we van een bestaande eindigheidsanalyse die in staat is om problematische oproepen – dit zijn de oproepen door dewelke de analyse de eindigheid van een programma niet kan bewijzen – als dusdanig te identificeren in een programma. Een voorbeeld van zulke analyse is die van Codish and Taboch (1999). De analyse construeert een eindige benadering van de *binaire ontvouw semantiek* van het programma waarvan de eindigheid bewezen moet worden. De binaire ontvouw semantiek van een programma bestaat uit een verzameling van binaire clauses die het atoom uit het hoofd van een clause uit het oorspronkelijke programma associëren aan een bepaald atoom uit de body van de clause. Om eindigheid van een doel met betrekking tot een programma P te bewijzen volstaat het (Codish and Taboch 1999) om aan te tonen dat een bepaalde eigenschap geldt voor (een bepaalde deelverzameling van) de binaire clauses die door de eindigheidsanalyse geconstrueerd worden. Indien de voorwaarde *niet* geldt op een bepaalde binaire clause, dan identificeert het enkele atoom in de body een mogelijk problematische oproep in het oorspronkelijke programma. We verkrijgen dan een bindingstijd-analyse door deze eindigheidsanalyse te itereren, waarbij in elke stap zo'n mogelijk problematische oproep uit het programma verwijderd wordt.

Voor zover we weten is de ontwikkelde analyse de eerste in zijn soort voor zuivere logische programma's die een meer liberale ontvouwstrategie mogelijk maakt door rekening te houden met eindigheid van *specialisatie*, eerder dan eindigheid van *evaluatie*. De analyse is superieur ten overstaan van eerdere analyses in het feit dat ze de constructie van *partiële* SLD-bomen tijdens specialisatie toestaat, in tegenstelling tot eerder werk zoals (Bruynooghe, Leuschel, and Sagonas 1998). Een eerste beperking van de huidige analyse is het feit dat ze *monovariant* is: voor elk predikaat wordt slechts één geannoteerde versie gecreëerd waarin voldoende atomen geresidualiseerd worden om eindigheid van ontvouwen garanderen voor *elke* oproep naar dat predikaat die tijdens specialisatie kan opduiken. Polyvariante analyses – waaronder (Bruynooghe, Leuschel, and Sagonas 1998) – kunnen in het algemeen leiden tot betere specialisatieresultaten omdat verschillende oproepen naar eenzelfde predikaat op een andere manier behandeld kunnen worden. Een andere beperking is het feit dat de analyse fundamenteel een booleaans domein gebruikt om bindingstijden voor te stellen: ofwel is een waarde voldoende geïnstantieerd met betrekking tot een bepaalde norm, ofwel niet. Bovendien is de gekozen norm vast tijdens de analyse en veronderstellen de meeste eindigheidsanalyses dat die norm gekozen wordt door de gebruiker. Het kiezen van een juiste norm – zelfs door een gebruiker – is niet triviaal en kan zelfs onmogelijk zijn, in

het bijzonder voor programma's die waarden van verschillende types manipuleren. Deze beperkingen worden niet opgelegd door de bindingstijd-analyse an sich, maar zijn eerder een gevolg van de onderliggende eindigheidsanalyse. We verwachten dan ook dat betere (meer precieze) eindigheidsanalyses zullen leiden tot betere (meer precieze) bindingstijd-analyses.

4 Afhankelijkheden in de Context van Getypeerde Logische Programma's

Afhankelijkheden spelen een belangrijke rol in programma-analyse. Een stelling als "de programmavariabele X heeft eigenschap p" kan voorgesteld worden door een propositionele variabele x^p en afhankelijkheden tussen eigenschappen van programmavariabelen kunnen uitgedrukt worden als booleaanse functies. Zo specificeert de functie $x^p \rightarrow y^p$ bijvoorbeeld dat wanneer X de eigenschap p heeft, ook Y de eigenschap p heeft. In vele gevallen wordt de precisie van een gegevensverloop (of data-flow) analyse verbeterd wanneer het onderliggende domein van de analyse afhankelijkheden kan vatten in termen van de eigenschap waarin we geïnteresseerd zijn. Een van de belangrijkste toepassingen in dit onderzoeksgebied is de analyse van zogenaamde *groundness* afhankelijkheden voor logische programma's waarbij de klasse van *positieve* booleaanse functies gebruikt wordt om afhankelijkheden voor te stellen. Het doel van de analyse is om na te gaan of een programmavariabele X op een bepaald programmapunt een unieke waarde heeft die niet veranderd kan worden. In de context van logisch programmeren betekent dit dat X *ground* is of, anders gezegd, een waarde heeft die geen variabelen bevat die op hun beurt geïnstantieerd kunnen worden. Dit is de eigenschap die in groundness analyse wordt voorgesteld door de propositionele variabele x. De klasse van positieve booleaanse functies, Pos, bestaat uit de verzameling booleaanse functies f waarvoor geldt dat $f(true, \ldots, true) = true$.

Een van de belangrijkste stappen in een analyse van groundness afhankelijkheden is het karakteriseren van de afhankelijkheden die opgelegd worden door de unificaties die tijdens de uitvoering kunnen gebeuren. Indien het programma een unificatie van de vorm $term_1 = term_2$ bevat en de variabelen in $term_1$ en $term_2$ zijn respectievelijk $\{X_1, \ldots, X_m\}$ en $\{Y_1, \ldots, Y_n\}$, dan is de groundness afhankelijkheid die overeen komt met deze unificatie $(x_1 \wedge \ldots \wedge x_m) \leftrightarrow (y_1 \wedge \ldots \wedge y_n)$. Deze functie specificeert dat de variabelen in $term_1$ ground zijn (of ground worden) als en slechts als de variabelen in $term_2$ ground zijn (of worden). Een belangrijk voordeel van het gebruik van afhankelijkheden voor groundness analyse is dat het berekenen van de afhankelijkheden oproep-onafhankelijk kan gebeuren. Inderdaad, de groundness relaties die door een predikaat gecreëerd worden zijn onafhankelijk van de concrete waarden uit een oproep naar dat predikaat; ze worden immers enkel bepaald door de definitie van het predikaat.

Het is mogelijk om de precisie van de analyse te verbeteren wanneer bijkomende informatie over de structuur van termen beschikbaar is. Als we bijvoorbeeld weten dat $term_1$ en $term_2$ allebei differentielijsten zijn van de vorm $H_1 - T_1$, respectievelijk $H_2 - T_2$, dan kan de unificatie $term_1 = term_2$ de afhankelijkheid $(h_1 \leftrightarrow h_2) \wedge (t_1 \leftrightarrow t_2)$ leveren, wat een meer precieze karakterisering is dan de formule $(h_1 \wedge t_1) \leftrightarrow (h_2 \wedge t_2)$ die zonder de bijkomende informatie afgeleid zou zijn. Het gebruik van een eenvoudige structuur-analyse om de precisie van andere analyses te verbeteren is het onderwerp van eerder werk geweest zoals van (Le Charlier and Van Hentenryck 1994) en (Codish, Marriott, and Taboch 2000). Hoewel het introduceren van structurele informatie meer precisie toelaat, is dit niet voldoende om het onderscheid te maken tussen bijvoorbeeld een begrensde lijst als $[1, X, 3]$ en een lijst met een open einde als $[1, 2|Z]$ omdat het open einde gesitueerd kan zijn op een willekeurige diepte. Het maken van zo'n onderscheid vereist het gebruik van type informatie. Type informatie kan afgeleid worden door analyse, zoals bijvoorbeeld in (Gallagher and de Waal 1994), gespecificeerd worden door de gebruiker maar geverifieerd worden door analyse, zoals in sommige Prolog systemen als Ciao (Hermenegildo, Bueno, Puebla, and López 1999), of gedeclareerd worden en als deel van de semantiek van een programma beschouwd worden zoals in Gödel (Hill and Lloyd 1994), Mercury (Somogyi, Henderson, and Conway 1996) en HAL (Demoen, García de la Banda, Harvey, Mariott, and Stuckey 1999).

In Hoofdstuk 6 van de thesis construeren we een analyse die type informatie combineert met een klassieke *Pos*-analyse om zo meer precisie te verkrijgen. Gebaseerd op het domein van onze bindingstijd-analyse voor Mercury introduceren we een op *Pos* gebaseerde "getypeerde" groundness analyse. Het belangrijkste verschilpunt met eerder werk in deze context is het feit dat we in ons werk een aparte *Pos*-analyse doen voor elk deeltype van de relevante types. Door de resultaten van de verschillende analyses te combineren verkrijgen we een karakterisering van een term die aangeeft in hoeverre de term geïnstantieerd is tijdens de uitvoering, met een gelijkaardige precisie als de bindingstijden van onze bindingstijd-analyse voor Mercury aangeven in hoeverre een term geïnstantieerd is tijdens de specialisatie. Een belangrijk probleem bij dit soort analyses die gebaseerd zijn op type informatie is het behandelen van polymorfisme. Bij een oproep-onafhankelijke analyse van een polymorf predikaat kunnen immers alleen afhankelijkheden geconstrueerd worden tot op het niveau van de typevariabelen uit de types van de argumenten. Wanneer we echter een oproep naar een polymorf predikaat beschouwen, is het vaak mogelijk om meer gedetailleerde afhankelijkheden af te leiden, tot op het niveau van de deeltypes van de (instanties van de) typevariabelen uit de types van de argumenten.

We ontwikkelen een schema waarin we afhankelijkheden op het niveau van de deeltypes uit de *actuele* types van een predikaatoproep afleiden door gebruik te maken van de afhankelijkheden op het niveau van de deeltypes uit de *formele* types van het predikaat. Als dusdanig blijft onze analyse oproep-onafhankelijk, terwijl de

afhankelijkheden die afgeleid worden uitgedrukt zijn op het niveau van de actuele, meer geïnstantieerde types van een predikaatoproep. Dit gaat ten koste van de accuraatheid: in het algemeen geldt dat de resultaten van onze oproep-onafhankelijke analyse minder precies zijn voor een oproep naar een polymorf predikaat, dan wat opgeleverd zou worden door een oproep-afhankelijke analyse van dat predikaat voor de specifieke oproep. In Hoofdstuk 6 definiëren we twee voldoende voorwaarden waaronder deze graad van precisie behouden blijft en waaronder – met andere woorden – onze analyse dus even precies is als een oproep-afhankelijke analyse. We vermoeden dat deze voorwaarden voldaan zijn voor een veelheid aan programma's.

5 Bottom-up Partiële Deductie van Logische Programma's

In het laatste deel van deze thesis bekijken we *on-line* partiële deductie, een onderzoeksgebied waar in de context van logische programma's reeds veel aandacht aan besteed geweest is. Het uitgangspunt van ons onderzoek is de partiële deductie van een bepaalde klasse van metavertolkers. Hoewel de specialisatie van zulke metavertolkers heel wat aandacht gekregen heeft (en in zekere zin één van de pijlers was van het hele onderzoek rond programmaspecialisatie), merken we dat *automatische* partiële deductie van zulke metavertolkers niet vanzelfsprekend is. Zo onderzoeken we in Hoofdstuk 7 van de thesis wat de fundamentele problemen zijn bij het bereiken van de gewenste specialisatie met een automatische techniek. We demonstreren dat een veel gebruikte en als zeer krachtig ervaren techniek (gebruik makend van de homeomorphic embedding relatie) niet in staat is om een bevredigend resultaat te halen voor een zeer eenvoudige vertolker. We tonen aan dat het automatische systeem wél de vooropgestelde resultaten kan halen indien het controlesysteem op een dusdanige manier wordt uitgebreid dat *globale* informatie in rekening wordt gebracht bij het nemen van *lokale* controlebeslissingen. Dit resultaat is belangrijk, aangezien het aantoont dat automatische specialisatie van de vertolker (met het beoogde resultaat) mogelijk is, maar een zeer gesofistikeerde en conceptueel moeilijke controletechniek vereist. Deze resultaten vormen de basis voor het onderzoek waarover in Hoofdstuk 8 gerapporteerd wordt. In dat hoofdstuk ontwikkelen we een nagelnieuwe techniek voor partiële deductie van logische programma's. De techniek concentreert zich – in tegenstelling tot de klassieke top-down partiële deductie – op de *bottom-up* gegevensstroom in een logisch programma. Waar een klassieke techniek voor partiële deductie een programma specialiseert door informatie uit de query top-down in het programma te propageren, is de nieuwe techniek – *bottom-up partiële deductie* – in staat om een logisch programma te specialiseren met betrekking tot een verzameling feiten door de informatie die in deze feiten vervat zit bottom-up doorheen het programma te propageren.

5.1 Naar de Grenzen van Top-Down Partiële Deductie

Het schrijven van metavertolkers is een welbekende techniek om de expressieve kracht van logische programma's te verbeteren. In essentie is een metavertolker een programma dat een ander programma manipuleert. Meestal wordt naar de vertolker verwezen als het *metaprogramma*, terwijl naar het programma dat verwerkt (of "vertolkt") wordt verwezen wordt als het *objectprogramma*. Een van de meest bekende een eenvoudige metavertolkers is de zogenaamde *Vanilla* metavertolker (Sterling and Shapiro 1986) waarvan we een variant reeds zijn tegengekomen in Paragraaf 3 van deze tekst.

In Hoofdstuk 7 bekijken we opnieuw de partiële deductie van zo'n metavertolker, maar nu in de context van de klassieke on-line specialisatie. We instantiëren het generische algoritme voor partiële deductie met concrete operaties voor abstractie die welbekend zijn en in het algemeen als zeer krachtig ervaren worden (Leuschel, Martens, and De Schreye 1998; Leuschel 1998; Sørensen and Glück 1995). Hoewel het resulterende algoritme goed presteert, zowel in het algemeen als in de context van de Vanilla vertolker, kan het niet garanderen dat alle interpretatie-overhead die geassocieerd is aan de metavertolker weggespecialiseerd wordt. Het basisprobleem is terug te voeren tot het feit dat zo'n automatisch systeem meestal generalisaties doorvoert wanneer het groeiende gegevensstructuren detecteert. Deze beslissing is gerechtvaardigd door de noodzaak om het specialisatieproces eindig te houden, en het feit dat deze groeiende gegevensstructuren vaak de aanleiding tot een niet-eindigend specialisatieproces zijn. De Vanilla metavertolker is echter een typisch voorbeeld van een programma waarvan de grootte van de gegevensstructuren *fluctueert* tussen verschillende recursieve oproepen. Inderdaad, de vertolker houdt een lijst bij van atomen die nog bewezen moeten worden in het objectprogramma. Het bewijzen van één atoom resulteert vaak in een aantal nieuwe atomen die nog moeten bewezen worden, wat zo het groeien en krimpen van de betrokken gegevensstructuur verklaart. Wanneer de specialisator nu twee `solve` oproepen naar de vertolker generaliseert terwijl de lijst van atomen in beide oproepen een verschillende lengte heeft, worden er onvermijdelijk atomen uit het objectprogramma ge generaliseert, wat typisch aanleiding zal geven tot een residueel programma waaruit niet alle interpretatie-overhead is weggespecialiseerd.

In het zelfde hoofdstuk bekijken we een mogelijke oplossing voor dit probleem. We proberen deze fluctuerende structuren in rekening te brengen door de *lokale* controlecomponent van het systeem gebruik te laten maken van informatie uit het *globale* niveau. Wanneer de lokale controle het ontvouwen wil stoppen, zal eerst gecontroleerd worden of op deze manier atomen op het globale niveau gegeneraliseerd zouden worden. Indien dat het geval is, zal de lokale controle toch trachten verder te ontvouwen, zolang dit resulteert in recursieve oproepen met kleinere gegevensstructuren. De resulterende controlestrategie blijft automatisch en algemeen, in de zin dat ze geen informatie gebruikt behalve informatie die automatisch vergaard werd tijdens het specialisatieproces, maar is desalniettemin sterk geïnspireerd op

de partiële deductie van de Vanilla vertolker die precies het soort fluctuerend gedrag vertoont dat door de nieuwe controle opgevangen wordt. Hoewel de nieuwe controlestrategie het verwachte resultaat bereikt op de Vanilla vertolker en op een uitbreiding hiervan die samenstellingen van logische programma's beschouwt (Brogi and Contiero 1997), valt te verwachten dat deze controlestrategie niet voldoende is om meer ingewikkelde metavertolkers op dezelfde voldoende wijze te specialiseren. Een ander nadeel van de resulterende controlestrategie is het feit dat ze door de vereiste interactie tussen het lokale en globale controleniveau de reeds subtiele wisselwerking tussen beide controleniveaus intenser maakt, wat het voorspellen van het effect van partiële deductie bemoeilijkt. Desalniettemin is het, voor zover ons bekend, de eerste volledig automatische controlestrategie voor partiële deductie die in staat is om de Vanilla metavertolker en eenvoudige varianten of uitbreidingen hiervan dusdanig te specialiseren dat alle interpretatie-overhead uit het gespecialiseerde programma verwijderd is.

5.2 Bottom-Up Partiële Deductie

Geïnspireerd door de problemen bij partiële deductie van de Vanilla metavertolker werd een nagelnieuwe techniek voor de partiële deductie van logische programma's ontwikkeld. Terwijl de klassieke partiële deductie gebaseerd is op de *top-down* uitvoering van logische programma's (SLD-resolutie), baseren we onze nieuwe techniek op de *bottom-up* uitvoering van logische programma's. Het resultaat is een specialisatietechniek waarmee een logisch programma niet langer gespecialiseerd wordt met betrekking tot een query, maar eerder gespecialiseerd kan worden met betrekking tot een verzameling feiten.

Voorbeeld 4 *Beschouw de volgende, niet traditionele implementatie van het predikaat* `append/3` *voor het verbinden van lijsten:*

```
append(X,Y,Y):-list_nil?(X).
append(X,Y,Z):-list_not_nil?(X),
               list_head(X,H), list_tail(X,T),
               append(T,Y,R), list_cons(H,R,Z).
```

Bovenstaand programma manipuleert geen termen rechtstreeks, maar eerder met behulp van een aantal operaties die een zogenaamd abstract gegevenstype implementeren. Bovenstaande implementatie kan gezien worden als "open" of onvolledig: om een oproep naar `append/3` te kunnen evalueren moeten de predikaatdefinities van het abstracte gegevenstype gegeven zijn. Een mogelijk voorbeeld van zo'n gegevenstype dat lijsten voorstelt is de volgende implementatie:

```
list_nil?([]).          list_head([X|_], X).     list_cons(X,Xs, [X|Xs]).
list_not_nil?([_|_]).   list_tail([_|Xs], Xs).
```

Het abstraheren van de concrete representatie van een gegevensstructuur is een gekende techniek om de herbruikbaarheid en aanpasbaarheid van software-componenten te verbeteren. Omwille van efficiëntieredenen heeft het echter zin om deze laag van abstractie (automatisch) te kunnen verwijderen, en in het bovenstaande voorbeeld de concrete representatie van lijsten te propageren in het `append/3` predikaat. Op deze manier kunnen alle oproepen naar het abstracte gegevenstype – en het daarmee gepaard gaande verlies aan efficiëntie – verwijderd worden. "Specialisatie" van het `append/3` predikaat met betrekking tot het abstracte gevenstype voor lijsten resulteert in de klassieke definitie van `append/3`:

```
append([],Y,Y).
append([X|Xs],Y,[X|R]):- append(Xs,Y,R).
```

Ook de definitie van de Vanilla metavertolker kan gezien worden als een "open" definitie, die gespecialiseerd kan worden met betrekking tot een objectprogramma. Deze vorm van specialisatie *kan* bereikt worden met een klassieke top-down specialisator. Het bereiken van de gewenste specialisatie met een *volledig automatische* specialisator is soms echter – zeker in het geval van de Vanilla metavertolker – verre van triviaal. Het basisprobleem is het feit dat de controle-informatie, die de specialisator nodig heeft om te beslissen het specialisatieproces al dan niet verder te zetten, vaak niet alleen top-down doorheen het programma loopt, maar ook bottom-up. In het bijna triviale `append/3` voorbeeld wordt dit geïllustreerd door het feit dat de query met betrekking tot dewelke het programma top-down gespecialiseerd moet worden geen of bijna geen informatie bevat en van de vorm `append(X,Y,Z)` is. Dankzij de eenvoud van dit voorbeeld zal haast geen enkele top-down specialisator hier moeilijkheden mee hebben, maar dat ligt anders bij meer ingewikkelde voorbeelden. Een ander extreem voorbeeld is het volgende, waarin de informatie die de recursie stuurt bottom-up doorheen het programma loopt. Het `make_list(T,I,R)` predikaat kan gebruikt worden om een lijst van een vaste lengte (voorgesteld door de parameter T) te maken waarin elk element geïnitialiseerd is met een bepaalde waarde (I); het resultaat wordt teruggegeven in R.

Voorbeeld 5

```
type(list1, [X]).          fill_list(L,T,I,L):- type(T,L).
type(list3, [X1,X2,X3]).   fill_list(L,T,I,R):- fill_list([I|L],T,I,R).
                           make_list(T,I,R):- fill_list([],T,I,R).
```

Voorbeeld 5 vertegenwoordigt een klasse van recursieve predikaten die een bepaalde structuur opbouwen die beslissend is om de recursie te stoppen. Alle top-down specialisatoren zijn gebaseerd op ontvouwtechnieken waarvan gekend is dat ze moeilijkheden hebben met dit soort predikaten. Vaak kan immers uit de partiële SLD-boom onder constructie niet afgeleid worden of verder ontvouwen al dan niet

zal eindigen. Het ontvouwen van een oproep `make_list(list3,a,R)` bijvoorbeeld geeft aanleiding tot de volgende sequentie van oproepen

```
fill_list([],list3,a,R),
fill_list([a],list3,a,R),
fill_list([a,a],list3,a,R),
...
```

waaruit elke zinvolle ontvouwregel (gebaseerd op de syntactische structuur van de atomen in de SLD-afleiding) potentiële non-terminatie zal besluiten en het ontvouwproces vroegtijdig zal stoppen. Wanneer bovenstaand recursief predikaat daarentegen op een bottom-up manier behandeld wordt, te beginnen door de feiten `type(list1,[X])` en `type(list3,[X1,X2,X3])` in het programma te propageren, wordt de aanwezige structuur *kleiner* tussen recursieve oproepen en worden de volgende feiten afgeleid:

```
make_list(list1,I,[I]).
make_list(list3,I,[I,I,I]).
```

wat precies het soort specialisatie is dat we met dit voorbeeld willen bereiken.

In Hoofdstuk 8 van deze thesis ontwikkelen we een formeel raamwerk voor de bottom-up partiële deductie van een logisch programma. We vertrekken van een bottom-up inferentie-operator op het semantische domein van Horn clauses, zoals die reeds geïntroduceerd werd in de context van de compositionele semantiek (Bossi, Gabbrielli, Levi, and Meo 1994). Door deze operator te combineren met een abstractiefunctie, verkrijgen we een *abstracte* bottom-up operator, A^c_P. Gegeven een abstractiefunctie definiëren we een *bottom-up partiële deductie* van een programma P als de verzameling clauses die het vaste punt is van de aan dit programma en deze abstractiefunctie geassocieerde A^c_P-operator. Twee belangrijke eigenschappen van een bottom-up partiële deductie zijn de *correctheid* en *volledigheid* van de transformatie. We bewijzen dat de transformatie *volledig* is: elk resultaat dat berekend wordt door het originele programma P wordt eveneens berekend door de bottom-up partiële deductie van P. We tonen aan dat de transformatie zonder bijkomende voorwaarden echter niet *correct* is, in de zin dat het getransformeerde programma meer-geïnstantieerde antwoorden kan berekenen dan het originele programma. We definiëren een voldoende voorwaarde waaronder de transformatie ook correct is. Correctheid en volledigheid worden formeel gedefinieerd in termen van de S-semantiek (Falaschi, Levi, Martelli, and Palamidessi 1989; Bossi, Gabbrielli, Levi, and Meo 1994).

In het vervolg van Hoofdstuk 8 creëren we een concrete instantie van het eerder gedefinieerde raamwerk. We definiëren een bottom-up operator die een gerichte grafe construeert waarin de oorzakelijke verbanden tussen de afgeleide clauses expliciet gemaakt zijn. Gebaseerd op deze concrete operator leiden we een generisch algoritme af voor het berekenen van een bottom-up partiële deductie. Het algo-

ritme is gedefinieerd in termen van dezelfde basisoperaties waarmee een generisch algoritme voor klassieke top-down partiële deductie gedefinieerd kan worden (zoals bijvoorbeeld (Leuschel, Martens, and De Schreye 1998; Leuschel and Bruynooghe 2001)). Als voorbeeld instantiëren we de basisoperaties met concrete operaties gebaseerd op de homeomorhic embedding relatie, zoals ze ook teruggevonden worden in veelgebruikte top-down controlestrategieën.

De resulterende techniek is interessant als techniek op zichzelf, aangezien ze in zekere zin de duale is van de klassieke top-down partiële deductie. Net zoals de top-down tegenhanger is het bottom-up specialisatieproces volledig te automatiseren, en we hebben als voorbeeld een volledig automatische specialisatiestrategie afgeleid. Voorbeelden tonen aan dat bottom-up partiële deductie in staat is om programmafragmenten te specialiseren die moeilijk of niet te specialiseren zijn met de klassieke en automatische top-down technieken. Bovendien wordt uitstekende specialisatie van de Vanilla metavertolker verkregen door de geconstrueerde bottom-up partiële deductie te alterneren met een klassieke top-down partiële deductie met een triviale controle. Zoals we eerder aangetoond hadden kan hetzelfde resultaat weliswaar verkregen worden door uitsluitend gebruik te maken van een top-down strategie, maar dit gaat ten koste van de elegantie en doorzichtigheid van de controlestrategie. Het feit dat dezelfde resultaten bereikt kunnen worden door de combinatie van twee minder gecompliceerde controlestrategieën – één die zich concentreert op de top-down gegevensstroom; de andere op de bottom-up gegevensstroom – doet vermoeden dat een combinatie van beide technieken aanleiding kan geven tot uitstekende specialisatieresultaten op een conceptueel meer elegante manier, en mogelijk de weg opent naar sterkere specialisatietechnieken die mogelijk in staat zijn meer gecompliceerde metavertolkers op een bevredigende manier te specialiseren.

6 Besluit

De belangrijkste bijdragen van deze thesis kunnen op drie vlakken gesitueerd worden. Ten eerste bekijken we off-line specialisatie van logische programma's. In tegenstelling tot de on-line aanpak, is off-line specialisatie voor logische programmeertalen slechts sporadisch bekeken geweest. Vooreerst ontwikkelen we een bindingstijd-analyse – het belangrijkste ingrediënt van een off-line specialisator – voor Mercury, een logische programmeertaal die veelvuldig van type- en modeinformatie gebruik maakt. Mercury is een uitstekende kandidaat voor de ontwikkeling van een bindingstijd-analyse. Inderdaad, enerzijds vertoont het uitvoeringsmodel van de taal voldoende gelijkenis met het uitvoeringsmodel van functionele talen om de grondig onderzochte concepten van bindingstijd-analyse in deze laatste context over te dragen naar Mercury. Anderzijds verzekert het feit dat Mercury een logische programmeertaal is met specifieke voorzieningen voor de ontwikkeling van grote toepassingen voldoende nieuwe uitdagingen en voegt het een nieuwe

dimensie toe aan de ontwikkeling van de analyse. De resulterende analyse propageert én gebruikt hogere-orde informatie en combineert een analysedomein van een zeer hoge precisiegraad met een symbolische aanpak, wat een modulaire analyse mogelijk maakt. Daarmee is de ontwikkelde bindingstijd-analyse een van de enige in zijn soort die al deze eigenschappen combineert. Een prototype implementatie van de analyse werd verwezenlijkt binnen de Melbourne Mercury compiler. Experimenten tonen aan dat de analyse in de praktijk toepasbaar is en de vooropgezette resultaten haalt, maar onthult tevens een aantal aandachtspunten die opduiken tijdens de analyse van grotere programma's. De aanpak voor Mercury steunt sterk op het feit dat Mercury een sterk gemodeerde taal is, wat maakt dat de analyse moeilijk of niet overdraagbaar is naar andere logische programmeertalen, in het bijzonder Prolog, die deze mode-informatie niet bevatten. In een tweede deel van ons werk rond off-line specialisatie herbekijken we bindingstijd-analyse binnen een context waar type- noch mode-informatie voorhanden is. We ontwikkelen een bindingstijd-analyse voor een zuivere deelverzameling van Prolog, en steunen de analyse op bestaande technieken voor eindigheidsanalyse.

In een tweede deel van deze thesis maken we een veralgemening van de ideeën die aan de basis liggen van het op types gebaseerde domein van bindingstijden en demonstreren we hoe een klassieke *Pos*-gebaseerde analyse verfijnd kan worden door gebruik te maken van type informatie. We tonen aan dat het combineren van type informatie met een klassieke, op *Pos*-gebaseerde, groundness analyse resulteert in een meer algemene analyse waarmee een waarde niet langer gekarakteriseerd wordt als al dan niet volledig ground, maar eerder gekarakteriseerd wordt door zijn graad van instantiatie. Onze analyse onderscheidt zich van soortgelijke, op types gebaseerde analyses door de manier waarop de type informatie in de analyse verwerkt wordt, en waardoor een nauwkeurige behandeling van polymorfe predikaten mogelijk is.

Tenslotte ontwikkelen we een nieuwe techniek, de zogenaamde bottom-up partiële deductie, waarmee een logisch programma gespecialiseerd kan worden met betrekking tot een verzameling feiten in plaats van met betrekking tot een query. De ontwikkeling van de techniek is geïnspireerd door de problemen die een klassieke top-down partiële deductie ondervindt, in het bijzonder bij de (automatische) partiële deductie van de Vanilla metavertolker. We formuleren een formeel raamwerk voor bottom-up partiële deductie, en geven een generisch algoritme dat geïnstantieerd kan worden met een concrete controlestrategie. De controle is online, en we instantiëren het algoritme met concrete operaties waarvan het abstractiegedrag gelijkaardig is aan de manier waarop klassieke top-down partiële deductie soms gecontroleerd wordt. Afgezien van het feit dat de techniek op zichzelf interessant is, argumenteren we dat een combinatie van bottom-up en top-down partiële deductie eenzelfde (of zelfs een betere) specialisatiegraad kan halen dan één enkele strategie. Daarenboven geven experimenten aan dat bij zo'n gecombineerde aanpak de benodigde strategieën minder gesofistikeerd moeten zijn dan wanneer

slechts één enkele techniek (bottom-up of top-down) gebruikt wordt. We hebben dit in de thesis geïllustreerd aan de hand van de specialisatie van de Vanilla metavertolker.

Referenties

Andersen, L. (1993). Binding-time analysis and the taming of C pointers. In *PEPM93*, pp. 47–58. ACM.

Apt, K. R. (1990). Logic programming. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pp. 493–574. Elsevier Science Publishers B.V.

Bossi, A., M. Gabbrielli, G. Levi, and M. C. Meo (1994). A compositional semantics for logic programs. *Theoretical Computer Science 122*(1-2), 3–47.

Brogi, A. and S. Contiero (1997). Specialising meta-level compositions of logic programs. In J. Gallagher (Ed.), *Proceedings LOPSTR'96*, Stockholm, pp. 275–294. Springer-Verlag, LNCS 1207.

Bruynooghe, M., M. Leuschel, and K. Sagonas (1998). A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin (Ed.), *Programming Languages and Systems, Proc. of ESOP'98, part of ETAPS'98*, Lisbon, Portugal, pp. 27–41. Springer-Verlag. LNCS 1381.

Codish, M., K. Marriott, and C. Taboch (2000). Improving program analyses by structure untupling,. *Journal of Logic Programming 43*(3), 251 – 263.

Codish, M. and C. Taboch (1999). A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming 41*(1), 103–123.

Consel, C. and F. Noël (1996). A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, pp. 145–156.

Cousot, P. and R. Cousot (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pp. 238–252.

De Schreye, D., R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen (1999). Conjunctive partial deduction: Foundations, control, algorithms and experiments. *Journal of Logic Programming 41*(2-3), 231 – 277.

Demoen, B., M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey (1999). An overview of HAL. In J. Jaffar (Ed.), *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Volume 1713 of *LNCS*, pp. 174–188. Springer.

Falaschi, M., G. Levi, M. Martelli, and C. Palamidessi (1989). Declarative modeling of the operational behaviour of logic programs. *Theoretical Computer Science 69*, 289–318.

Gallagher, J. (1993). Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, pp. 88–98. ACM Press.

Gallagher, J. and M. Bruynooghe (1990). Some low-level source transformations for logic programs. In M. Bruynooghe (Ed.), *Proceedings Meta'90*, Leuven, pp. 229–244.

Gallagher, J. and D. A. de Waal (1994). Fast and precise regular approximations of logic programs. In *Proc. ICLP'94*, pp. 599–613.

Hermenegildo, M., F. Bueno, G. Puebla, and P. López (1999). Debugging, and optimization using the ciao system preprocessor. In D. D. Schreye (Ed.), *Proceedings of ICLP'99*, pp. 52–66.

Hill, P. and J. Lloyd (1994). *The Gödel Programming Language*. MIT Press.

Jones, N. D., C. K. Gomard, and P. Sestoft (1993). *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

Le Charlier, B. and P. Van Hentenryck (1994). Experimental evaluation of a generic abstract interpretation algorithm for PROLOG. *ACM Transactions on Programming Languages and Systems 16*(1), 35–101.

Leuschel, M. (1998). On the power of homeomorphic embedding for online termination. In G. Levi (Ed.), *Static Analysis. Proceedings*, Volume 1503 of *Lecture Notes in Computer Science*, Pisa, Italy, pp. 230–245. Springer-Verlag.

Leuschel, M. and M. Bruynooghe (2001). Control issues in partial deduction: The ever ending story. Draft.

Leuschel, M., B. Martens, and D. De Schreye (1998). Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems 20*(1), 208 – 258.

Lloyd, J. W. (1987). *Foundations of Logic Programming*. Springer-Verlag.

Lloyd, J. W. and J. C. Shepherdson (1991). Partial evaluation in logic programming. *Journal of Logic Programming 11*(3&4), 217–242.

Marriott, K. and H. Sndergaard (1993). Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems 2*(1-4), 181–196.

Mogensen, T. and A. Bondorf (1993). Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement (Eds.), *Proceedings LOPSTR'92*, pp. 214–227. Springer-Verlag, Workshops in Computing Series.

Somogyi, Z., F. Henderson, and T. Conway (1995). Logic programming for the real world. In *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming*.

Somogyi, Z., F. Henderson, and T. Conway (1996). The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29*(1–3), 17–64.

Sørensen, M. H. and R. Glück (1995). An algorithm of generalization in positive supercompilation. In J. Lloyd (Ed.), *Proceedings ILPS'95*, Portland, Oregon, pp. 465–479. MIT Press.

Sterling, L. and E. Shapiro (1986). *The Art of Prolog*. MIT Press.

Weise, D., R. Conybeare, E. Ruf, and S. Seligman (1991). Automatic online partial evaluation. In J. Hughes (Ed.), *Proceedings of the 3rd ACM Conference on Functional Programming Languages and Computer Architecture*, Cambridge, Massachusets, USA, pp. 165–191. Springer-Verlag, LNCS 523.