UNIVERSITEIT GENT

FACULTY OF SCIENCES

Towards Flexible Goal-Oriented Logic Programming

ir. Benoit Desouter

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Supervisors:
prof. dr. ir. Tom Schrijvers
prof. dr. ir. Marko van Dooren

Department of Applied Mathematics, Computer Science and Statistics
Faculty of Sciences, Ghent University

Acknowledgments

As it feels more natural to thank people in the language that we have used on a daily basis during the journey towards completing this PhD thesis, I'll use Dutch for most of the next few pages.

In de eerste plaats wil ik mijn promotoren Tom en Marko bedanken. Tom, bedankt voor het geduld als ik occasioneel iets maar half begreep, en om er altijd vertrouwen in te blijven hebben. Je hebt me heel wat kansen aangereikt, waardoor ik van heel wat onderwerpen iets heb opgestoken. Marko, hoewel je er pas halverwege bijkwam, toonde je al snel interesse voor het onderwerp en heb je er vanuit je eigen expertise heel wat aan toegevoegd. Je deur stond altijd voor me open als ik even een tussentijdse statusupdate wou geven. Bedankt voor de babbels over vanalles en nog wat, en om me grondig te betrekken bij het geven van Programmeren 1. Ik heb nog heel wat bijgeleerd over objectoriëntatie door jouw visie, slides en codevoorbeelden. Daarnaast ook bedankt om mijn lokale LaTeX-goeroe te zijn: niettegenstaande ik LaTeX al tien jaar lang gebruik, heb ik voor het precies goed krijgen van deze thesis heel wat nieuwe pakketten en trucjes moeten gebruiken, met regelmatig vreemde output of cryptische foutmeldingen tot gevolg die ik niet altijd alleen kon oplossen — of het zou me op zijn minst veel meer tijd gekost hebben.

Ook wil ik graag Bart Demoen, Jan Wielemaker, Christophe Scholliers, Kris Coolsaet en Gunnar Brinkmann bedanken voor het grondig nalezen van mijn thesis en voor hun kritische feedback.

In het bijzonder wil ik Bart bedanken voor zijn ondersteuning van hProlog, het experimentele Prolog-systeem waarop ik zoveel heb gevloekt. Zonder hProlog zou het meeste van deze thesis echter niet mogelijk geweest zijn. Meer dan eens heb je geduldig geantwoord op mijn emails met vragen over het "vreemde" gedrag van hProlog. Bedankt voor de stroom aan Prolog tips en WAM-expertise.

Het is bijzonder leuk als je werk ook gebruikt wordt door iemand anders dan jezelf, en zeker als die persoon een alom gerespecteerde voortrekker is in dat gebied. Daarom wil ik Jan bedanken om onze implementatie van tabling te porten naar SWI-Prolog, de datastructuren te herschrijven in C en het geheel op te nemen in de officiële development release. Daardoor heeft SWI-Prolog nu een experimentele tabling implementatie die stapsgewijs verbeterd kan worden. Het was heel onverwachts om zoveel positieve reacties te zien uit de SWI-Prolog community.

Gunnar en Kris, bedankt om de administratie op jullie te nemen; ik weet dat dat niet altijd de leukste taak is.

I would also like to thank LogicBlox, Inc. for giving us the ability to investigate the integration of Datalog and constraint programming in their system.

Ik heb heel graag gewerkt op de vakgroep WE02 omwille van de goede sfeer. De verzameling van TWISTers die ik moet bedanken is in de loop der jaren behoorlijk groot geworden. Om die verzameling te kunnen neerschrijven, heb ik noodgedwongen een volgorde moeten kiezen, hoewel ik dat liever niet had moeten doen. Op andere momenten in de tijd was de volgorde misschien anders geweest. Mezelf kennende, ben ik waarschijnlijk ook wel iemand onterecht vergeten, daarvoor al vast een sorry en dankjewel voor wie hieronder geen specifieke vermelding heeft, maar zich toch aangesproken voelt.

Herman, ik had me geen betere bureaugenoot kunnen indenken. Merci voor de komische noot die nooit ver weg was, de ernst wanneer het nodig was en voor niet-ophoudende babbels over geeky en minder geeky onderwerpen. Lynn en Sofie, bedankt voor de vele aangename gesprekken, het aanhoren van mijn occasionele gezeur, en voor de inside jokes over het oprichten van een eigen vakgroep bestaande uit de mensen van de "vaagheid" en mezelf. Aan de wandelzoektocht die we met ons drie hebben ondernomen op de personeelssportdag 2015, heb ik hele goede herinneringen. Lynn, bedankt om op de zonnige meidag toen Bart, Sofie en wij op het grasveld achter S9 zaten te werken (wegens de veel te warme bureaus), bij het zien van een eerdere versie van dit manuscript, spontaan suggesties te maken voor het verbeteren van mijn wiskundige notatie. Virginie, bedankt voor je enthousiasme, zowel bij badminton als op het werk, en voor het hart onder de riem wanneer ik daar nood aan had. Om gelijkaardige redenen een grote dankjewel voor het het een"man"spraesidium van onze niet-erkende "studentenvereniging": Catherine. Daarbovenop nog een dankjewel voor het organiseren van spelletjesavonden, weekendjes en dergelijke. Je besefte van bij het begin als geen ander het belang van teambuilding en je slaagt er telkens opnieuw in om elke vreemde eend in de bijt zich al snel

thuis te laten voelen. Bij toekomstige organisaties zal ik zeker van de partij zijn; jullie zijn nog niet van mij af, beloofd! Charlotte, dankjewel voor babbels over verbouwingen, over je schattige dochtertje Emilina (bij de laatste aanpassing van dit dankwoord, ben je nog maar net bevallen van Jolan) en zoveel meer. Machteld en Jens, dankjewel voor het tweewekelijks reserveren van de badmintonpleintjes in het GUSB en voor goede babbels. Dieter en Bart, jullie waren als “de informatica-assistenten” best wel een voorbeeld voor mij. Bedankt daarvoor! Felix, ik ben er zeker van dat ook jij een voorbeeldassistent zult zijn voor toekomstige collega’s. Katia en Ann, de koffiepauzes waren voor mij de manier om even alle dagelijkse beslommeringen rond onderzoek en onderwijs uit mijn hoofd te zetten. Bedankt om die momenten nog aangenamer te maken. Hilde, bedankt voor alle petten die jij droeg. Kris, om het er na zoveel keer inwrijven dat ik qua diploma een ingenieur ben, op je geheel eigen wijze toch je appreciatie voor me uit te drukken: bedankt! Glad, niet alleen tijdens je S9 tijd, maar telkens we elkaar op straat tegenkwamen, was je bereid je ervaring met me te delen. We hebben samen behoorlijk de draak gestoken met alles wat maar enigszins naar bureaucratie rook. Nico, ook uit jouw ervaring kon ik rijkelijk putten. Joyce en jij zijn bovendien fantastische TWIST-organisatoren en Minions! Michaël, officieel was je de peter van Herman, maar tijdens die eerste onzekere weken, nu 4 jaar geleden, deed je meer dan je best om ook mij op m’n gemak te stellen. Jean-Marie, Jan en Dominiek, jullie enthousiasme en inzet was aanstekelijk!

Annick, Davy, Nico, Niels en Bert, voor het samen geven van en uitgebreid discussiëren over Programmeren 1. De cursus van vier jaar geleden lijkt in het niets meer op de cursus van vandaag; dat bracht vaak een hoop last minute werk en kinderziektes met zich mee, die we samen hebben aangepakt en overwonnen. We hebben honderden studenten naar de eindmeet geleid. Annick, dankjewel voor het delen van pure onderwijservaring, je vele voorbereidingswerk in het weekend, en voor de aangename babbels tussenin! Ik keek altijd uit naar een les Programmeren 1. Davy, bedankt voor de technische hulpmiddelen waarmee we de uitdaging konden aangaan om eerstejaarsstudenten zo goed mogelijk de basis van objectoriëntatie bij te brengen. Ik bedoel natuurlijk Indianio en Capthook. Ook merci voor de luchtige insteek die bij jou nooit ver te zoeken was. Nico en Niels, steeds als ik weer eens dreigde bedolven te worden onder de berg oefeningen en bijhorende testen die we in de loop der jaren opgesteld en gewijzigd hebben, kon ik op jullie hulp rekenen.

Sarah, je verdient een bijzondere vermelding voor je inspirerende persoonlijkheid. Zelden heb ik iemand ontmoet met zo’n geduld, rustig, realistisch en supervriendelijk. Bedankt voor alle korte en langere topgesprekken, voor je verstandige opmerkingen en om me meer dan eens met m’n mond vol tanden te zetten. Al ben je dan al bijna twee jaar vol toewijding aan je doctoraat aan het schrijven, je blijft de verstandigste student aan wie ik ooit les heb mogen geven. Als halftime TWIST-er, halftime VIB-er en nog met een promotor in het verre Granada, heb je het misschien niet altijd even gemakkelijk, maar ik heb je niet één keer horen klagen. Integendeel, je benadert alles vanuit je positieve maar kritische ingesteldheid.

Aster, Cara en Malin (in alfabetische volgorde), ik ben behoorlijk trots op jullie; daarom kunnen jullie hier ook niet ontbreken. Tot slot wil ik ook mijn ouders bedanken voor alle goede zorgen en voor alle kansen. Jullie hebben me altijd gesteund, al snapten jullie vaak helemaal niks van wat ik aan het doen was, en jullie stonden ook altijd klaar voor me.

Contents

Chapter 1

Introduction

Declarative programming is about telling the computer what to do, but not how. An exact definition of the notion has been the subject of much debate, but the term certainly includes the logic programming paradigm, a paradigm originating in the application of first order logic to problems in artificial intelligence in the early seventies.

Nowadays, a wide range of subdomains exists [7] like traditional logic programming, answer set programming [89], probabilistic logic programming [105] and many more. This thesis deals with traditional logic programming. It is by itself already an extremely interesting domain. The reason for this is of course the support for nondeterminism, the ability to deal with multiple next steps, that is lacking from imperative and functional languages.

Kowalski's Adage Logic programming is the realisation of the ideal of R. Kowalski, captured in his well-known adage [86]:

$$\textsc{Algorithm} = \textsc{Logic} + \textsc{Control}$$

According to this adage, the algorithms that make up computer programs consist of two disjoint parts: one logic part encoding the domain-specific knowledge and another part instructing the computer how to reason with that logic. The ideal of Kowalski is that the programmer should be able to only focus on the former, as it deals with describing the essence of the problem and not with the details of how to calculate a solution. Indeed, modelling the problem itself is for all but the most trivial problems already a challenging task. Given a model of the problem, Kowalski believes that many computational challenges that arise can be solved with computer-supplied reasoning.

There are two difficult and long lasting issues with the practical realisations of this idea, which form the topic of this thesis:

- For pragmatic reasons like performance, a strict separation between logic and control cannot be maintained in state-of-the-art logic programming.
- There are several situations where the standard computer-supplied control is too weak and this shines through to the logic layer. In these situations, the programmer needs to adapt itself to the control, rather than the inverse.

The Importance of Kowalski's Adage Kowalski's adage is important for several reasons. First of all, the adage has a clear focus on modelling domain-specific knowledge, as can also be seen in flowcharts, UML-diagrams and design documents typically used in the planning stage of software development. It results in high-quality software that fulfills the customer's need. This focus on domain-specific knowledge is also advocated for use in the implementation phase in various other settings:

- For safety-critical applications, formal specification languages are typically used [12]. The verification of the resulting model is performed by the computer.
- Bertrand Meyer's design-by-contract approach [98] is now part of every self-respecting computer science curriculum. At runtime, the computer verifies that the behaviour of the program remains within the terms of the contract.
- In a sense, even unit tests can be seen as as an executable specification of domain specific knowledge. But of course, tests do not realise any customer-usable functionality of the system.

Secondly, letting the computer supply the control considerably raises the abstraction level of a programming language, so that we can tackle more difficult problems without having to cope with an overwhelming amount of low-level details. Indeed, in many other areas of technology and engineering, abstraction has been the key to success stories:

- Integrated circuits are an abstraction of individual transistors. Thanks to integrated circuits, we are able to build supercomputers.

- Operating systems abstract over the underlying hardware. Thanks to operating systems, we do not have to worry about our specific hardware. We can use time-sharing to maximally use the computer's processing power and so forth.

Even in our day-to-day life we use abstraction:

- A network map of the London Underground is an abstraction of the underlying physical layout of the lines and tube stations.
- A credit card is an abstraction of money, that is an abstraction of bartering goods.

Third, logic has long been used to model the human thought process. As such, it forms a natural basis for instructing computers. The lack of mutable state (in the most pure flavours) of logic programming that many novice users find so peculiar is then a consequence of this model: it is not more peculiar than the presence of mutable state in object-oriented programming which naturally arises from mimicking real-world objects.

Goal-Directed Logic Programming There exist two major computation strategies in logic programming: goal-directed (also known as top-down) and bottom-up. A goal-directed evaluation starts from a query and tries to prove it by using the rules in the program to expand the query until only facts remain. The actual process used is called refutation. A bottom-up evaluation starts from facts and derives all consequences using the rules in the program until the query is derived.

In this thesis, we exclusively focus on the goal-directed approach. It is used in the Prolog language, which is the oldest and best-known logic programming language. Prolog has been developed as a direct consequence of Kowalski's ideas. However, even today, Prolog has not fully realised the adage due to the limitations of the techniques chosen to instantiate it.

Prolog models the logic part of Kowalski's equation using Horn clauses and simple builtins. This choice exposes a uniform and simple interface to the programmer. The limited number of language constructs makes it easy to reason formally about a program. Moreover the homoiconic nature of the language considerably simplifies automated manipulation.

The control part of Kowalski's adage is modelled with SLD-resolution [85] that corresponds to a top-down and left-to-right traversal of an implicit search tree defined by the logic rules in the program. SLD-resolution is both conceptually simple and resource efficient, the latter of which was especially important given the limited processing power at the time Prolog was designed. Nowadays Prolog virtual machines are heavily optimised for SLD-resolution. The architecture of a Prolog virtual machine is nearly always based on a variant of Warren Abstract Machine (WAM) [171, 4].

The choice to model logic with Horn clauses — complemented with builtins — is not without its drawbacks. Horn clauses do not offer any way to capture frequently occurring patterns in program code. As a consequence, maintaining a code base of a reasonable size quickly becomes a tedious and boring task. Therefore, programmers have frequently used program transformation (exploiting Prolog's homoiconic nature) to avoid this. However transformations are not an ideal solution, as they interact with each other. This seriously slows down the development and adoption of new language features.

Neither is the choice of SLD-resolution for control optimal. SLD-resolution by itself does not offer any flexibility neither in the parts of the search tree scoured, nor in the traversal order. The search is always exhaustive and its order does not dynamically adapt itself based on the properties of the specific program.

For many real world problems an exhaustive search of the problem space is simply infeasible due to sheer size. The standard approach in computer science is the use of heuristics to attempt finding a solution, rather than all solutions, or a near-optimal solution instead of the optimal solution, but within reasonable time and with reasonable resources. For Prolog, this is no different, but currently the only way to implement these heuristics is by adapting the problem logic to explicitly encode a heuristic of choice. This goes against the ideal of logic programming and has deep practical consequences:

- no reuse of the heuristics is possible;
- the entanglement of logic and control affects maintainability: typically experimentation with many different heuristics is needed, so that the programmer ends up with many different versions of the same program.

In summary: the need to use search heuristics in the SLD-setting creates an overlap between logic and control that is not present in Kowalski's ideal.

Secondly, SLD-resolution frequently requires a procedural — as opposed to declarative — reading of the program rules. Programs incompatible with the procedural reading loop infinitely, despite the fact that they are logically sound. For example: the inability of Prolog to deal with left-recursive rules surprises many newcomers [149, 148].

Tabling [20] has been developed as a means to tackle the shortcomings of SLD-resolution, yet maintain its goal-directedness. It combines the efficiency of top-down SLD-resolution with the cycle insensitivity of a Datalog-style bottom-up computation. At the same time it avoids recomputation. Unfortunately, the barrier for adoption has proven to be too high, as only a few Prolog implementations currently provide tabling.

1.1 The Purpose and Structure of this Dissertation

The purpose of this thesis is to investigate alternatives for both the limited modelling capabilities and the control issues in contemporary Prolog. To tackle these issues, we borrow techniques from the functional programming paradigm and adapt them to the Prolog setting.

Introduction to Prolog In Chapter 2 of this thesis, we give a brief introduction to Prolog. As one focus of this thesis is to propose alternatives for and enhancements to the SLD-resolution strategy, we explain this form of computer-provided control. We then focus on the more advanced concepts needed in the rest of this thesis, like meta-interpretation, definite clause grammars and nonbacktrackable variables. We do not extensively discuss the implementation of a Prolog virtual machine, but limit ourselves to pointing out a few relevant parts of the WAM.

Modular Search In Chapter 3 on modular search heuristics, we provide a hook into Prolog's disjunction. This hook allows us to execute a handler at every disjunction. The handler then implements the desired search heuristic. In this way, logic and heuristic are no longer entangled, effectively providing a solution to the reuse and maintainability issues discussed previously. We have titled this approach Tor. However, the hook-based approach is a very operational approach. We also provide a much more elegant declarative approach based on the observation that each heuristic in isolation already defines a search tree of a form that is typical for that particular heuristic.

Delimited Control In Chapter 4, we introduce delimited control for logic programming. Delimited control has an almost mythical status in the functional programming world, but it had never been introduced to logic programming. This form of control cannot only be used to implement various features commonly found in existing Prologs, it also provides the foundations for the easy development of new language features. In this light, the effect handlers approach provides a structured approach on top of the foundations with ongoing research towards achieving more efficiency [78]. In particular, we use delimited control in Chapter 7 where we develop a lightweight implementation of tabling. Of course, we cannot introduce delimited control without providing a formal semantics, a study of how it can be implemented and an investigation of what interactions it has with other language features. We discuss its low-level implementation and verify the performance of the resulting engine.

Modular Search revisited In Chapter 5, we return to the problem of modular search heuristics. We develop a formal specification in Haskell for the problem using the free monad transformer. One particularly interesting result is that we can derive an efficient Prolog implementation from this specification. The process is nontrivial: although the free monad transformer in theory forms a good basis for a Prolog implementation of modular search heuristics, implementing the free monad transformer itself would be very challenging. Fortunately, it is exactly delimited control that provides an isomorphic replacement, and as we will have seen by then from the discussion and examples in the preceding chapter, delimited control suits Prolog very well.

Tabling with Delimited Control The next three chapters discuss tabling. Tabling is perhaps the most widely studied extension of standard Prolog, as it considerably raises the declarative nature of the language. Tabled resolution strategies have better termination properties than standard SLD-resolution and their engines may drastically improve performance for solving problems that can be expressed in terms of smaller instances of themselves.

In particular, Chapter 6 is an introductory chapter for tabled resolution. We study existing implementation mechanisms for tabled resolution and point out their strengths and weaknesses. We conclude that there is no tabling implementation that offers good performance while keeping implementation efforts at a minimum.

Chapter 7 tackles the challenge identified in the previous chapter: we show how a lightweight tabling mechanism can be implemented on top of an existing Prolog. We keep WAM changes at an absolute minimum by requiring only support for delimited continuations to be implemented natively. Indeed, delimited continuations provide exactly the suspension capabilities that lie at the heart of any tabling implementation. But in contrast to existing suspension mechanisms, delimited control does not require major architectural changes to the underlying virtual machine. We implement all surrounding infrastructure such as call variant and answer tries in Prolog itself. In particular we come up with a simple datastructure that avoids recomputation. Although several approaches exist that do not avoid recomputation and have nonetheless exhibited good performance, a lot of the implementation effort on the world's most performant tabling engines has been spent on avoiding it, leaving the recomputing engines behind. The resulting tabling implementation is not only a big step forward in lowering the adoption threshold, but additionally serves as a validation for delimited control in Prolog.

In Chapter 8 we investigate answer subsumption for tabled logic programming. For many practical programs, it not necessary to store all answers like standard tabling does, as we are often only interested in answers that are optimal with respect to a particular criterion. To this end various techniques have been developed that discard suboptimal answers during the computation. These techniques are generally referred to as answer subsumption. However, none of the existing techniques has been defined formally, so that it is not clear when the techniques can be safely applied. With the development of a mathematical framework, we provide more insight in the soundness of answer subsumption.

We conclude in Chapter 9 with an overview of what we have achieved, pointing out future work in this area. We also look at the big picture by identifying several other areas in which logic programming falls short and can learn from both the successes and failures of other paradigms.

1.2 Scientific Output

Publications

Peer-reviewed Journals and Conference Proceedings

- Benoit Desouter, Marko van Dooren, Tom Schrijvers, and Alexander Vandenbroucke. Tabling as a Library with Delimited Control. International Joint Conference on Artificial Intelligence (IJCAI), Sister Conferences Best Paper Track (on invitation), 2016.

- Benoit Desouter, Marko van Dooren, and Tom Schrijvers. Tabling as a Library with Delimited Control. Theory and Practice of Logic Programming (TPLP), 2015. Proceedings of the 31st International Conference on Logic Programming (ICLP).

- Tom Schrijvers, Nicolas Wu, Benoit Desouter, and Bart Demoen. Heuristics Entwined with Handlers Combined: from Functional Specification to Logic Programming Implementation. Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP), 2014.

- Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. Tor: Modular Search with Hookable Disjunction. Science of Computer Programming (SoCP), 2014.

- Tom Schrijvers, Bart Demoen, Benoit Desouter, and Jan Wielemaker. Delimited Continuations for Prolog. Theory and Practice of Logic Programming, 2013. Proceedings of the 29th International Conference on Logic Programming (ICLP).

- Benoit Desouter. Implementing LP Systems with CP Techniques. Online Supplement of Theory and Practice of Logic Programming (TPLP), 2013. Proceedings of the 29th International Conference on Logic Programming (ICLP). Doctoral Consortium.

- Benoit Desouter, and Tom Schrijvers. Integrating Datalog and Constraint Solving. Proceedings of the 13th International Colloqium on Implementation of Constraint and Logic Programming Systems (CICLOPS), 2013.

Other Publications

- Tom Schrijvers, Bart Demoen, and Benoit Desouter. Delimited Continuations in Prolog: Semantics, Use and Implementation in the WAM. Report CW 631 of KU Leuven, Department of Computer Science, 2013.

- Benoit Desouter, and Tom Schrijvers. Tor: Modular Search with Hookable Disjunction. Newsletter of the Association for Logic Programming (ALP), 2013.

Presentations

Presentations at International Conferences

- 31st International Conference on Logic Programming (ICLP). Cork, Ireland. 2015. Talk entitled: Tabling as a Library with Delimited Control.

- 29th International Conference on Logic Programming (ICLP). Istanbul, Turkey. 2013. Talk entitled: Delimited Continuations for Prolog. Talk given together with Bart Demoen.

- Doctoral Consortium of the 29th International Conference on Logic Programming (ICLP). Istanbul, Turkey. 2013. Talk entitled: Implementing LP Systems with CP Techniques.

- 13th International Colloqium on Implementation of Constraint and Logic Programming Systems (CICLOPS). Istanbul, Turkey. 2013. Talk entitled: Integrating Datalog and Constraint Solving.

Other Presentations

- Wetenschappelijke Onderzoeksgemeenschap (WOG) Declaratieve Methoden in de Informatica. Houthem-Sint Gerlach, the Netherlands. 2013. Talk entitled: Zipping Trees for Modular Search.

Chapter 2

Goal-directed Logic Programming

2.1 Introduction

Kowalski's well-known adage [86] captures the essence of programming in the equation:

$$\textsc{Algorithm} = \textsc{Logic} + \textsc{Control}$$

Logic programming embraces this adage and interprets it strictly. As such, the ideal of programming is threefold:

- model problems (by offering a declarative formulation for nondeterminism and negation based on logic);
- separate logic from control as much as possible;
- provide automatic control

The best-known logic programming language is Prolog. Prolog was created in the early seventies by Alain Colmerauer. The goal of this chapter is to familiarize the reader with the necessary knowledge of this language to understand the rest of the thesis. A programming language has both a syntax, defining what sequence of symbols make up a valid program, and a semantics defining the meaning of syntactically valid programs. We begin with the definition of Prolog's syntax.

Syntax for Data A Prolog term is either a constant (an atom or a number), a variable or a term $f(t_1, t_2, \ldots, t_n)$ with f a function[1] symbol of arity n $(n > 0)$[2] and t_i terms ($i \in [1, n]$).

Variables start with an upper case letter and atoms with a lowercase letter.

Given a term $f(t_1, t_2, \ldots, t_n)$, the symbol f is called its functor, and its arity is n. The notation f/n is commonly used to denote such a term. Functor names also start with a lowercase letter.

Syntax for Code A literal has either the form l or $l(t_1, t_2, \ldots, t_n)$ $(n > 0)$ with l the predicate symbol (or functor), t_i terms (data) and arity n. Two literals are said to belong to the same predicate iff they have the same predicate symbol and arity. The literal *true* denotes logical truth, and *fail* denotes logical falsehood.

A rule has the following form:

$$p \leftarrow q_1, q_2, \ldots, q_n \quad (n \geq 0)$$

It consists of a head p, which is a predicate, and a body, which is the conjunction $q_1, q_2, \ldots, q_n$. Each q_i is a literal r. A fact is special case of a rule where there are no prerequisites $(n = 0)$ and thus is a literal. As in first-order logic, a predicate is a boolean valued function, where its rules and facts define the values mapped to *true*. A Prolog program itself constitutes of rules and facts.

A Prolog term or literal is said to be ground iff the term/literal does not contain any variables.

We have now defined what a syntactically valid Prolog program looks like.

Unification Unification [167] is the process in which Prolog terms are made syntactically equal by instantiating (or binding) variables. Consider the terms `married(alice,bob)` and `married(X,Y)`. Then `married(alice,bob)` unifies with `married(X,Y)` by binding X to `alice` and Y to `bob`. In contrast, the terms `married(alice,bob)` and `married(carol,bob)` do not unify, as clearly Alice is a different person from Carol. The process performs the least amount of work that is necessary to make the terms syntactically equal. Thus, unification computes the most-general unifier (MGU). For example, given the terms `married(alice,X)` and `married(Y,Z)`, the most general unifier is $\{X \leftarrow \text{alice}, Y \leftarrow Z\}$.

[1] As syntactic sugar, Prolog also allows to use an infix or prefix notation. This is commonly used for builtin, mostly mathematical, operations.

[2] SWI-Prolog allows compound terms with zero arguments. Classical Prolog does not. See `http://www.swi-prolog.org/pldoc/man?section=ext-compound-zero`

Put formally, the unification algorithm in Prolog obeys the following rules:

- If t_1 and t_2 are constants, they unify if and only if they are the same.
- If t_1 is a variable, then bind t_1 to t_2. Similarly if t_2 is a variable.
- If $t_1 = f(t_{1_1}, t_{1_2}, \ldots, t_{1_n})$ and $t_2 = g(t_{2_1}, t_{2_2}, \ldots, t_{2_m})$ then t_1 and t_2 unify if and only if $f = g$, $n = m$ and $\forall i \in [1, n] : t_{1_i} = t_{2_i}$ unify .

From the formal description, it is clear that variables can be unified with each other without having been assigned a value. Once a variable has been bound, its value cannot be changed.

Occurs Check In theory, the unification of a variable `V` and a term `T` should fail if `T` contains `V`. Checking this property is known as the "occurs check". Many Prolog implementations leave out this check by default for reasons of performance. This may even cause unification to loop forever. The Prolog ISO standard requires the presence of `unify_with_occurs_check/2` for sound unification [27]. Some implementations can optionally perform the occurs check for all unifications (for example SWI-Prolog).

High-level Semantics A Prolog program models a world where some things can be proved and others cannot. A simple query or goal is a regular literal that the Prolog engine will attempt to prove. In addition, the Prolog engine can also prove compound queries. A compound query is a conjunction of simple queries and is considered provable if all of its constituent simple queries are provable. A provable query is also said to be satisfiable. Facts are the base literals that are taken to be unconditionally true. Other terms can be proved via rules: the body of a rule represents the prerequisites that must be provable before the head is taken to be true.

If a ground query can be proved, the system will answer "true". If the query is nonground, the system will in addition provide an answer substitution. An answer substitution tells the user which variables must be unified. If a query can be proved in different ways, the system will provide multiple answer substitutions.

The Prolog engine attempts to find a proof for a given query using a fixed control mechanism. The control mechanism is called SLD-resolution [85]. To prove a query, the engine first selects the matching clauses: those rules for which the head unifies with the query. Given a query q and two matching clauses p_1 and p_2, SLD resolution will try both clauses in the order they appear in the source code. Given a clause p consisting of the conjunction

q_1, q_2, SLD-resolution will try both conjuncts from left to right. Thus the matching clauses, and the clauses for each of their body terms together with their defining clauses and so on, define an implicit search tree that Prolog scours in depth-first fashion. The term goal is used to indicate any part of the search tree that the engine currently tries to prove.

Example 1. *For example, to prove the query* `parent(X,bill)` *given the following small database of facts:*

```
parent(alice,bob).
parent(carol,bill).
parent(bob,bill).
```

the matching clauses are `parent(carol,bill)` *and* `parent(bob,bill)`*. Both clauses will be tried successfully in the given order. The associated substitutions are* $\{X \leftarrow carol\}$ *and* $\{X \leftarrow bob\}$ *respectively.*

Example 2. *In the following example there are two rules defining the concept of an ancestor.*

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

When those rules are taken together with the facts above, the implicit search for the query `ancestor(X,bill)` *looks like Figure 2.1, where we have used* `a/2` *as an abbreviation for* `ancestor/3` *and similarly* `p/2` *for* `parent/2`*. The lightning sign (ϟ) denotes failure to prove the goal and the square (□) indicates success. For reasons of clarity, we have not labeled the edges with the answer substitutions made.*

Note that in the above example, we have deliberately chosen the order of the rules so that the nonrecursive rule comes first (rule ordering). Also, we have ordered the goals within the second rule so that there is no left-recursion (goal ordering).

There are three alternative orderings:

1. The recursive rule is the first one, and uses right recursion.

2. The rule with the base case comes first, and the inductive rule uses left recursion.

3. The inductive rule is the first one, and uses left recursion.

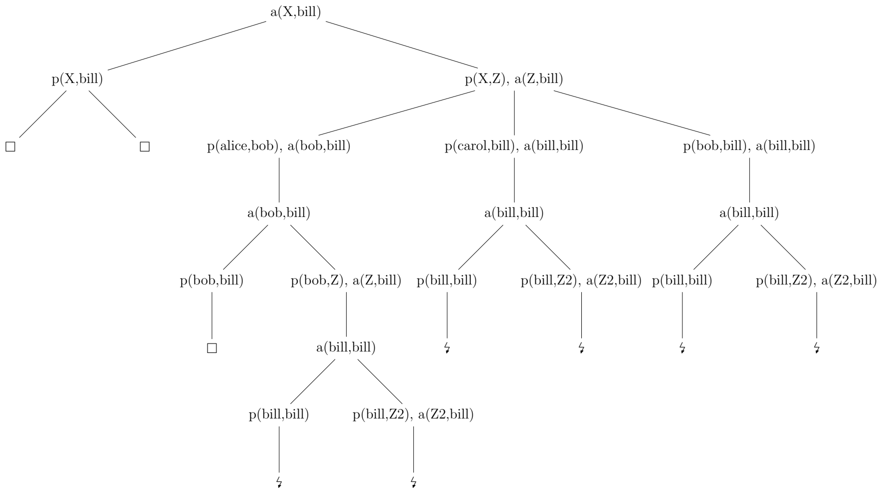


Figure 2.1: Resolution tree for the query `ancestor(X,bill)`.

When we run the same query `ancestor(X,bill)` on each of the alternatives, alternative 1 also works as expected. Alternative 2 yields the correct answers, but then loops infinitely. Alternative 3 loops infinitely. The last two alternatives loop infinitely because SLD-resolution searches the implicit search tree in a depth first manner: once it gets stuck in an infinite branch, it can never escape.

The SLD-control strategy has multiple advantages:

- it is easily understandable for the programmer, although he needs to be aware of rule and goal ordering;

- Prolog implementations can be (and are) heavily optimised for this strategy so that only a limited amount of memory is needed.

Several chapters in this thesis are devoted to the disadvantages of the SLD-resolution and a study of the alternatives. In particular, Chapter 3 discusses how search heuristics that are compatible with an exhaustive depth-first search can be modularly implemented in Prolog. In Chapter 7 we discuss the implementation of a tabling mechanism, a series of control mechanisms developed specifically to tackle the fundamental weaknesses of depth-first search.

Negation So far, we have defined Horn clause programs. Horn clause programs or definite programs are the class of Prolog programs that do not use negation or aggregation. Over the years, negation and some aggregation predicates have been added. We do not discuss the latter as they are not important for the remainder of the thesis, and do not represent a fundamental change. We cannot, however, ignore the former.

The negation used in Prolog is negation-as-failure: the negation of a goal succeeds if the goal itself cannot be proved. As such, it implements a closed world assumption: anything that cannot be proved is assumed to be false. In standard Prolog, there is no three valued logic where some things can be unknown, and truth is always absolute.

To support negation, we need to adapt the allowed syntax for rules. A rule now has the following form:

$$p \leftarrow q_1, q_2, \ldots, q_n \quad (n \geq 0)$$

The head p, is unchanged with respect to the previous definition. The body is still a conjunction $q_1, q_2, \ldots, q_n$ but each q_i is now either a literal r or its negation $\backslash + r$.

Disjunction, Cut and Selection Typical Prolog implementations allow three more constructs: disjunction, cut and selection.

Disjunction is denoted with a semicolon and is allowed only in rule bodies. It is not a fundamental construct, as any rule containing disjunction can be rewritten as a set of rules with the same head. The definition used in the SWI-Prolog documentation is:

```
Goal1 ; _Goal2 :- Goal1.
_Goal1 ; Goal2 :- Goal2.
```

A cut, denoted by an exclamation point, makes the interpreter commit to the clause in which it appears and discards any choice points made to the left of the cut in the current clause. Cut is controversial as it does not have a semantics backed by first-order logic, but only an operational semantics.

For selection, the syntax is somewhat unusual:

```
( If ->
  Then
;
  Else
).
```

The definition used in the SWI-Prolog documentation is:

```
If -> Then; _Else :- If, !, Then.
If -> _Then; Else :- !, Else.
```

Implementation Virtually all modern Prolog implementations implement a virtual machine that is some variant of the Warren Abstract Machine (WAM) as proposed by D. H. D. Warren [171, 4].

A Prolog implementation typically distinguishes three (logical) stacks, that we explain in a simplified form below. This will help the reader understand the implementation of delimited control in the Prolog virtual machine discussed in Section 4.5 on page 79. The three stacks are:

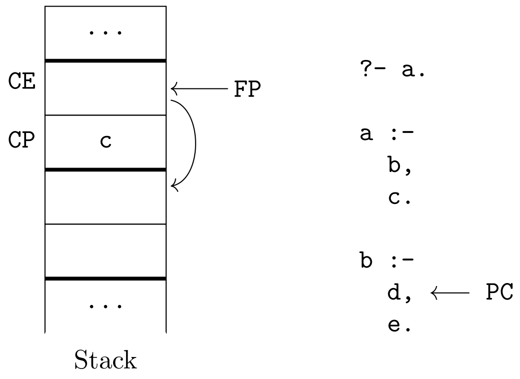
- an environment stack[3]
- a choicepoint stack
- a trail stack

Other important components in an implementation are the heap, and a set of registers to store data temporarily.

Local Stack The local stack has the same function as the call stack known from imperative programming languages. Figure 2.2 shows a simple program with two rules. There are nor arguments nor variables in this program. The query is `?- a.`; the situation right before the call to predicate `d` is shown, as indicated by the program counter `PC`. Every predicate call generates a frame on the local stack. The frame pointer `FP` points to the top of stack. A frame contains a pointer to the calling frame, normally indicated by `CE` (for "continuation environment") and a pointer to the goal that must be called if the current rule is proved successfully. This pointer is called `CP`, for "continuation point". It fulfils the same task as the return address in stack frames of imperative languages.

If a rule contains a predicate with arguments, the layout of the stack frame becomes more difficult. In Figure 2.3 we show a modified version of the program. The query is again `?- a.`. For every argument there is one additional slot containing a pointer into the heap. The argument itself is represented on the heap (which is used as a stack of slots because of the implementation of backtracking explained below, hence the alternative name "global stack") in a straightforward way.

[3]Also called the local stack.

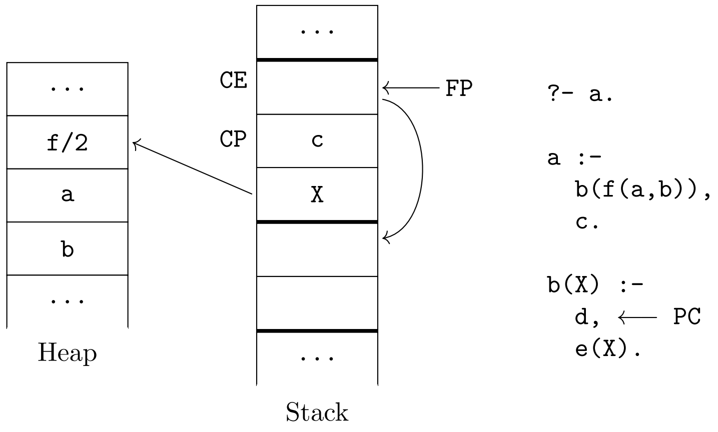


```
?- a.

a :-
  b,
  c.

b :-
  d, ⟵ PC
  e.
```

Figure 2.2: Illustrating the function of the local stack (part 1).



```
?- a.

a :-
  b(f(a,b)),
  c.

b(X) :-
  d, ⟵ PC
  e(X).
```

Figure 2.3: Illustrating the function of the local stack (part 2).

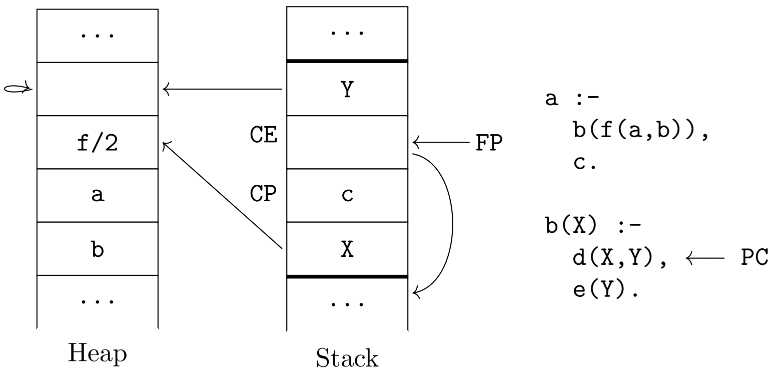


Figure 2.4: Illustrating the function of the local stack (part 3).

If a rule contains local variables, the layout of the stack frame is again adapted accordingly. Figure 2.4 shows such a situation. There is one additional slot per local variable containing a pointer into the heap. In this case, the local variable is still unbound and there is just one heap cell pointing to itself.

Choicepoint and Trail Stack Perhaps the main distinguishing feature of Prolog is its built in support for nondeterminism. Let us consider a rule with disjunction to see how this is implemented at the engine level (Figure 2.5). The situation depicted shows the engine before the unification in the first branch of the disjunction. We see that a choicepoint has been allocated on the choicepoint stack. This choicepoint contains a next clause pointer (`BP`) so that the engine knows what the alternatives for the choicepoint are. The current trail pointer (`TR`) points to the top of the trail stack. This top is the boundary to unwind bindings on backtracking to the next alternative. The top of heap, pointed to by the current top of heap (`H`) slot, is a boundary to reclaim terms on backtracking. On the heap, we see that `X` is unbound.

Figure 2.6 shows the same rule just after the execution of the unification in the first branch. We clearly see that `X` has been bound to `f(a)`, but that the trail stack records that this binding must be undone on backtracking. Similarly, the choicepoint stack contains the information on information that on backtracking can be garbage collected on the heap.

Unification Unification is implemented by aliasing variables. Figure 2.7 shows the result of the unifications `X = Y`, `Z = Q` and `Q = Y`.

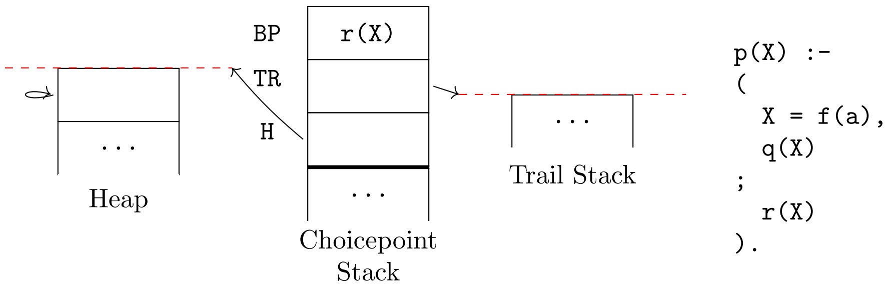


Figure 2.5: Illustrating the function of the choicepoint and trail stacks (part 1).

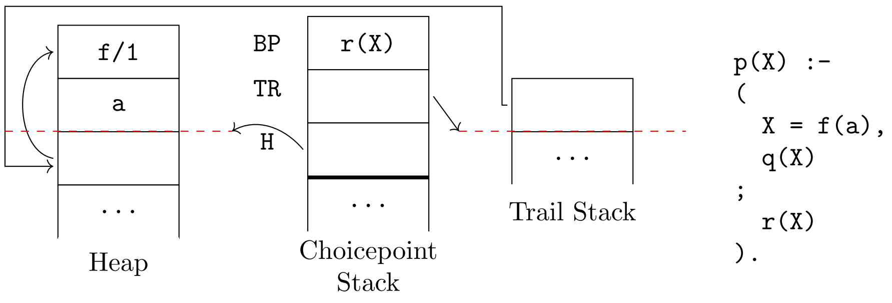


Figure 2.6: Illustrating the function of the choicepoint and trail stacks (part 2).

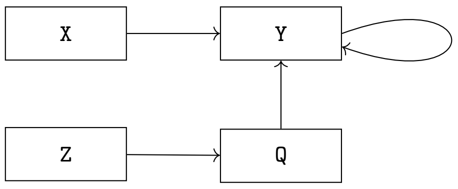


Figure 2.7: Implementing unification by pointer aliasing.

2.2 Vanilla Meta-Interpreter

Prolog is well-suited for meta-programming, as code has the same syntactic form as data. A textbook example is writing a meta-interpreter for Prolog itself [150]. This technique can be used to formally define the semantics of new Prolog language constructs, as shown in Section 4.3. It is also useful for quickly implementing proof-of-concepts of these constructs at the cost of limited execution speed.

```
eval(G) :-
  (G = (G1,G2) ->
    eval(G1),
    eval(G2)
  ;
  G = (T -> G1; G2) ->
    (eval(T) ->
      eval(G1)
    ;
      eval(G2)
    )
  ;
  built_in_predicate(G) ->
    call(G)
  ;
    clause(G,Body),
    eval(Body)
  ).
```

Figure 2.8: A vanilla meta-interpreter for Prolog.

The vanilla meta-interpreter enhanced with support for builtins is show in Figure 2.8. The meta-interpreter is structured as a large case-switch over the constructs that may appear in a syntactically valid Prolog program. As an alternative for the case-switch, multiple rules may be used together with the cut construct. However, correctly limiting the scope of the cuts is tricky, therefore the switch-case version should be preferred. Let us examine each of the cases in the meta-interpreter separately:

Conjunction The first case is conjunction of two goals. So it suffices to evaluate the first conjunct in the meta-interpreter and, in the case of success, evaluate the second conjunct. In short: conjunction in the vanilla meta-interpreter is interpreted as standard Prolog conjunction.

Selection The second case is the Prolog if construct. Again this is interpreted as the regular Prolog if construct, which is not different from the selection statement or expression found in other programming languages. Hence, its semantics is deterministic commitment to either `G1` or `G2`. This is unusual for Prolog, as this language is fundamentally nondeterministic.

Builtins Rather than handling each builtin separately, we assume the existence of the `built_in_predicate/1` that tests whether a particular predicate is a builtin. If this is true, we call the builtin.

Rules In all other cases, we assume that the predicate is defined by a rule, of which we obtain its defining body using the `clause/2`-builtin. Note that facts can be viewed as rules with body `true`. This body is then recursively evaluated in the meta-interpreter.

Note that disjunction is not a fundamental construct, as it can be simulated by using multiple rules with the same head. We have also not included the cut as this construct only has an operational meaning and is not present in traditional Horn-clause programs. Also note that conjunction is binary. Although the user is allowed to write the conjunction of $n, n \geqslant 3$ goals without parentheses, it is internally represented as nested binary conjunctions, which is of course very convenient for meta-interpretation.

2.3 Definite Clause Grammars

DCGs [26, 108] are a well-known Prolog extension to sequentially access the elements of an implicit list. They are a good example for the effect handler's approach discussed in Section 4.2.2 on page 71, which is why we explain them in this introductory chapter.

DCGs provide a convenient notation for formally writing grammars and enable parsing a given input string from that grammar. The grammar rules can be automatically expanded into Prolog clauses. Suppose we want to recognize the language $a^*b^*c^*$, we can write the following grammar:

```
s --> a, b, c.

a --> [a], a.
a --> [].

b --> [b], b.
b --> [].
```

```
c --> [c], c.
c --> [].
```

Each rule consists of a head and a body. The head is a nonterminal symbol and each body item is either a terminal or a nonterminal. Terminal operations are enclosed within lists, and the empty list `[]` corresponds to the empty operation [150]. The query to test whether the sentence `aabbcc` is valid, is `s([a,a,b,b,c,c],[])..` One can also generate all valid sentences in the language using the query `s(X,[])`.

Of course, just recognizing or generating a language is not very interesting. Suppose we want to count the number of occurrences of each letter, then we can modify the grammar as follows:

```
s(Na,Nb,Nc) --> a(Na), b(Nb), c(Nc).

a(N) --> [a], a(N0), {N is N0 + 1}.
a(0) --> [].

b(N) --> [b], b(N0), {N is N0 + 1}.
b(0) --> [].

c(N) --> [c], c(N0), {N is N0 + 1}.
c(0) --> [].
```

Here we add arguments to the nonterminal symbols carrying the number of occurrences of the respective symbol, and we use braces to indicate goals that must be called by Prolog directly. The builtin `is/2` is used to evaluate expressions and is typically used in infix notation. The expression must be ground at the time of evaluation.

2.4 Nonbacktrackable Variables and Mutation

The tabling implementation introduced in Chapter 7 heavily relies on nonbacktrackable global variables and nonbacktrackable term mutation. Moreover, they are also used occasionally in Chapter 3. We first describe the semantics and implementation effort of the predicates for nonbacktrackable global variables. Such variables are available in many popular Prolog implementations. The overhead of these features is only a small constant. Afterwards, we describe nonbacktrackable mutation. The descriptions are adapted from the SWI-Prolog website as it appeared on June 10, 2015.

2.4.1 Global Nonbacktrackable Variables

nb_setval(+Name, +Value) In hProlog, this predicate associates `Value` with the atom `Name` without copying it. The semantics on backtracking to a point before creating the link are poorly defined for compound terms. The principal term is always left untouched, but backtracking behaviour on arguments is undone if the original assignment was trailed and left alone otherwise, which implies that the history that created the term affects the behaviour on backtracking. A `copy_term/2` can be used to avoid this. In SWI-Prolog, `nb_setval/2` binds a copy of `Value` to avoid the ill-defined behaviour with respect to arguments of compound terms. For special purposes, `nb_linkval/2` binds the raw term and is equivalent to hProlog's `nb_setval/2`.

nb_getval(+Name, -Value) Get the value associated with the global non-backtrackable variable `Name` and unify it with `Value`. Note that this unification may further instantiate the value of the global variable.

Native implementation Implementing `nb_setval/2` is not hard, and does not introduce significant overhead in the WAM:

- In hProlog the implementation differs from that of `b_setval/2` by not trailing its argument and freezing the heap in the case of a list or struct. The additional cost is in computing the new frozen heap top. In the worst case, this is linear in the number of choicepoints, but the work required is (for each choicepoint) a simple addition. So this overhead is really insignificant.

- Frozen heap is only reclaimed by garbage collection, but does not make more space reachable, so that the garbage collector's copying phase is unaffected.

- The heap backtrack pointer `HB` must be set to the top of the heap. This must also happen at every backtrack, for which one may introduce an extra WAM-register `FH` (a so-called freeze register). In that case the garbage collection phase also needs to be made aware of this extra register.

Nonnative implementation Mutable variables (also known as *reference cells*) can also be implemented nonnatively by means of mutable terms, as proposed by Aggoun and Beldiceanu [3]. Our implementation for creating, reading and writing such variables comes in both a backtrackable and a non-backtrackable version, and is as follows:

```
new_bvar(InitialValue,Var) :-    new_nbvar(InitialValue,Var) :-
  var(Var),                        var(Var),
  Var = bvar(InitialValue).        Var = nbvar(InitialValue).

b_put(Var,Value) :-              nb_put(Var,Value) :-
  Var = bvar(_),                   Var = nbvar(_),
  setarg(1,Var,Value).             nb_setarg(1,Var,Value).

b_get(bvar(Value),Value).        nb_get(nbvar(Value),Value).
```

The nonnative implementation of mutable variables is available as a separate library on the following webpage:

`http://www.swi-prolog.org/pack/list?p=mutable_variables`.

2.4.2 Nonbacktrackable Mutation

For nonbacktrackable mutation, hProlog and SWI-Prolog provide the predicate `nb_setarg/3` that has the semantics defined below[4]. This predicate uses the same technique as `nb_setval/2`.

nb_setarg(+Arg,+Term,+Value) In hProlog, this predicate assigns the `Arg`th argument of the compound term `Term` with the given `Value`. On backtracking the assignment is not reversed. The term `Value` is not duplicated before assignment. In SWI-Prolog, `Value` is duplicated to avoid the same issues as described for `nb_setval/2`, and `nb_linkarg/3` is provided as a special-purpose alternative.

The implementation can be made thread-safe, reentrant and capable of handling exceptions. Realising these features with a traditional implementation based on assert/retract or `flag/3` is much more complicated.

[4]In GNU Prolog the `setarg/4` predicate can be used to the same effect.

Chapter 3

Modular Search with Hookable Disjunction

3.1 Introduction

In Prolog, the logic part of a problem specification is captured in programmer-supplied rules or clauses that have a first-order logic interpretation. The control component is supplied by the Prolog engine and essentially consists of *search*. In order to answer queries, a Prolog engine performs a backward-chaining depth-first tree search.

Prolog's default search strategy is in practice inadequate to effectively scour large search spaces. As a consequence, the programmer often has to complement Prolog's control with additional hints or heuristics in the form of extra code. This is particularly prevalent in the context of Constraint Logic Programming where it is common practice for the programmer to complement a constraint model with a search specification.

Unfortunately, it is not all that easy to cleanly separate logic and control when implementing search heuristics in Prolog. When one discovers that Prolog's control is ineffective, it is often impossible to orthogonally add one's own control without touching the existing logic. The problem is that syntactically logic and control in Prolog are tightly coupled, and adding different control means cross-cutting existing code.

In this chapter, we present a novel approach to adding control in an orthogonal manner. Our solution features the following properties:

```
label([]).
label([Var|Vars]) :-
  ( var(Var) ->
      fd_inf(Var,Value),
      ( Var #= Value,
        label(Vars)
      ;
        Var #\= Value,
        label([Var|Vars])
      )
  ;
      label(Vars)
  ).
```

```
label([],_).
label([Var|Vars],D) :-
  ( var(Var) ->
      D > 0,
      ND is D - 1,
      fd_inf(Var,Value),
      ( Var #= Value,
        label(Vars,ND)
      ;
        Var #\= Value,
        label([Var|Vars],ND)
      )
  ;
      label(Vars,D)
  ).
```

Figure 3.1: Labeling predicate: plain (left) and with depth bound (right).

- It is a lightweight library-based approach that is easily portable to different Prolog systems: it is currently an SWI-Prolog library [174] available at `http://www.swi-prolog.org/pack/list?p=tor`.
- Our approach has all the benefits of modularity: search methods can be composed and the library of these heuristics is (user-)extensible.
- Its overhead is minimal, as we demonstrate on benchmarks: this is achieved through `term_expansion/2`, a feature present in most Prolog systems.

With Tor, we capture all common search methods in CLP(FD) libraries such as ECLiPSe's `search/6` that provides ten fixed (combinations of) search methods [133]. This approach is indeed particularly suitable for Constraint Logic Programming, but also useful for general Prolog programs with a large search space.

3.2 Problem Statement

We illustrate the heart of the matter on a simple labeling predicate `label/1` written against SWI-Prolog's `clpfd` library [162] (see Fig. 3.1, left). A labeling predicate assigns a value to each variable in the given list of variables (`Vars`).

`label/1` defines a search tree where the branches are created by disjunction.[1]

Suppose that for a certain call `label([X`$_1$`,...,X`$_n$`])` the search tree is too large to fully explore. In order to get some useful answers, certain parts of the tree can be left unexplored, effectively pruning the tree. One particular way in which this can be done is by reaping the low-hanging solutions only, and pruning the subtrees that are below a certain depth. This is achieved by imposing a *depth bound* on Prolog's depth first search. Figure 3.1 shows on the right a variant of `label/1` that implements this idea; the additional parameter is the depth bound.

Imposing a depth bound may or may not be a successful approach to getting useful answers. If it turns out to be unsuccessful, other pruning strategies can be tried, like imposing a node bound or a discrepancy bound. Each of these requires rewriting the `label/1` predicate to incorporate a different pruning technique. In general, an explorative process takes place whereby several different variants of the labeling code are written and evaluated until an effective pruning strategy is found.

3.2.1 Problems with this Approach

The problems with the above approach should be apparent:

- The approach follows the well-known copy-paste-modify anti-pattern. Variants of the labeling code are copied all over the place, potentially propagating bugs and rendering maintenance into a nightmare. Working code is modified.
- The same heuristic is implemented over and over in different settings (different applications, different labeling predicates, different Prolog systems, ...). This process is error-prone, wastes precious programmer time and is bound to yield non-optimal code quality.
- The effort and expertise required to combine working labeling code with various search heuristics is non-trivial. This means that fewer combinations are explored by programmers under time pressure or unfamiliar with particular heuristics. The end result is that suboptimal solutions are obtained.
- As soon as the labeling code spans several different predicates or multiple invocations of the same predicate, the complexity of adding search heuristics increases drastically.

[1] `fd_inf/2` returns the smallest value in a variable's finite domain. The operators starting with a hash impose arithmetic constraints.

3.2.2 Current Solutions

Most of the current solutions are specific to CLP, and we are aware of one general Prolog approach.

CLP Solutions In the context of CLP ECLiPSe [133] copes with this problem by providing a number of search methods in the `search/6` predicate. This predicate lets the user control through its various arguments the selection method, the choice method and the search method: the former two decide on which variable is used during labeling, and which value it is assigned first. They do not concern us here. The search method controls how the search tree is explored, e.g., depth-bounded, node-bounded or limited discrepancy search. Apart from individual search methods, only a fixed number of compositions is supported, such as changing the strategy when a depth bound is reached. In this setting users can extend the set of supported heuristics and combinations by reprogramming parts of the `search/6` predicate.

The same approach can be found in other Prolog systems' CLP(FD) libraries, albeit to a more limited extent. SICStus Prolog [17] allows imposing discrepancy and time limits, and B-Prolog [175] provides a time limit. GNU Prolog [41] and Ciao's [62] new `clpfd` library provide no way to limit the search on top of depth-first traversal.

All CLP(FD) libraries do provide one extra search method: optimization with respect to an objective value. Optimization is typically implemented as either branch-and-bound or by restarting the whole search with a new bound whenever a solution is found.

Typically these approaches only support adding search heuristics to a simple goal made up of a labeling predicate defined in the corresponding CLP library. This means that complex goals made up of a conjunction of labeling calls or custom labeling predicates are not supported. ECLiPSe is the only system that provides one search method, branch-and-bound, independent from a particular labeling predicate.

Prolog Solution We are aware of only one other approach to modify Prolog's own search method: the breadth-first and iterative deepening program transformations in Ciao [62]. These modify annotated predicates in place and are not compositional.

All in all the available library support that Prolog systems provide is very limited indeed. As soon as users face a (constraint) problem that requires a non-trivial search method, they are forced to write all their search code from scratch, and it can be very daunting to combine different search methods.

Non-Prolog Solutions There are a range of effective search techniques that are not based on (depth-first) tree search like local search, genetic algorithms, simulated annealing, ... However, these techniques are out of scope of this chapter. We only consider search methods that are compatible with Prolog's depth-first search.

Chapter Organization The rest of this chapter is structured as follows. First, Section 3.3 outlines the Tor approach. Next, Section 3.4 reviews Tor's standard library of search methods. Then, Section 3.5 covers the Tor's implementation. Section 3.6 discusses how Tor allows to observe the search tree for (performance) debugging purposes. We illustrate the application of Tor on an example Prolog problem in Section 3.7. Next, Section 3.8 evaluates the Tor implementation. In Section 3.9, we present a simple automatic specializer that mitigates overhead even in applications without constraint propagation using Tor. Section 3.10 addresses related work and Section 3.11 concludes.

3.3 Solution Overview

3.3.1 User Perspective

Tor divides search code into two parts: a) the code that defines the *search tree*, and b) the code that defines the *search method*. The user defines these separately (or reuses existing definitions from a library) and combines them into a search goal.

Search Tree Code The search tree code shown in Figure 3.1 sets up the problem specific search tree using `label/1` as an example. To fit in the Tor framework one provision has to be made: the code must use Tor's custom disjunction `tor/2` rather than `;/2`. For instance, `tor_label/1` is the Tor-compatible variant of `label/1`.

```
tor_label([]).
tor_label([Var|Vars]) :-
  ( var(Var) ->
      fd_inf(Var,Value),
      (    Var #= Value,
           tor_label(Vars)
      tor
           Var #\= Value,
           tor_label([Var|Vars])
```

```
            )
        ;
            tor_label(Vars)
        ).
```

Search Methods A search method is defined as a predicate that captures the essence of that method in a declarative way, as a bare-bones search tree without any useful work (such as labeling variables). For instance, dbs_tree/1 captures the depth-bounded search method.

```
dbs_tree(Depth) :-
  Depth > 0,
  Depth1 is Depth - 1,
  ( dbs_tree(Depth1)
  tor
    dbs_tree(Depth1)
  ).
```

Just like the search tree code, the search method code must respect syntactic restrictions:

- The search method code must be defined as a predicate with a single clause.
- This clause must contain at most one invocation of tor/2.
- Moreover, each of the two branches of that disjunction may contain at most one directly recursive invocation.
- Finally, there may be no indirectly recursive calls and no indirect invocations of tor/2.

The reason for these restrictions is explained in Section 3.5 on page 41.

Combining Search Tree and Search Method The user imposes a search method on a search tree by calling the predicate tor_merge(MGoal,TGoal), where MGoal is a call to the search method predicate and TGoal is a call to the search tree predicate. Conceptually, tor_merge/2 overlays or merges the search trees of the two goals, synchronizing their tor/2 disjunctions.

An example of tor_merge's behavior is graphically depicted in Figure 3.2. The top left search tree is that of dbs_tree(4), where all the red (northeast-striped) leaves at level 5 denote failures. The top right search tree is that

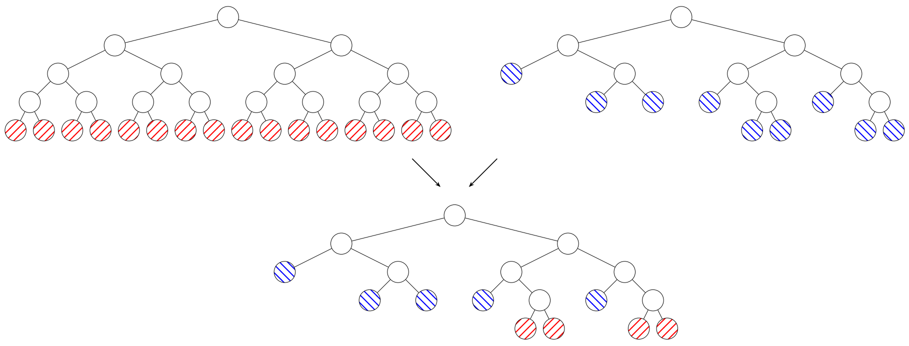


Figure 3.2: Search trees of `tor_merge(dbs_tree(4),tor_label([X,Y]))`.

of `tor_label([X,Y])`, where the blue (northwest-striped) leaves at various levels denote solutions. The bottom search tree is obtained by merging both other trees. The corresponding leaves are overlaid. When an internal node is overlaid with a leaf, the leaf wins out. If both nodes are internal nodes, the resulting node is an internal node. When both nodes are leaves, the leaf from the left tree wins out.

To facilitate reuse, we generally recommend to encapsulate the application of `tor_merge/2` to a particular search method in a separate predicate, like `dbs/2` for `dbs_tree/1`.

```
dbs(Depth,Goal) :-
  tor_merge(dbs_tree(Depth),Goal).
```

This makes for more concise calls, like `dbs(4,tor_label(Vars))`.

Wrapping Up In the final step, the Tor predicate `search(Goal)` is used to, conceptually, replace all the occurrences (merged or not) of `tor/2` by proper Prolog disjunctions.

In summary, the behavior of `label/2` of Fig. 3.1 is recovered as follows:

$$\begin{array}{c}\texttt{search(dbs(Depth,tor\_label(Vars)))}\\ \equiv \\ \texttt{label(Vars,Depth)}\end{array}$$

3.3.2 Modularity Aspects

The big contribution of the Tor approach is its *modularity*. Here we look in more detail at the modularity aspects of Tor that are not found in any of the existing systems.

Decoupling of Search Tree and Search Method

The first modularity advantage of TOR is that it decouples the code that defines the *search tree* from the code that defines the *search method.* This decoupling means that new search methods and new search tree code can be written without awareness of one another and without the modification of any existing code. This means that, once developed, new search methods and labelling code can easily be reused in many different settings.

Contrast this with ECLiPSe's `search/6` predicate. It tightly couples the options for setting up the search tree (like variable and value selection strategies) with those for the search method. For instance, the CLP(FD) search is hard wired and thus the available search methods do not support pure Prolog search. Users cannot add new search methods or labeling approaches without adapting the existing code.

Finally, we note that this decoupling does not exclude already supported forms of modularity. In particular, various problem-specific heuristics exist for deciding how to build the search tree. Well-known examples are variable and value selection strategies in CLP(FD) and these are an essential part of an effective search. There are already good solutions for modularizing variable and variable selection strategies in CLP(FD) libraries and TOR does not duplicate their effort. Nevertheless TOR is inherently compatible with these modular solutions: the strategies can easily be integrated in the search tree code. We refer to the companion code library for several examples.

Modular Combination of Search Tree Code and Search Method

Because their implementations are decoupled, there is no inherent restriction on the combination of search tree code with search method code. To make matters more concrete, let us consider an additional search method `lds/1` (short for limited discrepancy search, explained in Section 3.4.3 on page 37) and an additional search predicate `tor_member/2` (the TOR variant of the well-known `member/2`). We can now express four different search scenarios by varying both the search tree and search method code:

```
?- search(dbs(10,tor_label([X1,...,Xn]))).
?- search(dbs(10,tor_member(X,[X1,...,Xn]))).
?- search(lds(tor_label([X1,...,Xn]))).
?- search(lds(tor_member(X,[X1,...,Xn]))).
```

More concretely, any other search method and labeling predicate can be combined in the same way, whether they originate from the Tor library or are defined by the user. Of course, it is still up to the user to assess which composition is effective for his problem. No CLP(FD) library we are aware of provides this functionality.

Advanced Compositions

Beyond the basic combinations illustrated above, Tor supports the modular composition of multiple search methods and/or multiple labeling goals. None of these are readily expressible in existing CLP(FD) systems.

Composition of Labeling Goals A user can define a complex labeling goal as the conjunction of two invocations of `tor_label/1`.[2]

```
?- search(lds((tor_label([X1,...,Xn])
              ,tor_label([Y1,...,Ym])))).
```

This example becomes more interesting when the two lists of variables are labeled with different variable and value selection strategies.

Composition of Search Methods With nested invocation the user can compose two (or more) existing search methods into a new one. This composition denotes that both search methods are simultaneously active in every node of the search tree.

For instance, we can simultaneously apply a depth limit and perform a limited discrepancy search:

```
?- search(dbs(10,lds(tor_label([X1,...,Xn])))).
```

Contrast this with the non-modular approach where the user would face the much more complex task of writing a combined search heuristic `dbs_lds/2` from scratch.

Putting Everything Together Finally, the compositional nature of the notation can be exploited to its fullest potential to obtain sophisticated search specifications. For instance, the goal

[2]Observe that this example is fundamentally distinct from the simpler goal `?- search(lds((tor_label([X1,...,Xn])))), search(lds((tor_label([Y1,...,Ym])))).`

```
?- ...,
   search(lds((dbs(XsLimit,tor_label(Xs))
              ,dbs(YsLimit,tor_label(Ys))))).
```

applies limited discrepancy search to the whole search tree, and additionally imposes one depth-limit on the search of the `Xs` and another to that of the `Ys`.

3.4 Search Method Library

Following the Tor approach, it is easy to write various search methods in a modular way. While the user can write custom ones himself, Tor already provides a substantial library of search methods. We cover several of them here.

3.4.1 Discrepancy-Bounded Search

The discrepancy-bounded search heuristic is a small variant of depth-bounded search: the bound is only updated in right branches.

```
dibs(Discrepancies,Goal) :-
  tor_merge(dibs_tree(Discrepancies),Goal).

dibs_tree(Discrepancies) :-
  ( dibs_tree(Discrepancies)
  tor
    Discrepancies > 0,
    NDiscrepancies is Discrepancies - 1,
    dibs_tree(NDiscrepancies)
  ).
```

3.4.2 Iterative Deepening

Iterative deepening emulates breadth-first search by means of increasing depth-bounds. The implementation consists of a driver `id_loop/3` that initiates an iteration with a given depth bound and, if pruning occurred, starts the next one with an incremented depth-bound.

An iteration consists of a search with a variant of the depth-bounded heuristic, `id_tree/3`; it differs from depth-bounded search in that it reports its pruning in the non-backtrackable mutable variable `PVar` (see Section 2.4). This variable communicates to the driver whether a new iteration should be started or not.

```
id(Goal) :-
  new_nbvar(not_pruned,PVar),
  id_loop(Goal,0,PVar).

id_loop(Goal,Depth,PVar) :-
  nb_put(PVar,not_pruned),
  ( tor_merge(id_tree(Depth,PVar),Goal)
  ;
    nb_get(PVar,Value),
    Value == pruned,
    NDepth is Depth + 1,
    id_loop(Goal,NDepth,PVar)
  ).

id_tree(Depth,PruneVar) :-
  ( Depth > 0 ->
      NDepth is Depth - 1,
      ( id_tree(NDepth, PruneVar)
      tor
        id_tree(NDepth, PruneVar)
      )
  ;
    nb_put(PruneVar,pruned),
    false
  ).
```

The problem with this definition is that the upper parts of the search tree are repeatedly searched. This is because in iteration $i + 1$ there is no means to restart the search from those parts that were cut away in iteration i. In the following chapter, we propose a method that could capture those subtrees, instead of cutting them away. The search could then be restarted from there.

3.4.3 Limited Discrepancy Search and Factored Iteration

The traditional limited discrepancy search [61] is a minor variant of iterative deepening. It applies the depth-bound only in right branches. Put differently, limited discrepancy search is to discrepancy-bounded search what iterative deepening is to depth-bounded search.

With some abstraction (`with_pruned` and friends — defined below), we can factor out the common iteration part of iterative deepening and limited discrepancy search:

```
iterate(PGoal) :-
  with_pruned(
    iterate_loop(0,PGoal)).

iterate_loop(N,PGoal) :-
  (
    call(PGoal,N)
  ;
    is_pruned,
    reset_pruned,
    M is N + 1,
    iterate_loop(M,PGoal)
  ).
```

This iteration pattern runs a goal `PGoal` that is parameterized by a natural number `N`. The goal uses this number as a bound and applies pruning when the bound is exceeded. The iteration repeatedly restarts the goal with successive values for `N` until the goal completes without pruning.

With this iteration pattern we can express iterative deepening and limited discrepancy search as follows:

```
id(Goal)  :- iterate(flip(dbs,Goal)).
lds(Goal) :- iterate(flip(dibs,Goal)).

flip(Goal,Y,X) :- call(Goal,X,Y).
```

There is only one complicating factor: we need to communicate the pruning from the handler to the iteration. Fortunately, global variables allow us to do that.

```
prune :-
  set_pruned(true),
  fail.

reset_pruned :-
  set_pruned(false).

is_pruned :-
  get_pruned(true).

get_pruned(Flag) :-
  nb_getval(pruned,Flag).
```

```
set_pruned(Flag) :-
  nb_setval(pruned,Flag).

with_pruned(Goal) :-
  get_pruned(OldFlag),
  ( reset_pruned,
    call(Goal)
  ;
    set_pruned(OldFlag),
    fail
  ).
```

With the imperative ugliness hidden in the above definitions, the following new definition of `dbs_tree` handler subsumes both `id_tree/2` and the previous `dbs_tree/1` definitions.

```
dbs_tree(Depth) :-
   ( Depth > 0 ->
       Depth1 is Depth - 1,
       ( dbs_tree(Depth1)
       tor
         dbs_tree(Depth1)
       )
   ;
       prune
   ).
```

Node-Bounded Search

A node-bounded search is much like a depth-bounded search, except that the decrements of the limit are not backtracked. Hence, as an optimization we abort the whole search at once by throwing an exception rather than gradually failing out of the search tree.

```
nbs(Nodes,Goal) :-
  new_nbvar(Nodes,NodesVar),
  catch(
    tor_merge(nbs_tree(NodesVar),Goal),
    out_of_nodes(NodesVar),
    fail
  ).

nbs_tree(Var) :-
  nb_get(Var,N),
  ( N > 0 ->
    N1 is N - 1,
    nb_put(Var, N1),
    ( nbs_tree(Var)
    tor
      nbs_tree(Var)
    )
  ;
    throw(out_of_nodes(Var))
  ).
```

3.4.4 Branch-and-Bound Optimization

This well-known optimization approach posts constraints in the intermediate nodes of the search tree to find increasingly better solutions. Our implementation uses Tor to access those intermediate nodes and generate increasingly larger values of the `Objective` variable. It uses two variables, `BestVar` and `Current`. The former keeps track of the overall best solution so far, while the latter is the solution that the current node tries to improve upon.

Both the overall and current best solution are initialized to a value smaller than the infimum of the objective variable's domain. Whenever a solution is found, the overall best solution is updated. Whenever we backtrack into a Tor choice point, the heuristic synchronizes the current best solution with the overall best solution. If the current best solution was out of sync, the handler also imposes a new lower bound on the objective variable. Note that `inf` denotes negative infinity.

```
bab(Objective,Goal) :-
  fd_inf(Objective,Inf),
  LowerBound is Inf - 1,
  new_nbvar(LowerBound,BestVar),
  Current = inf,
  tor_merge(bab_tree(Objective,BestVar,Current),Goal),
  nb_put(BestVar,Objective).

bab_tree(Objective,BestVar,Current) :-
  nb_get(BestVar,Best),
  ( Best \= inf , (Current == inf ; Best > Current ) ->
      Objective #> Best,
      NCurrent = Best
  ;
      NCurrent = Current
  ),
  ( bab_tree(Objective,BestVar,NCurrent)
  tor
    bab_tree(Objective,BestVar,NCurrent)
  ).
```

3.4.5 More Search Methods

We have implemented many other orthogonal search methods with Tor, including all those offered by ECLiPSe's `search/6` predicate. These can be found in the companion code.

3.5 Tor Infrastructure Implementation

3.5.1 Hookable Disjunction

Tor is built around one core predicate, `tor/2`, which replaces the regular Prolog disjunction in search tree code. The predicate is defined as:

```
G1 tor G2 :-
  ( b_getval(left,Left),
    call(Left,G1)   % conceptually: Left(G1)
  ;
    b_getval(right,Right),
    call(Right,G2)  % conceptually: Right(G2)
  ).
```

This definition provides two hooks into the disjunction by means of global variables `left` and `right`.[3] In these hooks the programmer installs *handlers* for the left and right branches to control the search. These handlers are *higher-order* predicates that take a goal and execute it in a (possibly) modified manner.

We obtain standard Prolog disjunction, if we use `call/1` as handler:

```
?- findall(X, ( X in 1..10
              , b_setval(left,call)
              , b_setval(right,call)
              , tor_label([X])
              ), Values).
Values = [1,2,3,4,5,6,7,8,9,10].
```

The point of Tor is of course to install more interesting handlers.

3.5.2 From Search Methods to Handlers

More interesting handlers originate from the search method. The `tor_merge/2` predicate transforms their high-level definitions into pairs of low-level handlers, before it installs those handlers. This transformation proceeds in two phases. First the search method definition is normalized, and then the handlers are extracted.

[3]Note that `b_getval/2` and `b_putval/2` are SWI-Prolog builtins for reading and writing global mutable variables, whose names are atoms. Their non-backtrackable counterparts are `nb_getval/2` and `nb_putval/2`.

Search Method Normalisation

In the first phase, the `rewrite/2` predicate rewrites the search method definition into a normal form

```
sm(X1,...,Xn) :-
  ( Left
  tor
    Right
  ).
```

If `tor/2` is defined as the usual disjunction, both arguments of `rewrite/2` have (on success) the same logical interpretation.

```
rewrite((Head :- Body),(Head :- Left tor Right)) :-
  split(Body,Left,Right).

split(tor(GL,GR),GL,GR) :- !.
split((G1,G2),(GL1,GL2),(GR1,GR2)) :- !,
  split(G1,GL1,GR1),
  split(G2,GL2,GR2).
split((Test -> G1 ; G2),
        (Test -> GL1 ; GL2),(Test -> GR1 ; GR2)) :- !,
  split(G1,GL1,GR1),
  split(G2,GL2,GR2).
split((G1;G2),(GL1;GL2),(GR1;GR2)) :- !,
  split(G1,GL1,GR1),
  split(G2,GL2,GR2).
split(G,G,G).
```

Handler Extraction

The left and right handlers are derived from the `Left` and `Right` branches of the search method's normal form:

```
sm_left(X1,...,Xn,Goal) :-
  NLeft.
sm_right(X1,...,Xn,Goal) :-
  NRight.
```

where `NLeft` and `NRight` are derived from `Left` and `Right` by replacing any recursive calls with `call(Goal)`. Moreover, if any of the recursive calls features parameters that are not the same as in the head, that parameter is wrapped in a mutable variable. For instance, the `Depth` parameter of `dbs_tree/1` changes to `Depth1` in the recursive calls. Hence, the following handlers are derived:

```
dbs_tree_left(MDepth,Goal) :-
  b_get(MDepth,Depth),
  Depth > 0,
  Depth1 is Depth - 1,
  b_put(MDepth,Depth1),
  call(Goal).
dbs_tree_right(MDepth,Goal) :-
  ... % identical
```

Finally, a `tor_merge(sm(T1,...,Tn),Goal)` goal is rewritten into the appropriate invocation of `tor_handlers/3`:

```
tor_handlers(Goal, sm_left(T1,...,Tn), sm_right(T1,...,Tn)).
```

In case any of the parameters need to be wrapped in a mutable variable, `tor_merge/2` also takes care of that. For instance,

```
?- tor_merge(dbs_tree(4),tor_label(Xs)).
```

becomes

```
?- new_bvar(4,MVar),
   tor_handlers(tor_label(Xs),dbs_tree_left(MVar)
                             ,dbs_tree_right(MVar)).
```

3.5.3 Handler Infrastructure

Default Handler

The predicate `search/1` sets up the default handler for both hooks: `call/1`.

```
search(Goal) :-
  b_setval(left,call),
  b_setval(right,call),
  call(Goal).
```

With this default handler, `tor/2` corresponds simply to plain disjunction `(;)/2`.[4] For instance, with `search/1` we recover the behavior of `label/1` of Fig. 3.1 from the Tor-variant:

$$\texttt{search(tor\_label(Vars))} \equiv \texttt{label(Vars)}$$

[4]Apart from the scope of any cuts in the alternative branches

Extending Installed Handlers

In order to facilitate installing new handlers, TOR provides a convenient predicate: `tor_handlers/3`.

```
tor_handlers(Goal,Left,Right) :-
  b_getval(left,LeftHandler),
  b_getval(right,RightHandler),
  b_setval(left,compose(LeftHandler,Left)),
  b_setval(right,compose(RightHandler,Right)),
    call(Goal),
  b_setval(left,LeftHandler),
  b_setval(right,RightHandler).

compose(G1,G2,Goal) :- call(G1,call(G2,Goal)).
  % conceptually: G1(G2(Goal))
```

This predicate assumes that there are already handlers installed, either by `search/1` or a previous invocation of `tor_handlers/2`. It does not replace the installed handlers by the new ones, but composes them with `compose/3`.[5] This accounts for the ability to compose search methods, discussed in Section 3.3.2.

Finally, `tor_handlers/2` also scopes the effect of the new handlers: they are only active in the provided goal. After execution of the goal, the old handlers are reset.

3.5.4 Custom Low-Level Handlers

In addition to writing high-level search methods, expert users can also exploit TOR's low-level infrastructure and write custom low-level handlers that don't fit the search method pattern. Here we show two such cases.

Higher-Order Search Methods

ECLiPSe's `search/6` provides several *higher-order search methods*. These are search methods that are parameterized by other search methods.

An example of this is the following `dbs/3` variant on depth-bounded search. When it reaches the depth bound, it does not prune the remaining subtree, but activates the search method `Method`. A typical example is to limit the discrepancy once we reach a certain level in the search tree. This is achieved with `dbs(Level,lds(Discrepancies),Goal)`.

[5]While `compose` is a ternary predicate, recall that it has to be used in partially applied form in `left` and `right`.

```
dbs(Level, Method, Goal) :-
  new_bvar(yes(Level),Var),
  tor_handlers(Goal,dbs_handler(Var,Method)
                   ,dbs_handler(Var,Method)).

dbs_handler(Var,Method,Goal) :-
  b_get(Var,MDepth),
  dbs_handler_(MDepth,Var,Method,Goal).

dbs_handler_(yes(Depth),Var,Method,Goal) :-
  ( Depth > 1 ->
      NDepth is Depth - 1,
      b_put(Var,yes(NDepth)),
      call(Goal)
  ;
      b_put(Var,no),
      call(Method,Goal)
  ).
dbs_handler_(no,_,_,Goal) :-
  call(Goal).
```

The first-order search `dbs/2` can then be redefined as `dbs(Level,prune,Goal)` where:

```
prune(Goal) :- prune.
```

In ECLiPSe, only a fixed number of parameters can be supplied to these higher-order search methods, and `search/6` explicitly caters for each separate combination in its implementation. Not so with TOR. There is no restriction on the possible combinations; the higher-order search methods are truly parametric.

Parallel Search

It turns out that the comparatively simple interface of TOR is even general enough to express at least a naive implementation of parallel search. The query `?- search(parallel(tor_label(Vars),5))` uses 5 processes to explore parts of the search tree in parallel. It is based on the definition of `parallel/2` below.

```
parallel(Goal,N) :-
  set_available_processes(N),
  tor_handlers(Goal, tor_fork, call).
```

```
tor_fork(Goal) :-
  ( i_am_a_child ->
      call(Goal)
  ;
      wait_for_available_process,
      fork(PID),
      PID == child,
      call(Goal)
  ).
```

For a left branch, the code uses the `fork/1` predicate to duplicate the current Prolog process,[6] yielding a so-called *parent* process and a concurrent *child* process. The child process (determined via `i_am_a_child/0`) explores the goal. Since the goal `PID == child` fails in the parent process, this parent process backtracks and considers the right branch which is delegated by the installed handler to the built-in `call/1` predicate, and whose left tor-branches are again subject to `tor_fork/1`.

We have used three more predicates that need explanation:

- `set_available_processes/1` initializes the number of available (sub-) processes,
- `i_am_a_child/0` succeeds if and only if the current process is not the main Prolog process, and
- `wait_for_available_process/0` waits until a process is available and then succeeds: any time a process is forked, the number of available processes goes down by one, and when a process finishes, the number of available processes goes up by one.

All three predicates can be implemented in an ad-hoc way in SWI-Prolog.

To illustrate the parallel exploration of two independent branches in a simple and self-contained example, consider the query:

```
?- search(parallel(repeat tor X = 2,1)).
```

which yields `X = 2` on the toplevel (a shared resource among all created processes), whereas this specific solution cannot be obtained with regular Prolog disjunction because it is hidden by an infinite branch due to the goal `repeat`.

Clearly, the possibilities of search parallelism based on the Tor framework are worth exploring further, in particular regarding communication between processes, and using threads instead of processes for portability and efficiency.

[6]`fork/1` is available in SWI-Prolog on Unix platforms.

3.6 Search Tree Observation

The original purpose of Tor was to allow the manipulation of search tree traversal by various search heuristics. It turns out that Tor also enables various ways to observe the search tree, so that one can gain insight in the search process itself, e.g., for (performance) debugging purposes. We illustrate in the next sections plain statistics and visualization.

3.6.1 Statistics

Similar to SWI-Prolog's `profile/1`, `time/1` and `statistics/0` predicates, we can provide different components that monitor various metrics of the search tree and provide us with a convenient summary. In the following example, we constrain 4 finite domain variables to the domain $1, \ldots, 4$ via the library's `ins/2` constraint and emit all solutions found by labeling, including accompanying statistics:

```
?- length(Xs,4), Xs ins 1..4,
     search(tor_statistics((tor_label(Xs),writeln(Xs)))),
     false.
[1,1,1,1]
% Number of solutions: ......... 1
% Number of nodes: ............. 4
% Number of failures: .......... 0
  ...
[4,4,4,4]
% Number of solutions: ....... 256
% Number of nodes: ........... 510
% Number of failures: .......... 0
```

The code for `tor_statistics/1` is in the Tor library.

To support users who want to check whether they have successfully replaced all regular disjunctions with Tor, we also provide a tool that uses SWI-Prolog's choice point inspection primitive `prolog_current_choice/1` to verify this.

3.6.2 Visualization

In addition to summarized data of the search tree, we can also visualize the actual search tree itself with Tor. For that purpose, we provide a predicate that emits a textual representation, a log, of the search tree:

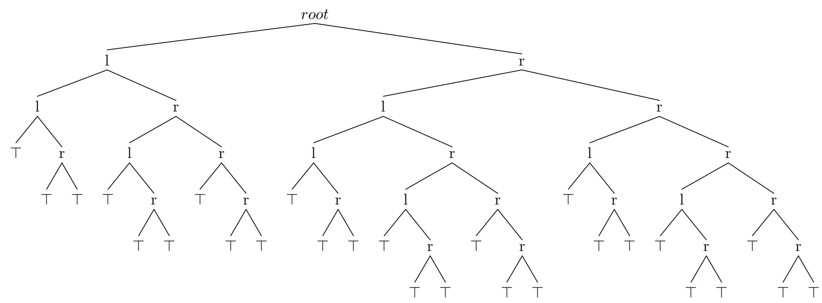


Figure 3.3: Search tree for labeling 3 variables with domains of size 3 that are not involved in any constraints.

```
log(Goal) :-
  tor_merge(log_tree,Goal),
  writeln(solution).

log_tree :-
  ( ( writeln(left)
    tor
      writeln(right)
    ),
    log_tree
  ;
    writeln(false),
    false
  ).
```

A complimentary tool that turns this log into a PDF image is also available from our public code repository. Due to our concise decision to transform the textual logs to scalable vector graphics in PDF format, there is no inherent limit on the sizes of search trees that users of Tor can visualize with this tool.

Fig. 3.3 shows the complete search tree for labeling 3 variables with domains of size 3 that are not involved in any constraints: `Xs = [_,_,_], Xs ins 1..3, search(log(tor_label(Xs)))`. The symbol $\top$ denotes that a solution is found at this node, while l and r denote internal nodes generated by left and right branches of `tor/2` respectively.

Fig. 3.4 shows two search trees for the 8-queens puzzle: The left one was created with depth limit (search strategy `dbs`) 4 and contains no solutions. The right one was created with depth limit 7 and stopped the search after

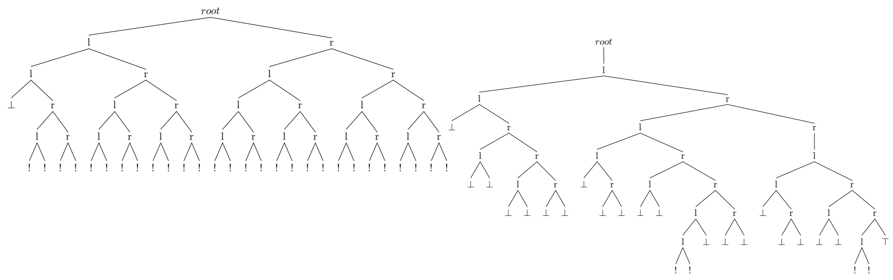


Figure 3.4: Search trees of 8-queens with depth bound 4 and 7.

finding the first solution. Hence, only the right-most leaf is a solution. The symbol ⊥ denotes pruning due to constraint propagation, and ! denotes a node that is not explored because the depth limit is exceeded at this level of the search tree.

It would be interesting to further integrate the logging output with the more powerful CP visualization tool CP-Viz [144].

3.7 Plain Prolog Example

While the application of Tor to CLP problems is obvious, we wish to emphasize that Tor is not limited to CLP.

For that reason we illustrate the use of Tor on the well-known problem of the wolf, the goat and the cabbage. The following code, adapted from Sterling and Shapiro [150], implements this decision problem in plain Prolog (without constraints). Naive depth-first execution of this code loops infinitely.

```
wgc :-
  initial_state(State),
  wgc(State).

wgc(State) :-
  final_state(State), !.
wgc(State) :-
  move(State,Move),
  update(State,Move,State1),
  legal(State1),
  wgc(State1).

initial_state(wgc(left, [wolf, goat, cabbage], [])).
```

```
final_state(wgc(right, [], [wolf, goat, cabbage])).

move(wgc(Bank, Left, Right),Move) :-
  ( Bank == left,
    tor_member(Move, Left)
  tor
    Bank == right,
    tor_member(Move, Right )
  tor
    Move = alone
  ).

:- tor tor_member/2.
tor_member(X,[X|_]).
tor_member(X,[_|Xs]) :- tor_member(X,Xs).

update(wgc(B,L,R), Cargo, wgc(B1, L1, R1)) :-
  update_boat(B, B1),
  update_banks(Cargo, B, L, R, L1, R1).

update_boat(left, right).
update_boat(right, left).

update_banks(alone, _B, L, R, L, R) :- !.
update_banks(Cargo, left, L, R, L1, R1) :- !,
  select(Cargo, L, L1),
  insert(Cargo, R, R1).
update_banks(Cargo, right, L, R, L1, R1) :-
  select(Cargo, R, R1),
  insert(Cargo, L, L1).

insert(X,[Y|Ys], [X,Y|Ys]) :-
  precedes(X,Y), !.
insert(X, [Y|Ys], [Y|Zs]) :-
  precedes(Y,X), !,
  insert(X,Ys,Zs).
insert(X, [], [X]).

precedes(wolf, _X).
precedes(_X, cabbage).

legal(wgc(left, _L, R)) :- \+ illegal(R).
legal(wgc(right, L, _R)) :- \+ illegal(L).

illegal(Bank) :- memberchk(wolf, Bank),
                 memberchk(goat, Bank).
illegal(Bank) :- memberchk(goat, Bank),
                 memberchk(cabbage, Bank).
```

The nondeterministic enumeration in this code is situated in the `move/2` and `tor_member/2` predicates.[7] In order to use Tor, we have replaced ordinary Prolog disjunction with `tor/2`.

To avoid the non-termination, we can apply a depth-bound and discover in finite time that the problem has a solution.

```
?- search(dbs(17,wgc)).
   true.
```

Of course this is not the only search method that solves the problem. Thanks to Tor, it is convenient to explore many others and to determine the most effective one for the problem at hand.

3.8 Evaluation

To study Tor's overhead, we have performed a number of benchmarks on a MacBook Pro, with a 2.4 GHz CPU and 4 GB RAM, running Mac OS X 10.6.7. We compare two Prolog systems with different performance characteristics. On the one hand we consider SWI-Prolog 5.11.7, a feature-rich, but relatively slow Prolog system with a CLP(FD) solver written in Prolog. On the other hand, we consider B-Prolog 7.5#3, one of the fastest Prolog systems with a highly optimized CLP(FD) implementation.

3.8.1 Pure Search

Table 3.1 considers the extreme situation where the search is pure enumeration of *unconstrained* constraint variables:

```
length(N,Vars), Vars ins 1..D
```

Hence, no constraint propagators are activated due to choices. Values are simply enumerated.

[7]The `tor/1` declaration implicitly adds Tor-disjunctions between the clauses of a predicate.

	our `label/1`		`clpfd`'s `label/1` B-Prolog's `labeling/1`		`search/6`	
	man	Tor	man	Tor	man	Tor
SWI-Prolog						
`N=6,D=8`	1.80 s	240 %	2.08 s	151 %	2.55 s	132 %
`N=6,D=9`	3.63 s	249 %	4.20 s	153 %	5.09 s	135 %
`N=6,D=10`	6.82 s	269 %	7.87 s	155 %	9.53 s	137 %
`N=7,D=8`	14.44 s	244 %	16.63 s	153 %	20.40 s	134 %
`N=7,D=9`	32.80 s	269 %	37.80 s	155 %	46.04 s	136 %
`N=7,D=10`	68.27 s	278 %	78.63 s	157 %	94.30 s	139 %
B-Prolog						
`N=6,D=8`	0.49 s	156 %	0.09 s	276 %	0.12 s	223 %
`N=6,D=9`	0.99 s	157 %	0.18 s	283 %	0.23 s	221 %
`N=6,D=10`	1.87 s	160 %	0.32 s	291 %	0.44 s	219 %
`N=7,D=8`	4.56 s	144 %	0.71 s	306 %	0.94 s	220 %
`N=7,D=9`	8.90 s	163 %	1.59 s	301 %	2.06 s	225 %
`N=7,D=10`	18.64 s	163 %	3.25 s	332 %	4.37 s	220 %

Table 3.1: Labeling benchmarks without propagation: execution times.

	our `ff_label/1`		`labeling/2`		`search/6`	
	man	Tor	man	Tor	man	Tor
SWI-Prolog						
`allinterval`	4.03 s	101 %	4.02 s	101 %	4.01 s	101 %
`golf`	3.93 s	99 %	3.92 s	100 %	3.96 s	99 %
`mhex`	18.59 s	102 %	18.61 s	101 %	18.46 s	101 %
`n_queens`	2.03 s	103 %	2.05 s	102 %	2.09 s	102 %
`sudoku`	2.14 s	101 %	2.15 s	101 %	3.40 s	100 %
B-Prolog						
`allinterval`	1.14 s	100 %	0.81 s	112 %	0.89 s	109 %
`knapsack`	3.94 s	125 %	2.11 s	175 %	2.17 s	172 %
`knight`	0.67 s	101 %	0.71 s	100 %	0.91 s	100 %
`mhex`	0.23 s	106 %	0.19 s	107 %	0.23 s	104 %
`n_queens`	1.01 s	107 %	0.89 s	107 %	1.03 s	106 %

Table 3.2: Labeling benchmarks with propagation: execution times. Note that the problem sizes of the benchmarks are not the same for SWI-Prolog and for B-Prolog.

The first column denotes the problem size, expressed in the number of variables `N` and their domain size `D`. The other three pairs of columns denote different implementations of labeling: 1) `label/1` as listed in this paper, 2) `label/1` from SWI-Prolog's `clpfd` library and the corresponding `labeling/1` provided by B-Prolog, and 3) `search/6` ported from ECLiPSe to SWI-Prolog and B-Prolog with minimal changes. For each of these, we show the absolute runtime of the standard/manual version (man) and the relative runtime of the Tor version (tor).

In both SWI-Prolog and B-Prolog the impact of Tor is pretty consistent across the problem sizes, but depends on the labeling implementation. In SWI-Prolog, the overhead is most prominent (140-180 %) in our bare-bones `label/1`, while it is less so (50-60 %) in `clpfd`'s `label/1`. The latter delegates to `labeling/2`, which involves more generic option processing. Finally, in `search/6` Tor compensates its overhead further (to 30-40 %) by not collecting search statistics when these are not demanded. In ECLiPSe's implementation, these statistics are collected regardless of demand.

In B-Prolog, the performance characteristics of the labeling predicates are markedly different. Firstly, the cost of the inequality `(#\=)/2` in our `label/1` is relatively high, which keeps the overhead of Tor low (60%). In contrast, the two other labeling predicates rely on B-Prolog's `domain_inst_next/3` for enumeration, which compiles down to a single abstract machine instruction. As a result the overhead of Tor is much higher, more so in the tight `labeling/1` (170%-230%) than the more bloated `search/6` (120%).

In summary, in these propagation-free benchmarks, the overhead of Tor goes up to about a factor three for tight labeling loops, but is lower for option-rich labeling predicates. Moreover, Tor is better behaved in SWI-Prolog than in B-Prolog. All in all, we find that this is a very reasonable price to pay for the extra flexibility that Tor provides. Still, invoking Tor's specializer (see the next section) can get rid of all overhead.

3.8.2 Search vs. Propagation

While the performance penalty of Tor is limited in the previous benchmarks, the performance-wary user may not be willing to accept the overhead. However, the previous benchmarks are not representative of realistic CLP problems, that spend a lot of time on constraint propagation in every node of the search tree. All this extra work easily dwarfs the overhead of Tor. Table 3.2 illustrates this observation on a number of typical CLP benchmarks.

For added realism, the benchmarks use the first-fail variable selection strategy, with hand-written labeling code `ff_label/1`, the two library predicates

	plain	lds	dibs-1	dibs-2	credit/bbs
`N= 95`	2.11 s	0.66 s	0.45 s	**0.28 s**	0.33 s
`N= 96`	**0.65 s**	4.98 s	4.89 s	1.13 s	1.04 s
`N= 97`	T/O	3.68 s	**3.56 s**	22.66 s	4.08 s
`N= 98`	T/O	15.67 s	† 5.71 s	10.16 s	**2.50 s**
`N= 99`	T/O	2.42 s	**2.22 s**	9.85 s	2.57 s

† no solution

Table 3.3: N-Queens benchmarks with various search methods: execution times.

`labeling/2` (SWI-Prolog) and `labeling_ff/1` (B-Prolog), and the ported `search/6`. Because B-Prolog's CLP(FD) solver is orders of magnitude faster than SWI-Prolog's, it makes little sense to use exactly the same benchmarks for the two platforms. Instead, we resorted to different problem sizes or different benchmarks altogether.

In the case of SWI-Prolog, we see that Tor introduces no (significant) overhead; its runtime is marginal compared to that of constraint propagation. In the case of B-Prolog, the overhead of Tor is more noticeable, in the order of 10% for most benchmarks. Only in the case of the knapsack problem does it go up to 75% for the tightest labeling loop.

In summary, we see no performance reason to avoid the use of Tor for most CLP problems. Especially in SWI-Prolog there is no runtime price to pay. In the setting of B-Prolog, an extra 10% runtime is a low price for the extra flexibility that Tor provides. Moreover, in the next section we will see how we can eliminate Tor's overhead to the extent that we don't pay for it if we don't use the capabilities it provides.

3.8.3 Search Methods

Finally, Table 3.3 illustrates once more why we want to use different search methods: they can significantly reduce the runtime while still leading to useful solutions. The table shows the runtime for finding the first solution of the n_queens benchmark in SWI-Prolog for 5 different problem sizes and 5 different search methods: (plain) plain depth-first search, (lds) limited discrepancy search, (dibs-1/-2) discrepancy bounds of 1 and 2, and (credit/bbs) credit-based search with 10,000 credits that switches to a bounded backtracking (1 backtrack) search when the credits are exhausted. The notation T/O stands for timeout.

3.9 Automatic Specialization

Tor encourages writing fairly abstract and generic code. This style clearly incurs some overhead (notably due to meta-calling) compared to specialized search code. Fortunately, in the case of CLP applications, this overhead is very modest compared to the cost of constraint propagation. However, in the case of applications without constraint propagation, we do observe an overhead that is significant. In order to mitigate that overhead, we exploit Prolog's homoiconic nature to provide a simple but effective automatic specializer.

Even though there is a large body of work on automatic program specialization for Prolog, notably involving partial evaluation, we decided to write our own program specializer. Its main tasks are 1) to perform *constant propagation* on the global variables `left` and `right`, 2) to replace instantiated meta-calls by direct calls and 3) to inline the handler code into the main search loop. For control we follow a lightweight approach based on declarations of what predicates to inline and specialize.

Example 1 Our specializer yields `label/1` for the generic composition `search(tor_label(Vars))`. Similarly, we recover SWI-Prolog's `labeling/2` by specializing its Tor variant. Hence, we do not pay if we do not modify the search.

Example 2 The specialized form of the goal

```
search(dbs(N, tor_label(Vars)))
```

is `new_bvar(N,DVar), label21(Vars, DVar)`, with:

```
label21([], _).
label21([Var|Vars], DVar) :-
  ( var(Var) ->
      fd_inf(Var, Val),
      ( b_get(DVar, Depth),
        Depth>0,
        NDepth is Depth+ -1,
        b_put(DVar, NDepth),
        Var#=Val,
        label21(Vars, DVar)
      ;
        b_get(DVar, G),
        G>0,
        NDepth is G+ -1,
        b_put(DVar, NDepth),
            Var#\=Val,
            label21([Var|Vars], DVar)
          )
      ;
          label21(Vars, DVar)
      ).
```

This code is slightly less efficient than that of `label/2`. Firstly, the overhead of mutable variables is not entirely eliminated here, as `DVar` is still present. Secondly, the two branches have some code in common that could be shared. However, there are no more meta-calls and all code is inlined in the recursive loop of `label21/2`.

In future work, we intend to get rid of the remaining inefficiencies by implementing additional transformations, including Peter Schachte's approach [131] for eliminating mutable variables adapted to our setting.

3.10 Related Work

We have already covered the most closely related work, existing approaches to search heuristics in Prolog, in Section 3.2.2. Here we cover other important related topics.

Combinators Tor is related to earlier work on Monadic Constraint Programming (MCP) [137] in the context of Haskell, and Search Combinators [138] in the context of C++ and the Gecode library[8]. In contrast to those works, Tor is tailored towards Prolog's built-in depth-first search and, as a consequence, consists of a much simpler and more elegant design.

Comet The imperative Comet language [164] features fully programmable search by means of search controllers [165]. There are two main differences between Tor and Comet's search controllers. Firstly, search controllers trade simplicity for flexibility, providing more hooks and first-class continuations to manipulate the search. Secondly, search controllers are not intended to be composed, in contrast to Tor's handlers that are explicitly designed to support composition.

Gecode Gecode [140] is a C++ library for constraint programming that provides two complimentary means to control the search: search engines and

[8]`http://www.gecode.org`

branchers. A valid search consists of a combination of one search engine and one or more branchers. The search engine determines how to navigate the search tree (e.g., depth first search, depth-first search with iterative deepening, ...) and the branchers define the search tree. A typical brancher is defined, like typical CLP(FD) labeling predicates, in terms of a set of variables, and a variable and value selection strategy. Multiple branchers denote a conjunction. Unlike Tor search engines cannot be composed, and all branchers are subject to one and the same search engine.

Aspect-Oriented Programming The Tor approach is closely related to aspect-oriented programming (AOP) [79, 92]. AOP provides a generic approach for modularly cross-cutting existing code with new code, so-called advice. This advice is injected in arbitrary join points (i.e., program points) based on a pointcut predicate.

Obviously Tor is more limited in scope, as only `tor/2` disjunctions are cross-cut and only at the positions of the two hooks. However, we believe that these "limitations" are actually Tor's strength: its simplicity makes it easy to express all common search methods and its discipline favors compositionality.

A Functional Specification of Modular Search In Chapter 5 we build a formal specification for the Tor approach based on functional techniques. More specifically, we reify the search tree as syntax using the free monad. The heuristics then act on this reified tree. Finally, the syntax is reflected back into the semantics.

We derive an actual Prolog implementation from this Haskell model; this is possible because of an isomorphism between the free monad and the delimited continuations monad. Implementing the free monad transformer in Prolog itself is challenging, but not needed because we have added delimited control to Prolog (see the following chapter), which is a much more natural fit for the language.

3.11 Conclusion and Future Work

We have presented Tor, a lightweight library-based approach for modifying Prolog's depth-first search with reusable and compositional search methods. While the notion of hookable disjunction has enabled a surprisingly large number of possibilities for modifying Prolog search, we still see a few areas that could be improved in future work:

Increased Expressivity Simplicity has been a guiding principle in the design of TOR. In order to minimize the threshold for users, we keep the effort and complexity of defining and using search methods low. We pay for this simplicity with a somewhat restricted expressivity. An example of a search method that cannot be expressed with TOR is swapping the order of branches in a disjunction. In order to overcome this limitation we would have to add extra complexity to the `tor/2` built-in in the form of an additional hook. However, we choose simplicity over additional expressivity. Nevertheless, TOR is remarkably expressive as it is, covering all of the commonly found search methods in CLP(FD) libraries.

It would be interesting to investigate how expressive TOR is in a parallel setting. As a starting point, the naive parallel search strategy from Subsection 3.5.4 could be adapted to use threads instead of processes.

On a more drastic account, we would like to investigate ways to replace TOR's underlying depth-first queuing strategy. The stack freezing functionality of tabling systems like XSB [152] and YAP [130] provides interesting perspectives for this purpose.

Multiway Disjunctions TOR currently only supports binary disjunctions; multiway disjunctions have to be decomposed into binary ones. For some applications, this decomposition can be somewhat unnatural. For instance, when enumerating all the values `V` of a constraint variable `X`, one might expect that all alternative assignments `X #= V` sit at the same level in the search tree. This is of course generally not the case in a binary decomposition.

Multiway disjunction could be provided by a more general hook replacing the hooks for the left and right branches that TOR currently supplies. For example, the following hook would do:

```
or(G1,G2) :- get_hook(P), P([G1,G2]).
mor(Gs) :- get_hook(P), P(Gs).
```

However, it is not clear how the elegant declarative interface based on archetypal search trees could be generalised to this setting. Each inner node of the archetypal search tree would have to dynamically adapt its number of children to match the number of branches in the multiway-disjunctions of the problem's search tree.

Declarative State Management We have hidden the operational aspects of TOR from the programmer with the use of the high-level programming interface for heuristics. Even though the underlying implementation relies on mutable variables, the interface provides a declarative view on state management.

Unfortunately, non-backtrackable state is not covered by the high-level interface; the programmer has to manage it explicitly in an imperative style. The problem is that non-backtrackable state updates are often followed immediately by failure. There is no idiomatic declarative alternative for this technique. However, we could turn to pure deterministic encodings of failure with non-backtrackable state, like Haskell's `ListT (State s)` monad [74] and use Filinski's reification/reflection technique [47] to translate to and from Prolog's native effects.

References

This chapter is based on our article on Modular Search: Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. Tor: Modular Search with Hookable Disjunction. Science of Computer Programming (SoCP), 2014. Benoit Desouter's contribution lies in defining and experimenting with the heuristics from this chapter, as well as the creation of an SWI-Prolog library. Benoit set up the initial assessment of the performance of the Tor approach and assisted in structuring the article's presentation.

Chapter 4

Delimited Control

Prolog, as defined in Chapter 2, is a very minimalist language. Essentially it consists of Horn clauses extended with simple built-in predicates. This minimality has several advantages, but infrastructure to capture common programming patterns is sometimes thoroughly missed.

In the past, this encoding of frequently occurring patterns has been addressed using meta-programming and program transformations. There are several well-known examples: definite clause grammars (DCGs) [26, 108], logical loops [132], extended DCGs [166], and structured state threading [69].

These are all non-local program transformations. For several reasons, they are not ideal for defining new language features:

- the effort of defining a transformation is proportional to the number of features in the language;
- program transformations are fragile: when the language evolves, they require amendments;
- when a new feature is introduced, the whole system may need transforming.

This makes the development and adoption of new language features a difficult case. By bringing a well-known control primitive from the functional world, delimited continuations [46, 31], to Prolog, we alleviate these problems.

Feature extensions based on delimited continuations are more lightweight and more robust with respect to change than traditional approaches. Introducing extensions based on delimited continuations does not require pervasive changes to an existing code base. These nice properties stem from the fact that they enable the definition of new language features at the program level (i.e. in libraries) rather than at the meta-level.

In contrast with BinProlog [157], where continuations are first-class, our primitives for delimited control do give continuations this status. In BinProlog, continuations are normally invisible, while we aim to make them available for library developers.

The Origin of Delimited Control Delimited continuations have their roots in functional programming, and their use in programs that explicitly pass continuations, continuation-passing style (CPS), is folklore. Felleisen introduced the reset and shift primitives for delimited control ("prompt applications") in direct style programs using the untyped lambda-calculus [46]. Danvy and Filinski [31] independently proposed similar operators. Felleisen defined the semantics via translation to a stack-machine, but did not provide an actual implementation. One of his examples was a yield-mechanism on a tree. Felleisen already pointed out the relation of continuations to stream-programming, although he did not distinguish yield as a separate operator. Duba et al. added first-class continuations to the statically typed ML language [43]. Flatt et al. implemented a production version in Scheme [48].

We define the control operators with a different signature and semantics from Danvy and Filinski's version. Our operators are more closely related to the `fcontrol` and `run` proposed by Dorai Sitaram [145].

The structure of this chapter is as follows: we first give an informal semantics for the primitives of delimited control. Next, we show a few applications. In a third section, we define a meta-interpreter semantics. Before diving into the actual implementation in the WAM (Section 4.5), we investigate the relationship of the primitives with `catch/3` and `throw/1`. We discuss some semantic intricacies in Section 4.6 and next move on to a presentation of related work. We evaluate the performance of our approach in Section 4.8 and conclude in Section 4.9.

4.1 Informal Semantics

We have implemented delimited control in Prolog by means of two interacting predicates: `reset/3` and `shift/1`. We now informally describe the semantics of this predicate pair. The pair does not turn continuations into first-class Prolog citizens, yet their usefulness will be made clear in the following section.

```
p :-                                ?- p.
  reset(q,Cont,Term1),              a
  writeln(Term1),                   qterm
  writeln(Cont),                    [$cont$(785488,[])]
  writeln(endp).                    endp

q :-
  writeln(a),
  shift(qterm),
  writeln(b).
```

Figure 4.1: Illustrating the semantics of `reset/3` and `shift/1`.

- `reset(Goal,Cont,Term1)` executes `Goal` and
  - if `Goal` calls `shift(Term2)`, its further execution is suspended and unified with continuation `Cont`. The `reset/3` predicate succeeds. A continuation is an unspecified Prolog term, which can be resumed using `call/1`. It can be called, saved, copied and compared like any other term, but it is opaque: from its representation we cannot determine anything about the actual goals it represents.
  - if `Goal` succeeds, then `reset/3` binds `Cont` and `Term1` to 0.
  - if `Goal` fails, then `reset/3` also fails.
- `shift(Term2)` unifies the remainder of `Goal` up to the nearest call to `reset/3` (i.e., the delimited continuation) with `Cont`, and its return value `Term2` with `Term1`. Finally, it returns control to just after the `reset/3` goal.

Example 3. *Figure 4.1 shows a first example. This example illustrates that `shift/1` unifies the last two arguments of `reset/3`: both are printed out. The continuation in this case represents the `writeln(b)` goal in the context of the activation of the clause for `q/0`. Since this continuation is not activated, this goal has no effect. The example also shows that execution is continued after the `reset/3` goal.*

One important aspect this example shows is that it is up to the continuation of the `reset/3` to call the continuation constructed by `shift/1`. A small adaptation of the above example actually calls the continuation:

```
p :-                                ?- p.
  reset(q,Cont,Term1),              a
  writeln(Term1),                   qterm
  call(Cont),                       b
  writeln(endp).                    endp
```

Compound Goal Given the clauses for `a/0` and `b/0` below,

```
a :-                               ?- a.
  reset((b,                        after_reset
         writeln(in_reset(Term))   after_shift
        ),Cont,Term),              in_reset(shifted)
  writeln(after_reset),
  call(Cont).

b :-
  shift(shifted),
  writeln(after_shift).
```

the result of the query `?- a.` on the right might surprise. However, it is completely in line with the description that `Cont` captures the whole of the continuation after the call to `shift/1` up to just after the `reset/3` goal. A simple way to see that this is the desired behavior is by noting that the clause for `a/0` above is equivalent to:

```
a :-
  reset(newpred(Term),Cont,Term),
  writeln(after_reset),
  call(Cont).

newpred(Term) :-
  b,
  writeln(inside_reset(Term)).
```

In Section 4.6 we discuss the actual semantic intricacies of the reset/shift pair.

4.2 Applications

4.2.1 Coroutines

As a first example of delimited control, we show how coroutines can be implemented. By the term coroutines, we do not mean the Prolog variety, but the

```
from_list([]).
from_list([X|Xs]) :-
  yield(X),
  from_list(Xs).

enum_from_to(L,U) :-
( L < U ->
  yield(L),
  NL is L + 1,
  enum_from_to(NL,U)
;
  true
).
```

```
sum(Sum) :-
  sum(0,Sum).

sum(Sum0,Sum) :-
  ask(X),
  ( X == eof ->
    Sum = Sum0
  ;
    Sum1 is Sum0 + X,
    sum(Sum1,Sum)
  ).
```

Figure 4.2: Some example iterators (left) and iteratees (right).

more general meaning. Coroutines are subroutines that can be suspended and resumed to communicate with another routine. There is growing consensus that coroutines are easier to understand than lazy evaluation [82, 81]. We too believe that Prolog coroutines are to be preferred over lazy evaluation à la Ciao [18], but both techniques allow for a flexible and modular design that emphasises reuse. The communication partners are coupled very loosely.

We can distinguish three different types of coroutines:

iterator An iterator[1] is a coroutine that suspends to output data. The iterator uses the *yield*/1 primitive to suspend and return an intermediate value.

iteratee An iteratee is a coroutine that suspends to request external input. The iteratee uses the *ask*/1 primitive to suspend.

transducer A transducer transforms iterators of one kind into another kind.

Figure 4.2 shows several examples of iterators (left) and iteratees (right). With `from_list/1`, values from the given argument list are generated one by one. With `enum_from_to/2`, one generates all values between the given bounds. This kind of coroutine-based iterators exists in many languages (e.g. Python). The `sum/0` iteratee adds up as many numbers as provided by its external input source.

[1]Quite often also referred to as a generator [90].

```
yield(X) :-                        ask(X) :-
  shift(yield(X)).                   shift(ask(X)).
```

Figure 4.3: Definition of *yield*/1 and *ask*/1.

```
% Iterator consumers               % Iteratee producers

with_write(Goal) :-                with_read(Goal) :-
  reset(Goal,Cont,Term),             reset(Goal,Cont,Term),
  ( Term = yield(X) ->               ( Term = ask(X) ->
    write(X),                          read(X),
    with_write(Cont)                   with_read(Cont)
  ;                                  ;
    true                               true
  ).                                 ).

                                   with_list(L,Goal) :-
                                     reset(Goal,Cont,Term),
                                     ( Term = ask(X) ->
                                       L = [X|T],
                                       with_list(T,Cont)
                                     ;
                                       true
                                     ).
```

Figure 4.4: Providing a context for iterators and iteratees with handlers: example handlers.

Handlers The implementation of the primitives *yield*/1 and *ask*/1 (Figure 4.3) relies on delimited control. The trick is to decouple the syntax of these operations from their semantics. So, the primitives are all syntax and simply shift their own term representation. The semantics is provided by a handler which can be seen as the context in which the coroutine runs. Figure 4.4 defines the `with_write` handler for our example iterators and the `with_read` and `with_list` handlers for our example iteratee:

- The `with_write` handler runs the given iterator in a delimited environment, which is provided by the `reset/3` predicate. Every time the yield operation is encountered, the execution of the iterator is interrupted and control is transferred to just after the reset goal. This is because `yield/1` is defined in terms of `shift/1`, which works together with `reset/3` to provide this behaviour.

  To distinguish between a yield and the iterator running out of elements, the `Term` argument of reset will be bound to yield in the first case and to 0 in the second case. For a yield, we print the yielded element to the terminal and call the handler recursively on the remainder of the iterator, which is bound to the `Cont` argument of the reset. In contrast, if the iterator runs out of elements, we succeed with `true`.

- The `with_read` and `with_list` handlers follow a similar pattern: if the iteratee finishes without asking for more elements, we succeed with `true`. Otherwise we read, respectively extract an element and call the handler recursively. If the user provides no element in the case of `with_read`, or if the list runs out of elements in the case of `with_list`, resolution simply fails.

  In the case of iterators, the context determines the sink of the data, hence we refer to the context as a consumer. With iteratees, the context determines the source of the data, hence we say that the context is a producer.

Consumers are independent from the particular iterator:

```
?- with_write(from_list([1,2,3]).
123
true.
?- with_write(enum_from_to(1,4)).
123
true.
```

The producers can be replaced independently from the iteratees:

```
?- with_list([1,2,3],sum(S)).
S = 6.
?- with_read(sum(S)).
|: 1.
|: 2.
|: 3.
S = 6.
```

Lockstep coroutining Iterators and iteratees can be combined to work together in lock-step. To this end, we define a handler `play/2` that uses both. This handler calls the first communication partner `G1` in a reset and whenever it yields or asks for an element, it is suspended. Next, `play/2` runs the other communication partner `G2` until yielding or asking. Then, the requests are synced: if `G1` asks an element and `G2` provides one, or vice versa, the communication is successful and the handler is invoked recursively on the remainder of both partners.

```
play(G1,G2) :-
  reset(G1,Cont1,Term1),
  ( Cont1 == 0 ->
    true
  ;
    reset(G2,Cont2,Term2),
    sync(Term1,Term2),
    play(Cont1,Cont2)
  ).
sync(ask(X),yield(X)).
sync(yield(X),ask(X)).
```

Flexible communication between both partners is made possible by `play/2`:

```
?- play(sum(S), from_list(1,2,3)).
S = 6.

?- play(sum(S), enum_from_to(1,4))
S = 6.
```

More generally, coroutines can mix `yield/1` and `ask/1` to communicate in two directions. Consider the following mapping and scanning predicates:

```
mapL([],[]).                      scanSum(Acc) :-
mapL([X|Xs],[Y|Ys]) :-              ask(X),
  yield(X),                         NAcc is Acc + X,
  ask(Y),                           yield(NAcc),
  mapL(Xs,Ys).                      scanSum(NAcc).
```

For instance, to compute a list of partial sums $y_j = \sum_{i=0}^{j} x_i$ for a given input list x_i one can use:

```
?- play(mapL([1,2,3,4],L),scanSum(0)).
L = [1,3,6,10].
```

Compare this coroutine-based approach to Sterling and Kirschenbaum's approach of applying techniques to skeletons [148]. The former are much more lightweight and uniform. In contrast, the latter rely on program transformation or meta-interpretation and are more ad-hoc.

Transducers A transducer transforms an iterator of one kind into an iterator of another kind. A transducer communicates with two parties: it asks values from an underlying iterator and uses these to produce other values that it yields to an iteratee.

The doubler/2 predicate is an example of a transducer that doubles the values it receives. Other examples are the mapL/2 and scanSum/1 predicates explained above.

```
doubler :-
  ask(Value),
  NValue is Value * 2,
  yield(NValue),
  doubler.
```

The transduce/2 predicate from Figure 4.5 applies a transducer to an iterator. The predicate runs the given transducer goal until a shift is encountered. The shift binds TermT to either yield(NValue) or to ask(Value). Then, transduce/2 calls a helper predicate. For a yield, the helper predicate actually yields the value and then calls transduce recursively on the continuation. For an ask, the handling is more complicated: the helper predicate runs the iterator goal, and if this goal yields a value, it supplies that value to the transducer goal before running transduce recursively on both continuations.

Here is an example how to use transduce/2:

```
?- play(sum(Sum),transduce(fromList([1,2]),doubler)).
Sum = 6.
```

```
transduce(IG,TG) :-
  reset(TG,ContT,TermT),
  transduce_(TermT,ContT,IG).

transduce_(0,_,_).
transduce_(yield(NValue),ContT,IG)) :-
  yield(NValue),
  transduce(IG,ContT).
transduce_(ask(Value),ContT,IG) :-
  reset(IG,ContI,TermI),
  ( TermI == 0 ->
     true
  ;
     TermI = yield(Value),
     transduce(ContI,ContT)
  ).
```

Figure 4.5: Definition of the `transduce/2` predicate for applying transducers.

Java-style iterators The `init_iterator/2` predicate packages a generator goal in an iterator structure that captures the last yielded element and the generator's continuation. This resembles the `iterator()` method from the Java `Iterable` interface very well. The `next/3` predicate extracts this element and builds the new iterator from the continuation, similar to the Java `next()` method on an iterator.

```
init_iterator(Goal,Iterator) :-
  reset(Goal,Cont,YE),
  ( YE = yield(Element) ->
      Iterator = next(Element,Cont)
  ;
      Iterator = done
  ).

next(next(Element,Cont),Element,Iterator) :-
  init_iterator(Cont,Iterator).
```

As in the examples above, consumers of iterators are independent of the particular generator:

```
sum(Iterator,Acc,Sum) :-
  ( next(Iterator,X,NIterator) ->
            NAcc is Acc + X,
            sum(NIterator,NAcc,Sum)
        ;
            Acc = Sum
        ).
```

and can be hooked up to many different ones:

```
?- init_iterator(fromList([1,2,3]),It), sum(It,0,Sum).
Sum = 6.

?- init_iterator(enumFromTo(1,3),It), sum(It,0,Sum).
Sum = 6.
```

4.2.2 Effect Handlers

Plotkin and Pretnar [110] have recently formulated a particularly insightful class of applications: *effect handlers*. Effect handlers are an elegant way to add many kinds of side-effectful operations to a language and are far less intrusive than monads [99].

We will see that the coroutines from the previous section are in fact all effect handlers; we explained them first because they are more familiar.

Implicit State Passing Figure 4.6 (left) defines an effect handler for an implicit state passing feature. The feature provides two primitive operations: `get/1` for reading the implicit state, and `put/1` for writing it. For instance, the predicate `inc/0` uses these primitives to increment the state.

```
inc :- get(S), S1 is S + 1, put(S1).
```

The effect handler decouples the syntax of the new operations from their semantics. The `put/1` and `get/1` predicates are all syntax and have no semantics; they simply `shift` their own term representation. The semantics is supplied by the handler predicate `run_state/3`. This handler predicate runs a goal and interprets the two primitive operations whenever they are shifted. For the interpretation, `run_state` recursively threads a state. Hence, a minimal example that uses implicit state is:

```
?- run_state(inc,0,S).
S = 1.
```

```
get(S) :- shift(get(S)).
put(S) :- shift(put(S)).

run_state(Goal,Sin,Sout) :-
  reset(Goal,Cont,Command),
  ( Cont == 0 ->
      Sout = Sin
  ; Command = get(S) ->
      S = Sin,
      run_state(Cont,Sin,Sout)
  ; Command = put(S) ->
      run_state(Cont,S,Sout)
  ).
```

```
c(E) :- shift(c(E)).

phrase(Goal,Lin,Lout) :-
  reset(Goal,Cont,Term),
  ( Cont == 0 ->
      Lin = Lout
  ; Term = c(E) ->
      Lin = [E|Lmid],
      phrase(Cont,Lmid,Lout)
  ).
```

Figure 4.6: Effect handler expressing the State monad (left) and effect handler for DCGs (right).

Definite Clause Grammars Figure 4.6 (right) shows a lightweight effect handler for definite clause grammars. They are conventionally defined by program transformation, for which they require special syntax to mark DCG clauses `H --> B` and to mark non-DCG goals {`G`} inside clauses. Our effect handler requires neither. It only introduces one new primitive operation `c(E)` to consume the current element `E` in the implicit list. For instance, the following predicate implements the grammar $(ab)^n$ and returns n.

```
ab(0).
ab(N) :- c(a), c(b), ab(M), N is M + 1.

?- phrase(ab(N),[a,b,a,b],[]).
N = 2.
```

Composing Effect Handlers Effect handlers can easily be made compositional. All it takes is for a handler to propagate unknown operations to the next one in line. For example we can mix the DCG and state features this way.

```
phrase(Goal,Lin,Lout) :-
  reset(Goal,Cont,Term),
  ( Cont == 0   -> ...
  ; Term = c(E) -> ...
  ;
    shift(Term),
    phrase(Cont,Lin,Lout)
  ).
```

```
ab.
ab :- c(a), c(b), inc, ab.

?- run_state(phrase(
      ab,[a,b,a,b],[]
    ),0,S).
S = 2.
```

4.3 Meta-Interpreter Semantics

4.3.1 Direct-Style

We now formalize the delimited continuations feature by extending the vanilla meta-interpreter from Section 2.2 with the `reset/3` and `shift/1` predicates. On purpose, we leave out the if-then-else case and postpone the discussion to Section 4.6.

```
eval(G) :-
  eval(G,Signal),
  ( Signal = shift(Term,Cont) ->
      format('ERROR: Uncaught `shift(~w)\'.\n',[Term]),
      fail
  ;
      true
  ).

eval(shift(Term),Signal) :- !,
  Signal = shift(Term,true).
eval(reset(G,Cont,Term),Signal) :- !,
  eval(G,Signal1),
  ( Signal1 = ok ->
     Cont = 0,
     Term = 0
  ;
     Signal1 = shift(Term,Cont)
  ),
  Signal = ok.
eval((G1,G2),Signal) :- !,
  eval(G1,Signal1),
  ( Signal1 = ok ->
      eval(G2,Signal)
  ;
```

```
        Signal1 = shift(Term,Cont),
        Signal = shift(Term,(Cont,G2))
    ).
  eval(G,Signal) :-
    built_in_predicate(G), !,
    call(G),
    Signal = ok.
  eval(G,Signal) :-
    clause(G,Body),
    eval(Body,Signal).
```

The meta-interpreter extends every goal with an extra output parameter `Signal`. It is instantiated to `ok` when the goal succeeds normally. The base case for this behavior is the `eval/2` clause for built-in predicates.

When a goal's evaluation is abruptly terminated by a call to `shift(Term)` before its continuation `Cont` can be executed, the `Signal` flag is instantiated to `shift(Term,Cont)`. The base case for this behavior is the `eval/2` clause for `shift(Term)`, where the empty continuation is represented by the goal `true`.

The clause for conjunction `(G1,G2)` evaluates the first goal. If it succeeds normally, the conjunction clause proceeds with `G2`. If `G1` is aborted by `shift/1`, then the whole conjunction case is aborted too and `G2` is added to the returned continuation.

The clause for `reset(G,Cont,Term)` evaluates `G` and binds `Cont` and `Term` to `0` when `G` terminates normally; otherwise, they are bound to the returned values.

4.3.2 Continuation-Passing Style

The following continuation-passing style meta-interpreter is an alternative formalization of the delimited continuation semantics. It materializes the call stack as a stack of *continuation frames*. Every frame consists of a list, representing a conjunction of goals.

The evaluation of a conjunction `(G1,G2)` adds the second conjunct `G2` to the front of the current frame's list at the top of the stack. The evaluation of `reset/3` pushes a new frame on top of the stack and `shift/1` pops the top frame from the stack.

```
  eval(G) :-
    eval(G,top([])).

  eval(reset(G,Cont,Term),Conts) :- !,
    eval(G,push([],Cont,Term,Conts)).
```

```
eval(shift(Term),Conts) :- !,
  eval_shift(Conts,Term).
eval(call_continuation(Cont0),Conts) :- !,
  add_conts(Cont0,Conts,NConts),
  eval_continue(NConts).
eval((G1,G2),Conts) :- !,
  add_cont(G2,Conts,NConts),
  eval(G1,NConts).
eval(true,Conts) :- !,
  eval_continue(Conts).
eval(Goal,Conts) :-
  clause(Goal,Body),
  eval(Body,Conts).

eval_continue(top([])).
eval_continue(top([G|Gs]) :-
  eval(G,top(Gs)).
eval_continue(push([],Cont,Term,Conts)) :-
  Cont = 0,
  Term = 0,
  eval_continue(Conts).
eval_continue(push([G|Gs],Cont,Term,Conts) :-
  eval(G,push(Gs,Cont,Term,Conts)).

eval_shift(top(Gs),Term) :-
  format('ERROR: Uncaught `shift(~w)\'.\n',[Term]),
  fail.
eval_shift(push(Cont,C,T,Conts),Term) :-
  C = Cont,
  T = Term,
  eval_continue(Conts).

add_cont(top(Gs),G,top([G|Gs])).
add_cont(push(Gs,Cont,Term,Conts),G,
              push([G|Gs],Cont,Term,Conts)).

add_conts([],Conts,Conts).
add_conts([G|Gs],Conts,NConts) :-
  add_cont(Conts,G,Conts1),
  add_conts(Gs,Conts1,NConts).
```

4.3.3 Program Transformation

By means of the second Futamura projection [50], we obtain a program transformation from the direct-style interpreter.

The general principle of this transformation is that each predicate gets one extra argument, which captures both the term of `shift/1` and the continuation, just as the second argument `Signal` does in the direct-style meta-interpreter. To keep the notation simple, we assume that the goals in the input program have arity zero, but the reader should be able to generalize easily.

We need to consider the transformation of a fact, and of three types of conjunctive clauses: below is at the left side the original form, and at the right the transformed form.

```
a.                       a(ok).

a :-                     a(X) :-
     b,                    b(Y),
     c.                    ( Y == ok ->
                             c(X)
                           ;
                             addcont(Y,c(_),X)
                           ).

d :-                     d(X) :-
  reset(e,Cont,Term),      e(Y),
                           (
                             Y == ok ->
                             Cont = 0, Term = 0
                           ;
                             s(Term,Cont) = Y
                           ),
  g.                       g(X).

h :-                     h(X) :-
  shift(Term), k.          X = s(Term,k(_)).

                         addcont(s(Term,Cont),G,s(Term,(Cont,G))).
```

This transformation works as follows:

1. The extra argument to a fact is `ok`.
2. For a rule consisting of the conjunction of two general body predicates, the extra argument is the extra argument of the second conjunct in case

the execution of the first conjunct was successful, and otherwise a term `s/2` containing the term shifted by the first conjunct and a composite continuation. This composite continuation, defined by the `addcont/3` fact consists of the continuation of the first conjunct, followed by the entire second conjunct. Remember from Section 2.2 that a conjunction of more than two goals is represented by nested binary conjunctions.

3. If the first goal in the rule is a reset, the extra argument of the entire rule is the extra argument to the second conjunct, regardless of the occurrence of a shift in the first conjunct. The difference lies in the bindings made to the extra argument of the first conjunct. In particular, `s(Term,Cont) = Y` binds the entire continuation to `Y`.

4. Although simple, the rule where the first conjunct is a shift, is perhaps the most interesting. The extra argument to the entire rule is a term `s(Term,Cont)` with `Term` the argument passed to the shift-operator, and `Cont` the continuation of that shift within the rule, which is `k`.

In this context, it is also worth giving the implementation of the predicates `call_continuation/1` and `call/1`. call_continuation/1 is defined trivially as:

```
call_continuation((G1,G2)) :- !,
  call_continuation(G1),
  call_continuation(G2).
call_continuation(G) :- call(G).
```

and the transformed version is obtained as defined above. Because the implementation is trivial, in our WAM version the user can just use `call/1` for continuations. After all, continuations are just like any other goal.

For `call/1`, we write the transformed version by hand:

```
call(X,Goal) :-
  arg(1,Goal,X),
  call(Goal).  % the normal call/1
```

It is fairly obvious that the first argument of `call/2` must be the first argument of the called goal.

4.4 Relation to `catch/3` and `throw/1`

The usual way to call `catch/throw` is in combination with each other, so that `catch(Goal,Ball1,Handler)` corresponds to a `throw(Ball2)`.

`catch/throw` has similarities with `reset/shift`, the important differences being:

1. `throw/1` discards both the forward and backtracking continuation up to the matching call to `catch/3`; `shift/1` makes only the forward continuation available to the (reset) handler. In other words: `shift/1` does not delete choicepoints.

2. `throw/1` makes a copy of `Ball2` (let's name it `Ball2Copy`) and then undoes the bindings up to the `catch/3`; `shift/1` does not make a copy of its argument and does not undo bindings.

3. if `Ball1` and `Ball2Copy` do not unify, a matching call to `catch/3` higher up in the execution is searched for; if `Term1` does not unify with `Term2`, this results in failure of the continuation of the `reset/3` goal, and backtracking occurs, potentially into `Goal`.

4. as for `catch/throw`, it is as if the handler is `true`.

The following code shows how `catch/3` and `throw/1` can be implemented with reset/shift:

```
catch(Goal,_Catcher,_Handler) :-
  nb_setval(thrown,nothrow),
  catch1(Goal).
catch(_Goal,Catcher,Handler) :-
  nb_getval(thrown,Term),
  Term = ball(Ball),
  nb_setval(thrown,nothrow),
  (Ball = Catcher ->
    call(Handler)
  ;
    throw(Term)
  ).
```

```
catch1(Goal) :-
  reset(Goal,Cont,Term),
  (Cont == 0 ->
    true  % no ball was thrown
  ;
    !,
    nb_setval(thrown,Term),
    fail
  ).

throw(Ball) :-
  copy_term(Ball,BC),
  shift(ball(BC)).
```

Note that we use the fact that when there is no `shift/1` inside the `Goal` of `reset/3`, `Cont` is unified with the integer 0.

One might wonder whether implementing `reset/shift` with `catch/throw` is possible. One showstopper is that `throw/1` undoes all bindings up to the `catch/3`, while `shift/1` does not. The other is that `throw/1` copies (at least in its ISO compliant mode[2]) while `shift/1` does not make a copy of its argument. Both are related of course.

[2]SWI-Prolog copies the ball just like ISO. It only searches for a catcher with matching ball before copying.

4.5 Implementation

This section presents an implementation of `reset/3` and `shift/1` in the WAM [4, 171], more specifically in the context of hProlog [36].

4.5.1 The hProlog Implementation

There are three main issues in the implementation:

1. the representation of a (delimited) continuation,
2. the change of control involved in `shift/1`, and
3. how to pass the continuation and the argument of `shift/1` to `reset/3`

They are described at the abstract machine level, using the hProlog WAM variant that originates in the XSB implementation [152]: the name of several abstract machine instructions reflects that. Still, the code below should be easily readable to anyone acquainted with the WAM.

Relevant hProlog Implementation Choices hProlog uses a separate environment and choicepoint stack, WAM argument registers are numbered starting at 1, and a free variable (a self-reference) never occurs in an environment. Also, we do not use the WAM instruction names from [4], but adopt the naming convention actually used in hProlog: the instruction names are truncated, yet annotated with their classification. Possible classifications are `t` (for temporary) and `p` (for permanent).

Reset The hProlog code of `reset/3` is shown below on the left; `sysh:asm/1` is a variant of the C `asm` command for generating inline WAM instructions. The corresponding WAM code is on the right: for each instruction, it shows the code address. The `getpvar` instruction places the contents of the given register into the given permanent variable and always continues with the next instruction. The `getpval` instruction unifies the given variable and argument register. Backtracking occurs on failure; on success, the next instruction is executed.

```
reset(Goal,Cont,Term) :-            000 allocate 4
                                    016 getpvar Y2 A3
                                    032 getpvar Y3 A2
  call(Goal),                       048 call call/1  4
  reset_marker,                     080 builtin_reset_marker_0
```

```
    sysh:asm(getpval(Cont,1)),          088 getpval Y3 A1
    sysh:asm(getpval(Term,2)).          104 getpval Y2 A2
                                        120 dealloc_proceed
```

The `builtin_reset_marker_0` serves two roles:

- If no `shift` is executed inside a succeeding `Goal`, execution returns to the `reset_marker`. It is then responsible for putting the default value 0 in the WAM argument registers 1 and 2. The `getpval` instructions subsequently unify these registers with the appropriate arguments of `reset`.
- If a `shift` is executed inside `Goal`, the code for `shift` puts the correct values of `Cont` and `Term` in argument registers 1 and 2. The `reset_marker` then acts as a marker in the stack for `shift`, which returns to the `getpval` right *behind* the `marker` so that the `reset_marker` itself is not executed.

Shift The implementation of `shift/1` — together with that of a helper predicate `get_chunks/3` — is listed below. It performs two tasks: 1) to capture the continuation up to the nearest enclosing `reset/3` in a heap term `Cont`, and 2) to unwind the local stack to pass control back to that `reset/3`.

```
shift(Term) :-
  % 1) capture continuation
  nextEP(first,E,P),
  get_chunks(E,P,L),
  Cont = call_continuation(L),
  % 2) pass control
  sysh:asm(putpval(Cont,1)),
  sysh:asm(putpval(Term,2)),
  unwind_environments.

get_chunks(E,P,L) :-
  ( points_to_reset_marker(P) ->
      L = []
  ;
      get_chunk(E,P,TB),
      L = [TB|Rest],
      nextEP(E,NextE,NextP),
      get_chunks(NextE,NextP,Rest)
  ).
```

1. The first task is handled by the auxiliary predicate `get_chunks(E,P,L)`. It captures the delimited continuation as a list `L` of *continuation chunks* by traversing the local stack and constructing with `get_chunk/3` a continuation chunk for each environment on the way. The structure of such a chunk is explained below.

   The traversal is made possible by the new `nextEP(E,NextE,NextP)` primitive that retrieves both the next environment pointer `NextE` and next continuation pointer `NextP` stored in the given environment `E`. The traversal starts at the current environment, aliased by the atom `first`, and ends at the environment of the `reset/3` call, which is reached when the continuation pointer points to the `reset_marker` (identified by the new primitive `points_to_reset_marker/1`). The current environment `first` is the start of the chain that leads to `builtin_reset_marker`.

   Finally, `shift` wraps the resulting list in the functor of the predicate `call_continuation/1` (shown later) so that it can be directly meta-called.

2. For the second task, `shift` first sets up `Cont` and `Term` in the first and second WAM argument registers (with `putpval` that places the content of the given permanent variable into the given register and then continues with the next instruction) where the two `getpval` instructions at the end of `reset/3` can find them. Then it passes control to `reset/3` with the new primitive `unwind_environments/0`. This primitive unwinds the environment stack up to the environment of the first enclosing `reset/3` call in a similar way as `get_chunks/3` traverses it. Then it sets the WAM E register[3] to point to this environment, and the WAM P register (the "next instruction" pointer) to point to just *after* the `builtin_reset_marker_0` instruction so that it does not get executed.

   Note that `unwind_environments/0` is careful not to upset the WAM argument registers set up by `shift/1` and that `unwind_environments/0` leaves the choice points unchanged, so that later backtracking could bring the execution back in the scope of the `reset/3` goal. This is compatible with the meta-interpreter semantics in Section 4.3.

Continuation Chunks The predicate `get_chunk(E,P,TB)` builds a continuation chunk in its `TB` argument. Such a chunk captures in a heap term all the necessary information to resume the unexecuted remainder (the tail) of a

[3]The WAM E global register keeps the address of the latest environment on top of the stack.

predicate body. This information consists of 1) the code to execute, and 2) the data to execute with.

The first part is easy: the code to execute starts at `P`. The second part is more involved: the code may refer to data in both argument registers and environment variables. Fortunately, it is a WAM invariant that no argument registers are live at a code point, like `P`, right after a call. Hence, we only need to capture the set of live environment variables (LEV) in the environment `E`.

In hProlog, just as in Yap [130] and possibly other systems, the LEV set at a continuation point P is determined at compile-time, and can at runtime be retrieved from P. This basically follows the ideas of Branquart and Lewi [14], but there are many ways to implement them. For hProlog, a bitmap of fixed size describing which slots are live at that point in the execution of a clause, is an argument of each `call` instruction. If this fixed bitmap size is too small to represent the active Yvars[4], it points to a piece of memory after the code for the predicate (but still in the code zone), where there is enough space. The loader takes care of this. The bitmaps are not shown in the WAM code above. Hence, in summary, the term built by `get_chunk/3` in its `TB` argument is `$cont$(P,LEV)`.

The dual of `get_chunk/3` is `call_chunk($cont$(P,LEV))`: it builds a new environment on the local stack from the continuation chunk. The size of the new environment can be found in the call-instruction right before `P`, and the live variables LEV can be filled in the appropriate slots of the environment by using the position information provided by `P`.

The predicate `call_continuation/1` extends `call_chunk/1` to a list of continuation chunks:

```
call_continuation([]).
call_continuation([TB|Rest]) :-
  call_chunk(TB),
  call_continuation(Rest).
```

Example 4. *The example of Fig. 4.7 on the facing page provides more insight in the representation of a continuation. The example shows Prolog code on the left, and the corresponding hProlog WAM instructions on the right. The* ***putpvar Yi Aj*** *initialises the i-th stack variable in the current environment to unbound and also lets* ***Aj*** *point to it; the meaning of the rest of the other instructions should be clear from their opcode. The continuation captured in the example consists of two chunks:*

[4]Permanent variables are traditionally denoted as `Yi` and temporary variables as `Xi`, hence the terms `YVars` and `XVars`.

```
p :-                                 176  allocate 4
                                     192  putpvar Y2 A2
                                     208  putpvar Y3 A3
                                     224  put_atom A1 q
  reset(q,Cont,Term),                248  call reset/3  4
                                     280  putpval Y2 A1
  writeln(Cont),                     296  call writeln/1  4
                                     328  putpval Y3 A1
  writeln(Term),                     344  call writeln/1  4
                                     376  putpval Y2 A1
  call(Cont).                        392  deallex call/1

q :-                                 584  allocate 2
  r,                                 600  call r/0  2
                                     632  put_atom A1 endq
  writeln(endq).                     656  deallex writeln/1

r :-                                 680  allocate 3
                                     696  putpvar Y2 A1
  foo(Y),                            712  call foo/1  3
                                     744  put_atom A1 shiftterm
  shift(shiftterm),                  768  call shift/1  3
                                     800  putpval Y2 A1
  writeln(Y).                        816  deallex writeln/1

foo(bla(_)).

?- p.
call_continuation([$cont$(800,[bla(_165)]),$cont$(632,[])])
shiftterm
bla(_165)
endq
```

Figure 4.7: Example Prolog and WAM code.

1. *`$cont$(800,[bla(_165)])` mentions: 1) the address (800) of the first instruction following the `shift/1` goal, and 2) the one active Yvar (Y2) of `r/0` at that point. The existence of the latter is derived from the preceding `call 3` instruction (768): 3 is the length of the environment at that point (E,CP and Y2). Hence, the (dereferenced) reference to Y2 is copied in the list; the term is not copied with `copy_term/2`. During*

`call_continuation/1`, this reference is put in the appropriate environment slot in a new environment.

2. *`$cont$(632,[])` points to the instruction 632 right after the call to `r/0` in the body of `q/0`. Since that clause has no permanent variables, the LEV is empty.*

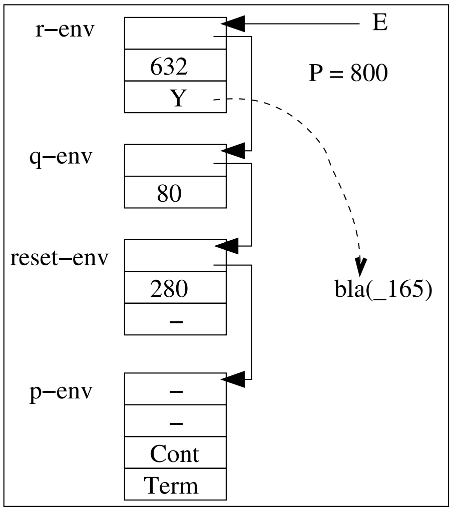


Figure 4.8: The local stack and the `E` and `P` pointers at the moment `shift/1` is called.

Figure 4.8 completes the example. It shows the environments of the activations of `p/0`, `reset/3`, `q/0` and `r/0`, at the moment that `shift/1` is constructing the continuation. The reader can check the values of all code pointers in the figure. Note that the term `bla(_165)` resides on the heap. Some environment entries are not shown as they are not relevant here.

4.5.2 ZIP Implementation

As a proof of concept, Wielemaker has implemented delimited continuations in SWI-Prolog [174]: it is based on the simpler ZIP [11]. This leads to the following differences:

1. Since an SWI-frame — similar to a WAM environment — records which predicate it belongs to, the `reset_marker` is not needed for finding the corresponding reset activation.
2. SWI-Prolog uses code scanning [172] to determine the live frame variables. Code scanning is performed once while constructing the delimited continuation, and avoided while reconstructing the frame with its correct slots. The SWI-Prolog analogue of the hProlog `$cont$/2` term is a `$cont$/3` term with as first argument a clause reference for keeping the clause corresponding to the chunk alive long enough. The second argument is a program counter similar to the first argument of the hProlog `$cont$/2`. The third argument is a list of `Offset-Term` pairs identifying the frame slot and the value of the live frame variables.

The interaction between the two implementations lead to the exploration of three alternatives:

Reinstalling the whole continuation in one go at `call_continuation`. This can lead to repeatedly scanning (a copy of) the same continuation, and sometimes leads to changing the runtime complexity of the program from linear to quadratic.

Leaving continuations on the local stack instead of copying them to the heap. The obvious advantages are: a) creating the continuation in `shift/1` is cheap, and b) no data is created on the heap that must be garbage collected. After implementing this alternative, we made the following observations:

1. It can lead to the same complexity increase as installing the whole continuation in one go, unless one introduces an extra WAM-register or global scoped variable that remembers the environment with the CP that points to the marker: in this way, no scanning of the stack is needed to find the limits of the continuation.
2. The recursive reactivation of a delimited continuation is no longer possible.
3. To make the approach work, one needs to implement a local stack garbage collector. We have refrained from doing so.
4. Heap garbage collection needs some modification to scan the live continuations.
5. In case the delimited continuation is small, in the order of a few frames, the approach yields no performance advantage.

6. In SWI-Prolog, there are two advantages: a) the clause reference in the `$cont$/3` term is no longer needed. b) It does no longer interfere with SWI-Prolog's meta-calling of control structures by means of a special temporary clause on the local stack.

In conclusion, keeping continuations on the stack is feasible, but whether it is desirable depends on the design of the virtual machine.

Non-selective Environment Saving We have also tried saving all environment slots in the `$cont$/2` structure, rather than selectively saving only the live variables. As a consequence, heap garbage collection becomes slightly more complicated but can still be accurate. Whether this approach saves time and space depends theoretically on the ratio between the sizes of the environment and the LEV, but in practice, it turns out to be only slightly more efficient, and we have abandoned it.

4.6 Semantic Intricacies

4.6.1 Cut and If-then-else

WAM-based implementations usually store information on how far to cut in the environment. This distance may no longer be appropriate when the cut is captured in and executed as part of a delimited continuation. Special care must be taken not to inadvertently cut unrelated choicepoints.

Two cases are common: a cut not appearing as the first goal in a clause[5], and if-then-else with a non-simple test. Both are exemplified below:

```
                                    p :- savecp(B), p(B).
p :- a, !, body1.                   p(B) :- a, cutto(B),
                                                body1.
p :- body2.                         p(_) :- body2.

q :- b, (c -> d ; e), f.            q :- b, savecp(B),
                                         (c, cutto(B), d ; e),
                                         f.
```

On the left, the user-written code is shown, on the right the equivalent code using the non-ISO predicates `savecp/1` and `cutto/1`. The predicate `savecp/1`

[5]If cut is the first goal in the clause, the relevant choicepoint can be detected in a different way.

unifies the current choice point pointer `B` with its argument. Later, `cutto/1` uses this pointer to cut up to the correct choice point. In both cases, `B` is a permanent variable that resides in the environment of that clause activation. The above is hProlog specific (and actually follows the XSB implementation), but a similar thing happens in many other Prolog implementations.

Since a continuation saves the active permanent variables, it is possible that the value of such a `B` is captured. The situation in which the `cutto(B)` goal is later executed in the delimited continuation must be treated carefully: the choice point `B` refers to might not exist any longer, and even if it does, it would be strange to cut all choicepoints up to `B` away, as there could be new choicepoints that are related to the continuation of the `reset/3` goal.

A choice needs to be made, and we have decided that a cut in a captured continuation can only cut up to, but not including, the youngest choicepoint before invoking the continuation. The two examples in Fig. 4.9 on the next page show this.

In the implementation, one needs to take care that any active permanent variable whose value represents a choice point, is replaced by the appropriate choice point on executing a continuation containing that cut.

4.6.2 Re-activation

A continuation can be called more than once, so the question arises: what variables do those different activations share? In our approach, this depends on which optimizations are performed.

```
a :- shift(x), nonvar(X), X = 1.
a :- X = X, shift(x), nonvar(X), X = 1.
```

E.g. in the first clause the variable `X` is not live at the moment `shift/1` is called. Hence, the variable `X` is not shared between different invocations of the continuation. However, in the second clause sharing depends on whether `X=X` is optimized away or not. The meta-interpreter does not have this problem, and this is the only point where the low-level implementation differs from the meta-interpreter. Here is a more detailed example:

```
p0 :-                                     ?- p0, fail.
  reset(q0,Cont,Term),                    q_1
  writeln(Term),                          fromq_1
  call(Cont).                             endq_1
                                          q_2
q0 :-                                     fromq_2
  writeln(q_1),                           endq_2
  shift(fromq_1), !,
  writeln(endq_1).

q0 :-
  writeln(q_2),
  shift(fromq_2),
  writeln(endq_2).

p1 :-                                     ?- p1, fail.
  reset(q1,Cont,Term),                    q_1
  writeln(Term),                          fromq_1
  call(Cont).                             endq_1
                                          q_2
q1 :-                                     fromq_2
  (                                       endq_2
    writeln(q_1), shift(fromq_1) ->
    writeln(endq_1)
  ;
    writeln(q_2), shift(fromq_2),
    writeln(endq_2)
  ).
```

Figure 4.9: A cut in a captured continuation can only cut up to the youngest choicepoint before invoking the continuation: example situations.

```
calltwice1 :-
  reset(f1,Cont,_),
  writeln(Cont),
  call(Cont),
  call(Cont).

f1 :-
  foo(Y),
  shift(ignored),
  writeln('Y' = Y),
  Y = 1.

foo(_).

?- calltwice1.
[$cont$(787800,[_263])]
Y = _263
Y = 1
```

```
calltwice2 :-
  reset(f2,Cont,_),
  writeln(Cont),
  call(Cont),
  call(Cont).

f2 :-
  shift(ignored),
  writeln('Y' = Y),
  Y = 1.

?- calltwice2.
[$cont$(788000,[])]
Y = _282
Y = _300
```

In both pieces of code, `Y` is a permanent variable. However, in `f1/0`, `Y` is initialized before the call to `shift/1`, so it appears in the continuation (one can see that the variable `_263` occurs in the output twice), while in `f2/0`, `Y` is initialized after the call to `shift/1`: so the initialization of `Y` in the second case happens in the continuation, every time the continuation is activated. This explains the output in both cases as well.

One could argue that the WAM optimization which initializes variables as late as possible is no good in this context, and it makes the results dependent on other optimizations (e.g. inline the call `foo(Y)` to `Y = _` and then remove that unification as it has no effect). So one cannot rely on a particular sharing behavior of multiple invocations of the same continuation within the WAM. In the case of BinProlog as well, the exact form of a continuation can depend on optimizations, or the particularities of the binarizing transformation.

4.6.3 Nesting Catch/Throw and Reset/Shift

ISO Prolog has catch/throw as a scoped construct. Reset/shift is also scoped, so we must understand the interaction between both.

As long as the two constructs are properly nested, the resulting behavior is relatively easy to predict: the inner nested construct does not really interfere with the outer one.

When the two constructs are not properly nested, a little more thought is needed. The first case is exemplified by the code below: it shows a continuation containing a throw/1. This continuation is picked up by reset/3 and is subsequently called. The call to throw/1 is no longer in the scope of the initially corresponding catch/3 goal, so the throw remains uncaught.

```
p :-
  reset(q,Cont,Term),
  writeln(Term),
  call_continuation(Cont).

q :- catch(r,Ball,writeln(Ball)).

r :- shift(rterm), throw(rball).

?- p.
rterm
Uncaught exception(rball)
```

From the language design point of view, there might be different options to explore. We are satisfied because the current implementation behaves reasonably.

The other improper nesting is shown below:

```
a :-
  catch(b,Ball,writeln(Ball)).

b :-
  reset(c,Cont,Term),
  writeln(Term),
  call_continuation(Cont).

c :-
  throw(ballfromc),
  shift(notseen).

?- a.
ballfromc
```

Since throw/1 discards the forward continuation, it is clear that this situation does not pose any problems.

4.6.4 Shiftless Resets and Resetless Shifts

For the implementation in hProlog, we have chosen to unify the Cont and Term arguments of reset/3 with zero in the absence of shift, as this seems more useful than other options, and to have the toplevel catch shifts outside of a reset. It implies that in the code for `get_chunks/3` in Section 4.5 the test `points_to_reset_marker(P)` eventually succeeds. As such, there is no risk to cross the boundaries of the environment stack. Alternative semantics are easy to implement as a variation on the basic schema.

4.7 Related Work

4.7.1 BinProlog and Continuations

BinProlog [157] is based on explicit continuation passing: clauses are transformed to a binary form and carry the continuation as a first class citizen[6] in an extra argument. To be more explicit, binarization of the fact/clause/query on the left results internally in the constructs on the right:

```
a.                                     a(Cont) :- call(Cont).

a :- b, c.                             a(Cont) :- b(c(Cont)).

?- q.                                  ?- q(true).
```

While the continuation is normally invisible to the user, [158] describes how (still based on program transformation) the user can have access to the continuation, and then manipulate it. The special notation for that is by allowing multi-headed clauses of the form

```
p(foo), bar :- body.
```

whose meaning is: "if p/1 is called with first argument `foo`, and a continuation starting with `bar`, then execute body". The above clause is binarized using the built-in BinProlog predicate `strip_cont/3`, which splits a continuation — a conjunction of goals, but in binarized nested form — into its first goal and the rest of the continuation. As [158] says: `strip_cont/3` acts as if defined by

```
strip_cont(f(X1,...,Xn,Cont), f(X1,...,Xn), Cont).
```

for every `f/(n+1)`. `strip_cont/3` acts in a similar way to our `get_chunk/3` from Section 4.5.

Based on `strip_cont/3` and the implementation of catch and throw in BinProlog, we have built an implementation of `reset/3` and `shift/1` as follows:

[6]Unfortunately, being first class means it is an infinite term as soon as it is used explicitly.

```
reset(Goal,Cont,Term) :-
  call(Goal),
  marker(Cont,Term).

shift(Term) :-
  Marker = marker(Cs,Term),
  get_cont(Cont),
  consume_cont(Marker,(_,_,Cs),Cont,NewCont),
  call_cont(NewCont).

consume_cont(Marker,Gs,Cont,LastCont):-
  strip_cont(Cont,Goal,NextCont),
  (Goal = Marker ->
      LastCont = NextCont,
      Gs = true
  ;
      Gs=(Goal,OtherGs),
      consume_cont(Marker,OtherGs,NextCont,LastCont)
  ).

marker(0,0).
```

The predicate `consume_cont/4`, is similar to our `get_chunks/3`: it keeps peeling off the first goal of a continuation until the marker is found in the continuation. The fact `marker(0,0)` serves to catch the absence of a `shift/1` inside `Goal`. We used this code for benchmarking.

4.7.2 BinProlog and Logic Engines

BinProlog also provides a coroutine-like feature: *logic engines* [157, 33]. A logic engine is essentially an independent Prolog environment that can be queried for successive answers to a goal.

In spirit, the logic engines approach and our coroutines are quite similar: to consider concurrency decoupled from multi-threading. However, our coroutines are more lightweight as they live in the same engine and, e.g., share the same heap and choicepoint stack. Moreover, in our approach the interfaces are more symmetric: coroutines receive data with `ask/1` that was sent by another coroutine with `yield/1` and vice versa. Logic engines receive data with `from_engine/1` that was sent by `to_engine/2` and return data with `return/1` that was requested by `get/2`.

4.7.3 Conventional Prolog Coroutines

Various coroutine-like features have been proposed in the context of Prolog for implementing alternative execution mechanisms such as constraint logic programming: `freeze/2`, `block/1` declarations, ... Nowadays most of these are based on a single primitive concept: attributed variables [66, 88, 104, 35]. These attributed variables combine three useful aspects in one feature:

1. The ability to associate updateable data with a variable using the predicates `get_attr/3` and `put_attr/3`,
2. The ability to associate a goal, a call to the user-defined predicate `attr_unify_hook/2`, with the instantiation of the variable (the coroutine), and
3. The implicit and automatic invocation of the coroutine goal when the variable becomes instantiated or aliased to another attributed variable.

A significant difference with delimited continuations is that attributed variables allow only one way of transferring control: instantiation of a variable. In contrast, the reset/shift predicate pair offer explicit transfer of control.

Implementation-wise and conceptually, the attributed variable coroutines are not based on continuations. Typically, either a routine is triggered only once, or the same modified goal is triggered over and over again. Any behavior similar to continuations must be programmed explicitly.

In summary: apart from the common name "coroutine", attributed variable coroutines share very little with coroutines based on delimited continuations.

4.7.4 Environments on the Heap

In [37], Demoen and Nguyen describe an implementation of coroutining in which environments of certain declared predicates are put on the heap instead of on the local stack. The primitives `yield/1`, `leave/0` and `resume/1` proposed in that paper can be easily implemented with the constructs of the current paper. Figure 4.10 shows an action rules example from Demoen and Nguyen's paper, while Figure 4.11 shows the same example expressed with the constructs from this paper.

Without going in too much detail, it is fairly clear that our reset/shift are more general, and therefore not so efficient as their mechanism. However, the latter interferes more with other parts of the implementation like stack

```
:- suspension(foo/3).                ?- p.
                                     first(X,Y)
foo(X,Y,SuspTerm) :-                 next(1,Y)
  writeln(first(X,Y)),               next(1,2)
  yield(SuspTerm),
  writeln(next(X,Y)),
  leave.

p :-
  foo(X,Y,SuspTerm),
  X = 1,
  resume(SuspTerm),
  Y = 2,
  resume(SuspTerm).
```

Figure 4.10: Original action rules example.

```
foo(X,Y) :-
  writeln(first(X,Y)),
  shift(SuspTerm),
  writeln(next(X,Y)).

p :-
  reset(foo(X,Y),Cont,SuspTerm),
  X = 1,
  call(Cont),
  Y = 2,
  call(Cont).
```

Figure 4.11: Action rules expressed using delimited control.

management, garbage collection, ... and is therefore perhaps not so attractive. Furthermore, much of the performance was achieved by implementing recurring patterns as low-level WAM instructions. Future work on the implementation might lead to a unified implementation which uses the best of both approaches.

4.7.5 Caml-based Languages

Masuko and Asai [94] describe the implementation of delimited continuations in the typed functional language MiniCaml. In later work [95], they present a direct implementation in the Caml Light system, a lightweight implementation of the Caml language. In contrast to MiniCaml, Caml Light is expressive enough to define actually interesting programs. Kiselyov [80] describes a library implementation for OCaml.

4.7.6 Experimental Languages

Eff [112] is a functional language where effects handlers are first-class citizens. The language is mainly experimental. Frank is another experimental language with typed algebraic effects, provided as a Haskell library [96].

4.7.7 Coroutines in Haskell

Our Prolog meta-interpreter closely resembles James and Sabry's implementation in terms of the coroutine monad [70]. The Haskell code for this implementation, from which we have omitted a few parts that may obfuscate the connection to our setting, is:

```
-- The coroutine monad

data Yield t a = Return a | Yield a (Yield t a)

instance Monad (Yield t) where
  return x              = Return x
  (Return x)     >>= f  = f x
  (Yield t cont) >>= f  = Yield t (cont >>= f)

yield :: t -> Yield t ()
yield t = Yield t (return ())

-- The shift/reset implementation
```

```
shift :: t -> Yield t ()
shift t  = yield t

reset :: Yield t a -> Yield t (Either (t, Yield t a) a)
reset (Return x)      = return (Right x)
reset (Yield t cont)  = return (Left (t,cont))
```

The Haskell constructors `Return` and `Yield` in this code correspond to the `Signal` parameter in the meta-interpreter from Section 4.3. Additionally, the definition of the monadic bind `(>>=)` closely corresponds to the meta interpreter's treatment of conjunction.

James and Sabry are not the first to study the coroutine monad. Different variants of the coroutine monad (transformer) [10] have been studied under different names: resumption monad [107], free monad [6] and step monad [71].

4.7.8 Coroutines in Mainstream Languages

Today, many mainstream languages like C♯, Ruby, JavaScript and Python, have some variant of a yield, although the operator is not widely described as a delimited continuation operator [2, 1, 102, 160]. Moreover expressivity greatly differs from language to language.

Statements vs. Expressions One distinguishing characteristic is whether the yield is an expression or a statement [70]. If yield is used as an expression, it means the iterator takes input from its calling context. This is possible in Ruby and Python 2.5. In earlier versions of Python, yield was a statement, as in Javascript 1.7 and C♯.

The continuations in our Prolog setting are essentially goals, which have a statement-like quality. However, as we have shown with iteratees, they are nevertheless able to take input from their context.

First-Class Iterators Another characteristic is whether iterators are first-class values. It is the case in *C♯*, although iterators are mostly used in combination with a foreach loop, as well as in JavaScript and Python. A first-class iterator is more flexible and it is easier to work with multiple iterators at the same time.

As we have shown, our iterators are first-class values in Prolog. They can be passed around freely.

4.8 Performance Evaluation

Although raw performance is clearly not the focus of delimited control, and a detailed evaluation is not necessary for the rest of this thesis, it is customary to include a short discussion of the performance. For more detailed benchmark results, we refer the interested reader to our technical report [40].

To assess the quality of our native implementation approach, we compare it to two other approaches for implementing delimited continuations:

- The transformation-based approach of Subsection 4.3.3. It adds a signal parameter to every predicate that is checked at every conjunction.
- The binarization approach uses the internal continuation-passing representation of Prolog clauses in BinProlog [157] and implements delimited continuations using the BinProlog built-ins.

The native and transformation approaches were implemented in hProlog and SWI-Prolog. It only makes sense to use the binarization approach in BinProlog.

We compare the three implementation approaches on two artificial benchmarks: (1) shift shifts a delimited continuation, and (2) exec calls a previously shifted continuation. We use three different sizes of continuations: 5,000, 10,000 and 20,000 chunks.

Table 4.1 shows the timing results (in milliseconds) obtained on an Intel Core2 Duo Processor T8100 2.10. Garbage collection times (only in SWI-Prolog) were not included, and the timings of empty loops were subtracted.

The shift benchmark in the upper half of the table shows that the native hProlog implementation is about 2.5 times faster than the transformed hProlog implementation. This shows that in hProlog the native implementation effort payed off. This is not the case in SWI-Prolog, partly because of the code scanning for constructing the LEV's, that makes the implementation more involved in the ZIP, and also because of other implementation choices made in SWI-Prolog. BinProlog's binarization does not exhibit an advantage compared to hProlog's transformation-based and native implementations. It is even outperformed by transformation in SWI-Prolog. Overall, we see that all implementations scale roughly linear with the size of the continuation, as expected.

The lower half of the table shows the exec benchmark which measures the time to execute the delimited continuation. This is contrasted — in brackets — with the time to meta-call a conjunction of equivalent goals. The same pattern shows here: SWI-Prolog performs better in transformed mode, while

	Native		Transformed		Binarization
Depth	hProlog	SWI-Prolog	hProlog	SWI-Prolog	BinProlog
shift					
5,000	64	1,965	164	505	1,120
10,000	128	3,950	328	1,028	2,230
20,000	268	8,388	664	2,037	4,450
exec					
5,000	248 (398)	1,951 (1,137)	480 (398)	1,415 (1,137)	260 (270)
10,000	492 (796)	3,886 (2,283)	964 (796)	2,847 (2,283)	530 (550)
20,000	992 (1,586)	7,780 (4,390)	1,932 (1,586)	5,688 (4,390)	1,040 (1,100)

Table 4.1: Benchmark results for shifting (top) and calling (bottom) continuations (ms).

hProlog performs better in native mode. hProlog even executes its native continuations faster than meta-calling equivalent conjunctions, but note that in hProlog and SWI-Prolog, meta-call suffers a performance penalty because of the ISO Prolog semantics. Calling continuations in BinProlog is almost as fast as in hProlog and on par with BinProlog's meta-call.

In summary, a native implementation of delimited continuations in the WAM is worthwhile. This does not seem true in the ZIP, or at least not within the overall design of SWI-Prolog.

4.9 Conclusion

This chapter has introduced a design of delimited continuations for Prolog that enables many useful applications. Alongside this design, it has described a complimentary implementation of the reset and shift operators in the WAM, which is to our knowledge the first implementation in a non-continuation passing style Prolog virtual machine. The implementation is lightweight, because it is independent of most of the rest of the system, and its interactions with other parts of the system were shown. The performance accommodates the applications: even in the very fast hProlog system, performance does not seem an impediment to using the enhanced expressivity.

We have nonetheless observed that the performance somewhat degrades if the delimited control operators are used to define deeply nested effect handlers. There is ongoing work by Hany Saleh [60] to develop a general technique to automatically collapse these nested handlers into a single monolithic one. His insights may lead to further improvements to the implementation.

References

This chapter is based on our article on Delimited Control and the companion technical report:

- Tom Schrijvers, Bart Demoen, Benoit Desouter, and Jan Wielemaker. Delimited Continuations for Prolog. Theory and Practice of Logic Programming, 2013. Proceedings of the 29th International Conference on Logic Programming (ICLP).

- Tom Schrijvers, Bart Demoen, and Benoit Desouter. Delimited Continuations in Prolog: Semantics, Use and Implementation in the WAM. Report CW 631 of KU Leuven, Department of Computer Science, 2013.

Benoit Desouter's contribution lies in experimentation with the effect handlers presented in this chapter and the initial performance assessment of the direct-style meta-interpreter as well as the native implementation. Also, Benoit investigated the relationship to coroutines in mainstream languages. Benoit assisted in writing both the technical report and article.

Chapter 5

A Functional Specification of Modular Search

5.1 Introduction

In Chapter 3, we have discussed TOR that constitutes the state of the art in separating the control mechanism from the problem logic while remaining faithful to Prolog's execution model. Other solutions offer greater flexibility but abandon Prolog's execution model altogether. TOR is a light-weight library-based approach that is easily portable to different Prolog systems. However, it suffers from a major deficiency: it lacks proper semantic basis, and so requires intimate knowledge of the implementation in order to be understood.

We resolve this deficiency by borrowing techniques from the world of functional programming—*monads* and *effect handlers*—to guide the design of a library that is both modular and based on sound principles. Indeed, exploiting the synergy between the functional programming and logical programming paradigms is essential for this work. The cross-pollination of ideas from both fields solves problems that are otherwise intractable. In this chapter we champion such multi-disciplinary work and present the full development of our solution from its functional specification in Haskell to logic programming implementation in Prolog, in order to bring both communities closer together.

Monads [170] have firmly established their place in functional programming as a practical solution to modelling computations that involve side effects and that have a notion of sequentiality. Instances of monads are abound, and much work has been done on reasoning about monadic programs in terms of the monad's implementation details [68], where the concrete instance is of interest.

An emerging approach is to treat monads as an interface, whose laws form a specification [51], which is closer to the formulation of universal algebras as Lawvere theories [87]. Here the implementation is hidden and the monad is *opaque* or abstract. Programs cannot exploit the monad's implementation details, but are restricted to the exposed interface.

In this chapter we develop a technique to extend the opaque monad that represents Prolog, in order to reason carefully about nondeterminism and heuristics. An important ingredient of our solution are *effect handlers* [110, 96, 9, 83, 13, 76]. They provide a new approach to defining monads that cleanly separates syntax from semantics, where a syntax tree is first built, and then interpreted in a semantic domain using a *handler*. We combine these effect handlers with the free monad transformer to model and extend an opaque effectful language expressed in monadic style.

Our technical contributions are as follows:

- We show how to obtain modular search heuristics by means of effect handlers, and, in particular, define an unusual *entwine* handler that allows us to express search heuristics as archetypal search trees much like TOR's `merge/2` combinator.

- We formulate a functional model that takes the feature-rich nature of modern Prolog systems into account. Furthermore, we use effect handlers and the free monad transformer to minimize the footprint of both the model and the final solution.

- We explain carefully how to transfer our functional solution for modular search heuristics to Prolog. This transfer involves a non-trivial isomorphism between the free monad transformer and the delimited continuation monad transformer.

5.2 The Challenge

Recall from Chapter 3 the standard practice to apply a search heuristic to a problem: it really means modifying the program to embody a different tree. For example, the `queens/2` program is in Figure 5.1 on the left. On the right is a modified version with the addition of a depth bound. Unfortunately,

```
queens(N, Qs) :-                queens2(N, Qs, DB) :-
  findall(C,between(1,N,C),L),    findall(C,between(1,N,C),L),
  go(L,N,[],Qs).                  go(L,N,[],Qs, DB).

go([],N,Qs,Qs).                 go([],N,Qs,Qs, _).
go([X|Xs],N,Acc,Qs) :-          go([X|Xs],N,Acc,Qs, DB) :-
  select(Y,[X|Xs],Ys),            select(Y,[X|Xs],Ys, DB, NDB),
  noThreat(Acc,N,1),              noThreat(Acc,N,1),
  go(Ys,N,[Y|Acc],Qs).            go(Ys,N,[Y|Acc],Qs, NDB).

select(Y,[X|Xs],Ys) :-          select(Y,[X|Xs],Ys, DB, NDB) :-
  (  Y = X,                       DB > 0,
     Ys = Xs                      (  Y = X,
  ;  Ys = [X|Zs],                    Ys = Xs,
     select(Y,Xs,Zs)                 NDB is DB - 1
  ).                              ;  Ys = [X|Zs],
                                     DB1 is DB - 1,
                                     select(Y,Xs,Zs, DB1, NDB)
                                  ).
noThreat([],_,_).
noThreat([M|Ms],R,C) :-
  abs(M-R) =\= C, NC is C + 1, noThreat(Ms,R,NC).
```

Figure 5.1: The n-queens search problem: plain (left) and with depth bound (right, with unmodified code in light gray).

there are obvious problems with this approach to search heuristics. Since the heuristic's code is entangled with the problem's code, it is hard to modularly reuse either. Furthermore, this entanglement encourages an error-prone and labour-intensive copy-paste-modify approach. What we really want is a modular approach where problems (i.e., the logic) and heuristics (i.e., the control) can be defined separately and combined effortlessly.

The Tor approach constitutes the current state of the art in solving this problem. With Tor the depth-bounded search heuristic is expressed as an independent predicate `dbs/2` and applied to `queens/2` as follows.

```
?- search(dbs(10,queens(8,Qs))).
```

The approach is based on using a hookable disjunction `tor/2` in `queens/2` instead of Prolog's regular disjunction `(;)/2`. There are two hooks: one for the left branch and one for the right branch of the disjunction. The `dbs/2` heuristic influences `queen/2`'s search by installing appropriate call-backs in the hooks.

However, the hook-based approach lacks proper semantic grounding. As a consequence, the approach lacks elegant algebraic properties that make its use predictable, in other words, its use requires intimate knowledge of the implementation.

In this chapter we aim for a more general and elegant solution that is based on proper semantic foundations. For this purpose we start from a functional model of the problem in Haskell and apply established techniques to solve the problem, then subsequently derive a practical implementation for Prolog.

5.3 From Prolog to Haskell

In this section we present the functional model of Prolog that will drive our developments. This already presents a first dilemma: modern Prolog systems provide a large set of primitive operations, the so-called *builtins*. Incorporating all of these in our model would be both extremely tedious and onerous. However, we also do not want to oversimplify our model and run the risk of obtaining results that do not work in actual Prolog systems.

We solve this dilemma with a standard functional programming technique, *abstract types*. Instead of specifying a concrete representation for Prolog computations, we assume the existence of an abstract type:

data *Prolog a*

Values of this type represent Prolog computations, also known as *goals*. As a small admission to functional programming, we will assume that goals have a return value which is reflected in the type variable a. Goals that return values of a particular type instantiate a accordingly. Proper Prolog goals are of type *Prolog* (): they do not return any value of interest, which is modelled by instantiating a with the unit type ().

We will assume that there are numerous ways to construct values of type *Prolog a*. However, we deliberately do not attempt to model them all and make no assumptions about how a Prolog goal is constructed. For our own purposes, we restrict our vocabulary to no more than four constructors (two primitives and two combinators) to build new goals.

The first two of these can be summarized by saying that *Prolog* has an instance of the monad class:

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
instance Monad Prolog
```

This means that *Prolog* supports a sequential composition operator ($\gg\!=$) and matching unit operator *return*. This notion of sequentiality closely corresponds to Prolog's conjunction operation `p , q`, which is satisfied when a solution exists for both `p` and `q`. In this setting order matters, and expressions are executed from left to right. More specifically `p , q` corresponds to $p \gg q$, which is an instance of $\gg\!=$ where the return value of p is of no interest to q.

$$p \gg q = p \gg\!= (\lambda x \to q)$$

A computation can end in one of two ways: either the search for a solution ends in success where the result is `true`, or in failure in which case the result is `false`. The behaviour of `true` interacts with the conjunction operator in the same way as *return* () with ($\gg$), as described by the left and right unit laws of a monad:

$$\mathit{return}\ () \gg p = p = p \gg \mathit{return}\ () \tag{5.1}$$

Given this relationship, we identify `true` with *return* ().

To model `false`, we introduce the operation

$$\mathit{fail} :: \mathit{Prolog}\ a$$

This operation comes equipped with the *left-zero* law, which dictates how *fail* interacts with the monadic bind:

$$\mathit{fail} \gg q = \mathit{fail} \tag{5.2}$$

Haskell	Prolog
$\mathit{return}\ ()$	`true`
fail	`false`
$p \gg q$	`p , q`
$p \mid\mid\mid q$	`p ; q`

Table 5.1: Prolog model in Haskell.

This is a perfect fit for `false` since in the setting of Prolog there is no right-zero of conjunction: side-effects performed before failure cannot be undone in general.

To model the disjunction `p ; q`, where either `p` or `q` must be satisfied, we introduce the operation:

$$(\mid\mid\mid) :: \mathit{Prolog}\ a \to \mathit{Prolog}\ a \to \mathit{Prolog}\ a$$

that satisfies the *left-distributivity* property:

$$(m \mid\mid\mid n) \gg\!= f = (m \gg\!= f) \mid\mid\mid (n \gg\!= f) \tag{5.3}$$

Moreover, $\langle \mathit{Prolog}\ a, (\mid\mid\mid), \mathit{fail} \rangle$ forms a monoid, where the following laws must hold:

$$x \mid\mid\mid (y \mid\mid\mid z) = (x \mid\mid\mid y) \mid\mid\mid z \tag{5.4a}$$

$$\mathit{fail} \mid\mid\mid x = x = x \mid\mid\mid \mathit{fail} \tag{5.4b}$$

The relationship between our model and the syntax of logic programming we are interested in is summarized in Table 5.1.

5.4 Background: Handlers and Transformers

Search heuristics are naturally expressed as transformations of search trees. For instance, the depth-bounded search prunes all subtrees below a given depth. Unfortunately, this view does not fit well with our monadic model of Prolog as the *Prolog* monad is opaque and we cannot observe the search tree structure of goals. A more effective technique for observing the syntactic structure of monadic programs is the *effect handlers* approach. This approach will be key to expressing heuristics in a modular way.

Effect handlers decouple the syntax and the semantics of side-effect primitives, which we call *operations* in the rest of the chapter. The syntactic operations themselves live in an abstract syntax tree, which is modelled by the free monad. The semantics are captured in so-called handler functions, or handlers for short, and we focus on those that can be expressed as folds over the abstract syntax tree.

The decoupling has a number of advantages: it facilitates both the modular definition of monads in terms of separately defined operations, and also the assignment of different semantics to the same syntax. In this dissertation we will add one more advantage to that list: it allows us to extend opaque monads that have not necessarily been defined in terms of the effect handlers approach with new capabilities.

The well-known free monad $f^{\star}$ denotes abstract syntax trees where the shape of the nodes is captured in the functor f.

$$\textbf{data}\, f^{\star}\ a = \mathit{Return}\ a \mid \mathit{Op}\ (f\ (f^{\star}\ a))$$

A new node of shape f and subtrees of type $f^{\star}\ a$ can be built with the *Op* constructor. The *Return* constructor marks a non-terminal, and $(\gg\!=)$ performs simultaneous substitution on all non-terminals in the tree.

$$\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (f^{\star})\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{return}\ a & & = \mathit{Return}\ a \\
\mathit{Return}\ a & \gg\!= f & = f\ a \\
\mathit{Op}\ n & \gg\!= f & = \mathit{Op}\ (\mathit{fmap}\ (\gg\!= f)\ n)
\end{array}
\end{array}$$

An important convention when using the free monad for modelling syntax trees is that each node represents an operation and its subtrees denote the possible continuations from that operation. In this way, the free monad's notion of substitution and the operational interpretation of sequential composition both coincide in $(\gg\!=)$.

As an example, lets consider an abstract syntax tree for expressions involving only the operations in *MonadState s m* [73], where the monad m uniquely determines the state s.

$$\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \rightarrow s\ \textbf{where} \\
\quad \mathit{get} :: \mathit{Monad}\ m \Rightarrow m\ s \\
\quad \mathit{put} :: \mathit{Monad}\ m \Rightarrow s \rightarrow m\ ()
\end{array}$$

This has two primitive operations, and the shape of the corresponding syntax is given by the functor *State s r*.

$$\begin{array}{l}
\textbf{data}\ \mathit{State}\ s\ r \\
\quad = \mathit{Get}\ (s \rightarrow r)
\end{array}$$

```
    | Put s (() → r)
instance Functor (State s) where
  fmap f (Get k)    = Get (f ∘ k)
  fmap f (Put s k) = Put s (f ∘ k)
```

Following the free monad convention, the parameter of the *Get* constructor takes a function that is the continuation after the get-operation. Similarly, the second parameter of the *Put* constructor is the continuation after the put-operation. Thus, *Op* (*Get k*) is the syntactic representation of *get* $\gg\!=$ *k* and *Op* (*Put s k*) of *put s* $\gg\!=$ *k*. If we want to represent the operations by themselves without any relevant continuation, we just instantiate *k* to *return*.

The following operations construct syntax trees that represent actions:

```
instance MonadState s ((State s)*) where
  get   = Op (Get return)
  put s = Op (Put s return)
```

5.4.1 The Free Monad Transformer

While the free monad approach forms an attractive basis for solving our search heuristics problem, it is not very appealing to replace the opaque *Prolog* monad with the free monad. After all, that would be akin to throwing away our existing Prolog system and engineering a new one from the ground up. Instead of this prodigious effort we would much prefer a more lightweight solution.

This solution comes in the form of the free monad transformer $f^\star_m$, which combines an existing base monad m with the free monad $f^\star$ by interleaving computations in m with syntactic operations from f. The return value has type a.

To define the free monad transformer, we generalise the free monad $f^\star$.

```
data Free f a x = ReturnF a | OpF (f x)
```

Here the type variable x represents the continuation after an f-operation. This generalisation is also a functor if f is a functor.

```
instance Functor f ⇒ Functor (Free f a) where
  fmap _ (ReturnF a) = ReturnF a
  fmap f (OpF x)      = OpF (fmap f x)
```

We can now define the free monad transformer by interleaving f with m and tying the knot.

$$\textbf{newtype}\ f^{\star}_{m}\ a = \mathit{Free}_T\ \{\mathit{unFree}_T :: m\ (\mathit{Free}\ f\ a\ (f^{\star}_{m}\ a))\}$$

When the base monad m is Id there is no additional structure so:

$$f^{\star}\ a \cong f^{\star}_{\mathit{Id}}\ a$$

In other words, $f^{\star}\ a$ is the fixpoint of $\mathit{Free}\ f\ a$. The star used in the notation reminds of the Kleene star.

The structure of $f^{\star}_{m}$ requires m to be a monad, and f to be a functor. We use this fact so often that we will use the following convenient notation for the required type constraint synonym:

$$\textbf{type}\ \vdash f^{\star}_{m} = (\mathit{Functor}\ f, \mathit{Functor}\ m, \mathit{Monad}\ m)$$

In other words, this constraint expresses that $f^{\star}_{m}$ is a well-formed free monad transformer.

We can define the fold over this structure:

$$\begin{aligned}&\mathit{fold} :: \vdash f^{\star}_{m} \Rightarrow (m\ (\mathit{Free}\ f\ a\ b) \to b) \to (f^{\star}_{m}\ a \to b)\\ &\mathit{fold}\ \mathit{alg} = \mathit{alg} \circ \mathit{fmap}\ (\mathit{fmap}\ (\mathit{fold}\ \mathit{alg})) \circ \mathit{unFree}_T\end{aligned}$$

To understand (and to come up with) this kind of definitions, one follows the types:

- $\mathit{fold}\ \mathit{alg}$ is a function from $f^{\star}_{m}\ a$ to b.

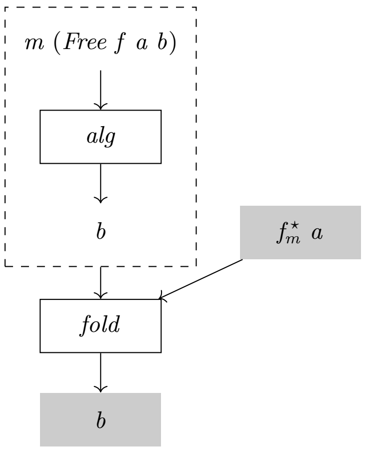


Figure 5.2: Illustrating the type correctness of the fold over $f^{\star}_{m}\ a$: left-hand side.

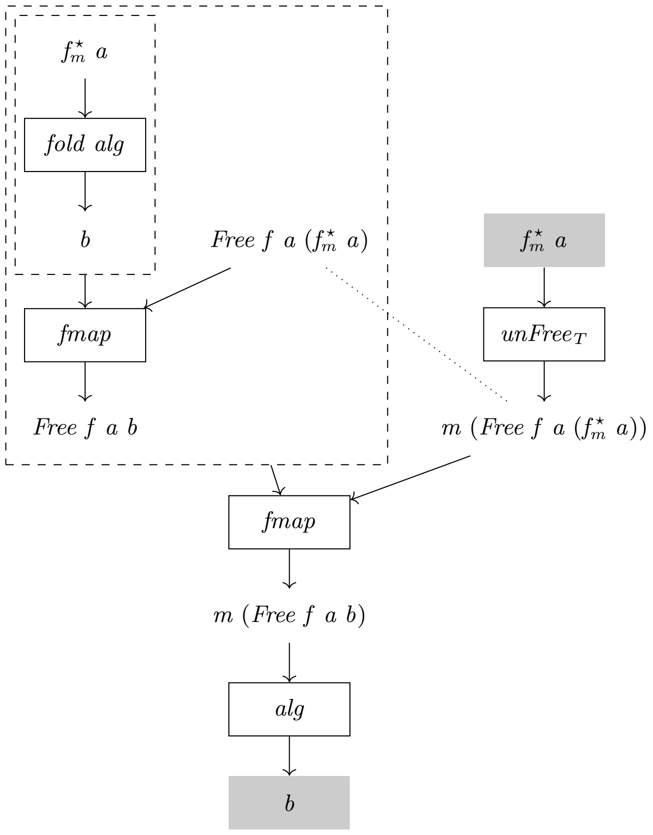
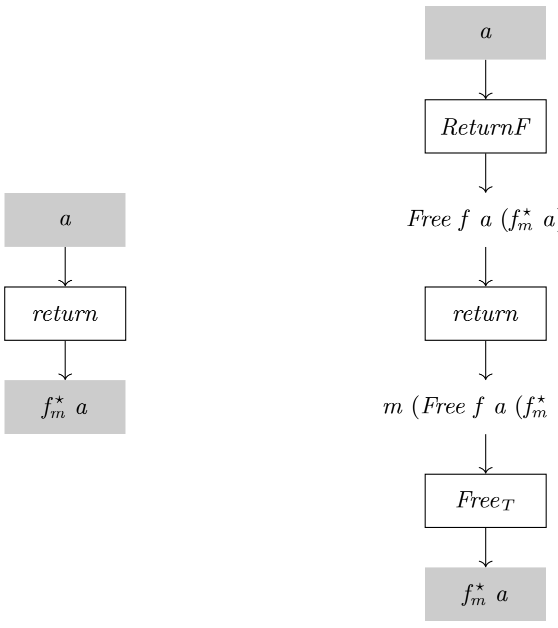


Figure 5.3: Illustrating the type correctness of the fold over $f_m^\star\ a$: right-hand side.

- $unFree_T$ is a function from $f_m^\star\ a$ to $m\ (Free\ f\ a\ (f_m^\star\ a))$.
- $(fmap\ (fold\ alg))$ is a function from $Free\ f\ a\ (f_m^\star\ a)$ to $Free\ f\ a\ b$, so $fmap\ (fmap\ (fold\ alg))$ is a function from $m\ (Free\ f\ a\ (f_m^\star\ a))$ to $m\ (Free\ f\ a\ b)$.
- The result of the expression $fmap\ (fmap\ (fold\ alg)) \circ unFree_T$ is thus a function from $f_m^\star\ a$ to $m\ (Free\ f\ a\ b)$.
- alg is a function from $m\ (Free\ f\ a\ b)\ to\ b$.
- The result of the entire right-hand side of $fold\ alg$ is thus a function from $f_m^\star\ a$ to b, which of course matches with the type of the left-hand side.

We have illustrated this in Figures 5.2 and 5.3.

Since any correct monad transformer must also be a monad, we must show that $f_m^\star$ is a monad. We can use the above fold to give a definition of $(\gg\!=)$ in the instance that shows this fact.



a
return
$f_m^\star\ a$
a
ReturnF
$Free\ f\ a\ (f_m^\star\ a)$
return
$m\ (Free\ f\ a\ (f_m^\star\ a))$
$Free_T$
$f_m^\star\ a$

Figure 5.4: Illustrating the type correctness of the $f_m^\star\ a$ monad instance: return case.

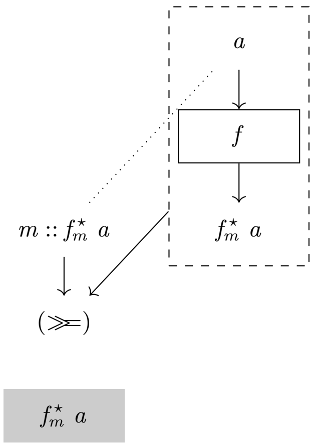


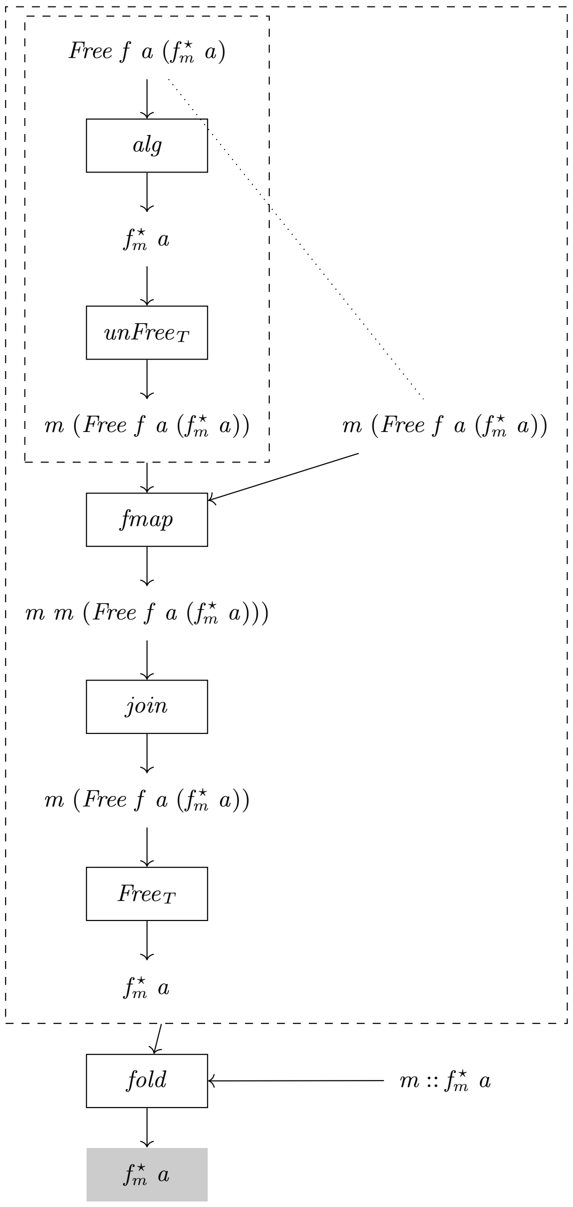
Figure 5.5: Illustrating the type correctness of the $f^{\star}_{m}\ a$ monad instance: left-hand side of the bind case.

$$
\begin{array}{l}
\textbf{instance} \vdash f^{\star}_{m} \Rightarrow \mathit{Monad}\ (f^{\star}_{m})\ \textbf{where} \\
\quad \mathit{return} \quad = \mathit{Free}_T \circ \mathit{return} \circ \mathit{ReturnF} \\
\quad m \gg\!= f = \mathit{fold}\ (\mathit{Free}_T \circ \mathit{join} \circ \mathit{fmap}\ (\mathit{unFree}_T \circ \mathit{alg}))\ m \\
\qquad \textbf{where}\ \mathit{alg}\ (\mathit{ReturnF}\ a) = f\ a \\
\qquad\qquad\quad\ \mathit{alg}\ (\mathit{OpF}\ \mathit{op}) \quad\ = \mathit{opF}\ \mathit{op} \\
\mathit{opF}\ \mathit{op} = \mathit{Free}_T\ (\mathit{return}\ (\mathit{OpF}\ \mathit{op}))
\end{array}
$$

Again, the type correctness of the definitions is not immediately obvious. We illustrate the *return*-case in Fig. 5.4 on page 111 and the bind-case ($(\gg\!=)$) in Fig. 5.5, and Fig. 5.6 on the facing page.

The following definition of *runStateF* is an example of a handler that eliminates the *State* syntax by interpreting it in terms of a state that is threaded through the computation. So, under the assumption that f is a functor and m is a monad (expressed by the constraint $\vdash (\mathit{State}\ s)^{\star}_{m}$), we can turn a syntax tree of type $(\mathit{State}\ s)^{\star}_{m}\ a$ back into the original base monad m by supplying an initial state of type s. The resulting value has type $m\ (a, s)$: it is a monadic value containing both the result value of computation and the final state.

$$
\begin{array}{l}
\mathit{runStateF} :: \vdash (\mathit{State}\ s)^{\star}_{m} \Rightarrow (\mathit{State}\ s)^{\star}_{m}\ a \to s \to m\ (a, s) \\
\mathit{runStateF}\ p\ s0 = \mathit{runFreeT}\ (\mathit{alg}\ s0)\ p\ \textbf{where} \\
\quad \mathit{alg}\ s\ (\mathit{ReturnF}\ x) \qquad = \mathit{return}\ (x, s) \\
\quad \mathit{alg}\ s\ (\mathit{OpF}\ (\mathit{Get}\ k)) \quad = \mathit{runStateF}\ (k\ s)\ \ s
\end{array}
$$



$Free\ f\ a\ (f_m^\star\ a)$
alg
$f_m^\star\ a$
$unFree_T$
$m\ (Free\ f\ a\ (f_m^\star\ a))$
$m\ (Free\ f\ a\ (f_m^\star\ a))$
$fmap$
$m\ m\ (Free\ f\ a\ (f_m^\star\ a)))$
$join$
$m\ (Free\ f\ a\ (f_m^\star\ a))$
$Free_T$
$f_m^\star\ a$
$fold$
$m :: f_m^\star\ a$
$f_m^\star\ a$

Figure 5.6: Illustrating the type correctness of the $f_m^\star\ a$ monad instance: right-hand side of the bind case.

$$\mathit{alg}\ s\ (\mathit{OpF}\ (\mathit{Put}\ s'\ k)) = \mathit{runStateF}\ (k\ ())\ s'$$

What the *alg*-function does, is intuitively clear: given a terminal node in the syntax tree and a state, we invoke the return operation of the base monad with the value in the terminal node and the same state. On encountering the syntactic encoding of a get, we execute that get and similarly for put.

The above definition makes use of the following auxiliary definition:

$$\begin{aligned}&\mathit{runFreeT} :: \vdash f^{\star}_{m} \Rightarrow (\mathit{Free}\ f\ a\ (f^{\star}_{m}\ a) \to m\ b) \to (f^{\star}_{m}\ a \to m\ b)\\ &\mathit{runFreeT}\ \mathit{alg}\ p = \mathit{unFree}_T\ p \gg\!\!= \mathit{alg}\end{aligned}$$

which runs the given computation up to the first syntactic operation, which it delegates to *alg*. In the case of *runStateF*, this *alg* handles the state operation appropriately, and at the end of the computation returns the result together with the final state.

In the remainder of the chapter we will also make use of the following variant of *runFreeT*:

$$\begin{aligned}&\mathit{step} :: \vdash f^{\star}_{m} \Rightarrow f^{\star}_{m}\ a \to (f\ (f^{\star}_{m}\ a) \to f^{\star}_{m}\ a) \to f^{\star}_{m}\ a\\ &\mathit{step}\ t\ \mathit{alg} = \mathit{Free}_T\ (\mathit{runFreeT}\ \mathit{alg}'\ t)\ \textbf{where}\\ &\quad \mathit{alg}'\ (\mathit{ReturnF}\ x) = \mathit{return}\ (\mathit{ReturnF}\ x)\\ &\quad \mathit{alg}'\ (\mathit{OpF}\ \mathit{op})\qquad = \mathit{unFree}_T\ (\mathit{alg}\ \mathit{op})\end{aligned}$$

The function *step* differs in two ways from *runFreeT*. Firstly, it does not promise to eliminate the syntactic operations altogether. Instead, it can be used to eliminate only some operations or to replace them by others (last line). Secondly, *step* preserves *ReturnF* (one-but-last line). As a consequence, the *alg* parameter only needs to concern itself with the operations.

We again demonstrate that the function is indeed type-correct in Fig. 5.7 on the next page, Fig. 5.8, and Fig. 5.9 on page 116.

5.5 Heuristics as Handlers in Haskell

The machinery of effect handlers gives us the tools we need to describe heuristics in a modular way. Our solution will be developed in four steps.

5.5.1 Step 1: Overloading

Our first step is to overload the operations of *Prolog*. Here we accomplish this with the *MonadProlog* type class:

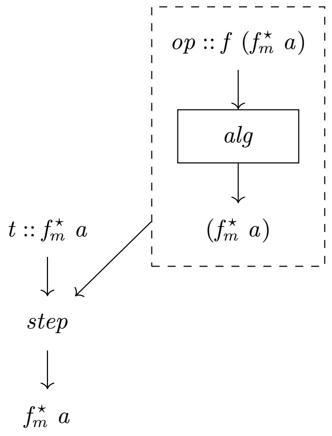


Figure 5.7: Illustrating the type correctness of *step*: left-hand side.

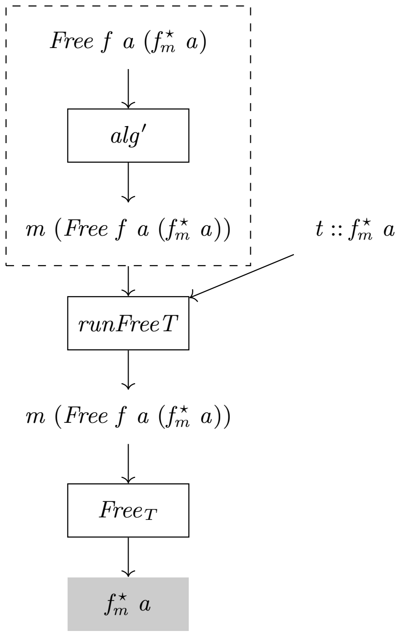

$Free\ f\ a\ (f_m^\star\ a)$
alg'
$m\ (Free\ f\ a\ (f_m^\star\ a))$
$t :: f_m^\star\ a$
$runFreeT$
$m\ (Free\ f\ a\ (f_m^\star\ a))$
$Free_T$
$f_m^\star\ a$

Figure 5.8: Illustrating the type correctness of *step*: right-hand side.

$$\begin{tikzcd}
x :: a \arrow[d] & op :: f\ (f_m^\star\ a) \arrow[d] \\
\boxed{\mathit{ReturnF}} \arrow[d] & \boxed{\mathit{alg}} \arrow[d] \\
\mathit{Free}\ f\ a\ (f_m^\star\ a) \arrow[d] & (f_m^\star\ a) \arrow[d] \\
\boxed{\mathit{return}} \arrow[d] & \boxed{\mathit{unFree}_T} \arrow[d] \\
m\ (\mathit{Free}\ f\ a\ (f_m^\star\ a)) & m\ (\mathit{Free}\ f\ a\ (f_m^\star\ a))
\end{tikzcd}$$

Figure 5.9: Illustrating the type correctness of the alg' helper function.

class *Monad* $m \Rightarrow$ *MonadProlog* m **where**
  $\mathit{fail} :: m\ a$
  $(|||) :: m\ a \to m\ a \to m\ a$

which inherits the left-zero and left-distributivity laws of *Prolog*,

$$\mathit{fail} \gg\!= q = \mathit{fail} \tag{5.5a}$$
$$(m\ |||\ n) \gg\!= f = (m \gg\!= f)\ |||\ (n \gg\!= f) \tag{5.5b}$$

However, importantly, we do not require that $\langle m\ \ a, (|||), \mathit{fail}\rangle$ forms a monoid. This relaxation is crucial to support search heuristics. After all, the monoid laws require that the shape of the search tree is irrelevant. For instance, according to the associativity law, the following two trees should be indistinguishable:

$t_1 = \mathit{return}\ x\ |||\ (\mathit{return}\ y\ |||\ \mathit{return}\ z)$
$t_2 = (\mathit{return}\ x\ |||\ \mathit{return}\ y)\ |||\ \mathit{return}\ z$

In contrast, the shape of the search tree is essential for search heuristics. Different shapes of trees will be affected differently by the same heuristic. For instance, the depth-bounded search heuristic may prune the solutions y and z in t_1, while it prunes x and y in t_2.

5.5.2 Step 2: Introducing Syntax

We proceed in the second step by capturing the relevant operations as syntax using the free monad transformer. While *MonadProlog* provides two operations, *fail* and ($|||$), we will see in the next step that we can get away with capturing only ($|||$) in syntax. This comes at the cost of somewhat more complicated handlers. However, Section 5.6 will bear out that keeping the syntactic footprint as small as possible is a good idea.

Hence, the functor *Or* captures only $p\ |||\ q$ with the syntax $Or\ p\ q$.

```
data Or x = Or x x
orF :: Monad m ⇒ $Or^{\star}_{m}$ a → $Or^{\star}_{m}$ a → $Or^{\star}_{m}$ a
orF p q = opF (Or p q)
instance Functor Or where
  fmap f (Or p q) = Or (f p) (f q)
```

Observe that unlike *Get* the constructor *Or* does not require a separate field for the continuation. Thanks to the left-distributivity property, we can express $(p\ |||\ q) \ggg k$ also as $(p \ggg k)\ |||\ (q \ggg k)$.

This data acts as a syntactic construction that gives us an instance of *MonadProlog* in terms of the free monad transformer:

```
instance MonadProlog m ⇒ MonadProlog ($Or^{\star}_{m}$) where
  fail    = lift fail
  p ||| q = orF p q
```

5.5.3 Step 3a: Adding Heuristics

The syntactic *Or* gives substance to the search tree that is implicitly embodied by a computation. Now we can truly write search heuristics as functions that transform this search tree.

```
type Heuristic m a = $Or^{\star}_{m}$ a → $Or^{\star}_{m}$ a
```

Below we capture a number of well-known heuristics in this form.

Depth-Bounded Search The Haskell version of the depth-bounded search heuristic from Section 3.3 looks like:

```
dbs :: MonadProlog m ⇒ Int → Heuristic m a
dbs 0 t = fail
dbs n t = step t alg where
  alg (Or x y) = (dbs (n - 1) x) ||| (dbs (n - 1) y)
```

Here we see our first use of the *step* function. It is used to decrement the depth bound at every *Or*. When the limit is exceeded, the whole remaining computation is replaced by failure. Remember that *Heuristic m a* is a synonym for a function from $Or^{\star}_{m}\ a$ to $Or^{\star}_{m}\ a$.

Discrepancy-Bounded Search Discrepancy-bounded search (Section 3.4) is a minor variant of depth-bounded search:

```
dibs :: MonadProlog m ⇒ Int → Heuristic m a
dibs 0 t = fail
dibs n t = step t alg where
  alg (Or x y) = (dibs n x) ||| (dibs (n - 1) y)
```

Node-Bounded Search The implementation of node-bounded search in Prolog requires the use of mutable references as explained in Section 2.4. We can capture such references in our model by using the type class *MonadRef*. This supports three operations: *newRef* creates a new reference *r a* within the monadic context *m*, *readRef* extracts a value from a reference into the context, and *writeRef* takes a reference and a value *a*, and writes the value into the reference.

```
class Monad m ⇒ MonadRef r m | m → r where
  newRef   :: a → m (r a)
  readRef  :: r a → m a
  writeRef :: r a → a → m ()
```

To illustrate the use of this interface, we implement *modifyRef*, which simply modifies the current value in some cell by applying a function *f* to its contents, and then returns the original value.

```
modifyRef :: MonadRef r m ⇒ r a → (a → a) → m a
modifyRef r f = do x ← readRef r
                   writeRef r (f x)
                   return x
```

This works by first reading the value *x* contained in the reference, writing the new value *f x* to the reference, and then returning *x*.

In addition to the usual properties of mutable references, we also explicitly stipulate the interaction with backtracking: the writes are not backtracked over.

$$\mathit{writeRef}\ \mathit{ref}\ x \gg (p \mathbin{|||} q) = (\mathit{writeRef}\ \mathit{ref}\ x \gg p) \mathbin{|||} q \tag{5.6}$$

Read from right to left this law expresses that writes in the left branch are also seen by the right branch. Contrast this with the characterization of backtracking behaviour:

$$\begin{gathered}\mathit{writeRef}\ \mathit{ref}\ x \gg (p \mathbin{|||} q) \\ = \\ (\mathit{writeRef}\ \mathit{ref}\ x \gg p) \mathbin{|||} (\mathit{writeRef}\ \mathit{ref}\ x \gg q)\end{gathered}$$

which expresses that writes in one branch are not seen by the other branch.

The support for references can be lifted straightforwardly from m to $Or^{\star}_{m}$.

```
instance MonadRef r m ⇒ MonadRef r (Or⋆m) where
  newRef x     = lift (newRef x)
  readRef r    = lift (readRef r)
  writeRef r x = lift (writeRef r x)
```

This support enables us to express a handler for node-bounded search.

```
nbs :: (MonadProlog m, MonadRef r m) ⇒ Int → Heuristic m a
nbs n t = newRef n ≫= go t where
  go t′ ref = step t′ alg where
    alg (Or x y) = do n ← modifyRef ref pred
                      guard (n > 0)
                      go x ref ||| go y ref
```

Failure-Bounded Search Failure-bounded search terminates the search when too many paths in the tree lead to dead ends. It may actually seem surprising that we can write this heuristic without being able to explicitly observe failure. Nevertheless, with a clever trick that relies on the underlying DFS we can observe failure indirectly.

```
fbs :: (MonadProlog m, MonadRef r m) ⇒
       Int → Heuristic m a
fbs n t = do ref ← newRef n
             fref ← newRef False             -- (a)
             x ← go ref fref t
```

```
          writeRef fref True                 -- (b)
          return x where
  go ref' fref' t' = step t' alg
    where alg (Or x y) = x |||
      (do b ← modifyRef fref' (const False)  -- (c)
          when (¬ b)                         -- (d)
            (do n ← modifyRef ref' pred
                guard (n > 0))
          y)
```

The mutable reference *fref* expresses whether the last explored path has terminated successfully. At the start of the search (a), we are on the first path but have not completed it yet. Hence initially *fref* holds the value *False*. Later, when a solution is found (b), the value *True* is written into the reference. After completing a path successfully or unsuccessfully, the search backtracks into the right branch of an *or*. At the start of the right branch (c) we can observe in *fref* whether the previous path was successful or not. We also write *False* into *fref* to capture the status of the new path we are on. If the previous path has failed (d), we subtract one from the failure bound and prune if we have failed too often already.

5.5.4 Step 3b: Adding Heuristics as Trees

The effect handlers approach distinguishes syntactic operations and handlers. Syntactic operations offer the flexibility of a deeply embedded domain-specific language (DSL); they can be freely analyzed, manipulated and interpreted in different ways. This is exactly the property we have put to good use with the definition of heuristics over search trees. In contrast, handlers like our heuristics are akin to a shallow embedding of a DSL: they can be used in one way only, by function application to a computation.

In this section we show how to recover much of the flexibility of deep embeddings, while simultaneously providing a more structured approach to defining search heuristics. This approach is a literal translation of the Prolog approach from Section 3.3.

1. We capture the essence of a heuristic in an archetypal search tree. For instance, the archetypal search tree for depth-bounded search is a perfect binary tree of depth n with failures at its leaves.

   ```
   dbsTree 0 = fail
   dbsTree n = dbsTree (n − 1) ||| dbsTree (n − 1)
   ```

2. We apply the heuristic to a search problem by means of an operator $(\curlyvee)$ called *entwine*, that combines two search trees: in this case, one given by the logic to solve the problem, and the other given by the heuristic.

   For instance, we recover *dbs* as follows:

   $$\begin{aligned}&\mathit{dbs}' :: \mathit{MonadProlog}\ m \Rightarrow \mathit{Int} \to \mathit{Heuristic}\ m\ a\\ &\mathit{dbs}'\ \mathit{db}\ t = \mathit{dbsTree}\ \mathit{db} \curlyvee t\end{aligned}$$

In summary, this approach refines Kowalski's slogan to:

$$\mathit{algorithm} = \mathit{control} \curlyvee \mathit{logic}$$

where both logic and control are expressed in a declarative rather than an operational style. Moreover, they are expressed in the same language of search trees.

The Entwining Operator The $(\curlyvee)$ operator is similar to the definition of the *zip* operation: zipping two lists truncates the longer one when their structure disagrees. Similarly, entwining two trees truncates the larger one when their structure disagrees. The truncation of trees is essentially the pruning of the search space.

$$\begin{aligned}&(\curlyvee) :: (\mathit{MonadProlog}\ m) \Rightarrow \mathit{Or}^{\star}_{m}\ a \to \mathit{Or}^{\star}_{m}\ a \to \mathit{Or}^{\star}_{m}\ a\\ &p \curlyvee q = p\ `\mathit{step}`\ (\lambda(\mathit{Or}\ x_1\ x_2) \to\\ &\qquad\qquad q\ `\mathit{step}`\ (\lambda(\mathit{Or}\ y_1\ y_2) \to\\ &\qquad\qquad\quad (x_1 \curlyvee y_1)\ |||\ (x_2 \curlyvee y_2)))\end{aligned}$$

This operation steps into the first tree, and inspects its structure for an *Or* constructor. When this is found, it steps into the second tree where it again inspects its structure for an *Or* constructor. If both are found, then a new tree is constructed, where children of the trees are entwined together.

Note that $\langle \mathit{Or}^{\star}_{m}\ a, (\curlyvee), \mathit{inf}\rangle$ forms a monoid, because $(\curlyvee)$ is associative and the infinitely branching tree *inf* is its identity:

$$\begin{aligned}&\mathit{inf} :: \mathit{Monad}\ m \Rightarrow \mathit{Or}^{\star}_{m}\ a\\ &\mathit{inf} = \mathit{opF}\ (\mathit{Or}\ \mathit{inf}\ \mathit{inf})\end{aligned}$$

Archetypal Trees and Handlers We can establish that $\mathit{dbsTree}\ n$ captures the essence of the dbs handler in an archetypal tree by showing that $\mathit{dbs}\ n$ and $\mathit{dbs}'\ n$ are equivalent. Our proof proceeds by induction on n.

For $n = 0$ we have that:

$$\begin{aligned}
& \mathit{dbs}\ 0\ t \\
= {} & \{ \text{ def. of } \mathit{dbs} \ \} \\
& \mathit{fail} \\
= {} & \{ \ \forall \mathit{alg}.\mathit{step}\ \mathit{fail}\ \mathit{alg} = \mathit{fail} \ \} \\
& \mathit{fail}\ \text{`}\mathit{step}\text{`}\ (\lambda(\mathit{Or}\ x_1\ x_2) \to \\
& \ \ t\ \text{`}\mathit{step}\text{`}\ (\lambda(\mathit{Or}\ y_1\ y_2) \to (x_1 \curlyvee y_1)\ |||\ (x_2 \curlyvee y_2))) \\
= {} & \{ \text{ def. of } (\curlyvee) \ \} \\
& \mathit{fail} \curlyvee t \\
= {} & \{ \text{ def. of } \mathit{dbsTree} \ \} \\
& \mathit{dsbTree}\ 0 \curlyvee t \\
= {} & \{ \text{ def. of } \mathit{dbs}' \ \} \\
& \mathit{dbs}'\ 0\ t
\end{aligned}$$

Also, for $n = m + 1$ and induction hypothesis $\mathit{dbs}\ m = \mathit{dbs}'\ m$, we can show that:

$$\begin{aligned}
& \mathit{dbs}\ (m + 1)\ t \\
= {} & \{ \text{ def. of } \mathit{dbs} \ \} \\
& \mathit{step}\ t\ (\lambda(\mathit{Or}\ y_1\ y_2) \to (\mathit{dbs}\ m\ y_1)\ |||\ (\mathit{dbs}\ m\ y_2)) \\
= {} & \{ \text{ induction hypothesis } \} \\
& \mathit{step}\ t\ (\lambda(\mathit{Or}\ y_1\ y_2) \to (\mathit{dbs}'\ m\ y_1)\ |||\ (\mathit{dbs}'\ m\ y_2)) \\
= {} & \{ \text{ def. of } \mathit{dbs}' \ \} \\
& \mathit{step}\ t\ (\lambda(\mathit{Or}\ y_1\ y_2) \to (\mathit{dbsTree}\ m \curlyvee y_1)\ |||\ (\mathit{dbsTree}\ m \curlyvee y_2)) \\
= {} & \{ \ \forall \mathit{alg}\ t_1\ t_2.\mathit{alg}\ (\mathit{Or}\ t_1\ t_2) = \mathit{step}\ (t_1\ |||\ t_2)\ \mathit{alg} \ \} \\
& (\mathit{dbsTree}\ m\ |||\ \mathit{dbsTree}\ m)\ \text{`}\mathit{step}\text{`}\ (\lambda(\mathit{Or}\ x_1\ x_2) \to \\
& \ \ t\ \text{`}\mathit{step}\text{`}\ (\lambda(\mathit{Or}\ y_1\ y_2) \to (x_1 \curlyvee y_1)\ |||\ (x_2 \curlyvee y_2))) \\
= {} & \{ \text{ def. of } \curlyvee \ \} \\
& (\mathit{dbsTree}\ m\ |||\ \mathit{dbsTree}\ m) \curlyvee t \\
= {} & \{ \text{ def. of } \mathit{dbsTree} \ \} \\
& \mathit{dbsTree}\ (m + 1) \curlyvee t
\end{aligned}$$

$$\begin{array}{ll} = & \{ \text{ def. of } \mathit{dbs}' \ \} \\ & \mathit{dbs}'\ (m+1)\ t \end{array}$$

Modular Definition of Heuristics Figure 5.10 shows that there is an archetypal search tree hidden in all the heuristics of Section 5.5.4. Yet, the main advantage of archetypal search trees is that we can define them in a convenient modular style.

For instance, we can define *dbsTree n* as $\mathit{delay}\ n \gg \mathit{fail}$, where *delay* returns *return* () after a given depth:

```
delay :: MonadProlog m ⇒ Int → Or★_m ()
delay 0 = return ()
delay n = delay (n − 1) ||| delay (n − 1)
```

Here, *return* () is a placeholder for another heuristic that can be plugged in with ($\gg$) and becomes active at depth n in the search tree.

This allows us to define iterative deepening as follows:

```
itd :: (MonadProlog m, MonadRef r m) ⇒ Heuristic m a
itd t = do  newRef False >>= go t 0 where
  go t n ref = (t ⋎ (delay n ≫ prune ref)) |||
                      do b ← readRef ref
                         if b then do writeRef ref False
                                      go t (n + 1) ref
                              else  fail
  prune ref = writeRef ref True ≫ fail
```

Iterative deepening *itd* repeatedly runs a depth-bounded search, incrementing the depth bound on each iteration. The iterative process stops when no pruning happened in the last iteration. The heuristic *prune ref* performs immediate pruning and records this in the mutable reference to remember it across backtracking. With $\mathit{delay}\ n \gg \mathit{prune}\ \mathit{ref}$ the immediate pruning is delayed to depth n.

While TOR provides an operator `merge/2` similar to ($\curlyvee$), that operator does not satisfy the same elegant algebraic properties (e.g, forming a monoid) and, as consequence, cannot express *delay*ed heuristics in this modular fashion.

Limitation While ($\curlyvee$) is very convenient and captures a large class of search heuristics, not all heuristics can be expressed in this way. In particular, consider the random reordering of branches,

```
dibs′ :: (MonadProlog m) ⇒ Int → Heuristic m a
dibs′ db t = dibsTree db ⋎ t where
  dibsTree 0 = fail
  dibsTree n = dibsTree n ⋎ dibsTree (n − 1)


nbs′ :: (MonadProlog m, MonadRef r m) ⇒
        Int → Heuristic m a
nbs′ n t = do ref ← newRef n
              t ⋎ nbsTree ref where
  nbsTree ref = do n ← modifyRef ref pred
                   guard (n > 0)
                   nbsTree ref ||| nbsTree ref


fbs′ :: (MonadProlog m, MonadRef r m) ⇒ Int → Heuristic m a
fbs′ n t = do ref ← newRef n
              fref ← newRef False
              x ← t ⋎ fbsTree ref fref
              writeRef fref True
              return x where
  fbsTree ref fref = fbsTree ref fref |||
    do b ← modifyRef fref (const False)
       when (¬ b) (do n ← modifyRef ref pred
                      guard (n > 0))
       fbsTree ref fref
```

Figure 5.10: Search heuristics expressed as entwined trees.

```
$muddle :: (MonadRandom\ m, MonadProlog\ m) \Rightarrow Heuristic\ m\ a$
$muddle\ t = step\ t\ alg$ where
  $alg\ (Or\ x\ y) =$ do $b \leftarrow getRandom$
                         if $b$ then $muddle\ x\ |||\ muddle\ y$
                            else $muddle\ y\ |||\ muddle\ x$
```

which is a popular heuristic to randomize the search tree. This heuristic cannot be expressed with $(\curlyvee)$ because it always keeps left branches on the left and right branches on the right.

5.5.5 Step 4: Reflecting Syntax Back into Semantics

Finally, the *semOr* handler reflects the syntactic *Or* back into the semantic $(|||)$ of the underlying monad m.

```
$semOr :: MonadProlog\ m \Rightarrow Or^{\star}_{m}\ a \rightarrow m\ a$
$semOr = runFreeT\ alg$ where
  $alg\ (ReturnF\ a)\qquad = return\ a$
  $alg\ (OpF\ (Or\ x\ y)) = semOr\ x\ |||\ semOr\ y$
```

We can now recover $queens'$ as:

$$queens''\ n\ db = semOr\ (dbs\ db\ (queens\ n))$$

provided that *queens* is written against the type class *MonadProlog* rather than the opaque monad *Prolog*.

In this scheme, we start with the original tree generated by *queens*, but interpreted under the $Or^{\star}_{m}$ monad. The ensuing tree is then pruned by the function *dbs* before it is finally reflected back into the underlying monad m.

5.6 From Haskell to Prolog

This section sets up the means to transfer our Haskell-based solution for modular search heuristics to Prolog.

On the outset, there are several compelling reasons why basing our approach on the free monad transformer would make it well-suited for implementation in Prolog:

1. In the $Or^{\star}_{m}$ type, we can choose m to be the complex (but implicit) monad that underpins Prolog.

2. The approach allows us to conveniently reuse Prolog's primitive implementations for *return* () and *fail* by lifting.

3. We can lift individual feature extensions that we have modelled as additional class constraints, such as mutable references and random number generation. This could be applied to other extensions we have not covered explicitly in our model: predicates (i.e., goal abstractions), mutable databases, I/O, and many more.

However, implementing the free monad transformer itself in Prolog is challenging. As the *Prolog* monad is implicit in the Prolog language, it does not lend itself to transformation. So we now direct our efforts to overcoming this obstacle.

5.6.1 Meta-Interpreter

As explained in Section 2.2, meta-interpreters are the most common way of modelling Prolog language extensions in Prolog. The signature of a plain Prolog meta-interpreter is `eval/1`, where `eval(Goal)` conceptually denotes the type m (). However, our meta-interpreter needs to capture computations of type m (*Free Or* () ($Or^{\star}_{m}$ ())). Hence, we extend the interpreter's signature with an extra (output) argument: `eval(Goal,Flag)`. `Flag` is either `return` (corresponding to *Return* ()) or `or(Goal1,Goal2)` (corresponding to *opF* (*Or* g_1 g_2)).

With this signature it is straightforward to port the free monad transformer implementation to Prolog (see Figure 5.11). However, this meta-interpreter requires us to reify *all* of the syntax in Prolog which we are interested in. For our small fragment this is very manageable, but there are many other features in Prolog, such as built-ins and user-defined predicate calls, and with a growing list, this approach will soon become tedious and unmaintainable. We clearly need an approach that is orthogonal to the existing language features.

5.6.2 Delimited Continuations

We do not have to look very far for an alternative approach. The delimited continuations from the previous chapter provide an isomorphic replacement of the free monad transformer.

Prolog provides an idiosyncratic interface of two operators for capturing delimited continuations: `shift/1` and `reset/3` which are conventionally modelled by:[1]

[1]Note that these control operators are have different signatures and semantics than those

```
semOr(Goal) :-
  eval(Goal,Flag),
  ( Flag = return ->
      true
  ; Flag = or(G1,G2) ->
      ( semOr(G1)
      ; semOr(G2)
      )
  ).

eval(Goal,Flag) :-
  ( Goal = fail ->
      fail
  ; Goal = true ->
      Flag = return
  ; Goal = (Goal1,Goal2) ->
      eval(Goal1,Flag1),
      ( Flag1 = return ->
          eval(Goal2,Flag)
      ; Flag1 = or(Left,Right) ->
          Flag = or((Left,Goal2),(Right,Goal2))
      )
  ; Goal = (Goal1;Goal2) ->
      Flag = or(Goal1,Goal2)
  ; Goal = entwine(Goal1,Goal2) ->
      eval(Goal1,Flag1),
      ( Flag1 = return ->
          Flag = return
      ; Flag1 = or(Left1,Right1) ->
          eval(Goal2,Flag2),
          ( Flag2 = return ->
              Flag = return
          ; Flag2 = or(Left2,Right2) ->
              eval((entwine(Left1,Left2)
                   ;entwine(Right1,Right2)
                   ),Flag)
          )
      )
  ).
```

Figure 5.11: Prolog meta-interpreter supporting `entwine/2`.

```
class Monad m ⇒ MonadDelCont f m | m → f where
  shift_P :: f b → m b
  reset_P :: m a → (a → m b) →
             (Susp f (m a) → m b) → m b
```

The shift_P operation takes a reason and returns a computation. The reset_P operation takes a computation of type $m\ a$, a conversion function of type $a \to m\ b$, a handler function of type $\mathit{Susp}\ f\ (m\ a) \to m\ b$ and returns a computation $m\ b$.

These operations can be seen as a generalization of *throw* and *catch* from the well-known error monad. Like *throw*, the shift_P operation terminates the ongoing computation abruptly, with a value that indicates the reason. Like *catch*, the reset_P operation makes it possible to observe whether a subcomputation terminates normally or abruptly.

The big difference between both interfaces is that *catch* only exposes the reason for the abrupt termination. In contrast, reset_P gives us, nicely packaged up in a *Susp*(ension), both the reason (as a value of type $f\ a$), and the unfinished part (the continuation given by $a \to r$) of the subcomputation.

```
data Susp f r where
  S :: f a → (a → r) → Susp f r
instance Functor (Susp f) where
  fmap f (S d r) = S d (f ∘ r)
```

Note that the type of the reason $f\ a$ is indexed by the type a expected by the continuation $a \to r$.

The reset_P and shift_P control operations satisfy two laws that regulate their interaction.

$$\mathit{reset}_P\ (\mathit{shift}_P\ d \ggg\!= f)\ r\ h = h\ (S\ d\ f) \tag{5.7a}$$

$$\mathit{reset}_P\ (\mathit{return}\ x)\ r\ h = r\ x \tag{5.7b}$$

The first law shows that a shift under a reset is handled by h, which has access to the suspended computation. The second law shows that the result of a successful computation under a reset is handled by the conversion function r, which has access to the result. In either case, reset_P returns a computation of type $m\ b$.

originally introduced by Danvy and Filinski under those names [31]. They are more closely related to Sitaram's *fcontrol* and *run* operators [145].

5.6.3 The Delimited Continuations Transformer

An instance of *MonadDelCont* can be obtained from any monad m by making use of the delimited continuations monad transformer, written $f^{\dagger}_{m}\ a$, and this will serve as our replacement for the free monad transformer that fits the functionality exposed by Prolog.

```
$\textbf{newtype}\ f^{\dagger}_{m}\ a = DC_T\ \{runDC_T :: \forall r.(a \to m\ r) \to$
                              $(Susp\ f\ (f^{\dagger}_{m}\ a) \to m\ r) \to m\ r\}$
```

Its representation takes two continuations, the return continuation of type $a \to m\ r$, and the handler continuation of type $Susp\ f\ (f^{\dagger}_{m}\ a) \to m\ r$.

The transformed monad's *return* method invokes the return continuation, and its $(\gg=)$ extends both continuations:

```
$\textbf{instance}\ Monad\ m \Rightarrow Monad\ (f^{\dagger}_{m})\ \textbf{where}$
  $return\ x = DC_T\ (\lambda r\ h \to r\ x)$
  $m \gg= f\ \ = DC_T\ (\lambda r\ h \to$
    $runDC_T\ m\ (\lambda x \to runDC_T\ (f\ x)\ r\ h)\ (h \circ fmap\ (\gg= f)))$
```

Remember that the $runDC_T$ function takes three arguments:

- the first is the computation m to extend;
- the second is the return continuation, in this case it it a function that takes the result of the computation and runs $f\ x$;
- the third is the handler continuation, which is $h \circ fmap\ (\gg= f)$.

The $reset_P$ method sets the handler continuation and the $shift_P$ method invokes it.

```
$\textbf{instance}\ Monad\ m \Rightarrow MonadDelCont\ f\ (f^{\dagger}_{m})\ \textbf{where}$
  $reset_P\ m\ r\ h = DC_T\ (\lambda r'\ h' \to$
    $runDC_T\ m\ (\lambda x \to runDC_T\ (r\ x)\ r'\ h')$
              $(\lambda p \to runDC_T\ (h\ p)\ r'\ h'))$
  $shift_P\ d = DC_T\ (\lambda r\ h \to h\ (S\ d\ return))$
```

5.6.4 The Isomorphism

We will now establish the relationship between the free monad transformer and the delimited continuations monad transformer. This will enable us to adapt our existing infrastructure formulated in terms of the former to Prolog-compatible infrastructure in terms of the latter.

For all intents and purposes the two transformers are isomorphic, but there are two significant technical wrinkles that must be ironed out before formally establishing this isomorphism.

First, we must enforce that the base functor f of the delimited continuation transformer is only applied to monadic values. These values are after all meant to be the continuations of the syntactic operations. We can impose this restriction by pre-composing f with the monad and thus use $f_m^{\ddagger}$ instead of $f_m^{\dagger}$. Remember that any monad is a functor and that the composition of two functors is also a functor.

$$\mathbf{data}\, f_m^{\ddagger}\ a = L\,\{\,runL :: (f \circ f_m^{\ddagger})_m^{\dagger}\ a\,\}$$

where $(\circ)$ is the well-known functor composition:

$$\mathbf{data}\ (f \circ g)\ a = Comp\,\{\,runComp :: f\ (g\ a)\,\}$$

The corresponding functor instance is:

```
$\mathbf{instance}\ (Functor\ f, Functor\ g) \Rightarrow Functor\ ((\circ)\ f\ g)\ \mathbf{where}$
  $fmap\ f\ (Comp\ x) = Comp\ (fmap\ (fmap\ f)\ x)$
```

Second, the suspension $S\ s\ k$ breaks up a continuation into two parts: one part that sits under the syntactic construction s, and another part k that represents the following execution. Consequently, continuations that have been broken up at different points are distinguishable. However, if we are careful to always treat these parts as one atomic continuation, then this does not pose a problem. We can enforce this atomic treatment by *normalize*-ing all $f_m^{\ddagger}\ a$ computations.

```
$normalize :: \vdash f_m^{\star} \Rightarrow f_m^{\ddagger}\ a \rightarrow f_m^{\ddagger}\ a$
$normalize\ m =$
  $L\ (DC_T\ (\lambda r\ h \rightarrow runDC_T\ (runL\ m)\ r\ (h \circ norm)))$
  $\mathbf{where}$
    $norm\ (S\ x\ k) =$
      $S\ ((Comp \circ fmap\ join \circ runComp \circ fmap\ (L \circ k))\ x)\ return$
```

Taking the above two points into consideration we can formulate the two witnesses of the isomorphism.

The *to* function turn a value of type $f_m^{\star}\ a$ into a value of type $f_m^{\ddagger}\ a$, while *from* does the inverse.

```
$to :: \vdash f^{\star}_{m} \Rightarrow f^{\star}_{m}\ a \to f^{\ddagger}_{m}\ a$
$to\ m = L\ (DC_T\ (\lambda r\ h \to$
  $\mathbf{do}\ x \leftarrow unFree_T\ m$
      $\mathbf{case}\ x\ \mathbf{of}$
        $ReturnF\ a \to r\ a$
        $OpF\ s \qquad \to h\ (S\ (Comp\ (fmap\ to\ s))\ return)$
        $))$
$from :: \vdash f^{\star}_{m} \Rightarrow f^{\ddagger}_{m}\ a \to f^{\star}_{m}\ a$
$from\ m\ = Free_T\ (runDC_T\ (runL\ m)\ r\ h)$ where
  $r = \ return \circ ReturnF$
  $h = \ return \circ OpF \circ fmap\ from \circ collapse$
  $collapse\ (S\ s\ k) = fmap\ (\ggg L \circ k)\ (runComp\ s)$
```

These functions indeed witness the isomorphism if quotiented by *normalize*:

$$from \circ to = id \tag{5.8}$$

$$normalize \circ to \circ from = normalize \tag{5.9}$$

Finally, using the isomorphism it is possible to derive that the $f^{\ddagger}_{m}$ equivalent of *opF* can be defined simply as:

$$opF'\ s = shift_P\ s$$

In other words, a syntactic operation can be modelled directly in Prolog using `shift/1`.

Similarly, we can derive the counterpart of *step* as:

$$step'\ m\ h = reset_P\ m\ return\ h$$

With opF' and $step'$ we have all we need to make the transition from Haskell to Prolog.

5.7 Heuristics as Handlers in Prolog

Finally, we have a Prolog-friendly approach that is both light-weight and enables a mostly native execution of search problems.

5.7.1 Delimited Continuations

The actual Prolog interface to delimited continuations is as follows: The built-in predicate `shift(D)` corresponds to $shift_P\ t$. The $reset_P$ operation has signature `reset(P,K,D)`, which corresponds more or less to

Haskell	Prolog
Left ()	`K = 0`
Right (*S d k*)	`K =` $[\![k]\!]$`, D =` $[\![d]\!]$

Table 5.2: Interpretation of delimited continuations in Prolog.

$$\begin{array}{l} \mathit{reset}_P :: M\ () \to M\ (\mathit{Either}\ ()\ (\mathit{Susp}\ f\ (M\ ()))) \\ \mathit{reset}_P\ p\ (\mathit{return} \circ \mathit{Left})\ (\mathit{return} \circ \mathit{Right}) \end{array}$$

The input argument is the computation `P` and the other two arguments encode in an untyped way the output, as is shown in Table 5.2.

5.7.2 The **`entwine/2`** Infrastructure

With the help of the delimited continuation primitives, we can implement the infrastructure for `entwine/2` in Prolog, which corresponds to the ($\curlyvee$) operation.

Since Prolog does not allow us to override the disjunction primitive `(;)/2`, we are forced to use a new name, `or/2`, for expressing its syntactic form.

```
or(G1, G2) :- shift(or(G1, G2)).
```

Prolog's plain disjunction `(;)/2` remains available, allowing programmers to choose between disjunction that can be observed by our framework, and that which cannot. Capturing and handling of syntactic disjunctions is implemented with `reset/3`.

```
step(G, Pattern, Handler) :-
  reset(G, K, D),
  ( K = 0
  -> true
  ;  D = or(G1, G2),
     Pattern = or((G1, K), (G2, K)),
     call(Handler)
  ).
```

This enables a straightforward port of the ($\curlyvee$) implementation:

```
entwine(G1, G2) :-
  step(G1, or(GL1, GR1),
    step(G2, or(GL2, GR2),
      or(entwine(GL1, GL2), entwine(GR1, GR2))
    )
  ).
```

Finally, the reflection of toplevel syntactic disjunctions into Prolog's original disjunction is handled by `semOr/1`:

```
semOr(G) :-
  step(G, or(G1, G2), (semOr(G1) ; semOr(G2))).
```

While the meta-interpreter must cater for all features in the languages, this delimited continuations-based approach is nicely orthogonal to other features. The code is substantially shorter and clearly requires less maintenance. Moreover, even though raw performance is not the main objective, this approach is almost 3 times faster than the meta-interpreter on search intensive code that does not use ($\curlyvee$).

A small caveat is in order: In our Haskell model we expect that the following property holds for $p :: \mathit{MonadProlog}\ m \Rightarrow m\ a$:

$$p = \mathit{semOr}\ p$$

The Prolog equivalent of this statement is only true if the programmer avoids calling `or/2` inside a small subset of Prolog's control operations like Prolog's special `catch/3`. Fortunately, this requirement is generally not a heavy burden when solving search problems.

5.7.3 Search Heuristics

Now that we implemented the entwining infrastructure in Prolog, it is possible to define well-known search heuristics in the same concise and high-level style as in Haskell. As an example, the following Prolog code implements the depth-bounded search heuristic:

```
dbs(Depth,Goal) :- entwine(Goal,dbs(Depth)).

dbs(Depth) :-
  Depth > 0,
  Depth1 is Depth - 1,
  ( dbs(Depth1) or dbs(Depth1) ).
```

We have also implemented other heuristics such as discrepancy-bounded, node-bounded and failure-bounded search, as well as branch-and-bound, limited discrepancy search, and iterative deepening.[2]

As we have remarked in Section 5.5.4, not all search heuristics can be expressed in terms of `entwine/2`. Fortunately, we can still write custom handlers. One such handler is `muddle/1`:

[2]`http://users.ugent.be/~bdsouter/heuristics.html`

```
muddle(G) :-
  step(G, or(GL, GR),
    ( random(2) > 0
    ->  or(muddle(GL),muddle(GR))
    ;   or(muddle(GR),muddle(GL))
    )).
```

5.7.4 Multi-Way Disjunctions

Multi-way disjunctions are useful to express for instance that all the alternatives generated by a call to `select/3` are at the same level in the search tree and thus should be treated equally by depth-bounded search.

With the effect handlers approach, multi-way disjunction can easily be expressed as a generalization of binary disjunctions: the multi-way disjunction predicate `mor/1` takes a list of goals rather than two goals.

```
mor(Gs) :- shift(mor(Gs)).

mstep(G, Pattern, Handler) :-
  reset(G, K, D),
  ( K = 0
  ->  true
  ;   D = mor(Gs),
      maplist(extend(K),Gs,EGs),
      Pattern = mor(EGs),
      call(Handler)
  ).

extend(K,G,(G,K)).
```

A multi-way disjunction can be interpreted in terms of Prolog's binary disjunction.

```
semMor(G) :- mstep(G, mor(Gs), alts(Gs)).

alts([G|Gs])  :-  (G ; alts(Gs)).
```

However, first we can apply heuristics, like depth-bounded search. While it is not obvious how to extend `entwine/2` to multi-way disjunctions, it is easy enough to write regular handlers.

```
mdbs(DB,G) :-
  mstep(G,mor(Gs),
    (DB > 0,
        NDB is DB - 1,
        maplist(mdbs_rec(NDB),Gs,NGs),
        mor(NGs))).

mdbs_rec(DB,G,mdbs(DB,G)).
```

5.8 Related Work

5.8.1 Search

FP Models of LP Spivey's algebraic model of logic programming's combinatorial search [147] is very similar to *MonadProlog*. The model was first described by Seres et al. [142] as a way to allow both depth-first and breadth first strategies.

It has long been known that Prolog-like features can be embedded in Haskell using monads and monad transformers. For instance, Hinze [64] provides the equivalent of *MonadProlog* as well as a pruning primitive *once*. We can implement this using ($\curlyvee$).

Hinze [65] has also derived a backtracking monad transformer using the techniques of term and context passing. Both are systematic ways to derive a program implementation from its specification. The technique thus builds on the laws one imposes on the monad at hand to eliminate the need for a deus ex machina.

Kiselyov et al. [84] derive two implementations of a backtracking monad transformer. The first manages continuations explicitly, while the second does this implicitly using delimited control operations. Unlike our work, their monad transformer provides several extra operations, among which are fair conjunctions and disjunctions, and allows selecting an arbitrary number of answers.

Erwig [44] compares Prolog and Haskell-style approaches to solving search problems. He argues that the Haskell style (which comprises lazy evaluation, static typing and multi-parameter type classes) is better suited. However, search heuristics are not discussed.

Functional Logic Programming Typically, Functional Logic Programming (FLP) systems support nondeterminism in the same way as Prolog, with a fixed depth-first search strategy. In order to provide more flexibility, various FLP researchers [15, 93] have investigated *encapsulated search*. Encapsulated search reifies the search tree of a nondeterministic computation in a datastructure similar to $Or^{\star}_{m}$. This reified tree can be explored by programmer-supplied search strategies instead of the default depth-first search.

Given the tree-based interface of FLP encapsulated search, it is a perfect platform for the ideas of this paper: the declarative definition of search heuristics as archetypal search trees, and the modular composition of search trees with the $(\curlyvee)$ operator.

Constraint Programming The constraint logic programming libraries of many Prolog systems [17, 175, 41, 62] provide search heuristics that offer limited reusability: they are hardwired in a generic labelling predicate that can be used to solve particular classes of problems. The one exception is the branch-and-bound heuristic of ECLiPSe [133], which is not tied to a labelling predicate.

Schrijvers et al. [137] present Monadic Constraint Programming, a monadic model of Constraint Programming. This model features an explicit search tree datastructure that is manipulated by search heuristics. Compositionality of search heuristics is achieved by defining them in terms of a set of hooks. This approach is more complex and operational in nature than the one in this chapter, which makes it harder for the programmer to define new heuristics and reason about the behaviour of compositions. The hook-based approach is further explored in C++ [139] ("search combinators") and Prolog [134] settings, where it suffers from similar problems.

MiniSearch [119] is also a combinator based meta-search language that acknowledges the significant engineering effort in [139]. This complexity is because the original search combinators approach interacts with the solver at every node. MiniSearch is designed with minimal requirements on the solvers: any FlatZinc [103] solver will do, although an emulation layer for incrementally adding constraints and variables and for emulating interprocess communication is then needed. To minimize solver requirements, MiniSearch only interacts with the solver at every solution. It cannot express heuristics like limited discrepancy search and iterative deepening that can be expressed in our approach. Our approach provides access to the entire search tree and provides flexibility via lightweight delimited continuations. Similar to our approach, variable and value ordering heuristics cannot be expressed. The considerable improvements achieved with MiniSearch for solution quality within a two minute time limit once again demonstrate the importance of being able to specify custom search heuristics flexibly.

Nordin and Tolmach [106] describe a lazy functional framework for solving constraint satisfaction problems. As in our approach, it is straightforward to express and combine algorithms to prune the search space, using both fixed and dynamic variable ordering. They note an imperative implementation of several combinations of these algorithms is known to be tricky. However, they stress the importance of being able to experiment with them, since the best combination of features tends to depend on the particular problem.

Continuations The explicit use and manipulation of continuations in continuation passing style programs for implementing search is folklore. In the late 1980's, Felleisen [46] and Danvy & Filinski [31] independently proposed operators for delimited continuations in direct style programs. The latter is the reset/shift approach we have adopted in this article, which has a simple static interpretation in terms of continuations.

The CP language *Comet* [164] is a particularly interesting application of this technique: it features fully programmable search [165] based on continuations that make it easy to capture the state of the solver and write non-deterministic code.

5.8.2 Algebraic Effect Handlers

Plotkin and Pretnar [110] have introduced the concept of handlers for algebraic effects as a generalization of exception handlers. Their approach applies handlers on the free monad. Based on this idea, two entirely new programming languages, Frank [96] and Eff [9], have been created from the ground up around algebraic effect handlers; in these languages the computation monad is implicit.

More recently, three proposals show how to implement algebraic effect handlers on top of existing functional programming languages: Kiselyov et al. [83] provide a Haskell implementation in terms of the free monad, in combination with the codensity transformer to obtain better performance for $(\gg=)$. Brady [13] provides a layered implementation: a syntactic monad is interpreted into what is essentially the delimited continuation-based approach of Section 5.6. Finally, underneath it all is an arbitrary monad m; while Brady only uses this underlying monad in the handler definitions, his handler infrastructure is in fact a monad transformer. Kammar et al. [76] present several different implementations in Haskell, OCaml, SML and Racket. For Haskell, the free monad and a continuation-based approach are considered. For the other languages, the delimited continuation approach is taken.

5.8.3 Monads

Monadic Zip The literature covers a number of *zip*-like monadic operations similar to our $(\curlyvee)$: Giorgidze et al. [52] introduce a monadic zip operator *mzip* to support parallel monad comprehensions, a generalization of parallel list comprehensions. Their *mzip* is subject to two laws: it must have a partial inverse *munzip*, and it must be associative.

As part of their *Joinad* concept, Petricek et al. [109] define monadic operations similar to ours, including a monadic *zip*. However, the laws associated with their operations make them different in important ways from ours. Notably, while our $(\curlyvee)$ and $(|||)$ commute, for *Joinad*s the *zip* operator left-distributes over *or*.

Monad Laws Gibbons and Hinze [51] promote reasoning about code that is polymorphic in the monad by means of laws, which is the starting point of this paper. They illustrate law-based reasoning on several related monadic effects: failure, nondeterministic choice and probabilistic choice.

Free Monad Transformer The free monad (transformer) is also known by various other names, emphasizing different aspects: coroutine monad [10], resumption monad [107] and step monad [71]. The coroutine aspects is very relevant in our setting: in essence, the $(\curlyvee)$ operation treats two searches as coroutines that are synchronized at corresponding occurrences of $(|||)$.

5.9 Conclusion

This chapter has exploited the synergy between two declarative paradigms to tackle a challenging problem in logic programming with functional programming techniques. First it has shown how to cleanly separate logic and search heuristics in a functional model of Prolog by means of effect handlers and the free monad transformer. Then it has derived an actual Prolog implementation from this functional specification.

We are keen to use effect handlers to further extend Prolog with control operations that interact with the ones presented in this work. Of particular interest is Spivey's *wrap* [147] that groups multiple binary disjunctions into a single multi-way disjunction.

References

This chapter is based on our article on Entwining Heuristics: Tom Schrijvers, Nicolas Wu, Benoit Desouter, and Bart Demoen. Heuristics Entwined with Handlers Combined: from Functional Specification to Logic Programming Implementation. Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP), 2014. Benoit Desouter assisted in writing both the final as well as early versions of the article, with special focus on the introduction and motivation sections. He was responsible for a study of related work.

Chapter 6

Introduction to Tabled Resolution

In Chapter 4, we have introduced delimited control and we have shown how it can be used to flexibly add new language constructs. The language constructs we have added so far are meant to be used in rule bodies and as such, do not change the evaluation order of an entire set of rules. This kind of flexibility and how to obtain it, is the topic of this and the following chapter.

6.1 Problems with SLD-resolution

As mentioned in Chapter 2, the standard control mechanism offered by SLD-resolution, has several advantages. However, it has also severe restrictions:

1. it may go into an infinite loop if the input data contains cycles;

2. it may go into an infinite loop if clauses contain left recursion;

3. it is easy to construct programs where subgoals are repeatedly evaluated. For large inputs, this leads to a combinatorial explosion of the number subgoals to explore.

We now discuss each of those problems in more detail.

Cyclic Input Data Consider a trivial reachability problem with a cycle in the input data:

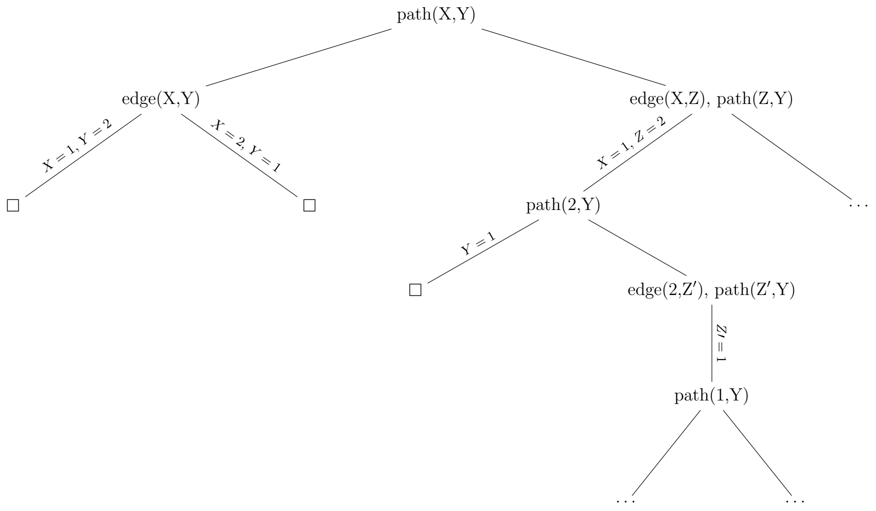


Figure 6.1: SLD-resolution gets trapped due to cyclic input data.

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).

edge(1,2).
edge(2,1).
```

For the query `path(X,Y)`, SLD-resolution repeats the correct answer substitutions an infinite number of times. We illustrate this in Figure 6.1. Thus due to the cycle, the interpreter gets trapped.

Left Recursion Consider the following formulation of the reachability problem:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z), path(Z,Y).

edge(1,2).
edge(2,3).
```

Although for this trivial problem, no cycles occur in the input data, standard SLD-resolution goes into an infinite loop[1] after finding the correct solu-

[1]i.e. until the virtual machine runs out of stack space.

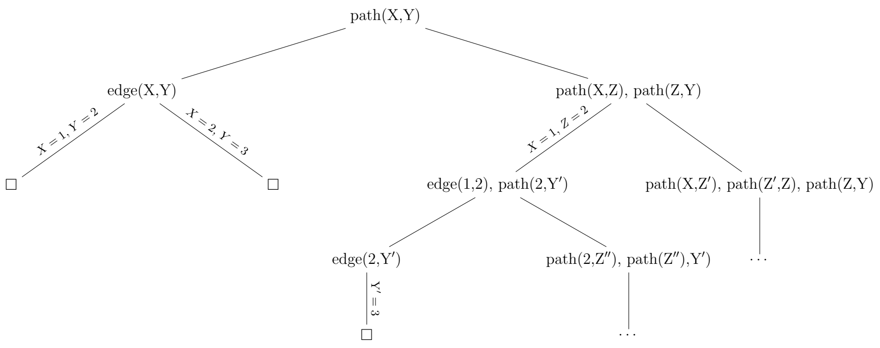


Figure 6.2: SLD-resolution gets trapped due to left recursion.

tions for the query `path(X,Y)`. This is because the second rule is left recursive: the rule is defined in terms of itself where the head occurs as the first body goal. We illustrate this in Figure 6.2. When we swap the first and second rule, no answers are ever found before the occurrence of the loop. Left-recursion often occurs in problem domains like parsing [100].

In general, nontermination for certain programs occurs no matter what the *static* computation strategy is. Under a static computation strategy, the decision which clause to use next for resolution, does not depend on any run-time properties of the program, but is fixed beforehand. We need a strategy that is *dynamic* in nature [53].

Combinatorial Explosion To illustrate the possibility of combinatorial explosion in standard SLD-resolution, it suffices to write a Prolog program that calculates the well-known Fibonacci numbers:

```
fibo(0, 1).
fibo(1, 1).
fibo(N, F) :-
  N > 1,
  N1 is N - 1,
  N2 is N - 2,
  fib(N1, F1),
  fib(N2, F2),
  F is F1 + F2.
```

It is easy to see that during the calculation of the Fibonacci number F_n, the Fibonacci number F_{n-1} starts by calculating F_{n-2}, which is repeated immediately afterwards. This phenomenon is repeated for F_{n-2} and smaller instances. A complexity analysis shows that this algorithm has exponential complexity.

Indeed, in imperative programming, this problem has long been solved by dynamic programming. Even in Prolog, it is not very difficult to write a bottom-up version using the database manipulation predicate *assert*/1. Yet from a declarative point of view, the programmer should not be bothered with these control issues.

6.2 Denotational Semantics

Logicians have extensively studied the semantics of logic programming. Of interest for this thesis is the denotational semantics [163, 143].

Consider a definite clause program P (no negations anywhere). The Herbrand base H_P of P is the set of ground atoms in P. A Herbrand interpretation states which ground atoms from the Herbrand base are true and which are false. By convention, we simply indicate the atoms that are true in the Herbrand interpretation:

$$\forall a \in \mathrm{H}_P : I \models a \iff a \in I$$

A model of a program P is an interpretation of P in which every formula in the program is true.

Example 5. *Consider the following program P:*

```
p(a).
p(b).
q(X) :- p(X).
```

Then the Herbrand base is $\{p(a), p(b), q(a), q(b)\}$. *Two possible Herbrand interpretations are* $\{p(a), p(b)\}$ *and* $\{p(a), q(a)\}$.

Every program P has an immediate consequence operator $\mathrm{T_P}$ that maps Herbrand interpretations to Herbrand interpretations. This operator is defined as:

$$\mathrm{T_P} = \{\alpha \in \mathrm{H}_P \mid \alpha \leftarrow B_1, \ldots, B_n \text{ is a ground instance of a clause in } P \wedge \{B_1, \ldots, B_n\} \subseteq I\}$$

where I is an interpretation of P. This means that α follows directly from the given interpretation I by some rule in the program.

An interpretation I is a fixpoint of $\mathrm{T_P}$ iff $\mathrm{T_P}\,(I) = I$. The conventional denotational semantics for P is the unique interpretation I that is the least fixpoint of $\mathrm{T_P}$, also known as the least Herbrand model of the program (a result due to van Emden and Kowalski [163]). This interpretation contains those and only those atoms that follow from the program and that are not self-supported.

It can be shown that the least fixed-point of $\mathrm{T_P}$ is $\mathrm{T_P}{\uparrow}^{\omega}$ where $\mathrm{T_P}{\uparrow}^{\omega}$ is defined as:

$$\begin{aligned}
\mathrm{T_P}{\uparrow}^{0} &= \varnothing \\
\mathrm{T_P}{\uparrow}^{1} &= \mathrm{T_P}\left(\mathrm{T_P}{\uparrow}^{0}\right) \\
\mathrm{T_P}{\uparrow}^{2} &= \mathrm{T_P}\left(\mathrm{T_P}{\uparrow}^{1}\right) \\
&\dots \\
\mathrm{T_P}{\uparrow}^{\omega} &= \bigcup_{n \geqslant 0} \mathrm{T_P}{\uparrow}^{n}
\end{aligned}$$

This definition suggests a naive bottom-up evaluation strategy, which is used in an improved semi-naive form by Datalog systems. However, this strategy is impractical for query answering in the general Prolog setting. Firstly, compound terms give rise to both an infinite Herbrand model and an infinite least Herbrand model which cannot be practically computed. Secondly, the bottom-up strategy can be overly expensive because it derives more facts than necessary for answering the query at hand.

Hence, Prolog uses the top-down strategy of SLD resolution, essentially based on $\mathrm{T_P}^{-1}$ to reason backwards from the query and only consider relevant facts. Unfortunately, this backwards chaining strategy easily gets trapped in cyclic derivations.

6.3 Tabling and SLG-resolution

In order to alleviate the problems of SLD-resolution, D. S. Warren has introduced SLG-resolution [20]. The overall idea is called tabled evaluation or tabling, and now has many innovative uses in areas like model checking [114], program analysis and non-monotonic reasoning [53].

Tabling combines the efficiency of top-down SLD resolution with the cycle-insensitivity of a bottom-up least fixed-point computation. Its backbone is top-down resolution, but paired with active cycle detection. Tabling replaces infinite cycles with a forward-chaining least fixed-point strategy, not unlike the immediate consequence operator $\mathrm{T_P}$, but switches back to top-down resolution for previously unexplored queries.

Like in the bottom-up strategy tabling comes at the cost of storing the answers to intermediate queries. To mitigate this cost, most systems use SLD resolution by default and allow the programmer to enable tabling for individual predicates. This is often accomplished by a declaration of the following form:

```
:- table p/n.
```

During the evaluation of a query, several call variants of a tabled predicate are likely to occur. We precisely define the call variant notion below; for now it suffices to say that two terms are considered a variant if they are identical up to variable renaming. The idea of tabling is to keep a list of previously calculated answers for each call variant. When trying to prove a query of call variant v it is first checked whether the answers are not present in a table associated with v before the resolution starts using program clauses.

Tabling ensures termination for Horn clause programs. In several cases tabling selected predicates has a drastic influence on time complexity, sometimes a drop from exponential in some parameter n to polynomial in n. From the more theoretical point of view, tabling offers a consistent declarative and procedural semantics that is lacking from standard Prolog [53].

It is however not wise to table just any predicate, not only because of the memory usage, but also because it may be incompatible with the intended semantics. As a simple example, it would give very counterintuitive results to table a predicate that has side effects, because the side effects would be executed only at the moment of tabling [153].

Terminology An important notion is variance: informally, two terms are considered a variant of each other if they are identical up to variable renaming. For example, $p(1, X)$ and $p(1, Y)$ are variants of each other, while $p(1, X)$ and $p(2, X)$ are not.

A formal definition is given by the numbervars bijection [115] that standardises the representation of terms by representing each variable as a unique constant. For example, $\text{numbervars}(p(X, q(X, Y))) = p(\mathcal{V}_1, q(\mathcal{V}_1, \mathcal{V}_2))$.

The following definitions are based on the description in [128]:

Generator The generator is the first occurrence of a tabled subgoal. For a generator, the engine uses program clauses to derive answers for that subgoal.

Consumer A subgoal that was encountered before. The engine then does not use program clauses, but consumes answers from the table associated with this subgoal. It needs to be suspended when it has exhausted all answers currently in the table, and resumed when new answers have been derived. A generator also needs to act as a consumer of its table.

Scheduling component A set of subgoals that are possibly interdependent. This set has a unique leader.

Leader A leader is a generator subgoal G_L that depends on no subgoal older than itself. A goal A depends on another goal B if the answers to goal B influence the answers to goal A. Subgoals younger than G_L are allowed to depend on G_L. The leader schedules consumers in its scheduling component according to a particular scheduling strategy, and determines completion of a scheduling component. Scheduling strategies are discussed in Section 6.7.

Freeze register A freeze register is used to freeze a stack by setting the contents of the register to the current top of the stack. This is done so that the the execution state of suspended computations can be reconstructed. As a consequence, the frozen space will not be reclaimed until completion of the associated table.

Consumer choicepoint This type of choicepoint represents the suspended environment. It saves the state of several WAM-registers.

Forward trail Addresses and values used to restore variable bindings along the path to the suspended consumer.

With the above definitions, we have all the ingredients to describe how an SLG-engine suspends a consumer. SLG-resolution is based on the older OLDT-resolution ([156]), but adds support for negation. It is not our intention to provide a deep understanding of SLG-resolution, but rather to give the reader a general idea of how this resolution works. It corresponds to maintaining a forest of SLD-stacks [55]. The engine

1. freezes all the stacks by means of the freeze registers;
2. creates a consumer choicepoint;
3. fails, which causes backtracking to the previous choicepoint. Here, the freeze registers must not be reset.

Resuming a customer uses the information in the consumer choicepoint, in particular the forward trail to restore variable bindings along the path to the suspended consumer.

In the literature (e.g. [128]), five operations of an SLG-engine are distinguished:

New Subgoal For a given tabled subgoal G, this operation checks whether this is the first occurrence of G or not.

Program Clause Resolution This operation corresponds to ordinary Prolog resolution. It is used for generators.

New Answer Records answers in the table associated with G.

Answer Return As an alternative to program clause resolution, consumers use the answers from the table by means of this operation. Until completion, a tabled engine needs to be able to reconstruct the execution environments E of G so that the newly derived answers returned by ANSWER RETURN can be run in those environments E, potentially leading to more new answers.

Completion This operation determines whether all answers for a given set of subgoals are stored in the tables. To accomplish this, it needs to examine the dependencies between tables and the answers consumed so far. An SLG-engine maintains a stack of scheduling components to this end.

6.4 Implementation Challenges

Since its conception in the early nineties, tabling has only been implemented in a few Prolog systems, most notably in XSB [152]. Other well-known implementations are the Yap [130], Ciao [62] and B-Prolog [175] engines.

Each has its own particular ways of implementing tabling, but all the implementations have one thing in common: implementing tabling is hard.

Sagonas and Stuckey have very nicely summarised the core problems [128]:

> *...the generation and consumption of answers are asynchronous and interleaved events.*

Implementing tabling requires pervasive changes to the Prolog engine. Specifically for SLD-resolution, the changes required to the WAM are:

- the introduction of a set of freeze registers;
- the introduction of a forward trail stack;
- the introduction of a new memory area, the table space;
- changes to the garbage collector;

- new primitive operations;
- as a consequence of changing the number of memory areas, the adaptation of the tagging scheme.

These changes are not orthogonal to an existing SLD-engine, but deeply influence its architecture. Thus it should not surprise that, despite tabling being around for quite some time and despite its desirability, it has only been implemented in systems developed in the context of academia.

6.5 Data Structures

The data structures used for representing the table itself, have been extensively studied. Two approaches can be distinguished: the most common one is to use tries; an alternative is the use of hash tables, as in B-Prolog [180].

Tries A trie is a particular kind of tree where the number of children per node is variable across nodes and across time. Each entry is split into parts. The entries are stored in the trie so that the common prefixes of their parts are on a common path from the root.

Tries were originally defined to index dictionaries [115]. They provide complete discrimination for terms.

This datastructure allows for insertions and containment checks with good algorithmic complexity: both operations have complexity $\mathcal{O}(d)$ where d is the depth of the call pattern.

Example 6. *Suppose we have an answer `p(1,a(b))` for a call variant of `p(1, _)`. Then the path through a trie will be* ROOT*, 1, `a/1`, `b/0`. We obtain the trie from Figure 6.3 by adding the following answers to that trie: `p(1,a)`, `p(1,a(c))`, `p(1,a(b(c)))`, `p(1,a(b(d)))`.*

In the context of tabling, there are two usage scenarios for tries [115]:

Answer check / insert Each call variant is associated with its own trie for storing answers. This corresponds to the New Answer operation from Section 6.3.

Call check / insert There is a single trie that keeps the relation between call variants and their respective tables.

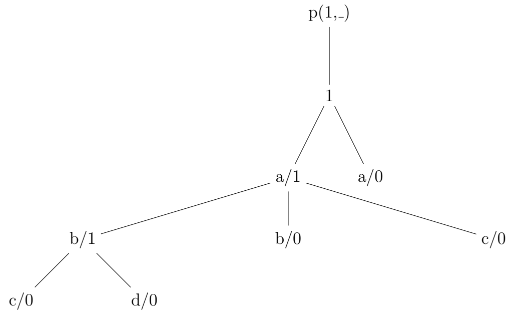


Figure 6.3: The structure of a trie: common prefixes are on a common path from the root.

Substitution factoring Substitution factoring (alternative: unification factoring) is a technique developed to avoid storing redundant data in tries, developed by Ramakrishnan et al. [115].

Given the table for a call variant G, all answers A in that table can be represented by means of an answer substitution θ_A as $G\theta_A$. Thus it suffices to only store the substitutions. Supposing that $\{V_1, V_2, \dots, V_m\}$ are the variables in G, then an answer substitution $G\theta_A$ has the following form:

$$\{V_1 \mapsto t_1, V_2 \mapsto t_2, \dots V_m \mapsto t_m\}$$

The sequence $\langle t_1, t_2, \dots t_m \rangle$ can easily be stored in a trie. The insertion and lookup operations then have a time complexity that is $\mathcal{O}(\|G\theta_A\|)$.

Implementing Tries in the WAM For efficiency, tries can be dynamically compiled to WAM instructions, so that looking up an answer in the trie constructs that answer on the stack.

6.6 Operational Problems

Common problems with tabling implementations are:

1. They add overhead for the evaluation of nontabled predicates, among the order of 10% [38]. This has been addressed by the CAT and CHAT approaches, discussed in Subsection 6.9.1.

2. Excessive memory consumption, due to:
    - the need to represent tables. Despite their good time complexity, standard tries are not particularly memory-friendly.
    - specifically for SLG-resolution and depending on the scheduling strategy (explained in the next section): stack space, because an arbitrary number of stacks may be frozen.
3. Incomplete tables, because in several applications not the answers to, but only the satisfiability of a given query are of interest. In that case it makes sense to stop the query evaluation as soon as an answer is found. However, this pruning operation results in incomplete tables. These tables cannot be trusted, because part of the computation has not been executed, and are typically thrown away. This is not only a memory problem, but it may also lead to many redundant computations, which we wanted to avoid in the first place.

Memory problems are recently becoming more of an issue [159, 113], because applications are made with increasingly large data sets. For example, Rocha et al. have applied tabling to inductive logic programming [123]. In this application domain, huge search spaces are explored and the number of tables gets too large for the available memory. In a tabling implementation, it is common to have a set of primitives to dynamically abolish some tables, but it is up to the programmer to use them. As it is difficult to make a good guess at what the least useful tables are without any extra information Rocha [121, 122] has introduced a memory management strategy based on a least recently used algorithm (LRU-strategy). In the same articles, Rocha proposes to keep incomplete tables around and to restart the evaluation from the beginning once the answers in the tables have been consumed. This idea bears some similarities with the work of Sagonas and Stuckey [128], discussed in Subsection 6.9.2 below. In this context, Sagonas and Stuckey have criticised attempts to integrate the Prolog cut-operator into SLG-resolution, as proposed in [54, 179].

6.7 Scheduling Strategies

A scheduling strategy determines the order of resuming suspended computations. Traditionally, two different strategies are distinguished [128, 32]:

batched scheduling In batched scheduling, a generator immediately consumes answers when they have been produced. This strategy thus returns answers one by one before computing them all, which in principle is better if only one answer or a subset of the answers is desired.

local scheduling In local scheduling, a generator postpones the consumption of generated answers until they have all been generated. Thus it collects all the solutions to a tabled predicate before making any one of them available outside the tabled computation.

The pros and cons of both scheduling strategies have been nicely summarised in [32]:

> *Batched evaluation is closer to SLD-evaluation in that it computes solutions lazily as they are demanded, but it may need arbitrarily more memory than local evaluation, which is able to reclaim memory sooner.*

In this article, de Guzmán, Carro and Warren propose swapping evaluation as a third strategy. Similar to batched evaluation, it returns answers one by one, but at a lower memory usage.

The importance of flexibly supporting multiple scheduling strategies is stressed by Rao et al. [117]. Depending on the scheduling strategy, tabled logic programs can show substantial differences in performance. The authors show that an optimal strategy unfortunately does not exist. Thus, tabling strategies can only improve specific classes of applications. Implementations should flexibly support multiple strategies.

6.8 Semantic Issues

Tabling does not go well with several Prolog constructs: cut, negation and aggregation predicates. Many papers ignore this, and only handle the class of definite programs.

The problem with cuts is the following [152]: suppose a goal G is called in two places and suppose the first of those two places has a cut. Then this cut might remove a choicepoint for an incomplete table. This could disallow the derivation of answers in the second place, leading to incompleteness[2].

Recent work attempts to provide pruning without relying on Prolog-style cut. One example is the JET-approach discussed in the next section. Chico de Guzmán, Carro and Hermenegildo [23] provide a recent overview of pruning benefits, its issues and mention five previous attempts.

[2]The best practical advice thus is not to use cuts in combination with tabling.

Negation has two problems: floundering and nonstratification [154]. The negation operator must only be applied to ground literals. In the other case, the program is said to “flounder”. Nonstratification means that a program has a set of rules with cycles through negation.

In addition, side effects must be used with care: a side effect will only be performed the first time that a subgoal is called.

6.9 Alternative Lowlevel Mechanisms

We already explained that implementing tabling is complex. In addition, the underlying idea leaves plenty of choice on how to implement different aspects. Finally, SLG-resolution is so specific that it is hard to reuse some aspects of its implementation in the context of other languages like functional or functional logic languages. Thus, over the years, a lot of alternative implementation mechanisms have been developed.

6.9.1 CAT and CHAT

The CAT is an alternative to the SLG-WAM used in XSB [38]. Rather than freezing memory areas, CAT uses incremental copies to preserve the execution state of suspended computations. The advantage of this approach is that the speed of the underlying abstract machine is not affected for nontabled execution. However, in the worst case, CAT must copy so much that its tabled execution becomes arbitrarily worse than that of the SLG-WAM. CHAT is an improved hybrid scheme incorporating some ideas from the SLG-WAM [39], but also respecting the important property that the execution speed of nontabled evaluation should not be affected. The scheme implements the suspension mechanism using a combination of freezing and copying to a separate memory area. CAT and CHAT do require changes to the WAM, but acknowledge that the complexity and scope of these changes should be kept limited. Demoen and Sagonas themselves consider CHAT’s design superior to CAT.

6.9.2 JET

The operational problems with incomplete tables and pruning discussed in Section 6.6 are the main motivation for the JET approach. JET allows suspension and resumption at an arbitrary point and, by means of a static analysis, detects points where pruning can occur without leading to recomputation. These points are called JET-points.

For example, in the following rule, taken directly from [128], and assuming batched scheduling, pruning could occur at the program points marked with ^:

```
    test :- start(S), t(S), tle(S,G), good(G).
                          ^                   ^
```

If the pruning is implemented like a Prolog cut, the only safe solution is to throw away the incomplete tables. JET saves these computations and allows them to be executed later. The implementation is based on CHAT. JET makes tabled evaluation much more demand-driven, and as such, we consider it an important step forward.

De Guzmán, Carro and Warren [32] mention that their swapping evaluation scheduling strategy was prefigured in the context of JET.

6.9.3 Recomputing Approaches

SLG's good performance comes from its suspension mechanism that avoids recomputation (explained below). However, this is also the hardest part in the implementation. Several approaches cleverly avoid the need to suspend.

To distinguish between tabled resolution strategies that do and do not perform recomputation, Sagonas and Stuckey have introduced the "proper tabling" terminology for strategies that do *not* perform recomputation [128].

Linear Tabling Mechanisms Linear tabling mechanisms [179], which implement the SLDT-resolution strategy, maintain a single execution tree. Instead of suspending, the approach steals choicepoints from a former variant call. Linear tabling is implemented in B-Prolog [178].

SLDT-resolution works in the same way as standard SLD-resolution, except when the call is a variant of some former call. In that case, SLDT uses the answers already available in the table. If the table is exhausted, the remaining clauses of the former call are tried, which is referred to as "stealing the choicepoint". Subgoals thus always extend the current computation, and no suspension is created. But this is not enough: after exhausting all answers and clauses, the call must be re-executed from the first clause of the predicate until no new answers can be derived. This re-execution is generally referred to as *recomputation*.

Each tabled call can thus be both a producer and a consumer. As an important property, there is no overhead for standard SLD-resolution, but the need for recomputation of subgoals cannot always be avoided. In some cases recomputation can be avoided with the Direct-Recursion Optimisation. Unlike for suspension-based mechanisms, the cut operator works for a class of useful programs.

Although simpler than SLG resolution, implementing SLDT still requires the addition of 4 new specifically designed WAM-instructions, a new frame structure and a new data area. As a drawback, the obtained efficiency is limited.

Dynamic Reordering of Alternatives Dynamic Reordering of Alternatives (DRA) [53, 55] also implements tabled evaluation without complex operations like stack-freezing. It postpones clauses containing variant calls at runtime, which is similar to suspension creation. Guo and Gupta have to this end introduced six new WAM instructions. The DRA-scheme has been developed with parallelism for logic programming in mind. Compared to XSB, Guo and Gupta's implementation of DRA has a significantly better space performance — due to a potentially large number of stacks/heaps that may be frozen in XSB, but a worse time performance. The authors cite as sources for XSB's better time performance that XSB avoids reconstructing the execution environment for applying looping alternatives, and secondly that XSB includes tabling in the compiling stage.

6.10 Transformation-based Approaches

6.10.1 Extension Tables

Back in 1987, S. W. Dietrich described extension tables in her PhD thesis [42]. In a later article [45] these tables were described as "a memo facility that the algorithm uses both to cut infinite derivation paths for complete evaluation and to optimise the evaluation of logic programs". Indeed, extension tables provide a lightweight implementation of tabling where the control flow is defined entirely by means of program transformation and database manipulations.

This approach cannot achieve satisfactory performance as suspended goals are always re-evaluated. The initial implementation used the `assert` and `retract` predicates for database manipulations. These predicates may be slow depending on term size and structure. A later version moved the data structures to C, but did not change the inherent recomputation behaviour.

6.10.2 Recomputation-Free Approaches

Ramesh and Chen [116] extend Prolog with new tabling primitives implemented in C through the foreign function interface. They are the first to aim for a *portable* method of integrating SLG-resolution into Prolog systems. A complicated program transformation introduces calls to these C routines at the appropriate points in tabled predicates. A limitation of their proposal is that it does not allow arbitrary interleaving of tabled and nontabled predicates [21]: except for the first call, tabled calls must appear in the body of a tabled predicate for their implementation to work correctly. Chico de Guzmán et al. remark that tabling all predicates between generators and consumers works around this problem, but that this can seriously impact efficiency.

More recently, Chico de Guzmán et. al. [24] have addressed the performance bottlenecks of Ramesh and Chen's approach. But while their improvement is successful in terms of performance and has good scaling characteristics, it does require lower-level C primitives, changes to the WAM's memory management, and an even more complicated program transformation. These changes further increase the cost of porting and maintaining the mechanism, while the development effort cannot be amortised over other features. Hence, the approach does not lower the threshold for adopting tabling.

Bridge Transformation Chico de Guzmán et al. have also presented an enhanced transformation removing the limitation that calls to tabled predicates must occur in the body of tabled predicates [21, 22]. They introduce the notion of bridge predicates: a predicate `B` is a bridge if for some tabled predicate `T`, `T` depends on `B`. The enhancement makes the transformation even more involved: a deep analysis of both tabled and bridge predicates is necessary and some code is duplicated. Due to the approximation used, some predicates may be unnecessarily marked as bridge predicates, which produces an additional average slowdown of three percent.

6.11 Call Subsumption

Traditionally, tabling implementations have been based on variant tabling. However, over the years, several alternatives have emerged that allow for more answer sharing. With variant-based tabling, an engine only reuses answers for a past goal that is a variant of the current goal (that is, identical modulo variable renaming) [72]. Subsumption-based tabling allows for more reuse: a goal G' subsuming a goal G contains in its table all answers for G. However not all answers for G' may be applicable, as G' is more general. However the mechanisms are more complex than the ones used in a variant-engine, hence more difficult to implement [28]. Increasing the level of abstraction will obviously make this task easier.

Johnson et al. [72] propose time-stamped tries that allow efficiently selecting the relevant answers for a subsumed goal G. The table space used in a time-stamped trie remains within a constant factor of the space used by a variant-based engine, but the engine is likely to create more choicepoints. Time-stamped tries are clearly superior to subsumptive tabling engines based on the older DTSA [118].

Cruz and Rocha [28] describe *retroactive* subsumption that allows sharing between subsumptive subgoals independent of their call-order. In a later paper ([29]), they propose a variation on the time-stamped trie: the Single Time Stamped Trie is tailored for retroactive subsumption. Although they report some overhead in programs stressing the drawbacks — when retroactive subsumption is not applicable, the simplicity of their design remains attractive.

6.12 Tabling in other Contexts

Haskell In Haskell, the traditional approach to deal with nondeterminism is to model computations by the list of their outcomes. However, this does not work when nondeterministic problems are recursive as the recursion may cause the lists to become infinite. Vandenbroucke et al. [168, 169] define a model that is able to deal with both non-determinism and recursion. They give a semantics to this model in terms of a least fixpoint computation, which is exactly what tabling does for the immediate consequence operator in logic programming.

Mercury Mercury is a pure logic programming language for the creation of large programs. The syntax is similar to Prolog, but semantically, the language is very different.

The main design criterium for tabling in Mercury was that tabled evaluation should not impact the execution speed of nontabled predicates [146]. Therefore the implementation borrows heavily from the CAT approach (discussed in Subsection 6.9.1). Somogyi and Sagonas state: "CAT is simply the tabling mechanism requiring the fewest, most isolated, changes to the Mercury implementation.". In particular, maximum performance for the tabled predicates themselves was not a design requirement. Chico de Guzman et al. [24] have stressed the technical differences in the implementation because of the differences in the base language, although both Mercury and their work implement tabling using external modules and program transformation, so as to keep the changes to the compiler and runtime system minimal.

Still, adding tabling to Mercury is hard and has only been implemented for one of several backends. The major reason for this complexity is because if-then-else and existential quantification are often used in Mercury. The corresponding constructs in Prolog are cuts and negation. In Section 6.8, we have explained that the interaction between these constructs and tabling are notoriously tricky. The Mercury approach partly motivated the JET approach [128].

The problem with existential quantification is that it discards stack frames once a witness is found. The discarded stack frames may include the frames of a generator, which leads to incomplete tables. Mercury has a new type of stack, the cut stack, that can detect this situation. In this case, the existence of noncomplete generators is erased, so that the table can be recomputed.

The problem with if-then-else is that in a tabling context, failure of the condition does not not imply that the condition is not satisfiable: some goal in the condition may have been suspended. By using a new type of stack, the possibly-negated-context stack, this situation can be detected. Rather can computing incorrect results, an exception can be thrown, but, to our knowledge, no solution to compute the correct answers has been proposed yet.

Picat Picat [176] is being promoted as a successor for Prolog. The language lacks Prolog's non-logical features, such as the cut operator and dynamic predicates. The Picat tabling system implements a linear tabling mechanism. This has been exploited to provide a framework for solving planning problems [8].

Tabling and Parallelism Logic programming is well-suited for automatic parallelization: logic programs often contain multiple clauses for a rule. These alternatives may be explored in parallel.

Integration of or-parallelism and tabling has been demonstrated by the Yap Prolog system [130].

References

This chapter is partially based on the following articles:

- Benoit Desouter, Marko van Dooren, and Tom Schrijvers. Tabling as a Library with Delimited Control. Theory and Practice of Logic Programming (TPLP), 2015. Proceedings of the 31st International Conference on Logic Programming (ICLP).

- Benoit Desouter, Marko van Dooren, Tom Schrijvers, and Alexander Vandenbroucke. Tabling as a Library with Delimited Control. International Joint Conference on Artificial Intelligence (IJCAI), Sister Conferences Best Paper Track (on invitation), 2016.

Chapter 7

Tabling with Delimited Control

7.1 Introduction

Tabling is one of the most widely studied extensions to Prolog because it considerably raises the declarative nature of the language. Tabling takes away the sensitivity of SLD resolution to rule and goal ordering, and allows a larger class of programs to terminate. As an added bonus, the memoisation that is done by the tabling mechanism may drastically improve performance in exchange for more memory.

Given all these advantages, it may come as a surprise that many Prolog systems still do not support tabling. The reason is however simple. In the previous chapter we have studied existing implementation approaches. All of them require pervasive changes to the Prolog engine. This is a substantial engineering effort that is beyond most systems [130].

We have also seen that several works have attempted to tackle this problem. The approach of Ramesh and Chen (Subsection 6.10.2), improved by Chico de Guzmán et. al.. is successful in terms of performance, but it does require lower-level C primitives, changes to the WAM's memory management, and an extremely complicated program transformation. Those changes increase the cost of porting and maintaining the mechanism, and the development effort cannot be amortised over other features. The extension tables from Subsection 6.10.1 provide a tabling mechanism that is implemented directly in Prolog. However, the approach cannot achieve satisfactory performance as suspended goals are always re-evaluated.

Santos Costa et al. [130] point out that *"Making it easy to change and control Prolog execution in a flexible way is a fundamental challenge for Prolog."*. In Chapter 4, we have already seen that delimited control, as a language construct for manipulating a program's control flow, does exactly that. We have shown that the impact of delimited control on the WAM is minimal. On top of that, the development effort of delimited control can be amortized over the range of high-level language features they enable, such as effect handlers.

In this chapter we show how delimited control can be used for a lightweight tabling mechanism. Both the tabling control flow and data structures are written entirely in Prolog enhanced with delimited control. It does not require deep custom changes to the Prolog engine, complicated program transformations, or meta-interpretation. As such our mechanism demystifies many aspects of implementing tabling.

Compared to existing state-of-the-art systems, our system needs more attention in terms of performance, but this does not outweigh the gain in flexibility: we bring tabling much closer to the masses. In contrast with extension tables, our approach does not require recomputation of suspended goals.

7.2 Shallow Transformation

In our approach, tabled predicates require no special notation, nor any syntactic analysis of the predicates being tabled. Predicates are written in the usual way, and transformed by a shallow program transformation.

```
:- table p/2.

p(X,Y) :- p(X,Z), e(Z,Y).
p(X,Y) :- e(X,Y).
```

Figure 7.1: Running example: transitive closure.

The use of tabling is illustrated in Figure 7.1. Predicate `p/2` computes the transitive closure of the `e/2` relation. The `table`-directive indicates that `p/2` will be tabled. Predicates without that directive are resolved using standard SLD-resolution.

The `table/1` directive performs a very shallow program transformation, the result of which is shown in Figure 7.2. This transformation introduces a new predicate `p_aux/2`, which we call the *worker* predicate. Its clauses are literal copies of the original clauses of `p/2`. Only the name in the head of the

```
p(X,Y) :- table(p(X,Y),p_aux(X,Y)).

p_aux(X,Y) :- p(X,Z), e(Z,Y).
p_aux(X,Y) :- e(X,Y).
```

Figure 7.2: Result of the transformation.

```
table(Wrapper,Worker) :-
  get_table_for_variant(Wrapper,Table),
  table_get_status(Table,Status)
  ( Status = complete ->
    get_answer_from_table(Table,Wrapper)
  ;
    ( exists_scheduling_component ->
      run_leader(Wrapper,Worker,Table),
      get_answer_from_table(Table,Wrapper)
    ;
      run_follower(Status,Wrapper,Worker,Table)
    )
  ).
```

Figure 7.3: The `table/2` predicate.

predicate is changed. Everything else is preserved, including recursive calls to the tabled predicate.

The new predicate `p/2`, which we call the *wrapper* predicate, is defined in terms of the dynamic tabling predicate `table/2`. This predicate receives the corresponding wrapper and worker call patterns as arguments and takes care of tabling that call. We stress that `table/2` does not receive any information about the static structure of `p/2` and manages the tabling fully dynamically. We explain how `table/2` can be implemented directly in Prolog in Section 7.3.

7.3 The `table/2` Predicate

Thanks to the shallow program transformation, the `table/2` predicate intercepts every call to a tabled predicate. Figure 7.3 shows that `table/2` retrieves the `Table` data structure for the given `Wrapper` call pattern. There is one table for every distinct call pattern encountered so far; if the current call pattern has not been encountered before, `get_table_for_variant/2` allocates

```
run_leader(Wrapper,Worker,Table) :-
  create_scheduling_component,
  activate(Wrapper,Worker,Table),
  completion,
  unset_scheduling_component.
```

Figure 7.4: Handling the leader call.

a fresh data structure for it.

Then `table/2` switches on the `Table`'s status. If the status is `complete`, it means that all answers for the `Wrapper` call pattern are already available in the table. The call is then answered by consuming the answers with the `get_answer_from_table/2` predicate.

Otherwise, we either start collecting answers (`run_leader/3`), or we are already in the process of collecting answers and proceed (`run_follower/4`). The call that initiates answer collection is called the *leader*. A leader is a call to a tabled predicate that has only non-tabled ancestors in the dynamic call graph. Other calls to tabled predicates during answer collection are called *followers*. Every follower has a leader as its ancestor. The leader and its followers make up a *scheduling component*. Multiple scheduling components can occur during program execution.

Example 7. *Consider the top-level call `?- p(X,Y).` for our running example. Then `p(X,Y)` clearly is the leader of a new scheduling component. The recursive call `p(X,Z)` in the first clause constitutes a follower in its scheduling component.*

The Leader The leader, defined in Figure 7.4, takes responsibility for computing all the answers of its scheduling component. To quickly identify whether there currently is a leader, we use a global non-backtrackable variable, that is checked by `exists_scheduling_component/0`. Similarly, the simple predicates `create_scheduling_component/0` and `unset_scheduling_component/0` set and unset the variable.

The job of the leader consists of two tasks: 1) it starts computing the answers of the scheduling component with `activate/3`, and 2) it computes the least fixpoint for the whole scheduling component with `completion/0`.

Followers Followers, defined in Figure 7.5, have fewer responsibilities than the leader. If the table of the follower is `fresh`, i.e. it is the first time the call

```
run_follower(fresh,Wrapper,Worker,Table) :-
  activate(Wrapper,Worker,Table),
  shift(call_info(Wrapper,Table)).

run_follower(active,Wrapper,Worker,Table) :-
  shift(call_info(Wrapper,Table)).
```

Figure 7.5: Handling a follower call.

pattern occurs, then the follower `activate`s the answer computation. Subsequently, it yields control with `shift/1`; this is explained in more detail in the next subsection. If the table is already `active`ly collecting answers, the follower immediately yields control.

7.4 Activation and Delimited Answer Computation

When a call pattern is encountered for the first time, the computation of its answers is activated with the predicate `activate/3`. This predicate, defined in Figure 7.6, alters the table status from `fresh`ly allocated to `active` and puts the `Worker` to work with the auxiliary `delim/3` predicate. Note that a failure driven loop is used to backtrack over all the alternatives of `Worker`.

```
activate(Wrapper,Worker,Table) :-
  table_set_status(Table,active),
  (
    delim(Wrapper,Worker,Table),
    fail
  ;
    true
  ).
```

Figure 7.6: Activation.

The body of a tabled predicate p/n is actually executed by predicate `delim/3`, defined in Figure 7.7. This predicate runs p/n's `Worker` in the context of a `reset/3`. If the `Worker` succeeds normally, the answer is added to the table with `store_answer/2`.

```
delim(Wrapper,Worker,Table) :-
   reset(Worker,Continuation,SourceCall),
   ( Continuation == 0 ->
     store_answer(Table,Wrapper)
   ;
     SourceCall = call_info(_,SourceTable),
     TargetCall = call_info(Wrapper,Table),
     Dependency = dependency(SourceCall,Continuation,TargetCall),
     store_dependency(SourceTable,Dependency)
   ).
```

Figure 7.7: Delimited execution.

However, if the `Worker` calls a tabled predicate q/m — with either the same or a different call pattern as p/n — then `Worker` does not terminate normally. The reason is that the q/m call is a follower, and `run_follower/4` always ends in a `shift/1` without producing an answer. Instead the `Worker` suspends, capturing the remainder in `Continuation`.

Example 8. *Consider the following clause from our running example:*

```
p_aux(X,Y) :- p(X,Z), e(Z,Y).
```

The worker `p_aux(X,Y)` for the call `p(X,Y)` immediately suspends at the recursive call `p(X,Z)` with `Continuation = e(Z,Y)`.

Through this suspension, we bypass the regular depth-first execution mechanism of Prolog and avoid its potential non-termination. We replace the depth-first search by the least fixpoint computation of the `completion` phase. For this purpose, we record the suspended computation in the form of a `dependency/3` structure. This structure expresses that given an answer for the q/m call, one may obtain answers for the p/n call by resuming the suspended continuation. We denote q/m as the *source call* and p/n as the *target call.* For the source call, it is sufficient to store the `SourceTable` to be able to retrieve an answer later. For the target call, we need the `Wrapper` in addition to the table, as the `Wrapper` contains the partial answer that running the continuation will instantiate. This explains the form of the `dependency/3` structure, which is stored in the table of the source call to be triggered whenever a new answer is added.

Example 9. *The dependency for our example above expresses that, given an answer for `p(X,Z)`, we may obtain answers for `p(X,Y)` by executing `e(Z,Y)`.*

```
completion :-
  ( worklist_empty ->
      set_all_complete,
      cleanup_tables
  ;
      pop_worklist(Table),
      completion_step(Table),
      completion
  ).

 completion_step(SourceTable) :-
   (
     table_get_work(SourceTable,Answer,
       dependency(Source,Continuation,Target)),
     Source = call_info(Answer,_),
     Target = call_info(Wrapper,TargetTable),
     delim(Wrapper,Continuation,TargetTable),
     fail
   ;
     true
   ).
```

Figure 7.8: The completion fixpoint.

For instance, if we get the answer `X = a, Z = b` for `p(X,Z)`, and we have the fact `e(b,c)` then we obtain the answer `X = a, Y = c` for `p(X,Y)`.

Example 10. *Assume that `e/2` is defined by the facts `e(a,b)` and `e(b,c)`. Then the query `?- p(X,Y)` yields not only the dependency on `p(X,Z)` through the first clause of `p_aux/2` but also the answers `p(a,b)` and `p(b,c)` through the second clause of `p_aux/2`. Since `p(X,Z)` is a variant of `p(X,Y)`, the dependency and the two answers are all associated with the same table.*

7.5 Completion

The completion phase, defined in Figure 7.8, computes the fixpoint over all answers and dependencies of the scheduling component. Just like Datalog's semi-naive approach [19], our implementation tries to avoid unnecessary recomputation.

We maintain a worklist of all tables for which at least one associated answer has not been fed into at least one associated dependency. This worklist is updated whenever a new answer or new dependency is associated with a table.

Predicate `completion/0` is the driving loop of the completion phase. It repeatedly pops a table from the worklist and calls `completion_step/1` to process answer/dependency pairs that have not yet been combined. When the worklist is empty, the completion fixpoint has been reached. On reaching the fixpoint, `set_all_complete/0` sets the status of every table in the scheduling component to `complete`. Finally `cleanup_tables/0` erases all the dependencies, as they are no longer necessary.

By calling `table_get_work/3`, predicate `completion_step/1` retrieves an unprocessed `Answer`/`Dependency` pair from the table. It instantiates the source of the dependency with the answer and resumes the dependency's continuation with `delim/3`, binding the variables in the partial answer `Wrapper` along the way. This process may lead to new answers or new dependencies that spur the fixpoint computation on. Here, a failure-driven loop is used to iterate over all answer/dependency pairs.

Example 11. *Let us consider the completion that follows Example 10. There is one entry in the worklist: the table for call variant `p(X,Y)`. This table has two unprocessed pairs:*[1]

```
p(a,b) / dependency(p(X,Z),e(Z,Y),p(X,Y))
p(b,c) / dependency(p(X,Z),e(Z,Y),p(X,Y))
```

The first pair yields the new answer `p(a,c)` with the help of the fact `e(b,c)`. The second pair yields nothing. The production of a new answer reschedules the table for `p(X,Y)` in the worklist. Yet the second completion round yields no new answers or dependencies and the fixpoint computation terminates with answer set {`p(a,b)`, `p(b,c)`, `p(a,c)`} for call `p(X,Y)`.

7.5.1 The Table Data Structures

The central data structure used by the tabling control flow explained above is the *table*. We maintain one such table per call variant, which can be retrieved from a global repository of all tables. This global repository is implemented in the form of a trie data structure, also known as the call trie, that maps call patterns to tables.

There is a second global data structure, the global worklist, which maintains a simple queue of tables for the completion algorithm explained in Section 7.5.

[1]We have abbreviated the call information for the sake of clarity.

The table itself consists of two parts: the answer trie and the local worklist:

- The answer trie is where `get_answer_from_table/2` finds its answers. Moreover, the trie allows `store_answer/2` to quickly check whether a newly produced answer has already been computed before, and to only store it in case it has not.

- The local worklist serves the `table_get_work/3` predicate. It retrieves pairs of answers and dependencies that have not been combined before. For this purpose we use a dequeue (i.e., a double-ended queue) that contains answers and dependencies.

  The dequeue maintains the invariant that an answer is to the left of a dependency if and only if they have not been combined. New answers are added on the left, because they have not been combined with any dependency yet. New dependencies are added on the right.

  For performance reasons, the dequeue batches consecutive answers into a single entry on insertion; the same happens to consecutive dependencies. Every batch contains homogeneous elements (either answers or dependencies) and is implemented as a list — the position of the elements in the list is insignificant. Batches of the same type are not merged if they become adjacent during the combination of answers and dependencies. Doing so would reduce the number of swaps, but at the cost of merging the lists.

  The `table_get_work/3` predicate retrieves a batch of answers immediately to the left of a batch of dependencies, swaps their positions and yields the elements of their Cartesian products for processing. Dependencies and answers that are created by the combination are also sent to the appropriate tables. A single step of this process is illustrated in Figure 7.9. The solid arrow denotes the transformation of the local worklist. The wavy line denotes the emission of new answers and dependencies that are generated by the completion step. The answers in the gray ellipse have been added to the local worklist, and will eventually move to the right of all dependencies.

Implementation Support The key implementation support for these tables are mutable terms and non-backtrackable mutations, discussed in Section 2.4. We also use a global variable for the table repository. The non-backtrackable nature is essential to retain the collected answers and dependencies across disjunctions.

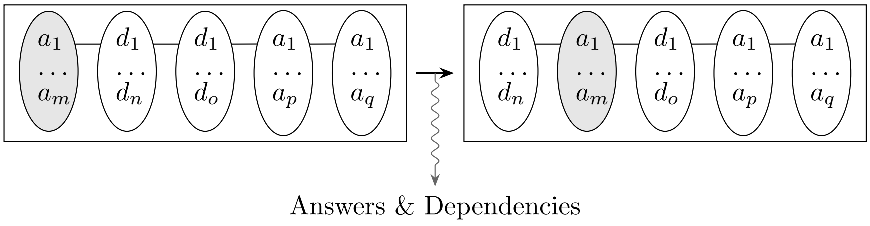


Figure 7.9: Combining answers and dependencies in a local worklist.

7.5.2 Completion of a Double Recursive Call

Example 12. *Consider a variant of our running example where the recursive clause is replaced by:*

```
r(X,Y) :- r(X,Z), r(Z,Y).
```

Figure 7.10 illustrates the computation of `?- r(a,Y)`*. Each table is a rectangle. The consecutive states of its worklist are shown from top to bottom. A dotted arrow shows the target of a dependency. The solid and wavy lines are as in Figure 7.9. In the explanation, the labels of the completion steps in the figure are written between parentheses. The call* `?- r(a,Y).` *gives rise to the dependency* `D1 = dependency(r(a,Z),r(Z,Y),r(a,Y))` *and the answer* `r(a,b)` *(left rectangle).*

Iteration 1 *In the first iteration of completion (1), the answer is fed into the dependency (wavy arrow α), hence* `D1` *and* `r(a,b)` *are swapped. This exposes the call* `r(b,Y)` *(middle rectangle). For this new call we immediately obtain the dependency* `D2 = dependency(r(b,Z1),r(Z1,Y),r(b,Y))` *and the answer* `r(b,c)`*. We also record* `D3 = dependency(r(b,Y),true,r(a,Y))` *between* `r(b,Y)` *and* `r(a,Y)`*. The* `true` *in* `D3` *represents the empty continuation: finding an answer for* `r(b,Y)` *gives an answer for* `r(a,Y)` *for free!*

Iteration 2 *During the second iteration, we feed the answer* `r(b,c)` *into the two dependencies* `D2` *(2.a, wavy arrow β) and* `D3` *(2.b, wavy arrow γ).*

β *In the* `D2` *case, we expose a new call* `r(c,Y)` *(right rectangle) yielding no direct answer. We obtain* `D4 = dependency(r(c,Z2),r(Z2,Y),r(c,Y))` *and a derived dependency* `D5 = dependency(r(c,Y),true,r(b,Y))` *instead.*

γ *In the* `D3` *case, we obtain the new answer* `r(a,c)` *for the top-level call.*

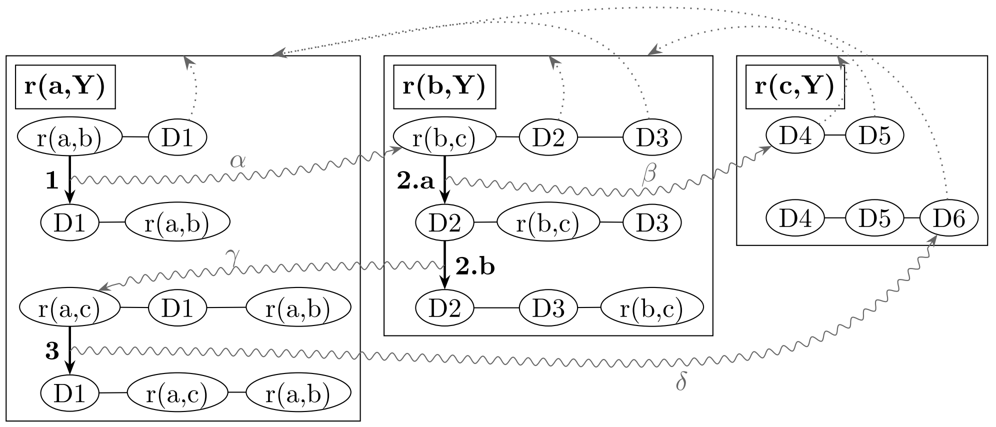


Figure 7.10: Illustration of the computation of `r(a,Y)`.

Iteration 3 *During the third iteration (3), we feed the new answer into dependency* `D1` *(wavy arrow δ). This yields the call* `r(c,Y)` *and the dependency* `D6 = dependency(r(c,Y),true,r(a,Y))`.

The fixpoint *Finally, there is no more work to be done: at the bottom of each rectangle, all Di are left of all answers. Hence, the fixpoint comprises the answer table* {`r(a,b)`, `r(a,c)`} *for the call pattern* `r(a,Y)`, *the answer table* {`r(b,c)`} *for the call pattern* `r(b,Y)` *and the empty answer table for* `r(c,Y)`.

7.6 Completion Details

This section gives more implementation details of the completion phase discussed in the previous section.

To get answers and dependencies that must be combined from the local worklist, we first extract the worklist from the table, and in preparation, set a flag indicating that we are busy working. The actual work is delegated to `table_get_work_/3`.

```
table_get_work(Table,_Answer,_Dependency) :-
  get_worklist(Table,Worklist),
  set_flag_executing_all_work(Worklist),
  table_get_work_(Worklist,Answer,Dependency).
```

In `table_get_work_/3`, we once more delegate the work nondeterministically, but once an answer/dependency pair is extracted from the local worklist, we copy the dependency to ensure that the original version is not modified. When the first rule eventually fails, the work in this local worklist is done for now, so we unset the flag.

```
table_get_work_(Worklist,Answer,Dependency) :-
  worklist_do_all_work(Worklist,Answer,Dependency0),
  copy_term(Dependency0,Dependency).
table_get_work_(Worklist,_Answer,_Dependency) :-
  unset_flag_executing_all_work(Worklist), fail.
```

The predicate extracting answer/dependency pairs executes a single step, and when this eventually fails, recursively calls itself unless all the work is done.

```
worklist_do_all_work(Worklist,Answer,Dependency) :-
  ( worklist_work_done(Worklist) ->
    fail
  ;
```

```
      worklist_do_step(Worklist,Answer,Dependency)
    ;
      worklist_do_all_work(Worklist,Answer,Dependency)
    ).
```

The job of the local worklist is done for now if the pointer to the cluster of answers that should be combined with dependencies, points to a dummy value. Alternatively, the work is done if there is no dependency cluster, in which case the next entry in the underlying double linked list representation of the worklist points to the dummy value.

```
worklist_work_done(Worklist) :-
  wkl_get_rightmost_inner_answer_cluster_pointer(Worklist,
    RiacPointer),
  ( wkl_is_dummy_pointer(Worklist,RiacPointer) -> true
  ;
    dll_get_pointer_to_next(RiacPointer,NextPointer),
    wkl_is_dummy_pointer(Worklist,NextPointer)
  ).
```

Taking the Cartesian product of an answer and a dependency cluster happens by first swapping the clusters in local worklist. Next, the pointers to these clusters are dereferenced by the underlying double linked list representation. Finally, one answer and one dependency are nondeterministically yielded for combination.

```
worklist_do_step(Worklist,Answer,Dependency) :-
  wkl_get_rightmost_inner_answer_cluster_pointer(Worklist,ACP),
  wkl_swap_answer_continuation(Worklist,ACP,SCP),
  dll_get_data(ACP, wkl_answer_cluster(AList)),
  dll_get_data(SCP, wkl_suspension_cluster(SList)),
  member(Answer,AList), member(Dependency,SList).
```

Swapping clusters is propagated to the underlying double linked list representation. Afterwards, the pointer to the answer cluster that must be swapped next, must be updated to the new location of the cluster.

```
wkl_swap_answer_continuation(Worklist,ACP,SCP) :-
  dll_get_pointer_to_next(ACP,SCP),
  dll_swap_adjacent_elements_(ACP,SCP),
  wkl_update_righmost_inner_answer_cluster_pointer(Worklist,ACP).
```

Updating the pointer to the answer cluster that must be swapped next, is a no-op if the cluster currently pointed to still has to be combined with dependency clusters, and hence can still propagate to the right. Otherwise, a new cluster is found by walking back in the underlying double linked list representation.

```
wkl_update_righmost_inner_answer_cluster_pointer(Worklist,ACP) :-
  ( wkl_answer_cluster_moved_completely(Worklist,ACP) ->
    wkl_find_new_rightmost_inner_answer_cluster_pointer(Worklist,
      ACP,ACP2),
    wkl_set_rightmost_inner_answer_cluster_pointer(Worklist,ACP2)
  ;
    true
  ).
```

7.7 Evaluation

7.7.1 Implementation Effort

Table 7.1 summarizes the implementation effort in lines of Prolog (LoC). The control flow shown in this chapter comprises 60 LoC, or less than 11% of the overall effort. The majority goes to the two kinds of data structures, the tries (40%) and the worklists (45%). Adding 25 lines of glue code, this amounts to an implementation for 577 Prolog LoC.

7.7.2 Performance

While raw efficiency is not the main objective of our lightweight implementation, it is nevertheless important to achieve a reasonable performance compared to the existing state-of-the-art tabling systems. In order to evaluate this, we compare our implementation in hProlog 3.2.38 against XSB 3.4.0 [152], B-Prolog8.1 [175], Yap 6.3.4 [130] and Ciao 1.15-2731-g3749edd [62] on a number of benchmarks.[2] Table 7.2 summarizes the results (for hProlog in ms, the others as a proportion) obtained on a Dell PowerEdge R410 server (2.4

[2]The description and code of the benchmarks can be found at `http://users.ugent.be/~bdsouter/tabling/`.

Category	**LoC**	**Category**	**LoC**
Control flow	60	Completion Worklists	259
Call and Answer Tries	233	Miscellaneous	25
Total	577		

Table 7.1: Code size in lines of code.

Benchmark	**Size**	**hProlog**		$\frac{\textbf{hProlog}}{\textbf{XSB}}$		$\frac{\textbf{hProlog}}{\textbf{B−Prolog}}$	$\frac{\textbf{hProlog}}{\textbf{Yap}}$	$\frac{\textbf{hProlog}}{\textbf{Ciao}}$
fib[a]	500	24	(13)	O/F	(—)	∞	∞	—
	750	33	(13)	O/F	(—)	17	41	—
	1,000	46	(13)	O/F	(—)	46	19	—
	10,000	982	(66)	O/F	(—)	3	44	—
recognize[a]	20,000	205	(73)	26	(1)	0.003	11	4
	50,000	503	(221)	30	(2)	0.001	14	4
n-reverse[a]	500	767	(138)	38	(5)	11	15	45
	1,000	2,800	(537)	31	(6)	6	8	34
shuttle[b]	2,000	44	(12)	∞	(2)	0.1	∞	9
	5,000	138	(14)	23	(2)	0.08	∞	12
	20,000	582	(29)	24	(4)	0.02	∞	10
	50,000	1,586	(72)	29	(6)	0.01	∞	12
ping pong	10,000	271	(16)	45	(2)	0.07	∞	14
	20,000	490	(28)	35	(4)	0.03	∞	8
path double first loop	50	653	(14)	19	(2)	13	∞	7
	100	4,638	(29)	17	(4)	10	∞	6
path double first	50	162	(12)	27	(2)	15	∞	14
	100	989	(16)	20	(3)	12	∞	10
	200	6,785	(53)	18	(7)	16	∞	10
	500	110,463	(267)	25	(14)	19	∞	14
path right last: pyramid 500	500	1,914	(104)	35	(7)	29	∞	27
path right last: binary tree 18	18	108,662	(4,120)	78	(5)	50	3,461	42
test large joins 2[c]	12	3,001	(237)	10	(5)	4	∞	12
joins mondial		6,444	(399)	8	(2)	7	224	6

Table 7.2: Results of the performance benchmarks: for hProlog in ms; the others as a proportion. In parentheses: maximum RSS, for hProlog in MB; XSB as a proportion.

Source: *a* [45] *b* [38] *c* Yap benchmark suite

GHz, 32 GB RAM) running Debian 7.6. In parentheses, we have indicated the maximum resident set size (RSS) in megabytes and the proportion of hProlog to XSB. The figures in the table are averages over five runs.

Discussion The XSB system is the reference system for tabling; it has invested most time and resources in the development of its tabling infrastructure. We see that it is 8 to 38 times faster than our implementation, but 45 to 78 times faster for two outliers (path right last: binary tree 18 and 10k ping-pong). It has a maximum RSS that is up to 7 times as large, and 14 times for path double first 500. In general, standard trie-based structures overload the memory because representation sharing is poor. This has been addressed by Raimundo and Rocha [113].

Since XSB does not support big integers, it was not meaningful to run the Fibonacci benchmark, recorded as O/F (for overflow). This is a case in point for wider tabling support in other systems: often we need both tabling and other non-standard features.

B-Prolog is only half as fast as XSB on many benchmarks, but is architecturally different: B-Prolog implements linear tabling and uses hash tables instead of tries. Moreover, in several cases B-Prolog is notably slower than XSB (i.e., n-reverse) and even much slower than our own implementation (recognize, shuttle, ping pong). Yet, unlike XSB, B-Prolog does support big integers and is substantially faster than our approach for the fib benchmark. All in all the results are mixed and point out several weaknesses in the B-Prolog implementation compared to our all Prolog implementation.

The Yap tabling implementation, which is based on that of XSB, is clearly the fastest: the underlying engine is much faster [125]. It outperforms our approach on all benchmarks, and the other systems on most. Many benchmarks take less than 1 ms, rounded down to 0 ms, hence the factor ∞ in the table.

The performance of Ciao lies between that of XSB and B-Prolog. Performance of our implementation is within a factor 4 to 14 of Ciao, with reverse and path right last as outliers. Running the Fibonacci benchmarks is currently not possible, as tabling and bignums currently do not operate together[3].

Summary We consider the performance results of our implementation very reasonable, especially if we take into account the stark contrast between our lightweight pure Prolog implementation and the complex integration in other systems. As part of future work, we think that advances in three areas may positively affect performance. Firstly, continuations are copied with

[3]Personal email communication with Manuel Carro.

`copy_term/2`. A special-purpose `copy_continuation/2` could do better by exploiting the known structure of these terms. Other applications using delimited control could benefit from this optimization as well. Secondly, we don't statically identify strongly connected components in the scheduling component. Doing so would allow the specialisation of completion. Finally, in contrast with state-of-the art implementations, our tries do not use substitution factoring.

7.7.3 Comparison with CHAT

We now compare the amount of work that must be performed for creating a suspension using delimited control to the work that must be performed in the Copy-Hybrid Approach to Tabling (CHAT) [39]. In this subsection, we assume familiarity with the WAM [171, 4], as summarised in Section 2.1 on page 11, as well as some familiarity with SLG-terminology, as explained in Section 6.3 on page 145.

Three different situations are of interest because they have a nontrivial cost: the creation of a suspension, its resumption, and the completion of a scheduling component. We discuss these situations in this order.

Creation of a suspension Without loss of generality, we assume a four stack WAM, i.e. the choicepoints and environments have their own separate stack[4]. On encountering a consumer of an arbitrary non-completed generator, both CHAT and our approach have to take action on some of the four stacks in the WAM:

Local stack To create a suspension with delimited control, the local stack must be traversed to capture the continuation of the call that triggered the suspension. The continuation is stored on the heap. The distance of the local stack that must be traversed depends on the continuation. Theoretically, this can be arbitrarily large, but is usually limited. Next, the stack is unwound over the same distance. How the live environment variables are determined depends on the specific implementation (see Section 4.5 on page 79): either at compile time or by code scanning.

Choicepoint stack In CHAT, a copy of the consumer choicepoint needs to be made in a CHAT-specific memory area. Additionally, all choicepoints between the consumer and its generator (inclusive) need to have their top of heap and top of local stack fields updated. The amount of work

[4]The other two stacks are the heap, also known as global stack, and the local stack.

depends on the distance between the consumer and its generator, but can be done in an incremental fashion. No such changes are needed in the delimited control implementation.

Trail stack In CHAT, a selective copy of the trail between consumer and generator needs to be made, together with the values pointed to by these trail entries. The amount of work once more depends on the distance between the consumer and its generator. There will be one trail entry per dynamical unification of a logical variable. No such copies need to be made in the delimited control implementation.

To summarize: the efficiency of CHAT depends on the choice point and trail stack distances between the consumer and its generator. For our tabling approach based on delimited control, the efficiency depends on the distance of the local stack that must be traversed, which depends on the continuation. It is hard to predict which distance will be the smallest as this greatly depends on the specific program.

Resumption of a suspension In CHAT, reinstalling a single consumer requires work involving each of the four WAM stacks:

Heap The top of heap must be adapted — an $\mathcal{O}(1)$-operation. No such change is needed in the delimited control implementation.

Local stack The top of the local stack must be adapted. This is again an $\mathcal{O}(1)$-operation. In our approach based on delimited control, a new environment must be created and the slots must be filled in with the live variables saved in the continuation. This operation is linear in the size of the continuation.

Choicepoint stack The consumer choicepoint from the CHAT-specific memory area must be copied back to the choicepoint stack. No such change is needed in the delimited control implementation.

Trail stack The trail is reinstalled by copying it from the CHAT-specific memory area to the trail stack and reinstalling the saved bindings. No change to the trail stack is needed in the delimited control implementation. However this operation serves the same purpose as the creation of a new environment on the local stack and filling it in with the live variables from the continuation. It also has the same complexity.

In summary: the amount of work to resume a suspension is similar for the CHAT and delimited control-based approaches.

Completion of a scheduling component In CHAT, on completion of a scheduling component, the CHAT-specific memory areas of all consumers of that component are freed. In our approach based on delimited control, all references to the continuations of the component are invalidated, the garbage collector can reclaim the space used for these continuations. Both tasks are similar. Additionally, in CHAT, the generator choice point of a leader is popped from the choicepoint stack, which is a constant time operation

In summary: the amount of work on completion of a scheduling component is similar for the CHAT and delimited control-based approaches.

7.8 Related Work

We have provided an thorough overview of existing tabling systems, their strengths and weaknesses in the previous chapter. So in this section, we limit ourselves to a discussing of delimited control in Prolog.

While delimited control is well-known in the functional programming world, it has not received much attention in the context of Prolog. We are the first ones to provide an unobtrusive implementation in the WAM, as discussed in Chapter 4. In the continuation-passing implementation [158] of BinProlog [157] adding delimited control is even easier. We also illustrate the power of delimited control by porting various effect handlers [111] to Prolog. As far as we know, this chapter shows the first Prolog-specific application.

7.9 Conclusion

In order to enable a more widespread adoption of tabling, we have presented a lightweight implementation of tabling on top of delimited control. In contrast to existing approaches, our approach is implemented entirely in Prolog and requires no deep modifications to the WAM or complex program transformations. While there is obviously a trade-off between the simplicity of the implementation and runtime performance, we believe that the current performance of our approach is reasonable. Of course, there is ample opportunity for improvement. The library approach also makes it feasible to study tabling outside of its original context, for example in functional programming.

References

This chapter is based on our article on Tabling with Delimited Control: Benoit Desouter, Marko van Dooren, and Tom Schrijvers. Tabling as a Library with Delimited Control. Theory and Practice of Logic Programming (TPLP), 2015. Proceedings of the 31st International Conference on Logic Programming (ICLP). Benoit Desouter was responsible for the implementation of the tabling library, as well as experimentation with different variations. Benoit conducted the performance evaluation and a study of the related work (presented in the previous chapter). He has written and structured most of the article.

Chapter 8

A Mathematical Formalisation of Answer Subsumption

8.1 Introduction

The original definition of tabling only uses answer variance: an answer A is added to the table for a particular goal G if and only if A is not a variant of any other answer already in the table [151]. This takes all arguments into account and is formally defined by the numbervars bijection [115] that standardises the representation of terms by representing each variable as a unique constant. An engine then only reuses answers for a past goal that is a variant of the current goal.

Although tabling greatly raises the expressivity of the language, another level of declarativeness is achieved by adding answer subsumption. Using plain tabling, the user must explicitly write the control logic for selecting optimal answers. Answer subsumption eliminates this. We say that a goal G' subsumes a goal G if G' contains in its table all answers for G. For example, `p(a,_)` subsumes `p(a,b)`. However, not all answers for G' may be applicable. As already mentioned in Chapter 6, subsumption-based tabling allows for more reuse than variant-based tabling.

To illustrate the merits of answer subsumption consider the following tabled predicate `p/3` that computes the transitive closure of the `e/3`-relation with the distance between the nodes:

```
:- table p/3.                     e(a,b,1).
p(X,Y,D) :- e(X,Y,D).             e(b,c,1).
p(X,Z,D) :-                       e(a,c,1).
  p(X,Y,D1),
  p(Y,Z,D2),
  D is D1 + D2.
```

The query `?- p(X,Y,D).` computes a table with four answers:

```
p(a,b,1).                         p(a,c,1).
p(b,c,1).                         p(a,c,2).
```

For this tiny example, the resulting table is small. But for many real-world problems the size of the table quickly becomes prohibitively large. Supposing we are only interested in the shortest paths between two nodes, the table contains unnecessary information on the longer paths. Moreover, the programmer must write code to select the shortest path:

```
shortest(X,Y,MD) :-
  findall(D,p(X,Y,D),Ds),
  list_min(Ds,MD).
```

Answer subsumption avoids storing unnecessary information and raises the declarative level: its directives allow the user to specify the characteristics of optimal answers and retain only them. Three varieties of answer subsumption exist nowadays. Their applications are in the domain of dynamic programming, the implementation of paraconsistent, quantitative and preference logics, as well as abstract analysis domains [151, 56].

Partial Order Answer Subsumption In the above example, any answer was added to the table regardless of the existence of another answer in the table describing a shorter path between the same nodes. Partial order answer subsumption [151] does not do so: an answer A is added to the table only if A is greater than the other answers in the table with respect to a user-defined partial order function. In addition, any answers worse (that is: smaller than A with respect to the partial order) are deleted from the table.

Using the notation from [151], also used in XSB, the tabling declaration for our working example can be modified as follows:

```
:- table p(_,_,po(</2)).
```

The partial order `</2` is then defined so that only answers `p(A,B,C1)` and `p(A,B,C2)` are ordered with respect to each other, based on the value of their third argument. Paths between different nodes cannot be compared.

The declaration results in a table with only three answers:

```
p(a,b,1).                       p(a,c,1).
p(b,c,1).
```

Partial order answer subsumption thus models an intuitive notion of preference.

Lattice Answer Subsumption Lattice answer subsumption [151] does not simply add an answer A satisfying some criterion: A is joined with another answer $A\prime$ and this join replaces $A\prime$ in the table. The user has the freedom to choose the join function.

Again using the notation in [151] for our working example, we have:

```
:- table p(_,_,lattice(min/3)).

min(X,Y,Z) :-
  Z is min(X,Y).
```

From the notation, the operational meaning is not immediately clear: answers `p(A,B,C1)` and `p(A,B,C2)` are joined and only the join is stored in the table. For our working example, this results in the same table as with partial order subsumption.

From a mathematical point of view, lattice answer subsumption defines an equivalence relation between sets of answers and stores only the representative of each equivalence class. The join operation can then be seen as union modulo the equivalence relation.

Mode-directed Tabling Mode-directed tabling [56, 57, 177, 129] is another form of answer subsumption where the user specifies which arguments are distinguishing (+), which are irrelevant (- or @) and which must be aggregated (usually a mnemonic). The user specifies an aggregation operation which is used to combine the values of that argument across answers with the same distinguishing arguments.

Using mode-direction, the table declaration for our working example looks like:

```
:- table p(+,+,0).
```

A plus sign denotes an indexed (= distinguishing) argument. A minus sign denotes an argument where the first instance seen is stored, whereas with the @-sign all instances are stored. In the original paper by Guo and Gupta [56], only two aggregate operations were allowed: minimum (indicated by 0 or "min") and maximum (indicated by 9 or "max").

The problem with answer subsumption Using answer subsumption the query `?- p(a,c,D).` yields only the shortest distance. It does so by greedily throwing away non-optimal intermediate results and in this way only considers finitely many paths, even if the graph is cyclic. In summary, answer subsumption makes tabling (sometimes infinitely) more efficient for our aggregation use-case.

Unfortunately, none of the existing implementations that we are aware of is generally sound. Consider the following pure logic program.

```
p(0). p(1).
p(2) :- p(X), X = 1.
p(3) :- p(X), X = 0.
```

The query `?- p(X).` has a finite set of solutions, {`p(0)`,`p(1)`,`p(2)`,`p(3)`}, the largest of which is `p(3)`. However XSB, Yap and B-Prolog all yield different (invalid) solutions when answer subsumption is used to obtain the maximal value. Both XSB and B-Prolog yield `X = 2`, with a maximum lattice and `max` table mode respectively. Yap (also with `max` table mode) yields `X = 0; X = 1; X = 2`, every solution except the right one.

Clearly, these results are unsound. This example is not the only erroneous one; we can easily construct more erroneous scenarios with other supported forms of aggregation. Hence, we must conclude that answer subsumption is in general not a semantics-preserving optimisation. Yet, as far as we know, the existing literature does not offer any guidance on when the feature can be relied upon. In fact, to our knowledge, its semantics have not been formally discussed before. Every variety is defined in an informal way, by means of examples, sometimes supplemented by the changes needed to an SLG-based tabling engine. This lack of formal definition makes it hard to implement answer subsumption in tabling systems that do not implement a variety of SLG-resolution. Moreover, if a system should support multiple varieties, their relationship and possible interactions must be understood thoroughly.

As a first step towards a remedy for this situation, we propose a framework for answer subsumption. This framework provides a common view on all existing varieties. Differences between the varieties can be expressed formally in this framework.

We start with an overview of the necessary mathematical background (Section 8.2). The framework itself is discussed subsequently in Section 8.3. We show how the existing approaches are an instance of the framework in Section 8.4 and discuss related work in Section 8.5. Section 8.6 concludes. We assume familiarity with tabling (see for example [20]).

8.2 Background: Lattices

In this section we introduce the mathematical concepts needed for formally defining answer subsumption. We start with the concept of a partial order. Next we introduce the poset concept. In Subsection 8.2.3 we define the least upper and greatest lower bound. We finish with the introduction of the lattice concept. This section is based on the Lattice Tutorial by N. Jovanovic [75] and the work of Lloyd [91].

8.2.1 Partial Order

A partial order relation is a binary relation R over the domain D that has the following additional properties:

$$\forall x \in D.\, R(x,x) \tag{Reflexivity}$$

$$\forall (x,y) \in D \times D.\, R(x,y) \wedge R(y,x) \implies x = y \tag{Antisymmetry}$$

$$\forall (x,y,z) \in D^3.\, R(x,y) \wedge R(y,z) \implies R(x,z) \tag{Transitivity}$$

For example, the relation $\leqslant$ over $\mathbb{R}$ satisfies all these properties. Therefore $\leqslant$ is a partial order relation. We denote an arbitrary partial order relation with $\sqsubseteq$, and we denote $x \sqsubseteq y$ for $\sqsubseteq (x,y)$.

8.2.2 Partially Ordered Set

A partially ordered set (short: poset) is a set D over which a partial order $\sqsubseteq$ is defined. It is denoted as $(D, \sqsubseteq)$, and D is called the ground set. If there is an order for all pairs of elements in the set, that set is a totally ordered set:

$$\forall (x,y) \in D \times D.\, x \sqsubseteq y \vee y \sqsubseteq x \tag{Totality}$$

Example 13. *The set $\{1,2,3\}$ is a totally ordered set under $\leqslant$. Its powerset, $\mathcal{P}(\{1,2,3\})$ is partially ordered set under subset inclusion $\subseteq$. To see that this set is not totally ordered, consider the pair $(\{1\},\{2\})$. Neither $\{1\} \subseteq \{2\}$ nor $\{2\} \subseteq \{1\}$ holds.*

8.2.3 Least Bounds

Two elements x and y in a poset $(D, \sqsubseteq)$ have an upper bound u iff $x \sqsubseteq u \wedge y \sqsubseteq u$. An upper bound is not necessarily unique and is not necessarily different from the elements it is computed for.

We use the following notation for the set of upper bounds of a subset S of D:

$$\mathrm{ubs}_D(S) = \{u \in D \mid \forall s \in S.\, s \sqsubseteq u\}$$

The least upper bound (supremum or join) of a subset S of D is an upper bound $\bigsqcup_D(S)$ that satisfies:

$$\forall u \in \mathrm{ubs}_D(S).\, \bigsqcup_D(S) \sqsubseteq u$$

This means that the least upper bound precedes any other upper bound. Traditionally, the following notation is used for the least upper bound when S has only two elements: $\bigsqcup_D(\{x,y\}) = x \sqcup y$. Equivalently, the following property holds for the least upper bound:

$$\forall s \in S.\, x \sqcup y \sqsubseteq s \iff x \sqsubseteq s \wedge y \sqsubseteq s$$

By analogy, one defines the notions of lower bound and greatest lower bound. The greatest lower bound is also known as the infimum or meet and denoted with $\bigsqcap_D(S)$ for $S \subseteq D$ and $x \sqcap y$ for $\{x,y\} \subseteq D$.

In what follows, we leave out the subscripts in the notation for supremum and infimum whenever they are clear from the context.

8.2.4 Lattice

A lattice $\langle L, \bigsqcup_S, \bigsqcap_S \rangle$ is a poset $(L, \sqsubseteq)$ in which all *nonempty finite* subsets have both a supremum and an infimum.

Example 14. *The above definition of lattice may seem trivial. Therefore it is instructive to give an example of a poset that is not a lattice. Consider the following set of three elements (that are sets themselves): $\{\{a\},\{b\},\{a,b\}\}$ with the inclusion relation $\subseteq$. There is no greatest lower bound for the elements $\{a\}$ and $\{b\}$, hence this poset is not a lattice.*

A complete lattice is a poset in which *all* subsets have both a least upper bound and a greatest lower bound. Stated mathematically [168]: a complete lattice is a partially ordered set such that for every $S \subseteq L$:

$$\left(\exists \bigsqcup S \in L. \forall x \in L. \left(\bigsqcup S \sqsubseteq x \Leftrightarrow \forall s \in S. s \sqsubseteq x\right)\right) \wedge \\ \left(\exists \bigsqcap S \in L. \forall x \in L. \left(x \sqsubseteq \bigsqcap S \Leftrightarrow \forall s \in S. x \sqsubseteq s\right)\right) \quad (8.1)$$

In the literature, it is described that existence of all possible infinite joins entails the existence of all possible infinite meets, and vice versa. In addition, we do not rely on the presence of a greatest lower bound in what follows. Therefore we work with upper semi-lattices, for which is sufficient to require that for every $S \subseteq L$:

$$\exists \bigsqcup S \in L. \forall x \in L. \left(\bigsqcup S \sqsubseteq x \Leftrightarrow \forall s \in S. s \sqsubseteq x\right) \quad (8.2)$$

Properties of complete lattices Consider a complete lattice $\langle L, \bigsqcup, \bigsqcap \rangle$. Then the following properties hold:

- The least upper bound $\bigsqcup S$ is unique $\forall S \subseteq L$.
- There is always a least element (bottom element) $\bot_L$, which corresponds to the least upper bound of the empty set $\varnothing$: $\bot_L = \bigsqcup \varnothing$.

All lattices with a finite number of elements are always complete lattices. Every non-empty finite lattice is bounded from below by taking the meet of all elements:

$$\bot_L := \bigsqcap \{l \mid l \in L\}$$

To every (other) lattice, one can adjoin an artificial bottom element.

8.3 Formalisation

In this section we describe our formalisation of answer subsumption. It is based on the definitions given above.

Suppose that the answers for a tabled predicate belong to a set A. We say that the tabled predicate has type A. In traditional tabling, the table t stores facts belonging to this predicate, and thus has type $\mathcal{P}(A)$. The fixpoint computation deals with answers of type A and suspended computations of type $C := A \to \text{Maybe}\,(A)$.

There are two motivations for answer subsumption: (a) reduction in table size, and (b) easier modelling of dynamic programming problems. Therefore, in general, the table simply has type L, denoted as $t : L$ and contains a single abstract answer. We assume the existence of a map from call patterns to the correct table. The type L is not just any type: we require the existence of an upper semi-lattice $\langle L, \sqcup \rangle$ where $\sqcup$ denotes the join operation $\sqcup : L \to L \to L$. The definition of an upper semi-lattice requires that the join operation $\sqcup$ is a total function. We use the notation $\sqsubseteq$ for the lattice's partial order function.

Moreover, to ensure termination, the upper semi lattice $\langle L, \sqcup \rangle$ must be bounded: it must have a least element, traditionally called bottom and denoted by $\bot_L$. As explained above, this is not a problem in practice.

Under the above requirements we define the following join operation on sets of abstract answers:

$$\bigsqcup : \mathcal{P}\,(L) \to L$$

$$\bigsqcup \{l_1, l_2, \ldots, l_n\} := \bot_L \sqcup l_1 \sqcup l_2 \sqcup \ldots \sqcup l_n \quad (8.3)$$

Equation 8.3 relies on the associativity of the join operation. Indeed:

$$(x \sqcup y) \sqcup z \sqsubseteq w \iff (x \sqcup y), z \sqsubseteq w \iff x, y, z \sqsubseteq w \iff$$
$$x, (y \sqcup z) \sqsubseteq w \iff x \sqcup (y \sqcup z) \sqsubseteq w \quad (8.4)$$

Injection Function We derive values of the type L from a fact of type A using an injection function $\alpha\prime : A \to L$. We require that this function is total:

$$\text{Dom}(\alpha\prime) = A \quad (8.5)$$

We then define the following injection operator on sets of nonabstract answers:

$$\alpha : \mathcal{P}\,(A) \to L$$

$$\alpha\,(as) := \bigsqcup \{\alpha\prime\,(a) \mid a \in as\} \quad (8.6)$$

We require that this function is total. The image of $\varnothing_A$ under α follows from Equation 8.3:

$$\alpha\,(\varnothing_A) = \bot_L \quad (8.7)$$

Note that the type of the set injection operator is not $\mathcal{P}\,(A) \to \mathcal{P}\,(L)$. Also note that in general the set injection operator α is *not* injective: an element of L will typically be mapped to by many elements of $\mathcal{P}\,(A)$.

$$\begin{tikzcd}
\mathcal{P}(A) \arrow[dd, "\alpha"'] \arrow[dr, "\mathcal{P}_{\alpha'}"] & \\
& \mathcal{P}(L) \arrow[dl, "\bigsqcup"'] \\
L &
\end{tikzcd}$$

Figure 8.1: Commutative diagram relating α and $\bigsqcup$.

$$\begin{tikzcd}
\mathcal{P}(A) \times \mathcal{P}(A) \arrow[r, "\cup"] \arrow[d, "\alpha\times\alpha"] & \mathcal{P}(A) \arrow[d, "\alpha"] \\
L \times L \arrow[r, "\sqcup"] & L
\end{tikzcd}$$

Figure 8.2: Commutative diagram relating α, $\cup$ and $\sqcup$.

Consequence From the definition of $\bigsqcup$ follows the following continuity property of α:

$$\alpha\left(\{a_1 \ldots a_n\} \cup \{a_{n+1} \ldots a_m\}\right) = \alpha\left(\{a_1 \ldots a_n\}\right) \sqcup \alpha\left(\{a_{n+1} \ldots a_m\}\right) \quad (8.8)$$

It means that α maintains the least-upper-bound.

As can be seen from Figure 8.1, $\alpha = \bigsqcup \circ \mathcal{P}_{\alpha'}$ where $\mathcal{P}_{\alpha'}$ applies α' to each element in the set it is given. Moreover, $\alpha \circ \cup = \sqcup \circ (\alpha \times \alpha)$ (Figure 8.2).

Morphism From Property 8.8 and the presence of a bottom element, it follows that α is an upper semi-lattice morphism from $\langle \mathcal{P}(A), \cup, \varnothing_A \rangle$ to $\langle L, \sqcup, \bot_L \rangle$.

Induced Relations The partial order relation on L induces a binary relation on A, defined as:

$$\precsim : A \to A \to \text{Bool}$$

$$a_1 \precsim a_2 \coloneqq \alpha'(a_1) \sqsubseteq \alpha'(a_2) \quad (8.9)$$

It is important to note that this binary relation is *not* a partial order as the antisymmetry property is *not* fulfilled. Figure 8.3 makes this clear: $1 \precsim 2$ and $2 \precsim 1$, but $1 \neq 2$.

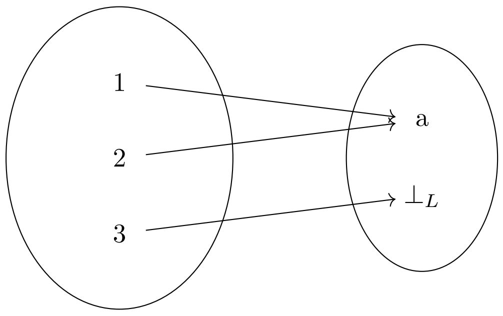


Figure 8.3: Example situation where is $\precsim$ *not* antisymmetric. The arrows define $\alpha\prime$.

The relation is reflexive, which follows from the reflexivity of $\sqsubseteq$. The relation is also transitive, which also follows from the transitivity of $\sqsubseteq$. Thus $\precsim$ is a preorder.

Similarly, the partial order relation on L induces a binary relation on $\mathcal{P}(A)$:

$$\begin{aligned}
&\boldsymbol{\precsim} : \mathcal{P}(A) \rightarrow \mathcal{P}(A) \rightarrow \text{Bool} \\
&\qquad\qquad \{a_{1_1}, a_{1_2}, a_{1_n}\} \boldsymbol{\precsim} \{a_{2_1}, a_{2_2}, a_{2_m}\} := \\
&\qquad\qquad\qquad \alpha\left(\{a_{1_1}, a_{1_2}, a_{1_n}\}\right) \sqsubseteq \alpha\left(\{a_{2_1}, a_{2_2}, a_{2_m}\}\right)
\end{aligned} \tag{8.10}$$

It is important to note that this binary relation is also *not* a partial order as the antisymmetry property is again *not* fulfilled. Figure 8.4 makes this clear since the following facts hold: $\varnothing_A \boldsymbol{\precsim} \{1\}$, $\{1\} \boldsymbol{\precsim} \varnothing_A$ but clearly $\varnothing_A \neq \{1\}$. The relation is reflexive, which follows from the reflexivity of $\sqsubseteq$. The relation is also transitive, which also follows from the transitivity of $\sqsubseteq$. Thus $\boldsymbol{\precsim}$ is a preorder.

The binary relations $\boldsymbol{\precsim}$ and $\precsim$ are of course related:

$$\{a_1\} \boldsymbol{\precsim} \{a_2\} \equiv a_1 \precsim a_2 \tag{8.11}$$

In theory many elements may map to $\perp_L$ as α is *not* injective. We are convinced that for many practical cases the following property holds:

$$\forall as \in \mathcal{P}(A) \,.\, \alpha(as) = \perp_L \implies as = \varnothing_A \tag{8.12}$$

The Table We now discuss creation and update for the table. A naive way to calculate the table would be to calculate the set $t : \mathcal{P}(A)$ of nonabstract answers and perform the abstraction step $\alpha(t)$ at the end. Because of

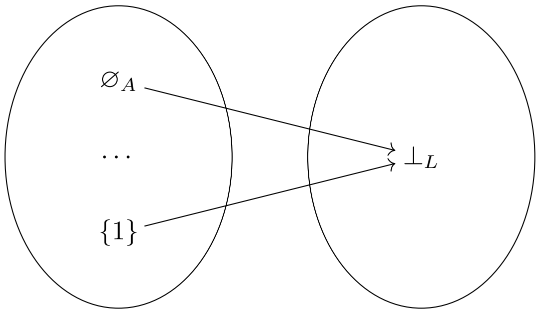


Figure 8.4: Example situation where is $\precsim$ *not* antisymmetric. The arrows define α.

property 8.8 we can interleave abstraction with the generation of nonabstract answers. This gives the following procedure:

Creation Procedure The table must be initialised with $\perp_L$.

Update Procedure On derivation of a new answer $a \in L$, the table t must be updated as follows: t contains an optimal abstracted answer l_{op}, the updated table $t\prime := t \sqcup \alpha\prime\,(a)$.

This procedure is much more interesting than the naive attempt for two reasons. The first reason is the storage needed: we assume that it is much more memory efficient to store an abstracted answer instead of a nonabstract answer. The second reason is the number of calculations performed. In the worst case, there may be an infinite number of nonabstract answers that do not matter for the calculation of optimal abstracted answers.

Monotonicity Requirement The entire procedure only works correctly on condition that the rules defining the tabled predicate are monotonic. This is related to the results given in [91]. If we model a rule as a function[1] $\mathrm{r} : A \rightarrow B$ then $\forall\,(a_1, a_2) \in A \times A.\, a_1 \precsim_A a_2 \implies \mathrm{r}\,(a_1) \precsim_B \mathrm{r}\,(a_2)$.

Suppose that we order the facts of Prolog predicates `p/1` and `q/1` by the natural ordering on their integer arguments. Then the following two rules constitute a program that does not satisfy the monotonicity requirement:

```
q(N) :- p(M), N is -1 * M.
```

[1]We feel it is not necessary to state precisely how to model rules as functions.

```
p(10).
p(N) :- p(M), N is M - 1, N > 0.
```

The Need for an Inverse Injection Function The fixpoint phase requires a more difficult treatment. Since the type of an arbitrary suspended computation is obviously fixed to C (defined above as $A \to \text{Maybe}\,(A)$), the type of its input must be A. Suppose we had to store every answer of type A just to serve as the input of the suspended computations, again we would not have gained anything at all. Therefore we assume the existence of an inverse injection function $\gamma : L \to \mathcal{P}\,(A)$ that constructs a set of possible nonabstract answers from a given abstract answer. This abstracted answer is an optimal abstracted answer stored in the table.

We require that γ is a total function:

$$\text{Dom}(\gamma) = L \tag{8.13}$$

Note that this definition allows that $\gamma\,(l_1) = \varnothing_A$.

Relationship between $\alpha\prime$ and γ We require the following relationship between the injection function $\alpha\prime$ and the inverse injection function γ by means of α:

$$\alpha \circ \gamma = \text{id} : L \to L \tag{8.14}$$

$$\gamma \circ \alpha = \text{paretoFrontier} \tag{8.15}$$

where $(g \circ f)(x) = g(f(x))$ and where paretoFrontier intuitively returns all minimal elements and is defined as:

$$\begin{aligned}&\text{paretoFrontier} : \mathcal{P}\,(A) \to \mathcal{P}\,(A)\\ &\qquad\qquad \text{paretoFrontier}\,(as) \coloneqq \{a_i \mid a_i \in as, \neg\exists a_j \in as.\, a_j \precsim a_i\}\end{aligned} \tag{8.16}$$

The rationale behind Equation 8.14 is that if we start from an abstracted answer and first derive a set of nonabstract answers using γ and then abstract these answers again using α, we should arrive at the same abstracted answer as we originally started from. After all, the deabstraction process should not introduce any new information.

The rationale behind Equation 8.15 is that given a set of nonabstract answers that is first abstracted using α and then deabstracted again using γ, we should not arrive at the original set of nonabstract answers, but at the subset of answers that are minimal.

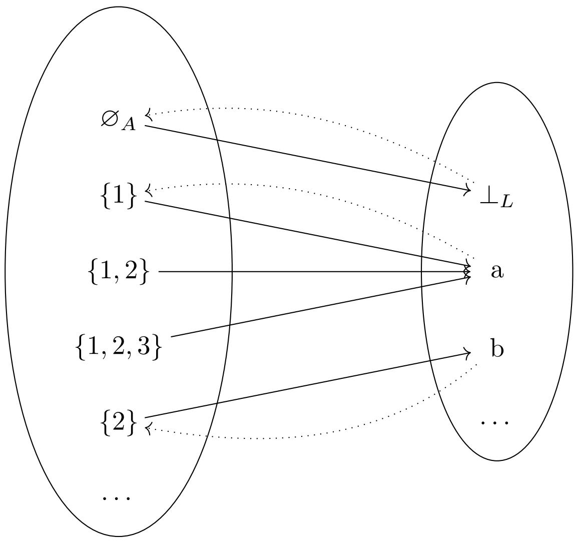


Figure 8.5: Visualising our requirements on $\alpha\prime$ (full line) and γ (dotted line).

For the definition of Equation 8.16 note especially that:

$$\neg\exists a_j \in as.\, a_j \precsim a_i \quad \not\equiv \quad \forall a_j \in as.\, a_i \precsim a_j \tag{8.17}$$

The reason for this is that not all elements $(a_i, a_j) \in A \times A$ have to be partially ordered with respect to each other. Figure 8.5 visualises the relationship between α and γ.

8.4 Framework Instances

We now show how the existing approaches to answer subsumption fit within our framework.

8.4.1 Regular Tabling

Tabling systems have been around since the early nineties. The best-known tabling engines, like XSB [152], Ciao [62], Yap [130] and B-Prolog [175], are based on SLG-resolution [20]. Several of these systems also support one or more variants of answer subsumption.

	Regular tabling	**Partial order**	
		without abstraction	with abstraction
L	$\mathcal{P}(A)$	$\mathcal{P}(A)$	$\mathcal{P}(B)$
$\boldsymbol{\alpha\prime}$	$\{a_1\}$	$\{a_1\}$	$\{\delta(\{a_1\})\}$
$\boldsymbol{\alpha}$	id	id	$\{\delta(x)\}$
$\boldsymbol{\gamma}$	id	id	$\cup\{\epsilon(y) \mid y \in x\}$
$\sqcup$	$\cup$	$\{x \mid x \in l_1 \cup l_2, (\nexists y \in l_1 \cup l_2 \mid x \neq y \wedge y \sqsubseteq x)\}$	similar

Table 8.1: Summary of the framework instances (part 1).

Regular tabling trivially fits in the framework by choosing $L := \mathcal{P}(A)$. The injection function is then $\alpha\prime(a_1) := \{a_1\}$. This means that $\alpha = \text{id}$. The inverse injection function is $\gamma := \text{id}$ as it must take type $L := \mathcal{P}(A)$ to type $\mathcal{P}(A)$. The join operation is then $\cup$. These choices are summarised in Table 8.1.

Table Data Structure The table has type $\mathcal{P}(A)$: it stores a set of answers. The standard datastructure is a trie, as it provides good insertion and lookup performance for individual answers. As an alternative, a hashtable is used in some implementations (e.g., B-Prolog).

8.4.2 Partial Order Answer Subsumption

Partial order answer subsumption is available in XSB [151]. It is a restricted form of lattice answer subsumption.

Partial order answer subsumption fits the framework by choosing

$$L := \mathcal{P}(A)$$

The injection function is again $\alpha\prime(a_1) = \{a_1\}$ and the inverse injection function is again $\gamma := \text{id}$. The join operation is

$$l_1 \sqcup l_2 := \{x \mid x \in l_1 \cup l_2, (\nexists y \in l_1 \cup l_2 \mid x \neq y \wedge y \sqsubseteq x)\}$$

This follows the existing informal semantics that

1. a_1 is only added to t iff a_1 is minimal with respect to other answers in t according to the partial order $\sqsubseteq$;

2. If a_1 is added, any answers that a_1 subsumes are deleted.

Also note that we need to deal with the case where elements cannot be partially ordered with respect to each other. The choices for this instance are again summarised in Table 8.1.

Table Data Structure The table again has type $\mathcal{P}(A)$. Here it is important to efficiently retrieve as well as delete answers that can be ordered with respect to a given answer a_1. Assuming that answers that be ordered with respect to each other (hereafter referred to as partially orderable answers) are sharing a common prefix, tries are particularly well-suited as their discriminatory property is based on common prefixes. In general, partially orderable answers should be stored in the same bucket.

Abstraction With partial order subsumption, an optional abstraction operation can be applied so that a set of answers can be represented by a single answer. This is needed when the program does not have a finite model. To support abstraction in XSB, the notation is extended to

```
    :- table p(_,_,po(order/2,abstr/2)).
```

where `order/2` is a partial order, as above, and `abstr/2` is the abstraction operation. This function is user-supplied. The abstraction is only taken when necessary. Swift and Warren briefly discuss a Net-style formalism as an example [151].

We designate the abstraction function with δ. It has the following type: $\delta : \mathcal{P}(A) \to B$. The abstract domain B thus is the user's choice and $L := \mathcal{P}(B)$. For the same reasons as outlined for the deinjection operator γ, we also need an inverse abstraction function ϵ. This function has type $\epsilon : B \to \mathcal{P}(A)$.

The set injection operator is then $\alpha(x) := \{\delta(x)\}$. This means that the injection operator is $\alpha\prime(a_1) := \{\delta(\{a_1\})\}$. The join operation is similar to above, but working on answers of type B instead of answers of type A. The deinjection operator is then $\gamma(x) := \cup\{\epsilon(y) \mid y \in x\}$.

8.4.3 Lattice Answer Subsumption

Lattice answer subsumption is available in XSB [151]. It can form the basis of multi-valued logics, quantitative logics, and of abstract interpretation [151].

Lattice answer subsumption fits the framework by choosing $L := \mathcal{P}(A)$. The injection function is then $\alpha\prime(a_1) := \{a_1\}$ and the inverse injection function

	Lattice subsumption	**Mode direction**
L	$\mathcal{P}(A)$	$\mathcal{P}(A)$
$\boldsymbol{\alpha\prime}$	$\{a_1\}$	$\{a_1\}$
$\boldsymbol{\alpha}$	id	id
$\boldsymbol{\gamma}$	id	id
$\sqcup$	arbitrary	$\{x \mid x \in l_1 \cup l_2,$ $x = \text{aggregate}($ $\text{setSameIndexedPositions}($ $l_1 \cup l_2, x))\}$

Table 8.2: Summary of the framework instances (part 2).

is $\gamma := \text{id}$. The join operation can be chosen arbitrarily. This follows the existing informal semantics that a_1 may not be added to t, but that the join is taken of a_1 and another answer in t. We summarize the choices made in Table 8.2. It is clear that lattice answer subsumption is the most general approach; mode direction requires the most difficult formalisation despite the ease with which it is applied in practice.

Table Data Structure Here the table should be able to store a set of answers a_i. Given the arbitrary choice of join operation, there is no clear candidate for an efficient data structure. As a general rule, it should be possible to efficiently retrieve those answers that should be joined with a given answer a_1.

8.4.4 Mode-directed Tabling

Mode-direction [56, 57] has been motivated by the underlying idea of dynamic programming: to define an optimal solution in terms of optimal solutions to subproblems. Finding optimal solutions using plain tabling is tricky. Secondly, mode-direction has been motivated by the need for evidence for the solution found.

Mode-direction was proposed by Guo and Gupta for use in the TALS system [53] (based on ALS Prolog). It has also been implemented in B-Prolog [177] and Yap [129]. Zhou et al. [177] extend the mode declaration for B-Prolog with a cardinality limit, and a new `nt`-declaration. The cardinality limit specifies the number of optimal answers that must at most be kept in the table. The `nt` mode allows arguments to be discarded for both variant checking and answer tabling, which is useful for passing global data to a predicate. It can be simulated using global variables, but the `nt` mode results in cleaner code. Also, in B-Prolog, a mode-directed predicate is allowed to produce multiple answers, so that the relation from input to output does not need to be a function. Unique to the Yap implementation is the mode "last", which is the opposite of the (-) operator (called "first" in Yap): the argument is not indexed, and the last instance seen will be stored.

An operational semantics for mode-direction has been defined by Guo and Gupta [57].

Mode-directed tabling fits the framework by choosing $L := \mathcal{P}(A)$. The injection function is again $\alpha\prime(a_1) := \{a_1\}$ and the inverse injection function is again $\gamma := \mathrm{id}$. Guo and Gupta allow two aggregation operations 'min' and 'max' either of which we denote with $\bowtie$. If B is the type of the arguments that must be aggregated, then $\bowtie$ has the type $\mathcal{P}(B) \to B$. We now introduce some auxiliary functions that we use for the definition of the join operation:

- The function indexedPositions returns a set containing the indices that the user has declared as indexed.
- The function nonIndexedPositions returns a set containing the indices that the user has declared as non-indexed.
- The function aggregationPosition returns the sole index that the user has declared as aggregated.
- The function sameIndexedPositions determines for two answers a_1 and a_2 whether their arguments at the indexed positions are equal. This function thus has type $A \to A \to \mathrm{Bool}$ and is defined as

$$\begin{aligned}&\mathrm{sameIndexedPositions}(a_1, a_2) := \\ &\qquad \forall i \in \mathrm{indexedPositions}.\, \arg_i(a_1) = \arg_i(a_2)\end{aligned} \quad (8.18)$$

- The function setSameIndexedPositions determines the set of answers that has the same arguments at the indexed positions as the given answer. This function has type $L \to A \to L$ and is defined as:

$$\begin{aligned}&\mathrm{setSameIndexedPositions}(l_1, a_1) := \\ &\qquad \{x \mid \mathrm{sameIndexedPositions}(x, l_1), x \in l_1\}\end{aligned} \quad (8.19)$$

$$l_1 \sqcup l_2 \coloneqq \{x \mid x \in l_1 \cup l_2, x = \text{aggregate}\left(\text{setSameIndexedPositions}\left(l_1 \cup l_2, x\right)\right)\} \tag{8.20}$$

where aggregate is defined as:

$$\text{aggregate} : L \to A$$
$$\text{aggregate}\left(x\right) \coloneqq \text{arg}_{\text{agg}}\left(x\right) \tag{8.21}$$

where agg is defined as:

$$\text{agg} : L \to B$$
$$\text{agg}\left(l_1\right) \coloneqq \bowtie\left(\left\{\text{arg}_{\text{aggregationPosition}}\left(x\right) \mid x \in l_1\right\}\right) \tag{8.22}$$

where $\text{arg}_{\text{aggregationPosition}}\left(x\right)$ selects the argument (of type B) to be aggregated from an answer x, and where $\bowtie$ converts a set of such arguments to an aggregated argument. We summarize the choices made for mode-direction in Table 8.2 on page 196.

In B-Prolog's implementation of mode-directed tabling, it is possible to specify that n optimal answers must be kept in the table. We can do the same by generalising the aggregation function $\bowtie$ to a relation.

Table Data Structure In this case, a trie is not unconditionally a good data structure. For example, suppose the first argument is non-indexed, while the rest of the arguments is indexed, then the answers with given indexed arguments are scattered all over the trie. A trie will work fine if the indexed arguments coincide with a prefix of the arguments.

8.5 Related Work

Lattice Answer Subsumption in Haskell Vandenbroucke et al. [168, 169] have added lattice answer subsumption to their tabling implementation in Haskell. It is based on the effect handlers approach.

Tabling and Probabilistic Inference Answer subsumption is used in the PITA probabilistic inference package for XSB. PITA computes the probability of queries by means of annotated disjunction (LPADs) and build explanations for the derivation steps. The binary decision diagrams (BDDs) used to represent those explanations have a natural lattice structure [120].

Tabling and Constraint Handling Rules Schrijvers et al. [135] argue that it is hard to implement constraint solvers using attributed variables, and therefore propose to use constraint handling rules (CHR) for developing new constraint solvers that can be integrated in a tabled logic programming system. In [135] they present a practical implementation of the framework. They use "call abstraction" or subsumption to reduce the number of tables, and discuss a novel way to reduce the size of answer sets in comparison to answer subsumption.

Tabling and Constraint Logic Programming Arias Herrero [5] gives a short overview of tabling combined with constraint logic programming (TCLP). The initial idea of TCLP is due to Kanellakis et al. [77] in the constraint databases community. A constraint database generalises atomic values to constraint variables to allow a range of values in the fields of the relation. TCLP's theoretical basis was set up in the context of Datalog [120]. However, a top-down approach normally uses less space and obtains its result faster than a Datalog-style bottom-up approach, but it can get stuck in infinite loops. Toman [161] proposed to use tabling to mitigate the downfalls of a top-down approach. The paradigm has since been applied to model checking, timed automata, and abstract interpretation.

Adding constraints to tabled logic programming allows to suspend calls that satisfy the following two conditions:

- the current call is a variant of an earlier call;
- the constraint store of the current call is entailed by the constraint store of the earlier call.

A constraint store S_a is entailed by S_b ($S_a \sqsubseteq S_b$) if any solution in S_a is also a solution in S_b.

TCLP is implemented in XSB [30], but the implementation does not use full entailment checking. The approach of de Guzmán et al. [25] uses entailment checking on both answers and calls. This allows termination in more cases than other approaches. It also improves in the area of constraint projection.

8.6 Conclusion and Future Work

Apart from classical tabling using only call variance, a few variants of answer subsumption have been described in literature. Compared to classical tabling, answer subsumption again raises the language's declarative expressiveness. Unfortunately, the semantics of these approaches has only been described informally.

We have presented a framework that can express all existing answer subsumption variants. This allows for a better understanding of each individual variant, but also sheds light on their mutual relationships.

A deep understanding of a particular answer subsumption variant is necessary to implement that technique robustly. Although the literature for most variants has also described the changes needed to SLG-based tabling engine to implement the variant, we felt that it is hard to fully grasp all implementation details due to a lack of formal definition. Therefore, anyone who wishes to add answer subsumption to a tabling system not based on SLG, is somewhat left in the dark. We are convinced that our framework makes this challenging task somewhat easier.

As future work we aim to develop a criterion that can check whether it is safe to use a particular form of answer subsumption, as preliminary findings has shown that is it easy to come up with situations where the existing mechanisms derive incorrect answers.

Chapter 9

Conclusions

We have started this thesis with an overview of the broad field of logic programming and then explained Prolog, the oldest and most widely known logic programming language, in more detail. Over the years, many successful extensions of traditional logic programming have been developed, which has lead to a wide variety of subdomains. However, implementation has required extensive efforts. This is because implementing a virtual machine for even plain Prolog is tremendously complex. Due to this complexity, the extensions have mainly been developed in isolation.

We believe that simpler implementation techniques could be a big leap forward in the progress of using logic programming for modeling ever larger and more difficult problems. These problems arise naturally in a wide range of scientific disciplines as well as industry. In this thesis we have focused on three different areas.

Modular Search Heuristics One of the most-heard criticisms about traditional Prolog is that its automated control, SLD-resolution, is too rigid, since it does not adapt itself to the situation and always performs an exhaustive search. Indeed an exhaustive depth-first search is not always feasible when problem domains become larger and larger. Out of necessity, the standard practice for changing the control is to modify the problem logic to include control information. However, intermingling custom heuristics with the logic describing the actual problem domain is clearly a violation of Kowalski's ideal. In practice, this is reflected in the development and maintenance effort.

Tor is a simple hook into disjunction that allows the clear separation between a custom heuristic and the logic of the problem domain. We have consciously kept the effort of defining a new (composed) search heuristic low. As a result, some search methods cannot be expressed, f.e. swapping the order of the branches in a disjunction. Nevertheless, Tor is remarkably expressive.

We have formalized Tor by constructing a functional model of Prolog and search heuristics. From this model we have derived an actual Prolog implementation based on delimited control. Delimited control is a well-known technique from the functional world. We have shown that this construct can be easily given a very useful Prolog-compatible semantics. We have shown how to implement delimited control in the WAM and verified the performance of the resulting engine.

Flexible language extensions Apart from search heuristics, delimited control enables the flexible implementation of a wide range of well-known programming techniques, such as DCGs, coroutines and implicit state passing. Notably, the effect handler's approach, which we see as a structured, yet lucid, way of using delimited control, has been a tremendous help. Indeed, also in the functional world itself, we perceive an increased interest in effect handlers, where they are seen as a simpler alternative for monads. In the logic programming world, the effect handlers approach provides the infrastructure to capture common program patterns.

In principle, the capturing of program patterns can also be achieved using meta-programming and program transformations, but these come with severe drawbacks:

- the effort of defining a transformation is proportional to the number of features in the language;
- programs transformations are fragile: when the language evolves, they require amendments;
- when a new feature is introduced, the whole system may need transforming.

Effect handlers using delimited control have none of these drawbacks: they can be developed in isolation and afterwards be composed at will. In view of what we have achieved with delimited control, its ease of both use and implementation, we believe that a more widespread use in the logic programming world, would be a step in the direction towards simple, yet powerful systems.

Tabling Probably the most widely studied adaptation of SLD-resolution is tabling. The promise of tabling is twofold: apart from the relaxed control-related requirements that tabling engines offer and the resulting improved termination properties, its memoisation effect can greatly improve performance. Both advantages come at the cost of memory. But also in this area, the implementation effort is a burden for its widespread adoption. For definite programs the technique itself is now well-understood, but investing several man-years in major architectural changes, is an effort that can only be made in an academic context. As a result, only a handful from a wide range of Prolog systems support tabling.

We have provided an extensive overview of existing implementation approaches. We have then shown how the issue can be tackled by a small library on top of a Prolog equipped with delimited control. In contrast with existing suspension mechanisms, the implementation of delimited control in the WAM can be done in a couple of hundred lines. The tabling implementation itself then takes under 600 lines and comes as a library. The main part of the effort does not go to manipulating the control flow (only 11% of the overall line count), but to providing efficient data structures for the fixpoint computation and for storing answers.

So far, our high-level tabling library has not yet achieved the performance level of a native implementation, but this was never our objective. We have definitely shown that delimited control is a viable construction for achieving the suspension functionality, which is by far the most difficult part of tabling systems. Thus all the expressivity of Prolog systems that do not have the resources to deeply reengineer their engine can now be combined with tabling.

Next to classical tabling where all distinct answers are stored in the table, the literature describes several techniques that allow to specify a preference across answers. Suboptimal answers then are discarded in favour of more optimal answers. This not only leads to higher space efficiency, but also raises the declarative nature of the language once again, as users do not need to concern themselves with the algorithmic behind the selection of optimal answers. Moreover, for some problems, these techniques, designated as answer subsumption, change the termination behaviour from looping infinitely to a much better-behaved finite runtime.

However, existing answer subsumption variants have not been defined formally, making it difficult to compare these techniques and implement them in a different setting than the one they were originally developed in. We have remedied these situation by defining a framework that provides a common setting.

9.1 Future Work

While several leaps further, we are by no means at the end of the path towards flexible goal-oriented logic programming: several interesting questions remain in areas close as well as more remote to this thesis. We begin with a discussion of the topics in close correspondence to this thesis and gradually move further away.

Modular Search Heuristics Concerning modular search heuristics, relaxing the requirement that search heuristics have to be compatible with Prolog's underlying depth first search mechanism would make way for a entirely new range of techniques. Our success with delimited control to manipulate the control flow provides an interesting perspective to this end. Secondly, Tor is limited to binary disjunction. Multiway disjunctions have to be encoded as a sequence of binary ones, which feels unnatural for some applications. It would be interesting to investigate how the model and implementation can be generalised; however it is clear that this can be achieved using a more complicated hook. Less clear is how this hook could be hidden behind a declarative interface similar to the archetypal search trees capturing the essence heuristics in the case of binary disjunction.

Flexible language extensions To help spread delimited control in logic programming, it would be beneficial to see what patterns commonly occur in actual Prolog programs and provide a library of abstractions. Unfortunately, deeply nesting effect handlers currently has an adverse effect on performance. There is ongoing work by Hany Saleh [60] on how nested effect handlers can be folded into a single monolithic one.

Tabling Specific for tabling, advances in several areas are still possible:

- Continuations are copied with the general `copy_term/2` predicate. A special-purpose `copy_continuation/2` could do better by exploiting the known structure of these terms. Other applications using delimited control could benefit from this optimization as well.

- In the scheduling component, we identify strongly connected components dynamically. Doing so statically would allow the specialisation of completion.

- In contrast with state-of-the art implementations, our tries do not use substitution factoring. Neither have we experimented with hash tables, as preferred by B-Prolog.

To shift the tradeoff between investment and performance more towards performance, implementing the supporting datastructures, like answer and call variant tries, at the lowest possible level is not the big issue for a skilled Prolog implementor, as these are orthogonal to the existing engine.

Given the improved understanding of techniques for answer subsumption offered by our formal framework, it would be interesting to extend our high-level tabling implementation with them. Our initial exploration shows that this does not require changes to the control flow, but only to the table data structure. However, defining a flexible architecture to switch between traditional tabling and the different answer subsumption variants requires more creativity, in addition to modularity support from the underlying Prolog. Ultimately, more practical experience could lead to the development of some guidelines on which answer subsumption technique to use in a given situation.

Parallel logic programming Exploiting parallelism in logic programs has long been a topic of interest. Gupta et al. provide an excellent survey [58]. The most promising method for non-tabled execution seems the exploitation of or-parallelism: exploring matching clauses in parallel, because this can be done implicitly and its implementation is reasonably simple.At the same time, or-parallelism obtains very good speedups [127].

Less work has been done on parallelising tabled execution. Freire et al. propose table parallelism where parallel execution can happen at each tabled subgoal [49]. They also design a parallel completion algorithm. Unfortunately, this does not exploit parallelism for non-tabled subgoals. To address this issue, Rocha et al. propose two new computational models: or-parallelism within tabling (OPT) and tabling within or parallelism (TOP) [124]. They have implemented the OPT model in Yap [126, 127].

In the long run, it would be interesting to investigate how the existing models for parallelism map to our lightweight tabling approach. As a first step, parallelising the completion phase may already yield significant performance improvements.

The cut construct Although a standard behaviour for cut in plain Prolog has been proposed as far back as 1986 [101], this extralogical construct still raises issues in the development of extensions, with tabling as a notorious example. In addition, we believe that the bad reputation of cut has certainly contributed to the current popularity of ASP. We are aware of at least two attempts to evict the construct from Prolog [67, 34]. On a less radical scale, analysis tools could flag and perhaps even propose refactorings in the style of Schrijvers et al. [141] for many occurrences.

User-orientation User-friendliness and adequate tooling support are still an issue in the whole logic programming community. Recently some interest has emerged there, as can be seen from the advent of IULP: the International Workshop on User-Oriented Logic Programming. More practically, the need for maintaining valuable large industrial applications triggers such developments [97].

Programming in the large For traditional logic programming itself there are still many challenges remaining in statical typing and portability. Pure Prolog has always been a dynamically typed language, but given the advantages of a statical type system, attempts to introduce new typed logic languages as well as attempts to gradually introduce types into existing Prolog variants have been made [136, 181, 63]. Portability is hard, as for historical reasons many interesting contributions to traditional logic programming have been scattered over a wide range of implementations. Since 2007, a basic compatibility framework has been established between Yap and SWI-Prolog [173].

Of prime importance to programming in the large is a powerful standardized module system: this is not only beneficial for the maintainability of large applications, but additionally stimulates library development [59]. Haemmerlé and Fages review the module system of several contemporary Prolog implementations [59]. Many of these systems derive from Quintus Prolog, yet differ in subtle ways.

For their module system review, Haemmerlé and Fages only use two properties: module protection and calling module protection. The former ensures that only visible predicates can be called, while the latter ensures that callbacks are impossible, unless explicitly allowed. Unfortunately, only the module systems of Ciao [16] and XSB [155] satisfy both properties. However, although they are unequivocally fundamental, many other issues may influence the design of a module system. Interesting examples are given in the design for the Ciao module system [16]: allowing efficient compilation and global analysis to name only two.

XSB's module system is remarkable in the sense that it only can only export and import predicates instead of any atom.

The popular SWI-Prolog has explicitly positioned itself as a system for programming in the large [174]. We believe that this is the right direction: with an ever increasing amount of heterogeneous data sources, flexibility and library support more than ever determine the success of not only a language, but of an entire paradigm.

Summary

In this thesis we have tackled several shortcomings of traditional goal-oriented logic programming. The de facto standard language for this class is Prolog. However, this language is not without its faults: one of the most often heard critics is its rigidness. Over the years of its long and venerable life, many extensions have been developed to deal with weak spots. Also, a rich variety of new subdomains in logic programming has its roots in Prolog.

However, Prolog language extensions have mainly been developed in isolation. This is because implementing those extensions is hard: a virtual machine for plain Prolog is already tremendously complex. We aim at more flexibility through simple implementation techniques. For finding these techniques we do not fear cross-pollination with another well-understood declarative paradigm: functional programming. Successfully combining ideas from these two paradigms could be a big leap towards modeling increasingly large and complex problems.

Prolog allows to specify a problem as a set of rules. Ideally, these rules define only the logic of the problem domain. The programmer does not need to worry about how the computer executes those rules. At least, that is the theory.

The control built into Prolog interpreters is SLD-resolution. SLD-resolution specifies that rules matching a given goal are executed in textual order: from top to bottom and from left to right. If the user is not constantly aware of this order, it is in practice very simple to write a program that traps the interpreter in an infinite loop. For instance, it suffices to write a left-recursive rule before its corresponding base case.

Yet, even when the user takes the built-in control into account, it is clear that the built-in control is very rigid. As it happens, the search strategy is not only predictable, but it is always exhaustive in addition. It is best to visualise the strategy as a (search) tree, where each alternative is represented by a branch. The leaves of the tree correspond to failures or successes. When

we write a left-recursive rule, we thus define an implicit tree with an infinite path on the left.

Modular Search SLD-resolution blindly chooses the left children in the infinite search tree over and over again and hence gets trapped on this path. Even when the search tree is finite, or when we make sure that the infinite branches in the tree are on the right, it can very well be that the finite part of the tree is already much too large to scour. Of course, there are many well-known heuristics to search only a part of the tree, for example imposing a bound on the length of the paths (depth-bounded search), on the number of nodes (node-bounded search), hybrids, etc..

The problem is however that the user cannot just use heuristics considering only a part of the tree: he has to encode the strategy himself, and that entangled with the logic describing the problem domain. Prolog only supplies a pruning operator (known as cut), that allows to prune alternative branches. It immediately strikes that this is a very primitive modus operandi: not a single form of reuse is possible, although the idea behind the heuristic is always the same. The programmer thus loses valuable time redefining similar code over and over again. In addition, it is often extremely important to experiment with lots of different heuristics. Each heuristic scours a different part of the tree. This can lead to finding a solution faster, or finding a better a solution.

The standard practice to experiment with heuristics is thus to copy the program and adapt it a bit. As a consequence bugs propagate throughout the different versions, and maintenance quickly becomes a real nightmare. Absence of a library of reusable search methods has an additional perverse effect: the programmer has to reinvent the wheel every time, which leads to him/her falling back on those methods that he/she is familiar with. He/she comes less into contact with alternative methods that a library would offer ready-to-use. A good software library has also been designed so that methods can work together easily; in our case this means that the search heuristics can be easily combined into a composite heuristic.

We address the non-reusability of search heuristics in Chapter 3. We provide a hook into Prolog's (binary) disjunction. This hook enables us to execute at each disjunction a handler that is specific for the search heuristic we want to obtain. We title this approach TOR. However, this is a very operational manner of doing things. We also provide a much more elegant declarative approach based on the observation that each heuristic in isolation already defines a search tree of a form that is typical for that particular heuristic. For example, depth-first search with a maximum allowed depth of n defines a balanced binary tree of depth $n+1$ of which the leaves. We can combine this heuristical tree with the search tree of the problem logic by overlaying them. To this end, it suffices to look at each node whether the shape of trees still match. If we encounter failure in the tree corresponding to the heuristic, then the result is failure. If we encounter a success node in the tree corresponding to the problem logic, then we have also found a solution for the combined heuristic.

Using the TOR methodology we define several well-known search heuristics. These search heuristics must all be compatible with Prolog's depth first execution strategy though. It is for example not possible to swap the order of the branches in the tree. The greatest power of TOR lies however in the way we can combine different heuristics into a new composite heuristic. It is for instance easy to create a heuristic that after exhaustively scouring the tree up to a given dept, switches to a strategy where only a given maximum number of nodes can be visited.

For a search problem, it is of course important how many tree nodes we can search in a given time interval. Therefore it is important that TOR does not cause too much overhead. In this respect, it is important to note that for a constraint programming problem the lion's share of the execution time is spent on propagation. With a slowdown factor of at most three, the performance of artificial benchmarks is reasonable given the extra flexibility, but for realistic benchmarks, we see that TOR does not introduce any significant overhead in the case of SWI-Prolog. For the faster B-Prolog the overhead of TOR is among the size of 10%. With automatic specialisation this overhead can be limited even more though.

Delimited Control In Chapter 4 we discuss delimited control. A Prolog program consists only of rules and facts. When we look at the form of an individual rule, we see that it in essence consists of a Horn clause, supplemented with simple built-in predicates. So Prolog is a minimalistic language. This has several advantages (it is for example easy to reason about the language), but at times infrastructure to encode frequently occurring patterns can be thoroughly missed. Therefore, in the past several methods to add extra constructs to the language have been developed. Beyond a doubt meta-programming and program transformations are among the most well-known. The use of both is encouraged by Prolog's homoiconic nature. Other examples from the literature are extended DCGs, logical loops and structured state threading. All these examples belong to the category of non-local program transformations.

Non-local program transformations are not ideal because they are fragile: each time a new construct is added to the language, the program transformation must also be adapted, even when semantically both are decoupled from each other. The amount of work to define such a program transformation is thus proportional to the number of constructs in the language. When a new construct is added, the entire system must be adapted. This makes defining a new language construct so unattractive that the alternative of copy-paste doesn't even seem that horrid.

In the functional world effect handlers have recently attracted a good deal of attention as a flexible but structured alternative to define new language constructs. However, for use in the logic programming world, an underlying mechanism is needed and therefore we can also draw inspiration from the functional world in the form of delimited control, that there hold an almost mythical status. We define two control operators `shift/1` and `reset/3` that offer a Prolog compatible form of delimited control. We define the semantics of these operations in the standard manner using a direct-style meta-interpreter, and using an interpreter in continuation-passing style. We discuss the different semantical points of attention that show up defining these constructs, including interaction with cut, selection and exception handling. We also present the full low-level implementation of the constructs in two standard architectures for Prolog virtual machines, the WAM and the ZIP. The constructs cannot only be used to implement well-known Prolog language extensions (such as DCGs), in addition we show how several ideas from other languages can easily be integrated. Effect handlers here serve as a structured approach on top of the lowlevel primitives for delimited control.

We compare the performance of the lowlevel implementation with an implementation based on program transformation. An implementation at low-level gives good results for the WAM, for the ZIP (or at least within the overall design of SWI-Prolog) this is not the case. There, an approach based on program transformation is faster.

Modular Search Specification In Chapter 5 we revisit the search heuristics from the third chapter. The Tor-approach is lightweight, efficient and can be easily ported to other systems. However, the approach has been defined quite operationally and therefore lacks a semantical model. We therefore define a model for Tor using functional techniques.

As a starter for the Tor model, we reify the search tree that is implicit in Prolog. Thus the search tree is now represented as syntax. This process happens by applying the Free monad. The search heuristics can do their job easily on this explicit representation. As a result, we obtain a modified tree. This tree can be reflected back into the semantics.

Working in the Free monad is very well-suited to a Haskell model. Implementing this monad in Prolog is on the contrary very challenging. Luckily, this is not needed. We show that there exists an isomorphism between the Free monad and the Delimited continuations monad transformer. We have created a Prolog implementation of delimited control in the previous chapter. As a consequence, using the isomorphism, we can port our Haskell results to Prolog.

Introduction to Tabled Resolution In Chapter 6 we give an overview of the existing techniques for tabling in logic programming. We already know that due to an incorrect rule order a standard Prolog interpreter based on SLD-resolution can get stuck in an infinite branch of the implicit search tree. The same problem can occur if the input data is cyclic, but then there is no simple solution possible like swapping the rules. Put more formally, SLD-resolution is not capable of calculating the least fixpoint of the immediate consequence operator. Tabling has been developed to tackle the problems of SLD-resolution. The underlying idea is to temporarily suspend calculations that would go into an infinite loop and resume them later in a controlled fashion. To this end, one keeps the collection of answers that have been found so far in a datastructure also known as the table.

However, implementing the tabling technique is extremely complex. The best-known technique is based on freezing the execution stacks of the virtual machine. It was developed by D.S. Warren and implemented in XSB. Besides, there are only a handful of implementations based on the same principle (Ciao and Yap). These have all been developed in an academic context. This is because implementing tabling requires extremely invasive changes to the architecture of the virtual machine, the so-called Warren Abstract Machine. That changes are for budgetary and time-related reasons simply impossible in industry. However, this low-level implementation offers a good performance.

As an alternative to a full-blown stack-freezing tabling implementation, a lot of different approaches have been developed based on complex program transformations. The most well-known are linear tabling mechanisms (B-Prolog). These avoid the necessity to temporarily suspend computations but thereby loose some performance. Given these complications, it is not a surprise that very few systems have a tabling implementation, despite the earliest research in tabling dating from the end of the eighties. It is however very desirable because tabling does not only dramatically increases the Prolog language's declarative level and termination properties drastically, but in addition can lead to a gain of an order of magnitude in execution speed. This is because tabling can be considered as a declarative variant of dynamic programming. In Chapter 7 we tackle this problem.

Tabling with Delimited Control After an overview of the existing techniques for tabling, it is clear that no implementation of tabling exists combining decent performance with ease of implementation. With ease of implementation, we mean a design that does not require drastic changes to the architecture of the virtual machine. In Chapter 7 we design an implementation that requires minimal changes to the WAM and is thus orthogonal to the already existing features. Typically, the suspension mechanism is the hardest part to provide; we realise that it possible to use the primitives for delimited control from Chapter 4 to this end. Apart from this we use mutable terms and non-backtrackable variables that are by default available in many Prolog implementations. These three features are the only that must be provided at low-level inside the WAM. The implementation itself is possible as a library in user space and thus is written in ordinary Prolog code. That makes our implementation very accessible and easily portable across the different Prolog implementations. In addition we have already shown that the primitives for delimited control cannot only be used to implement tabling. Because of this property the cost of implementing these primitives can be amortised over the applications they enable.

Unique to our work is the way in which we calculate the fixpoint. As input we have both a set of answers and a set of computations that were suspended, but that can lead to a new answer, given an already obtained answer. We describe a simple mechanism that does not unnecessarily duplicate work, but is at the same time simple: we consider a double-ended queue and at any point add obtained answers to one end and add suspended computations to the other end. Each time we combine an existing answer with a suspended computation, we swap both in the list. From this combination, new answers or new suspended computations can possibly be derived. We repeat the process until all answers are on that side that is used for adding suspended computations and vice versa.

We evaluate our tabling implementation by comparing to those Prolog implementations that have implemented tabling by freezing the execution stack (XSB, Ciao, Yap) and with the linear implementation of B-Prolog. The performance of our implementation is very reasonable when we consider the very deep architectural changes to the virtual machine that are necessary for the alternatives and compare this to our very lightweight implementation in only 577 lines of Prolog code. Of those 577 lines, less than 11% goes to the actual control flow. We spent 45% of the line count to the datastructure for the fixpoint computation and 40% to the tree datastructure for keeping the answers.

Answer Subsumption Answer subsumption is an extension of traditional tabling that allows to prefer certain, more suitable, answers above others. As a consequence, the less suitable answers no longer need to be stored in the table. This allows to express optimisation and planning problems in a much more declarative style, since the user is no longer responsible for selecting optimal (partial) solutions.

Several concrete variants of the general "answer subsumption" idea have been described in the literature. The problem is that they are not defined formally: the description is limited to giving the interface for a specific implementation, describing a few example problems, and an overview of the changes necessary to the specific tabling system (in practice always based on freezing the execution stacks) in which the variant has been developed. Hence, it is very difficult to implement the technique in another system: there is a lot of uncertainty about the exact behaviour.

In addition, it is also difficult to see the commonalities and differences between the variants. This makes a choice for a particular variant more of a guess, rather than an informed decision.

In Chapter 8 we remedy this situation: we define a general framework for answer subsumption in which all variants can be described. This framework is based on the mathematical concept of a lattice. We give an overview of the existing approaches to answer subsumption and show how they fit in the framework.

Nederlandstalige samenvatting

In deze thesis hebben we verschillende tekortkomingen van traditioneel doelgericht logisch programmeren behandeld. De feitelijke standaardtaal voor deze groep is Prolog. Deze taal heeft echter ook zijn gebreken: één van de meest gehoorde kritieken is z'n starheid. Tijdens Prolog's lange en eerbiedwaardige levensloop, zijn veel uitbreidingen ontwikkeld om de zwakke plekken aan te pakken. Daarnaast heeft ook een rijke variëteit aan nieuwe subdomeinen van logisch programmeren roots in Prolog.

Echter, taaluitbreidingen voor Prolog zijn meestal in isolatie ontwikkeld. De reden hiervoor is dat het implementeren van deze extensies moeilijk is: een virtuele machine voor pure Prolog is al zeer complex. Wij mikken op meer flexibiliteit door eenvoudige implementatietechnieken. Voor het vinden van deze technieken, schuwen we een tweewegsinteractie met een ander goed begrepen programmeerparadigma, functioneel programmeren, niet. Een succesvolle combinatie van ideeën uit deze twee paradigma's kan een grote stap zijn naar het modelleren van alsmaar grotere en complexere problemen.

Prolog laat toe een probleem declaratief te specifiëren als een verzameling regels. Deze regels definiëren idealiter alleen de logica van het probleemdomein. De programmeur hoeft zich niet te bekommeren om hoe deze regels worden uitgevoerd door de computer. Althans, dat is de theorie.

De controle die in Prolog-interpreters is ingebouwd, is SLD-resolutie. SLD-resolutie specificeert dat de regels die passen bij een gegeven doel worden uitgevoerd van boven naar onder, en dat de doelen binnen één regel zullen worden uitgevoerd van links naar rechts. Als de gebruiker zich niet voortdurend bewust is van deze volgorde, is het in de praktijk heel eenvoudig om een programma te schrijven waarbij de interpreter vast komt te zitten in een oneindige lus. Bijvoorbeeld, het volstaat om een links-recursieve regel neer te schrijven voor zijn basisgeval.

Maar ook wanneer de gebruiker wel rekening houdt met de ingebouwde controle, valt het duidelijk op dat deze heel rigide is. De zoekstrategie is namelijk niet alleen voorspelbaar, maar ook per default altijd exhaustief. De strategie valt het best te visualiseren als een (zoek)boom, waarbij elk alternatief zorgt voor een splitsing. De bladeren van de boom komen overeen met falingen of successen. Wanneer we een links-recursieve regel schrijven, definiëren we dus een impliciete boom met een oneindig lang pad links.

Modulair zoeken SLD-resolutie kiest telkens blindelings voor linkerkinderen in de oneindige zoekboom en komt dus zo vast te zitten op dit pad. Ook wanneer de zoekboom eindig is, of wanneer we ervoor zorgen dat de oneindige takken rechts in de boom zitten, kan het zijn dat het deel van de boom met eindige paden al veel te groot is om te doorzoeken. Uiteraard zijn vele heuristieken bekend om slechts een gedeelte van de boom te doorzoeken, bijvoorbeeld een limiet opleggen op de lengte van de paden (dieptebeperkt zoeken), op het aantal toppen dat mag bezocht worden in de boom (topbeperkt zoeken), combinaties enz..

Het probleem is echter dat de gebruiker heuristieken die slechts een deel van de boom beschouwen, niet zomaar kan opgeven: hij moet de strategie zelf coderen en wel vermengd met de logica die het probleemdomein definieert. Prolog voorziet enkel een snijoperator (bekend als cut), die toelaat om alternatieve takken weg te snijden. Het valt meteen op dat dit een zeer primitieve manier van werken is: er is geen enkele vorm van hergebruik mogelijk voor de heuristieken hoewel het idee altijd hetzelfde is. De programmeur verliest dus kostbare tijd door telkens opnieuw gelijkaardige code te moeten schrijven. Het is bovendien vaak heel belangrijk om met veel verschillende heuristieken te experimenteren. Elke heuristiek doorzoekt een ander deel van de boom. Dit kan er dus toe leiden dat een oplossing sneller gevonden wordt, of dat er een betere oplossing gevonden wordt.

De standaardmanier van werken tot nu toe is dus om het programma te kopiëren en een beetje aan te passen. Dat zorgt ervoor dat bugs zich verspreiden, en onderhoud wordt al snel een ware nachtmerrie. De afwezigheid van een bibliotheek van herbruikbare zoekmethoden heeft nog een bijkomend pervers effect: de programmeur moet telkens weer het wiel heruitvinden, wat er ook voor zorgt dat hij/zij steeds opnieuw terugvalt op die methoden die hij/zij kent. Hij/zij komt minder in contact met alternatieve methoden die een bibliotheek kant-en-klaar zou aanbieden. Een goede sofwarebibliotheek is ook zo ontworpen dat methoden gemakkelijk kunnen samenwerken; in ons geval betekent dit dat de zoekheuristieken gemakkelijk kunnen worden gecombineerd tot een samengestelde heuristiek.

Niet kunnen hergebruiken van zoekheuristieken pakken we aan in Hoofdstuk 3. We voorzien een omwegje (hook) in de (binaire) disjunctie van Prolog. Dit omwegje zorgt ervoor dat we bij elke disjunctie een bepaalde routine (handler) kunnen uitvoeren die specifiek is voor de zoekheuristiek die we willen bekomen. Deze aanpak noemen we TOR. Dit is echter een zeer operationele manier van werken. We voorzien ook een veel elegantere declaratieve manier gebaseerd op de observatie dat elke heuristiek op zichzelf al een zoekboom definieert met een vorm die typerend is voor de specifieke heuristiek. Zo definieert dieptebeperkt zoeken met een maximaal toegelaten diepte n een gebalanceerde binaire boom van diepte $n+1$ waarvan de bladeren overeenkomen met faling. Deze heuristiekboom kunnen we combineren met de zoekboom van de probleemlogica door ze te overlappen. Het volstaat hiervoor bij elke splitsing te kijken of de vorm van de bomen nog overeenkomt. Als we in de boom horend bij de heuristiek faling tegenkomen, dan faalt het geheel. Komen we in de zoekboom horend bij de probleemlogica een succestop tegen, dan hebben we ook een oplossing gevonden voor de gecombineerde heuristiek.

Aan de hand van de TOR methodologie definiëren we verschillende welbekende zoekheuristieken. Deze zoekheuristieken moeten echter wel allemaal compatibel zijn met de diepte-eerst strategie van Prolog. Het is bijvoorbeeld niet mogelijk om de volgorde van de takken in de boom om te wisselen. De grote kracht van TOR ligt echter in de manier waarop we de verschillende heuristieken kunnen combineren tot een nieuwe samengestelde heuristiek. Het is bijvoorbeeld eenvoudig om een heuristiek te maken die na het volledig doorzoeken van de boom tot op een gegeven diepte, voor de dieper gelegen toppen overschakelt op een strategie waarbij nog een maximaal aantal toppen mag worden bezocht.

Voor een zoekprobleem is het uiteraard belangrijk hoeveel toppen van de boom we in een gegeven tijdsspanne kunnen doorzoeken. Daarom is het belangrijk dat TOR niet al te veel overhead veroorzaakt. Hierbij is het belangrijk om op te merken dat voor een constraint programming probleem het leeuwendeel van de uitvoeringstijd gespendeerd wordt aan propagatie. De performantie voor artifiële benchmarks zonder propagatie is met een vertragingsfactor van hoogstens drie redelijk gezien de extra flexibiliteit, maar voor realistische benchmarks zien we TOR geen significante overhead introduceert in het geval van SWI-Prolog. Voor het snellere B-Prolog is de overhead van TOR in de grootteorde van 10%. Met automatische specialisatie kan deze overhead echter nog verminderd worden.

Begrensde controle In hoofdstuk 4 bespreken we begrensde controle (delimited control). Een Prolog programma bestaat enkel uit regels en feiten.

Wanneer we kijken naar de vorm van een individuele regel, dan zien we dat die in essentie bestaat uit een Horn-clausule, aangevuld met eenvoudige ingebouwde predikaten. Prolog is dus een minimalistische taal. Dit heeft verschillende voordelen (het is bijvoorbeeld eenvoudig om over de taal te redeneren), maar soms wordt infrastructuur om vaakvoorkomende patronen te kunnen encoderen nogal gemist. In het verleden zijn daarom verschillende manieren bedacht om extra constructies aan de taal toe te voegen. De bekendste hiervan zijn ongetwijfeld meta-programmeren en programmatransformaties. De toepassing van beide wordt aangemoedigd door de homoïconische natuur van Prolog. Andere voorbeelden uit de literatuur zijn extended DCGs, logical loops en structured state threading. Al deze voorbeelden vallen onder de categorie van niet-lokale programmatransformaties.

Niet-lokale programmatransformaties zijn echter niet ideaal, want ze zijn breekbaar. Telkens wanneer een nieuwe constructie aan de taal wordt toegevoegd, moet ook de programmatransformatie worden aangepast, zelfs wanneer beide semantisch gezien los van elkaar staan. De hoeveelheid werk om een dergelijke programmatransformatie te definiëren is dus recht evenredig met het aantal constructies in de taal. Wanneer een nieuwe constructie wordt geïntroduceerd, moet het hele systeem aangepast worden. Dit alles maakt het definiëren van een nieuwe taalconstructie zo onaantrekkelijk dat het alternatief copy-paste niet eens zo afschuwelijk lijkt.

In de functionele wereld hebben effect handlers de laatste tijd veel aandacht gekregen als een flexibele, maar gestructureerde manier om nieuwe taalconstructies te definiëren. Voor toepassing in de wereld van logisch programmeren is er echter een onderliggend mechanisme nodig en ook daarvoor kunnen we inspiratie halen uit de functionele wereld in de vorm van begrensde controle, die aldaar een haast mythische status geniet. Wij definiëren twee controleoperatoren `shift/1` en `reset/3` die een Prolog-compatibele vorm van begrensde controle aanbieden. We definiëren de semantiek van deze operaties op de standaard manier aan de hand van een directe metainterpreter en een interpreter in continuatiestijl. We bespreken de verschillende semantische aandachtspunten die bij het definiëren van deze primitieven opduiken, waaronder de interactie met cut, selectie en uitzonderingen. Ook bespreken we de volledige laagniveau implementatie van de constructies in twee standaardarchitecturen voor Prolog virtuele machines, de WAM en de ZIP. De constructies kunnen niet alleen gebruikt worden voor het aanbieden van bekende taaluitbreidingen van Prolog (zoals DCGs), maar we tonen ook aan hoe verschillende ideeën uit andere talen gemakkelijk kunnen worden toegevoegd. Effect handlers zorgen hierbij voor een gestructureerde aanpak bovenop de laagniveau primitieven voor begrensde controle.

We vergelijken de performantie van de laagniveauimplementatie met een transformatiegebaseerde aanpak. Een implementatie op laagniveau levert voor de WAM goede resultaten op, in de ZIP (of op z'n minst in het design van SWI-Prolog) is dit niet het geval. Daar is een transformatiegebaseerde aanpak sneller.

Een specificatie voor modulair zoeken In hoofdstuk 5 keren we terug op de zoekheuristieken uit het derde hoofdstuk. De TOR-aanpak is lichtgewicht, efficiënt en kan gemakkelijk worden overgedragen naar andere systemen. De aanpak is echter wel nogal operationeel gedefinieerd en daarom ontbreekt een semantisch model. We ontwerpen daarom een model voor TOR aan de hand van functionele technieken. Inzicht in zowel logische als functionele technieken is hier een noodzakelijke voorwaarde om het probleem tot een goed einde te brengen.

Als start van het model voor TOR reificeren we de zoekboom die in Prolog impliciet is. De zoekboom wordt dus nu voorgesteld door middel van syntax. Dit proces gebeurt door toepassing van de Free monad. Op deze expliciete voorstelling kunnen de heuristieken gemakkelijk werken. Als resultaat bekomen we een aangepaste boom. Deze boom kan terug worden geflecteerd in de semantiek.

Werken in de Free monad is erg geschikt voor een Haskell model. Deze monad daarentegen gaan implementeren in Prolog is erg uitdagend. Dit is gelukkig niet nodig. We tonen aan dat er een isomorfisme is tussen de Free monad en de Delimited Continuations monad transformer. We hebben in het vorige hoofdstuk een implementatie gemaakt voor begrensde controle in Prolog. Dankzij het isomorfisme kunnen we dus ook onze Haskell resultaten omzetten naar Prolog.

Introductie tot tabulatie In hoofdstuk 6 geven we een overzicht van de bestaande technieken voor tabelleren in logisch programmeren. We weten reeds dat door een verkeerde volgorde van regels, een standaard Prolog interpreter gebaseerd op SLD-resolutie vast kan komen te zitten in een oneindige tak van de impliciete zoekboom. Hetzelfde probleem kan optreden als de inputdata cyclisch is, maar dan is er geen eenvoudige oplossing mogelijk zoals het omwisselen van de regels. Meer formeel is SLD-resolutie niet in staat om het kleinste fixpunt te berekenen van de onmiddellijk-gevolg operator. Tabling is ontwikkeld om de problemen van SLD-resolutie aan te pakken. Het onderliggende idee is altijd om berekeningen die in een oneindige lus terecht dreigen te komen, af te breken en later op een gecontroleerde manier opnieuw uit te voeren. Daarvoor houdt men de verzameling van antwoorden bij die men tot nu toe heeft gevonden in een datastructuur die ook wel bekend staat als de tabel (table).

De tabulatietechniek implementeren is echter zeer complex. De meest bekende aanpak is gebaseerd op het bevriezen van de uitvoeringsstapels van de virtuele machine. Ze werd ontwikkeld door D.S. Warren en geïmplementeerd in XSB. Daarnaast zijn er slechts een handvol andere implementaties op hetzelfde principe gebaseerd (Ciao en Yap). Deze zijn alle ontwikkeld in een academische context. De implementatie van tabelleren vergt namelijk zeer ingrijpende veranderingen aan de architectuur van de virtuele machine, de zogeheten Warren Abstract Machine[1], die omwille van tijdsduur en budgettaire beperkingen simpelweg niet mogelijk zijn in de industrie. Deze implementatie op laag niveau biedt echter een goede performantie.

Als alternatief voor een volledige implementatie van tabulatie die de uitvoeringsstapels bevriest, zijn er heel wat andere manieren ontwikkeld gebaseerd op complexe programmatransformaties. Het meest bekend zijn de lineaire tabelleringsmechanismen (B-Prolog). Deze vermijden de noodzaak om berekeningen tijdelijk uit te stellen, maar boeten daardoor aan performantie in. Gegeven deze complicaties is het dan ook niet verwonderlijk dat, hoewel het vroegste onderzoek naar tabulatie dateert uit het einde van de jaren 80, zeer weinig systemen een implementatie van tabulatie hebben. Dit is echter zeer wenselijk omdat tabelleren niet alleen het declaratief niveau van de taal Prolog en zijn terminatieeigenschappen drastisch verhoogt, maar daarnaast ook kan zorgen voor een grootteorde aan verbetering van de uitvoeringssnelheid. Tabelleren kan namelijk beschouwd worden als een declaratieve variant van dynamisch programmeren. In hoofdstuk 7 pakken we dit probleem aan.

Tabulatie met begrensde controle Na een overzicht van de bestaande technieken voor tabelleren, is het duidelijk dat er geen enkele implementatie van tabulatie bestaat die een redelijke performantie combineert met implementatiegemak. Met implementatiegemak bedoelen we een ontwerp dat geen drastische wijzigingen vraagt in de architectuur van de virtuele machine. In hoofdstuk 7 ontwerpen we een implementatie die minimale wijzigen vraagt in de WAM en aldus orthogonaal staat op de reeds bestaande voorzieningen. Typisch het moeilijkst te voorzien is een onderbrekingsmechanisme; wij realiseren ons dat het mogelijk is om hiervoor de primitieven voor begrensde controle uit hoofdstuk 4 te gebruiken. Daarnaast gebruiken we muteerbare termen en niet-backtrackbare variabelen die reeds in heel wat Prolog imple-

[1]genoemd naar de pionier D.H.D Warren.

mentaties standaard zijn voorzien. Deze drie features zijn de enige die op laag niveau in de WAM moeten worden voorzien. De implementatie zelf is mogelijk als een bibliotheek op gebruikersniveau en is dus geschreven in gewone Prolog code. Dat maakt onze implementatie zeer toegankelijk en eenvoudig overdraagbaar tussen de verschillende Prolog implementaties. Bovendien hebben we reeds aangetoond dat de primitieven voor begrensde controle niet uitsluitend bruikbaar zijn voor het implementeren van tabulatie. Hierdoor kan de kost voor het implementeren van deze primitieven worden uitgemiddeld over hun verschillende toepassingen.

Uniek voor ons werk is de manier waarop we het fixpunt berekenen. Als input hebben we zowel een verzameling van antwoorden als een verzameling van berekeningen die eerder werden uitgesteld, maar gegeven een reeds berekend antwoord, kunnen leiden tot een nieuw antwoord. We beschrijven een eenvoudig algoritme dat geen werk onnodig dupliceert, maar tegelijk eenvoudig is: we beschouwen een dubbelgelinkte lijst en voegen steeds reeds berekende antwoorden toe aan het ene uiteinde en de uitgestelde berekeningen aan het andere uiteinde. Telkens wanneer we een bestaand antwoord combineren met een uitgestelde berekening, wisselen we beide om in de lijst. Door deze combinatie kunnen mogelijk nieuwe antwoorden of uitgestelde berekeningen aan de lijst worden toegevoegd. We herhalen het proces tot alle antwoorden aan die kant staan waar de uitgestelde berekeningen worden toegevoegd en vice versa.

We evalueren onze implementatie van tabulatie door te vergelijken met die Prolog implementaties die tabulatie hebben geïmplementeerd door het bevriezen van de uitvoeringsstapel (XSB, Ciao, Yap) en met de lineaire implementatie van B-Prolog. De performantie van onze implementatie is erg behoorlijk wanneer we rekening houden met de zeer ingrijpende veranderen aan de virtuele machine die nodig zijn voor de alternatieven en dit vergelijken met onze zeer lichtgewicht implementatie in slechts 577 lijnen Prolog code. Van die 577 lijnen code gaat minder dan 11% naar de eigenlijke control flow. We besteden 45% van het aantal lijnen aan de datastructuur voor de fixpuntberekening en 40% aan de boomdatastructuur voor het bijhouden van de antwoorden.

Selecteren van antwoorden Selecteren van antwoorden (answer subsumption) is een uitbreiding van het klassiek tabelleren die toelaat om bepaalde, meer geschikte, antwoorden te verkiezen boven andere. Als gevolg hiervan gaat men de minder geschikte antwoorden niet langer opslaan in de tabel. Dit laat toe om optimalisatie- en planningsproblemen uit te drukken op een veel declaratievere manier, aangezien de gebruiker niet zelf instaat voor het selecteren van optimale (deel)oplossingen.

Van het algemene idee “selecteren van antwoorden” zijn verschillende concrete varianten beschreven in de literatuur. Het probleem is echter dat deze niet formeel worden gedefinieerd: men beperkt zich tot het geven van de interface voor een specifieke implementatie, het beschrijven van een aantal voorbeelden, en een overzicht van de wijzigingen die nodig zijn in het specifieke (in de praktijk altijd op het bevriezen van de uitvoeringsstapel gebaseerde) tabelleringssysteem waarin de variant is ontwikkeld. Bijgevolg is het zeer moeilijk om de techniek te implementeren in een ander systeem: er heerst veel onduidelijkheid over het precieze gedrag.

Daarnaast is het ook moeilijk om de overeenkomsten en verschillen tussen de varianten te zien. Dat maakt een keuze voor een bepaalde variant meer een gok dan een geïnformeerde beslissing.

In hoofdstuk 8 pakken we deze situatie aan: we definiëren een algemeen framework voor answer subsumption waarin alle varianten kunnen worden beschreven. Dit framework is gebaseerd op het wiskundig begrip tralie. We geven een overzicht van de bestaande aanpakken voor selectie van antwoorden en tonen hoe deze binnen het framework passen.

Bibliography

[1] *Python PEP 342 — Coroutines via Enhanced Generators*, 2005. `http://www.python.org/dev/peps/pep-0342/`. ↑96

[2] *C♯ Language Specification Version 4.0*, 2010. `http://msdn.microsoft.com/en-us/library/x53a06bb.aspx`. ↑96

[3] Abderrahmane Aggoun and Nicolas Beldiceanu. Time stamps techniques for the trailed data in constraint logic programming systems. In *SPLT'90, 8ème Séminaire Programmation en Logique*, pages 487–510, 1990. ↑24

[4] Hassan Aït-Kaci. *Warren's Abstract Machine — a Tutorial Reconstruction.* `http://wambook.sourceforge.net/`, 1999. Online edition of the 1991 book published by MIT Press. ↑4, ↑17, ↑79, ↑177

[5] Joaquín Arias Herrero. Design and implementation of a modular interface to integrate CLP and tabled execution, 2015. ↑199

[6] S. Awodey. *Category Theory*, volume 49 of *Oxford Logic Guides*. Oxford University Press, 2006. ↑96

[7] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *The Journal of Logic Programming*, 1920, Supplement 1:73 – 148, 1994. Special Issue: Ten Years of Logic Programming. ↑1

[8] Roman Barták, Agostino Dovier, and Neng-Fa Zhou. On modeling planning problems in tabled logic programming. In *Proceedings of the 17th*

International Symposium on Principles and Practice of Declarative Programming, PPDP '15, pages 31–42, New York, NY, USA, 2015. ACM. ↑158

[9] Andrej Bauer and Maitja Pretnar. Programming with algebraic effects and handlers, 2012. `arXiv:1203.1539`. ↑102, ↑137

[10] M. Blazevic. monad-coroutine: Coroutine monad transformer for suspending and resuming monadic computations, 2010. `http://hackage.haskell.org/package/monad-coroutine`. ↑96, ↑138

[11] D.L. Bowen, L. Byrd, and W.F. Clocksin. A portable Prolog compiler. In *Proceedings of the Logic Programming Workshop*, pages 74–83, 1983. ↑84

[12] Marco Bozzano and Adolfo Villafiorita. *Design and Safety Assessment of Critical Systems.* Auerbach Publications, Boston, 1st edition, 2010. ↑2

[13] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, ICFP '13, pages 133–144. ACM, 2013. ↑102, ↑137

[14] Paul Branquart and Johan Lewi. A scheme of storage allocation and garbage collection for Algol 68. In *ALGOL 68 Implementation*, pages 199–238, 1970. ↑82

[15] Bernd Braßel, Michael Hanus, and Frank Huch. Encapsulating nondeterminism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004. ↑135

[16] Daniel Cabeza and Manuel Hermenegildo. The Ciao module system: A new module system for Prolog. *Electronic Notes in Theoretical Computer Science*, 30(3):122 – 142, 2000. Parallelism and Implementation Technology for (Constraint) Logic Programming (in connection with ICLP '99). ↑206

[17] Mats Carlsson and Per Mildner. SICStus Prolog - The first 25 years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012. ↑30, ↑136

[18] Amadeo Casas, Daniel Cabeza, and Manuel V. Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in LP systems. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *LNCS*, pages 146–162. Springer, 2006. ↑65

[19] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, 1989. ↑167

[20] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996. ↑4, ↑145, ↑185, ↑193

[21] Pablo Chico de Guzmán, Manuel Carro, and Manuel V. Hermenegildo. A sketch of a complete scheme for tabled execution based on program transformation. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 795–800. Springer Berlin Heidelberg, 2008. ↑156

[22] Pablo Chico de Guzmán, Manuel Carro, and Manuel V. Hermenegildo. A program transformation for continuation call-based tabled execution. *CoRR*, abs/0901.3906, 2009. ↑156

[23] Pablo Chico de Guzmán, Manuel Carro, and Manuel V. Hermenegildo. Supporting pruning in tabled LP. In *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages - Volume 7752*, PADL 2013, pages 60–76, New York, NY, USA, 2013. Springer-Verlag New York, Inc. ↑152

[24] Pablo Chico de Guzmán, Manuel Carro, Manuel V. Hermenegildo, Cláudio Silva, and Ricardo Rocha. An improved continuation call-based implementation of tabling. In *Practical Aspects of Declarative Languages, 10th International Symposium*, volume 4902 of *Lecture Notes in Computer Science*, pages 197–213. Springer, 2008. ↑156, ↑157

[25] Pablo Chico de Guzmán, Manuel Carro, Manuel V. Hermenegildo, and Peter Stuckey. A general implementation framework for tabled CLP. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming*, volume 7294 of *Lecture Notes in Computer Science*, pages 104–119. Springer Berlin Heidelberg, 2012. ↑199

[26] A. Colmerauer. Metamorphosis grammars. In Leonard Bolc, editor, *Natural Language Communication with Computers*, volume 63 of *Lecture Notes in Computer Science*, pages 133–188. Springer Berlin Heidelberg, 1978. ↑22, ↑61

[27] Michael A. Covington. ISO Prolog: A summary of the draft proposed standard. 1993. Last viewed online on May 10, 2016 at `http://fsl.cs.illinois.edu/images/9/9c/PrologStandard.pdf`. ↑13

[28] Flávio Cruz and Ricardo Rocha. Retroactive subsumption-based tabled evaluation of logic programs. In Tomi Janhunen and Ilkka Niemelä, editors, *Logics in Artificial Intelligence*, volume 6341 of *Lecture Notes in Computer Science*, pages 130–142. Springer Berlin Heidelberg, 2010. ↑157

[29] Flávio Cruz and Ricardo Rocha. A simple table space design for retroactive call subsumption. 2011. ↑157

[30] Baoqiu Cui and David S. Warren. A system for tabled constraint logic programming. In *Computational Logic CL 2000*, volume 1861 of *Lecture Notes in Computer Science*, pages 478–492. Springer Berlin Heidelberg, 2000. ↑199

[31] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 151–160. ACM, 1990. ↑61, ↑62, ↑128, ↑137

[32] Pablo Chico de Guzmán, Manuel Carro, and David S. Warren. Swapping evaluation: A memory-scalable solution for answer-on-demand tabling*. *Theory and Practice of Logic Programming*, 10(4-6):401–416, July 2010. ↑151, ↑152, ↑154

[33] Wolfgang De Meuter and Gruia-Catalin Roman, editors. *Coordination Models and Languages*, volume 6721 of *LNCS*, 2011. ↑92

[34] S. K. Debray and D. S. Warren. Towards banishing the cut from Prolog. *IEEE Transactions on Software Engineering*, 16(3):335–349, March 1990. ↑206

[35] Bart Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Comp. Sc., KU Leuven, Belgium, 2002. ↑93

[36] Bart Demoen and Phuong-Lan Nguyen. So many WAM Variations, so little Time. In *Computational Logic - CL2000, First International Conference, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000. ↑79

[37] Bart Demoen and Phuong-Lan Nguyen. Two WAM implementations of action rules. volume 5366 of *LNCS*, pages 621–635, December 2008. ↑93

[38] Bart Demoen and Konstantinos Sagonas. Cat: The copying approach to tabling. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin Heidelberg, 1998. ↑150, ↑153, ↑175

[39] Bart Demoen and Konstantinos Sagonas. Chat: The copy-hybrid approach to tabling. In Gopal Gupta, editor, *Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 106–121. Springer Berlin Heidelberg, 1998. ↑153, ↑177

[40] Bart Demoen, Tom Schrijvers, and Benoit Desouter. Delimited continuations in Prolog: semantics, use and implementation in the WAM. Report CW 631, Dept. of Computer Science, KU Leuven, Belgium, 2013. ↑97

[41] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):253–282, 2012. ↑30, ↑136

[42] Suzanne Wagner Dietrich. *Extension Tables for Recursive Query Evaluation.* PhD thesis, Stony Brook, NY, USA, 1987. AAI8815786. ↑155

[43] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 163–173, New York, NY, USA, 1991. ACM. ↑62

[44] Martin Erwig. Escape from Zurg: an exercise in logic programming. *J. Funct. Program.*, 14(3):253–261, May 2004. ↑135

[45] Changguan Fan and Suzanne Wagner Dietrich. Extension table builtins for Prolog. *Software: Practice and Experience*, 22(7):573–597, 1992. ↑155, ↑175

[46] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 180–190. ACM, 1988. ↑61, ↑62, ↑137

[47] Andrzej Filinski. Monads in action. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 483–494. ACM, 2010. ↑60

[48] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. *SIGPLAN Not.*, 42(9):165–176, October 2007. ↑62

[49] Juliana Freire, Rui Hu, Terrance Swift, and David S. Warren. *Programming Languages: Implementations, Logics and Programs: 7th International Symposium, PLILP '95 Utrecht, The Netherlands, September 20–22, 1995 Proceedings*, chapter Exploiting parallelism in tabled evaluations, pages 115–132. Springer Berlin Heidelberg, 1995. ↑205

[50] Yoshihiko Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. ↑76

[51] Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *Proc. ICFP*, pages 2–14. ACM, 2011. ↑102, ↑138

[52] George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing back monad comprehensions. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 13–22. ACM, 2011. ↑138

[53] Hai-Feng Guo and Gopal Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Proceedings of ICLP'01*, pages 181–196. Springer, 2001. ↑143, ↑145, ↑146, ↑155, ↑196

[54] Hai-Feng Guo and Gopal Gupta. Cuts in tabled logic programming. In Bart Demoen, editor, *Proceedings of the Colloqium on Implementation of Constraint and Logic Programming Systems (CICLOPS)*, pages 62–73, 2002. ↑151

[55] Hai-Feng Guo and Gopal Gupta. An efficient and flexible engine for computing fixed points. *CoRR*, abs/cs/0412041, 2004. ↑147, ↑155

[56] Hai-Feng Guo and Gopal Gupta. Simplifying dynamic programming via tabling. In Bharat Jayaraman, editor, *Practical Aspects of Declarative Languages*, volume 3057 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2004. ↑182, ↑183, ↑184, ↑196

[57] Hai-Feng Guo and Gopal Gupta. Simplifying dynamic programming via mode-directed tabling. *Software: Practice and Experience*, 38(1):75–94, 2008. ↑183, ↑196, ↑197

[58] Gopal Gupta, Enrico Pontelli, Khayri A.M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of Prolog programs: A survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, July 2001. ↑205

[59] Rémy Haemmerlé and Francois Fages. Modules for Prolog Revisited. Research Report RR-5869, INRIA, 2006. ↑206

[60] Amr Hany Saleh. Transforming delimited control: Achieving faster effect handlers. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming*, 2015. ↑99, ↑204

[61] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the 15th International Joint Conferences on Artificial Intelligence (IJCAI 1995)*, pages 607–613, 1995. ↑37

[62] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and German Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012. ↑30, ↑136, ↑148, ↑174, ↑193

[63] Patricia Hill and John Wylie Lloyd. *The Gödel programming language.* MIT press, 1994. ↑206

[64] Ralf Hinze. Prological Features In A Functional Setting — Axioms And Implementations. In *Third Fuji Int. Symp. on Functional and Logic Programming*, pages 98–122, 1998. ↑135

[65] Ralf Hinze. Deriving backtracking monad transformers. *SIGPLAN Not.*, 35(9):186–197, September 2000. ↑135

[66] Christian Holzbaur. Meta-structures vs. Attributed Variables in the Context of Extensible Unification. volume 631 of *LNCS*, pages 260–268, August 1992. ↑93

[67] Masaki Hoshida and Mario Tokoro. *Logic Programming '88: Proceedings of the 7th Conference Tokyo, Japan, April 11–14, 1988*, chapter ALEX: The logic programming language with explicit control and without cut-operators, pages 82–95. Springer Berlin Heidelberg, 1989. ↑206

[68] G. Hutton and D. Fulger. Reasoning About Effects: Seeing the Wood Through the Trees. In *Pre-proceedings of the Symposium on Trends in Functional Programming*, 2008. Unpublished, available at `http://www.cs.nott.ac.uk/~gmh/effects.pdf`. ↑102

[69] Dragan Ivanovic, José Francisco Morales Caballero, Manuel Carro, and Manuel Hermenegildo. Towards structured state threading in Prolog. In *CICLOPS 2009*, 2009. ↑61

[70] R. James and A. Sabry. Yield: Mainstream delimited continuations. In *First International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011)*, 2011. ↑95, ↑96

[71] Mauro Jaskelioff and Eugenio Moggi. Monad transformers as monoid transformers. *Theor. Comput. Sci.*, 411(51-52):4441–4466, December 2010. ↑96, ↑138

[72] Ernie Johnson, C.R. Ramakrishnan, I.V. Ramakrishnan, and Prasad Rao. A space efficient engine for subsumption-based tabled evaluation of logic programs. In Aart Middeldorp and Taisuke Sato, editors, *Functional and Logic Programming*, volume 1722 of *Lecture Notes in Computer Science*, pages 284–299. Springer Berlin Heidelberg, 1999. ↑156, ↑157

[73] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer Berlin Heidelberg, 1995. ↑107

[74] Mark P. Jones and Luc Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, Department of Computer Science, New Haven, Connecticut, December 1993. ↑60

[75] Nenad Jovanovic. Lattice tutorial, 2005. Last viewed on July 29, 2015 at `https://www.iseclab.org/people/enji/infosys/lattice_tutorial.pdf`. ↑185

[76] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, ICFP '13, pages 145–158. ACM, 2013. ↑102, ↑137

[77] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51(1):26–52, 1995. ↑199

[78] Steven Keuchel and Tom Schrijvers. Towards efficient implementations of effect handlers. In *IFL 2014*, 2014. ↑6

[79] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Christina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, pages 220–242, 1997. ↑58

[80] Oleg Kiselyov. Delimited control in OCaml, abstractly and concretely: System description. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 304–320. Springer Berlin Heidelberg, 2010. ↑95

[81] Oleg Kiselyov. Iteratees. volume 7294 of *LNCS*, pages 166–181, 2012. ↑65

[82] Oleg Kiselyov, Simon Peyton-Jones, and Amr Sabry. Lazy vs. yield: Incremental, lazy pretty-printing. In *APLAS*, 2012. ↑65

[83] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell '13, pages 59–70. ACM, 2013. ↑102, ↑137

[84] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In *Proc. ICFP'05*, pages 192–203. ACM, 2005. ↑135

[85] Robert Kowalski. Predicate logic as programming language. In *Proceedings of International Federation for Information Processing (IFIP)*, pages 569–574, 1974. ↑3, ↑13

[86] Robert Kowalski. *Logic for Problem Solving*. North-Holland, 1979. ↑1, ↑11

[87] Bill Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences of the United States of America*, 50(1):869–872, 1963. ↑102

[88] S. Le Houitouze. A New Data Structure for Implementing Extensions to Prolog. volume 456 of *LNCS*, pages 136–150, August 1990. ↑93

[89] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1594–1597. AAAI Press, 2008. ↑1

[90] Barbara Liskov. History of programming languages — II. chapter A History of CLU, pages 471–510. ACM, 1996. ↑65

[91] J. W. Lloyd. *Foundations of Logic Programming*, chapter 1: Declarative Semantics. Springer-Verlag New York, 1984. ↑185, ↑191

[92] Wolfgang Lohmann, Günter Riedewald, and Guido Wachsmuth. Aspect-Orientation in Prolog. In *Proceedings of the 16th International Symposium on Logic-based Program Synthesis and Transformation*, 2006. ↑58

[93] Wolfgang Lux. Implementing encapsulated search for a lazy functional logic language. In *Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *Lecture Notes in Computer Science*, pages 100–113. Springer, 1999. ↑135

[94] Moe Masuko and Kenichi Asai. Direct implementation of shift and reset in the MinCaml compiler. ML'09, pages 49–60, 2009. ↑95

[95] Moe Masuko and Kenichi Asai. Caml Light+ shift/reset= Caml Shift. *Theory and Practice of Delimited Continuations (TPDC 2011)*, pages 33–46, 2011. ↑95

[96] Conor McBride. The Frank manual, May 2012. Last viewed on 07/12/2015 at `https://personal.cis.strath.ac.uk/conor.mcbride/pub/Frank/TFM.pdf`. ↑95, ↑102, ↑137

[97] Edison Mera and Jan Wielemaker. Porting and refactoring Prolog programs: the PROSYN case study. *Theory and Practice of Logic Programming*, 13(4-5 Online Supplement), 2013. ↑206

[98] Bertrand Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, October 1992. ↑2

[99] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), 1991. ↑71

[100] Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the 1st North American Chapter of the Association for Computational Linguistics Conference*, NAACL 2000, pages 249–255. Association for Computational Linguistics, 2000. ↑143

[101] Chris Moss. *Third International Conference on Logic Programming: Imperial College of Science and Technology, London, United Kingdom, July 14–18, 1986 Proceedings*, chapter Cut & Paste — defining the impure primitives of Prolog, pages 686–694. Springer Berlin Heidelberg, 1986. ↑205

[102] Mozilla.org. *Javascript 1.7*, 2006. `https://developer.mozilla.org/en/New_in_JavaScript_1.7`. ↑96

[103] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. 4741:529–543, 2007. ↑136

[104] U. Neumerkel. Extensible unification by metastructures. In *META'90*, pages 352–364, April 1990. ↑93

[105] Raymond Ng and V.S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150 – 201, 1992. ↑1

[106] Thomas Nordin and Andrew Tolmach. Modular lazy search for constraint satisfaction problems. *J. Funct. Program.*, 11(5):557–587, September 2001. ↑136

[107] N. S. Papaspyrou. A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. Technical Report CSD-SW-TR-2-01, National Technical University of Athens, 2001. ↑96, ↑138

[108] Fernando C.N. Pereira and David H.D. Warren. Definite clause grammars for language analysis – A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231 – 278, 1980. ↑22, ↑61

[109] Tomas Petricek, Alan Mycroft, and Don Syme. Extending monads with pattern matching. In *Proc. Haskell'11*, pages 1–12. ACM, 2011. ↑138

[110] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 80–94. Springer-Verlag, 2009. ↑71, ↑102, ↑137

[111] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. ↑179

[112] Matija Pretnar. An introduction to algebraic effects and handlers. In *Proceedings of the 31st conference on the Mathematical Foundations of Programming Semantics*, Electronic notes in theoretical computer science, 2015. ↑95

[113] João Raimundo and Ricardo Rocha. Global trie for subterms. *Online Proceedings of the 11th International Colloquium on Implementation of Constraint Logic Programming Systems (CICLOPS 2011)*, 2011. ↑151, ↑176

[114] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David S. Warren. Efficient model checking using tabled resolution. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 143–154. Springer Berlin Heidelberg, 1997. ↑145

[115] I.V Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient access mechanisms for tabled logic programs. *The Journal of Logic Programming*, 38(1):31 – 54, 1999. ↑146, ↑149, ↑150, ↑181

[116] R. Ramesh and Weidong Chen. A portable method for integrating SLG resolution into Prolog systems. In *Proceedings of the 1994 International Symposium on Logic Programming*, ILPS '94, pages 618–632, Cambridge, MA, USA, 1994. MIT Press. ↑156

[117] Prasad Rao, C. R. Ramakrishnan, and I.V. Ramakrishnan. On the nonexistence of optimal scheduling strategies for tabled resolution. ↑152

[118] Prasad Rao, C. R. Ramakrishnan, and I.V. Ramakrishnan. A thread in time saves tabling time. In *Joint International Conference/Symposium on Logic Programming (JICSLP)*. MIT Press, 1996. ↑157

[119] Andrea Rendl, Tias Guns, Peter J. Stuckey, and Guido Tack. MiniSearch: A solver-independent meta-search language for MiniZinc. In *Proceedings of Principles and Practice of Constraint Programming — 21st International Conference (CP 2015)*, pages 376–392, 2015. ↑136

[120] Fabrizio Riguzzi and Terrance Swift. Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In *Proceedings of the 17th RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, Bologna, Italy, June 10-11, 2010*, volume 616 of *CEUR Workshop Proceedings*, Aachen, Germany, 2010. Sun SITE Central Europe. ↑198, ↑199

[121] Ricardo Rocha. Handling incomplete and complete tables in tabled logic programs. In Sandro Etalle and Miroslaw Truszczynski, editors, *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4079 of *Lecture Notes in Computer Science*, pages 427–428. Springer, 2006. ↑151

[122] Ricardo Rocha. On improving the efficiency and robustness of table storage mechanisms for tabled evaluation. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL '07, pages 155–169, Berlin, Heidelberg, 2007. Springer-Verlag. ↑151

[123] Ricardo Rocha, Nuno Fonseca, and Vítor Santos Costa. On applying tabling to inductive logic programming. In *Machine Learning: ECML 2005*, volume 3720 of *Lecture Notes in Computer Science*, pages 707–714. Springer Berlin Heidelberg, 2005. ↑151

[124] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. Or-parallelism within tabling. 1999. ↑205

[125] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. YapTab: A tabling engine designed to support parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000. ↑176

[126] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. *Logic Programming: 17th International Conference, ICLP 2001 Paphos, Cyprus, November 26 – December 1, 2001 Proceedings*, chapter On a Tabling Engine That Can Exploit Or-Parallelism, pages 43–58. Springer Berlin Heidelberg, 2001. ↑205

[127] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. On applying or-parallelism and tabling to logic programs. *CoRR*, cs.LO/0308007, 2003. ↑205

[128] Konstantinos Sagonas and Peter J. Stuckey. Just enough tabling. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '04, pages 78–89, New York, NY, USA, 2004. ACM. ↑146, ↑147, ↑148, ↑151, ↑154, ↑158

[129] João Santos and Ricardo Rocha. On the efficient implementation of mode-directed tabling. In Kostis Sagonas, editor, *Practical Aspects of Declarative Languages*, volume 7752 of *Lecture Notes in Computer Science*, pages 141–156. Springer Berlin Heidelberg, 2013. ↑183, ↑196

[130] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012. ↑59, ↑82, ↑148, ↑158, ↑161, ↑162, ↑174, ↑193

[131] Peter Schachte. Global variables in logic programming. In *Proceedings of the International Conference on Logic Programming (ICLP 1997)*, pages 3–17, 1997. ↑57

[132] Joachim Schimpf. Logical loops. volume 2401 of *LNCS*, pages 224–238, 2002. ↑61

[133] Joachim Schimpf and Kish Shen. ECLiPSe From LP to CLP. *Theory and Practice of Logic Programming*, 12(1-2):127–156, 2012. ↑28, ↑30, ↑136

[134] Tom Schrijvers, Bart Demoen, Markus Triska, and Benoit Desouter. Tor: Modular search with hookable disjunction. *Science of Computer Programming*, 84(0):101 – 120, 2014. ↑136

[135] Tom Schrijvers, Bart Demoen, and David Scott Warren. TCHR: A framework for tabled CHR. *Theory and Practice of Logic Programming*, 8(4):491–526, Jul 2008. ↑199

[136] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. Towards typed Prolog. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 693–697. Springer Berlin Heidelberg, 2008. ↑206

[137] Tom Schrijvers, Peter Stuckey, and Philip Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663–697, November 2009. ↑57, ↑136

[138] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter Stuckey. Search Combinators. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011)*, pages 774–788. Springer, 2011. ↑57

[139] Tom Schrijvers, Guido Tack, Pieter Wuille, Horst Samulowitz, and Peter Stuckey. Search combinators. *Constraints*, 18(2):269–305, 2013. ↑136

[140] Christian Schulte et al. Gecode, the generic constraint development environment, 2013. `http://www.gecode.org/`, accessed March 2013. ↑57

[141] Alexander Serebrenik, Tom Schrijvers, and Bart Demoen. Improving Prolog programs: Refactoring for Prolog. *Theory and Practice of Logic Programming*, 8(2):201–215, March 2008. ↑206

[142] Silvija Seres, Michael Spivey, and Tony Hoare. Algebra of logic programming. In *International Conference on Logic Programming*, pages 184–199. Palgrave MacMillan, 1999. ↑135

[143] Marek Sergot. Minimal models and fixpoint semantics for definite logic programs, January 2005. Last viewed on 04/02/2016 at `https://www.doc.ic.ac.uk/~mjs/teaching/KnowledgeRep491/Fixpoint_Definite_491-2x1.pdf`. ↑144

[144] Helmut Simonis, Paul Davern, Jacob Feldman, Deepak Mehta, Luis Quesada, and Mats Carlsson. A generic visualization platform for CP. In David Cohen, editor, *Proceedings of Principles and Practice of Constraint Programming — CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 460–474. Springer, 2010. ↑49

[145] Dorai Sitaram. Handling control. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 147–155, New York, NY, USA, 1993. ACM. ↑62, ↑128

[146] Zoltan Somogyi and Konstantinos Sagonas. Tabling in Mercury: Design and implementation. In Pascal Hentenryck, editor, *Practical Aspects of Declarative Languages*, volume 3819 of *Lecture Notes in Computer Science*, pages 150–167. Springer Berlin Heidelberg, 2006. ↑157

[147] John Michael Spivey. Algebras for combinatorial search. *J. Funct. Program.*, 19(3-4):469–487, July 2009. ↑135, ↑138

[148] L. S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In *Constructing Logic Programs*, pages 127–140. John Wiley, 1993. ↑4, ↑69

[149] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*, chapter Foreword by D.H.D. Warren. MIT Press, Cambridge, MA, 2nd edition, 1994. ↑4

[150] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, MA, 2nd edition, 1994. ↑21, ↑23, ↑49

[151] Terrance Swift and David S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In Tomi Janhunen and Ilkka Niemelä, editors, *Logics in Artificial Intelligence*, volume 6341 of *Lecture Notes in Computer Science*, pages 300–312. Springer Berlin Heidelberg, 2010. ↑181, ↑182, ↑183, ↑194, ↑195

[152] Terrance Swift and David S. Warren. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, January 2012. ↑59, ↑79, ↑148, ↑152, ↑174, ↑193

[153] Terrance Swift and David Scott Warren. *The XSB System – Volume 1: Programmer's Manual*, pages 91–94. Version 3.6.x edition, April 2015. ↑146

[154] Terrance Swift and David Scott Warren. *The XSB System – Volume 1: Programmer's Manual*, pages 80–136. Version 3.6.x edition, April 2015. ↑153

[155] Terrance Swift and David Scott Warren. *The XSB System – Volume 1: Programmer's Manual*, pages 24–30. Version 3.6.x edition, April 2015. ↑206

[156] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer Berlin Heidelberg, 1986. ↑147

[157] Paul Tarau. The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. *TPLP*, 12(1-2):97–126, 2012. ↑62, ↑91, ↑92, ↑97, ↑179

[158] Paul Tarau and Veronica Dahl. Logic Programming and Logic Grammars with First-order Continuations. In *LOPSTR '94*, volume 883, June 1994. ↑91, ↑179

[159] Christian Theil Have and Henning Christiansen. Efficient tabling of structured data using indexing and program transformation. In Claudio Russo and Neng-Fa Zhou, editors, *Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, pages 93–107. Springer Berlin Heidelberg, 2012. ↑151

[160] Dave Thomas and Andrew Hunt. *Programming Ruby: the Pragmatic Programmer's Guide.* Addison-Wesley Longman Publishing Co., Inc., 2000. ↑96

[161] David Toman. Memoing evaluation for constraint extensions of Datalog. *Constraints*, 2(3-4):337–359, 1997. ↑199

[162] Markus Triska. The finite domain constraint solver of SWI-Prolog. In *Proceedings of the 11th International Symposium on Functional and Logic Programming (FLOPS 2012)*, pages 307–316, 2012. ↑28

[163] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976. ↑144, ↑145

[164] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search.* MIT Press, 2005. ↑57, ↑137

[165] Pascal Van Hentenryck and Laurent Michel. Nondeterministic control for hybrid search. *Constraints*, 11(4):353–373, 2006. ↑57, ↑137

[166] Peter Van Roy. A useful extension to Prolog's definite clause grammar notation. 24(11):132–134, 1989. ↑61

[167] Maarten W. Van Someren. Understanding students' errors with Prolog unification. *Instructional Science*, 19(4-5):361–376, 1990. ↑12

[168] Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. Fixing non-determinism. ↑157, ↑187, ↑198

[169] Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. The table monad in Haskell. In *IFL 2015*, 2015. ↑157, ↑198

[170] P. Wadler. The essence of functional programming. In *POPL '92: 19th Symposium on Principles of Programming Languages*, pages 1–14, 1992. ↑101

[171] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983. ↑4, ↑17, ↑79, ↑177

[172] Jan Wielemaker and Ulrich Neumerkel. Precise garbage collection in Prolog. In *CICLOPS '08*, pages 1–15, 2008. ↑85

[173] Jan Wielemaker and Vítor Santos Costa. *Practical Aspects of Declarative Languages: 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, chapter On the Portability of Prolog Applications, pages 69–83. Springer Berlin Heidelberg, 2011. ↑206

[174] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *THEORY AND PRACTICE OF LOGIC PROGRAMMING*, 12(1-2):67–96, 2012. ↑28, ↑84, ↑207

[175] Neng-Fa Zhou. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218, 2012. ↑30, ↑136, ↑148, ↑174, ↑193

[176] Neng-Fa Zhou and Jonathan Fruhman. *A User's Guide to Picat*, 1.2 edition, June 2015. ↑158

[177] Neng-Fa Zhou, Y. Kameya, and T. Sato. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 2, pages 213–218, Oct 2010. ↑183, ↑196

[178] Neng-Fa Zhou, Taisuke Sato, and Yi-Dong Shen. Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming*, 8(01):81–109, 2008. ↑154

[179] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a linear tabling mechanism. In Enrico Pontelli and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages*, volume 1753 of *Lecture Notes in Computer Science*, pages 109–123. Springer Berlin Heidelberg, 2000. ↑151, ↑154

[180] Neng-Fa Zhou and Christian Theil Have. Efficient tabling of structured data with enhanced hash-consing. *Theory and Practice of Logic Programming*, 12:547–563, 7 2012. ↑149

[181] David Zook, Emir Pasalic, and Beata Sarna-Starosta. *Practical Aspects of Declarative Languages: 11th International Symposium, PADL 2009, Savannah, GA, USA, January 19-20, 2009. Proceedings*, chapter Typed Datalog, pages 168–182. Springer Berlin Heidelberg, 2009. ↑206

List of Figures

List of Tables

Index