Declarative Language Extensions for Prolog Courses

Ulrich Neumerkel
Technische Universität Wien, Austria
ulrich@complang.tuwien.ac.at

Markus Triska
Technische Universität Wien, Austria
Markus.Triska@tuwien.ac.at

Jan Wielemaker
Universiteit van Amsterdam, Netherlands
J.Wielemaker@uva.nl

Abstract

In this paper we present several extensions to support a more declarative view of programming in Prolog. These extensions enable introductory Prolog courses to concentrate on the pure parts of Prolog for longer periods than without. Even quite complex programs can now be written free of any reference to the more problematic constructs. Our extensions include an alternate way to handle the occurs-check, efficient side-effect free I/O with DCGs, and a uniform approach to integer arithmetic that overcomes the disadvantages of arithmetical evaluation and finite domain constraints, but combines and amplifies their strengths. All extensions have been included recently into the SWI-Prolog distribution[1].

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Constraint and logic languages; D.3.3 [*Programming Languages*]: Language Constructs and Features; K.3.2 [*Computers and Education*]: Computer and Information Science Education

General Terms Design, Human Factors, Languages

Keywords Teaching Prolog, side-effect free I/O, occurs-check, constraints

1. Introduction

The traditional approach to teaching Prolog codified in the late 1970s confronts the student with a collection of Prolog's weaknesses and imperfections. Ad hoc control constructs, incorrect implementations of negation and unification, side-effecting predicates, and moded arithmetic are all covered by introductory Prolog courses, obscuring the profound ideas behind. In fact, taking traditional Prolog, it is almost impossible to write a non-trivial program without any significant violation of the pure logic programming paradigm. In such a context, the learner has quite some difficulty to see through all the accumulated archaisms the actual declarative beauty. On the other hand, there has been significant progress within Prolog systems overcoming its original deficiencies.

It is one of the biggest challenges in Prolog courses to remove those thorny and distracting elements giving way to better understanding of the logic programming paradigm. As a first step in that direction one might reconsider the course's syllabus (Neumerkel 1995) to cover pure monotonic programs in the beginning as long as possible. This limits the first part to pure programs and DCGs with numbers represented as terms in Peano arithmetic style using `0` and `s/1`. Pure monotonic programs have many advantages compared to impure or non-monotonic ones. They allow advanced diagnostic slicing-techniques (Neumerkel and Kral 2002) to explain various errors. Even procedural properties like non-termination (Neumerkel and Mesnard 1999) can be succinctly explained. Alternate execution strategies like iterative deepening are also reserved to monotonic programs. We use such strategies in situations where Prolog's simple but efficient execution strategy ends up looping.

For the above reasons, staying with monotonic programs is beneficial. Unfortunately, monotonicity limits also applicability considerably, since the only way to get feedback from such programs are a shell's answer substitutions. A possible way out, chosen by one of the authors, was to develop a specialized programming environment (Neumerkel and Kral 2002) that permits to display answer substitutions in a side-effect free manner. This extended the cases of interesting pure programs somewhat and permitted to integrate many diagnostic programs. However, the shortcomings of this approach remain significant as the step from within the environment to the outside is discouraging.

In this paper we consider another, complementary approach that strengthens the pure parts of the Prolog language in general. The proposed extensions can be used within or without an environment. We enlarge Prolog's pure monotonic parts to cover restricted forms of side-effect free I/O in Sect. 3, as well as arithmetic that subsumes even finite domain constraints in Sect. 4. But first, we reconsider in Sect. 2 an often denied problem that already haunts beginners in their pure programs.

2. The occurs-check remains a problem

Queries like `?- X = s(X).` should fail, as there is no finite term that satisfies the equation. The unification algorithm employed requires a separate test to ensure that a variable does not occur as subterm of the term it should be bound to. This test is known as occurs-check.

Most Prolog implementations do not include the occurs-check within default unification for efficiency reasons. It is generally perceived to be a non-issue. Quite in contrast to this commonly held view, our experience in teaching shows that occurs-checks help identify typical beginner's errors. Further, termination results from proof- or inference-systems based on norms over the natural numbers—the overwhelming majority of current systems, like cTI (Mesnard and Neumerkel 2001), all assume that the analysed programs only produce first order terms, and thus most probably use unification with occurs-check. To profit from those advanced tools, there is no way around. For all these reasons we developed a new unification mode.

[1] http://www.swi-prolog.org

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FDPE'08, September 21, 2008, Victoria, BC, Canada
Copyright © 2008 ACM 978-1-60558-068-5/08/09. . . $5.00

2.1 The occurs-check in ISO Prolog

In the end 1980s, when ISO commenced to standardize Prolog (ISO 1995), most Prolog systems—including SWI-Prolog—contained an unreliable unification algorithm that sometimes worked, sometimes looped and sometimes aborted the entire Prolog system in error. Given such "current practice" for the very core operation of a programming language, ISO was faced with an almost unsolvable task. The existing systems could not be ruled out—they represented the overwhelming majority. Interest in adopting better unification algorithms was low for fear of reduced speed (LIPS-ratings)—the perceived selling argument at that time. It was a highly laborious and time consuming effort to find a way out to define unification around such undefinable behavior (Deransart 1990; Scowen 1990). The solution consisted in identifying a set of programs not subject to occurs-check—NSTO (Deransart et al. 1991) that can be reliably executed on all systems, regardless of their imperfections. Unifications that are STO, are required to be performed with the built-in predicate `unify_with_occurs_check/2`. For standard conforming programs, the programmer must add manually such extra goals, a tedious activity with no feedback about its outcome—be it success or failure. Even worse, the NSTO-property is undecidable for a given program. And to decide it for a single unification, an expensive nondeterminate algorithm is required, that incurs in the worst case exponential overheads (Deransart et al. 1991) compared to the usually employed unification algorithms. This quite complex situation and the absence of tools to decide (at least partially) the NSTO-property, persuaded Prolog implementers one by one, to swallow the bitter pill and implement a reliably terminating unification algorithm based on rational trees. In this manner, embarrassing bus errors were avoided, but problems with semantics still remained.

Several studies concluded (Deransart et al. 1991; Apt and Pellegrini 1992) that many well-known Prolog programs are NSTO. From these studies however, we can only conclude that the occurs-check is not a problem for most final, correct programs.

2.2 Sources of STO programs

These conclusions do not hold for the many intermediary incorrect states of a program prior to its completion. And even less for the intermediary states beginners produce. There are several reasons, why beginners produce erroneous STO-programs. The first is that the concept of the logic variable is rather difficult to grasp for students accustomed to imperative programming. Not surprisingly, they try to re-assign values to the same variable. For example, goals like `L = [El|L]` are written with the intention to add an element to a list. A further reason frequently surfacing in more advanced parts are misunderstandings concerning the nature of differences as they appear in difference lists. While some of those errors are detectable statically with an ad hoc analysis, the general case defies simple minded approaches. We are thus left in a state of uncertainty about the NSTO property of a program.

Apart from these errors, there are cases where the STO property leads to correct but highly inefficient programs. Consider `pairmatrix/1` describing quadratic matrices of pairs.

```
pairmatrix(S) :-
    maplist(same_length(S),S),
    maplist(maplist(pair),S).

same_length(Xs, Xs).
same_length([_X|Xs],[_Y|Ys]) :-
    same_length(Xs, Ys).

pair(_A-_B).
?- pairmatrix(M).
?- length(M,N), pairmatrix(M).
```

Ideally, `pairmatrix/1` finds for each length the most general quadratic matrix of pairs. The query `?- pairmatrix(M).` yields with rational tree unification the solution `M = []` before ending up in an infinite loop. When enabling unification with occurs-check the situation is only slightly improved. A single further solution `M = [[_A-_B]]` is found. Clearly, unification with occurs-check did not help much to improve this case. It happens quite frequently that programs with failing occurs-check end up in loops (Neumerkel 1992).

Querying with a given length, results in the desired answers. However, both unification modes get slow very quickly taking about 10^n inferences. For the length of 8, occurs-check unification requires more than one million inferences, and rational tree unification even 715 million inferences.

It is just problems like these, that are very hard to locate—not only for students. Our sophisticated termination analyzers are of no help in this situation—they simply agree that `pairmatrix/1` terminates for every given length. The situation in this particular case is even less intuitive, as execution with occurs-check is faster than rational tree unification. Commonly it is assumed that the occurs-check will incur overheads and not speedups. Silent failure during occurs-check is evidently no viable alternative. To locate the problem we need to know that there has been some unification that failed due to the occurs-check. The reader is invited to try to locate the error alone. In the appendix, we give an explanation to the problem.

2.3 An error mode for unification

We solved this problem by implementing a new execution mode that issues an error during those STO-unifications that are easy to detect, signaling that the program is STO. In this manner *any* creation of an infinite term is prevented. With our new unification, the problematic STO-situation in the example above is identified after 5 inferences. Note that in this example a type system might have helped us. However, to date, we are unaware of any type system for Prolog that produces actual guarantees as type systems do so reliably for functional languages.

Another kind of errors that are hard to locate and cannot even be located with type systems are incorrect uses of differences like the query `?- phrase([1],Xs,Xs).` Such cases correspond somewhat to infinite lists in pure functional programming languages.

2.4 ISO compliance

Our new execution mode differs from standard unification only in cases where the standard has explicitly foreseen undefined cases, cf. Note 3 of 7.3.4 (ISO 1995). It can be seen as a means to identify STO cases fulfilling partially the "not required" test of Note 4 of 7.3.4: *STO and NSTO are decidable properties for a single unification. However processors are not required to include such a test.*

Another decision requiring justification concerns the exact error to produce. At the first glance, type errors seem to be the most attractive. However, type errors in the sense of ISO (7.12.2 error classification clause b) assume that there is *one* incorrect culprit. In our case of an attempted unification, this could only be the resulting infinite term. But there are unifications that produce an error, but would fail with rational trees as for example `f(X,a) = f(s(X),b)`. Further, one central aim of this improved unification is to prevent the creation of infinite terms. By transmitting an infinite term for a type error, this guarantee would be undermined.

Another error candidate is the representation error which is foreseen for situations where an implementation defined limit is breached. Again, the error is not able to represent two arguments with arbitrary terms. As the error situation does not

fit into the existing standard errors, we use the new error term `occurs_check(TermA, TermB)` where the arguments are the terms prior to the attempted unification.

2.5 Implementation

For long, the occurs-check was avoided at any price (Colmerauer 1982)—and not without reason. Even very simple programs incur significant overheads. We have put thus considerable effort into reducing the overheads for the new occurs-check unification—many tiny benchmarks are even *faster* than without and most variable-term unifications are performed in constant time whereas general occurs-check unification requires time linear in the size of the bigger term. Focusing on an efficient implementation was very important as efficiency problems surface rapidly. To illustrate the overheads one faces, consider the minimal example of a grammar. The nonterminal `as//0` describes a sequence of the character a.

```
as --> [].
as --> "a", as.
```

For the above grammar, recent DCG-implementations (Moura 2008) will not incur any overhead due to occurs-check[2]. So recognizing a sequence `as` will take time linear to its length. However, by simply reformulating this grammar to the form below, quadratic overheads appear. For a sequence of thousand characters, the overheads between naive occurs-check and our implementation surpass two orders of magnitude. In more realistic programs, these overheads are often higher due to more complex nesting and nondeterminism. Unnecessary overheads are caused by differences passing the list back. As a rule of thumb, for every comma after a nonterminal in a grammar rule, a superfluous occurs-check with cost proportional to the remaining list is performed. With our optimized occurs-check implementation, the passing of a given list in a DCG does not perform any occurs-check to search the list.

```
as --> [].
as --> a, as.

a --> "a".
```

Despite all our efforts, overheads caused by occurs-check are still considerable. For this reason this new safe execution mode might remain reserved for learning, testing, and debugging. It is definitely preferable to use it in place of the classical occurs-check that results in silent failures and therefore leaves the programmer in the unknown. It also helps to predict the behavior under rational tree unification: If a goal does not produce an occurs-check error, it will behave exactly the same way with rational tree unification.

2.6 Proving termination

Most existing systems for proving or inferring termination properties in Prolog programs like cTI (Mesnard and Neumerkel 2001) assume that terms can be mapped onto the natural numbers according to a given norm. This implies that only finite terms can be modeled. Take as an example the most elementary norm known as term-size. Here all constants are mapped to 0 and function symbols count as 1 plus the sum of the arguments' sizes. The terms 0, s(0), t(s(0),0) are thus mapped onto 0, 1 (1+0), and 2 (1+(1+0)+0) respectively. The goal `s(0) = s(s(0))` would be mapped onto $1 = 2$. The failure is thus evident even by investigating only the terms' sizes. Now consider the goal `s(X) = X` which is mapped to $1 + X = X$, an equation without any solution in the natural numbers. From this we conclude that the goal `s(X) = X` does not have a solution. Certainly, this is not the case with rational tree unification. Here, this goal will succeed with `X` bound to a term that cannot be mapped consistently onto the natural numbers.

Our new execution mode shares all termination results with the classical occurs-check mode. Therefore, we can take results of termination inference literally.

2.7 Usage

Our new improved unification mode can be activated by putting the directive `:- set_prolog_flag(occurs_check,error).` into the startup file (`.plrc` or `pl.ini`). Alternatively, SWI-Prolog's unit-testing environment `plunit` (Wielemaker 2006) identifies STO-situations of all tests by comparing the results in the different unification modes.

3. Pure I/O via DCGs

I/O in Prolog is the first big show stopper to more interesting pure programs. There is no way around showing the deplorable side-effecting predicates when we want to parse some file. These predicates are the ideal attention-sink for students. Declarative reasoning simply does not work any more. The only debugging help for students are tracers that reinforce a procedural viewpoint by showing step-by-step the actual execution trace. Very quickly, the entire declarative stance one had built in a week-long path is unlearned within minutes. But even if we were willing to give up all purity, side-effecting programs remain very much prone to errors, as we have to fully understand and tame Prolog's nondeterminism. What did other languages do in that situation?

Pure declarative languages have no direct connection to the outside world. Their inner formalisms need to find their own meandering track to reach that state-ridden world—most often a search that is not obvious at its outset, as testified by the evolution of the I/O system (Hudak et al. 2007) in the Haskell programming language. When considering declarative languages with pure I/O, we see three different approaches: streams/continuations, monads and uniqueness-types. These approaches have been developed for functional languages, concurrent logic languages, and a moded logic language. Where Prolog differs so fundamentally, is its deep nondeterminism that we do not want to give up at any price. It makes a big difference that this nondeterminism is built-in. If backtracking would be provided in a monadic manner, things would be significantly simpler. This nondeterminism undermines the quasi single threadedness properties implicitly present in all three approaches and required to "do something". While it is obvious that Clean's (Barendsen and Smetsers 1993) and Mercury's (Somogyi et al. 1996) uniqueness-types guarantee single threadedness, this is also true for the other approaches. A stream element can only be reduced once, regardless of the number of referents. It seems that there is no way to perform general I/O operations as long as deep nondeterminism is in play. We have therefore settled with less than general interaction—I/O via DCGs on files (seekable devices). In this manner we can preserve Prolog's nondeterminism.

```
:- use_module(library(pio)).
... --> [] | [_], ... .

?- phrase_from_file((..., "searched", ...), file).
?- phrase_from_file((...,"a"|...,"b"), file).
?- phrase_from_file((...,("a"|"b")), file).
```

Consider searching a file for a given string. The non-terminal `...` ("ellipsis") describes an arbitrary sequence of arbitrary length. When we are interested in a particular string within a file, there must be an arbitrary sequence before and after that string. The entire file must be thus of the form `(...,"searched",...)`. Library `pio` gives us access to pure I/O via `phrase_from_file/2`

[2] Older DCG translations that used an auxiliary predicate `'C'/3` for terminal symbols had even here quadratic overheads.

and `phrase_to_file/2`. SWI-Prolog performs all of the above queries in constant space. Space requirements are thus independent of the file's size. Note that the second query above is a very naive way to assure that the last character in a file is a or b. Our naive formulation leads to two linear searches—still this operation runs in constant space. Quite symmetrically, output to a file is provided with `phrase_to_file(NonTerminal, Filename)`. In many cases our pure formulations are comparable in efficiency with direct side-effecting definitions.

Our approach to I/O is less powerful than the I/O systems in other declarative languages, but powerful enough to keep introductory Prolog courses clean from traditional I/O. While functional approaches are able to cover the entire spectrum of I/O activities, and are thus able to change and sense the world at the same time, we only allow unidirectional reading and writing alone. On the other hand, we allow in both parts full nondeterminism and profit from efficient buffering techniques that make overheads vanish without sophisticated compilation techniques. To make this approach effective, it was necessary to improve SWI-Prolog's garbage collector (Wielemaker and Neumerkel 2008).

4. Uniform arithmetic

We have three different ways to represent and use natural numbers in Prolog. The simplest way are terms in the style of Peano arithmetic. Then, we have the moded built-in predicates like `is/2` and its associated arithmetical comparison operators. Finally, there are the finite-domain predicates that define a different set of relations between integers. All three representations have certain strengths and weaknesses. To simplify matters, we unified the last two.

While moded arithmetic and finite-domain predicates share the same integers, their uses were originally quite orthogonal. In moded arithmetic the entire expression must be ground at evaluation time but integers can be of arbitrary size. Finite domain variables had originally (van Hentenryck 1989) to be defined using ranges that were declared statically. At runtime no further change (i.e. enlargement) to the declared range was possible. To change the size of a problem the file had to be edited manually. Subsequent implementations removed such strong restrictions step-by-step. One of the most significant generalizations was achieved 1996 with the first release of a finite domain system for SICStus Prolog (Carlsson et al. 1997). The equality relation `#=/2` could now be used in place of `is/2` for integers. Domains could be open (infinite), the restriction for finite domains was removed for general relations and only kept for labeling—the single place where this restriction makes sense. Extending labeling to infinite domains would make labeling nonterminating. For infinite domains, `labeling/2` issues an instantiation error, thereby enforcing termination. After a decade of using this quite advanced implementation in Prolog programming courses, a new implementation was created for SWI-Prolog that built upon our decade of experience using `clpfd` in programming courses and further generalizes and improves the state of the art.

4.1 Extending domains

Domain limits are no longer restricted to small values (typically below word sizes), but can be of arbitrary size, like in the query `?- abs(X) #=< 7^7^7.` Traditional finite domain applications will not profit from this extension, but it allows us to solve problems that have been known rather in integer programming circles. For example, the famous 7-11 problem[3] already surpasses the limits of SICStus on 32-bit systems. Also, this extension improves clpfd for general multi-moded functions.

[3] Attributed to Doug Brumbaugh but also to Don Edwards by (Pritchard and Gries 1983). See also (Michalewicz and Fogel 1998) chapter III.

4.2 Subsuming is/2 efficiency-wise

To eradicate is/2 from introductory courses completely, clpfd constraints are now specialized for the simply evaluable integer cases. For tiny integer intensive inner loops, clpfd's overhead is now about 30% compared to a direct usage of is/2. Prior to this optimization, overheads reached two orders of magnitude.

4.3 An always terminating clpfd

Termination within a finite domain constraint solver is trivially guaranteed as long as all domains are finite. By allowing infinite domains we gain significant expressibility but also may receive nonterminating propagation in return. Many clpfd-systems do not terminate for cases like `X#>abs(X)` or more generally `X#>Y,Y#>X,X#>=0`. This is not a problem for classical finite domain applications, that where developed at a time when infinite domains did not exist. However, it subverts or weakens termination proofs (Mesnard and Neumerkel 2001). All provers assume that a simple goal like `X = 1` will terminate, regardless of the constraint store accumulated by previous goals. An analysis taking into account such irregularities in existing provers would have to involve considerable complexity. It also makes explanations for nontermination more complex (Neumerkel and Mesnard 1999).

A frequent source of such "senseless" nonterminating constraints stems from misunderstandings of the nature of logical variables similar to cases requiring occurs-check, discussed in Sect. 2. These misunderstandings result in equations with multiple occurrences of variables on both sides that have no solution but do not terminate in many systems.

In SWI-Prolog's clpfd-implementation all predicates terminate always, regardless of the actual equations involved. A strict interpretation of the ISO error model facilitates both monotonicity and termination. This makes programming with constraints more predictable and helps to localize actual sources of nontermination (unrelated to clpfd). In particular, labeling always terminates. This permits us to construct simple termination proofs for predicates using labeling. Most predicates exploiting the actual constraint functionality consists of two parts: In the first called core-relation, the domains of variables are posted and the actual constraints are established. The second part contains the actual search using labeling. The search is quite often a time consuming activity—direct observation of universal termination of the entire predicate is therefore unrealistic. But testing for termination of the core-relation is realistic. A terminating core-relation is a proof for universal termination of the entire relation. Note that in systems with nonterminating unifications such proofs are invalid.

4.4 Syntactic disambiguation

Using SICStus Prolog's finite domain constraint solver in programming courses, it soon became apparent that the syntactic choice of operators added unnecessary confusion to students. The reification operator `#<=` is often confused with and used in place of the comparison `#=<`. This should come as no surprise since most programming languages except Prolog denote "smaller or equal" with `<=`, whereas Prolog uses `=<`. Typos like these cannot be reliably detected with type systems as both are relations between integers. Within SICStus we removed `#<=` which is symmetric to `#=>`. In SWI, we took the liberty to rename all reification relations to `#<==`, `#<==>`, and `#==>`.

4.5 Complete labeling for extrema

All preexisting implementations that provide maximization within labeling provide only an incomplete implementation for labeling extrema. In most cases only one (arbitrary) solution is produced. Such an incomplete implementation causes difficulties similar to

those caused by the !-operator used to prune the search space. Writing a completed labeling procedure based on an incomplete one is a relatively tedious task—definitely too difficult for beginners. However, for those rare cases where only a single solution should be produced, committing our complete implementation to the first solution is trivial.

5. Conclusions

We presented several improvements to Prolog programming especially well suited to dispose common impurities. We hope that the presented libraries for Prolog available in the current SWI-Prolog distribution will contribute to spread a purer and better programming style within introductory Prolog courses.

A. An explanation for `pairmatrix/1`

The predicate `pairmatrix/1` given in 2.2 shows in a prototypical situation how the absence and even the presence of the occurs-check leads to a highly inefficient program that can easily be identified as STO. The program is a slight simplification of an actual programming error. We used the higher-order `maplist/2` to keep the presentation short.

From outside, the user has the impression that `pairmatrix/1` is correctly defined as only correct solutions are shown. Also an algorithmic debugger (Shapiro 1983) has no chance to locate the error. To understand the problem we have to delve into the predicate's definition despite its correctness. The rule of `pairmatrix/1` is a conjunction of two goals. The first goal produces besides the actual correct solutions also incorrect solutions some of which contain infinitely nested lists. The second goal effectively eliminates all undesired solutions of the first goal and thus all infinite trees, by insisting on a list of list of pairs, whereas incorrect solutions have a list in place of where the pair structure `-/2` should be.

The first goal `maplist(same_length(S),S)` should define a squared matrix, i.e. a list of lists, all of the same length. However, the definition of `same_length/2` is too general. For example, the goal `?- same_length(unlist, unlist).` is true, while `unlist` is just a constant, but no list. It is the generalized fact `same_length(Xs, Xs)` that is responsible for the confusion. For each list of length n we have now n further solutions, all of which are unexpected. Those extra solutions set some elements of one list equal to elements in the other list. But one list occurs as element of the other. This means that the additional solutions all contain some list-of-list-of-list. Some of those solutions will contain infinitely nested lists, but not all of them. Consider the query `?- S = [_,_], maplist(same_length(S),S).` that produces eight additional and unexpected answers beside the expected answer `S = [[_,_],[_,_]]`. All of those unexpected answers contain infinitely nested lists, except `S = [[_,[_A,_B]],[_A,_B]]`. Considering only infinitely nested lists would therefore prevent to understand the actual problem. By specializing the fact to the single meaningful case that does not imply unwanted unification `same_length([], [])` the error is removed.

Generalizations as the fact `same_length(Xs,Xs).` might be considered a programming error, but it is quite common practice in Prolog to define relations slightly too general as witnessed by the idiomatical `append/3`-predicate. Similarly, consider the expansion of the DCG-rule `epsilon --> [].` which is the very fact `epsilon(Xs,Xs).` that we considered incorrect for the predicate `same_length/2`! The exact circumstances, where such a generalization is acceptable and where not, are not easily accessible to most programmers. Tools that help to clarify the actual role of terms by identifying STO-situations are therefore very helpful.

Acknowledgments

We would like to thank Tom Schrijvers for his solver `bounds` that we used as a starting point for the new `clpfd` implementation.

References

K. Apt, A. Pellegrini. Why the occur-check is not a problem. PLILP. 1992. LNCS 631.

E. Barendsen, J.E.W. Smetsers. Conventional and uniqueness typing in graph rewrite systems. FSTTCS. 1993. LNCS 761.

M. Carlsson, G. Ottosson, B. Carlson. An Open-Ended Finite Domain Constraint Solver. PLILP. 1997.

A. Colmerauer. Prolog and Infinite Trees. Logic Programming, K.L. Clark, S.-A. Trnlund (eds), 1982.

P. Deransart. The problem of unification in standard Prolog - a discussion paper in ISO/IEC JTC1 SC22 WG17 N59 Prolog, Vienna papers - 2 - 1990.

P. Deransart, G. Ferrand, M. Teguia. NSTO Programs (Not Subject to Occur-Check). ISLP. 1991.

P. van Hentenryck. Constraint Satisfaction in Logic Programming. MIT-Press, 1989.

P. Hudak, J. Hughes, S.Peyton Jones, Ph. Wadler. A History of Haskell: Being Lazy with Class. HOPL ACM 2007.

ISO/IEC 13211-1 Programming languages - Prolog - Part 1: General core. 1995. (7.3 Unification)

F. Mesnard, U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. SAS. 2001, LNCS 2126.

Z. Michalewicz, D.B. Fogel. How to Solve It: Modern Heuristics. Springer, 1998/2000.

P. Moura (ed.) ISO/IEC DTR 13211-3:2006 Definite clause grammar rules (Draft of April 2008).

U. Neumerkel. Pruning Infinite Failure Branches in Programs with Occur-Check. LPAR. 1992.

U. Neumerkel. How to teach Prolog. Teaching beginners Prolog. Tutorial PAP. 1995.

U. Neumerkel. Teaching Prolog and CLP, Tutorial ICLP. 1997.

U. Neumerkel, F. Mesnard. Localizing and explaining reasons for nonterminating logic programs with failure slices. PPDP. 1999. LNCS 1702.

U. Neumerkel, St. Kral. Declarative program development in Prolog with GUPU. WLPE. 2002.

P. Pritchard, D. Gries. The Seven-Eleven Problem. Cornell University TR 83-574 http://hdl.handle.net/1813/6414 1983.

R. Scowen. Unification - another discussion paper in ISO/IEC JTC1 SC22 WG17 N59 Prolog, Vienna papers - 2 - 1990.

E. Shapiro, Algorithmic Program Debugging. MIT Press, 1983.

Z. Somogyi, F. Henderson, Th. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. JLP 29(1-3), 1996.

J. Wielemaker. A programming environment for developing large applications. LPE. 1990.

J. Wielemaker. An overview of the SWI-Prolog programming environment. WLPE. 2003.

J. Wielemaker. Prolog Unit Tests. Manual. http://www.swi-prolog.org/packages/plunit.html 2006.

J. Wielemaker, U. Neumerkel. Garbage Collection for Pure Input Streams. Submitted paper. 2008.