Implementing LP Systems with CP Techniques

BENOIT DESOUTER
Department of Applied Mathematics, Computer Science and Statistics
Ghent University
Krijgslaan 281, S9
9000 Gent, Belgium
E-mail: Benoit.Desouter@UGent.be

submitted 12 May 2013; accepted 5 June 2013

Abstract

This article gives an overview of my research activities since October 2012. It discusses various areas of logic programming with constraint programming as a common denominator.

KEYWORDS: logic programming, constraint programming, Datalog, Leapfrog Triejoin, Prolog, heuristic tree search, delimited continuations, coroutines, monads

1 Introduction

Since the start of my doctorate on October 15, 2012, I have worked in several areas of logic programming with constraint programming as a common denominator.

During the orientation phase of my Ph.D. I have explored several logic programming topics: Prolog, Datalog, ASP, CLP, . . . I coauthored four papers, and one technical report. One article has been accepted at SoCP, a second at CICLOPS; we have recently reviewed a third that has been conditionally accepted at ICLP. Furthermore, I have written a contribution to the newsletter of the Association for Logic Programming (Desouter and Schrijvers 2013a).

Three of the above papers investigate how we can modify the Prolog control strategy to deviate from a standard depth-first search (DFS). Although DFS is a convenient and automatic choice for many, notably smaller problems, the need for different strategies often arises. We have explored approaches to implement these in a modular and composable way.

For one of those approaches, we have implemented delimited continuations in both a WAM-based and a ZIP-based Prolog. Delimited continuations are a famous control primitive that originated in the functional programming world. We have brought them to Prolog for the first time.

Apart from modular search, I also gained a strong interest in Datalog. LP is a common formalism for the field of databases and CSP, both at the theoretical level and the implementation level in the form of Datalog and CLP. In the past, close correspondences have been made between both fields at the theoretical level. Yet correspondence at the implementation level has been much less explored. I have worked on integrating them at the implementation level. Concretely, I have investigated the relationship between the efficient Leapfrog Triejoin execution algorithm of Datalog and a generic CP execution scheme. Exploring their unification in depth is the core topic of my Ph.D..

In the following sections I discuss the above mentioned points in reverse chronological order, but presenting the most closely related works together.

2 Integrating Datalog and Constraint Solving

2.1 Problem description

Datalog is a query language for deductive databases used in a variety of applications, such as retail planning, modelling, . . . based on first-order logic. Constraint programming (CP) is a well-known paradigm in which relations between variables describe the properties of a solution to the problem we wish to solve. The strategy how to actually compute these solutions is left to the computer. Thus, just like Datalog, constraint programming is declarative.

It is well known, at least at the theoretical level, that Datalog and CP, in particular CLP, have much in common (Vardi 2000). Yet correspondence at the implementation level has been much less explored.

We aim to show that Datalog and CP are also compatible at this level. We do so by starting from a standard CP implementation scheme, as formulated by Schulte and Stuckey (Schulte and Stuckey 2008), and specialize it to obtain a recently documented Datalog execution algorithm called *Leapfrog Triejoin* (Veldhuizen 2012). This opens up the possibility for further cross-fertilization between actual CP and Datalog systems. In particular, we aim to integrate CP propagation techniques in the Datalog join algorithm for query optimization.

In Datalog, tables are normally stored as trees, while in CP a union of intervals $\bigcup[lb_i, ub_i]$ is used. It is possible to implement an interval-based representation on top of trees. The basic idea is then that constraints are able to reduce the domains of the variables they are constraining by advancing the interval-based representation to the next possible legal value. In this context, the Leapfrog Triejoin algorithm can be seen as an n-ary constraint propagator for equality.

2.2 Goal of the research

I am adding support for constraints to a bottom-up Datalog evaluation engine, incorporating state-of-the-art techniques for constraint programming.

2.3 Current status

I have studied relevant literature and developed a first prototype. I started by studying the architecture of constraint programming systems at some level of detail. Secondly, I studied the Leapfrog Triejoin algorithm and investigated relevant parts of the LogicBlox codebase in which this algorithm has been implemented.

My supervisor and I have developed a prototype that demonstrates the feasibility of integrating this algorithm in a constraint programming context. A paper describing our approach in detail has been accepted at CICLOPS (Desouter and Schrijvers 2013b).

I started implementing these ideas in the LogicBlox' bottom-up evaluation engine. In this light, I have closely collaborated with Daniel Zinn, member of LogicBlox' runtime team. I have also attended several presentations and lectures about their core runtime architecture.

In the near future, overcoming the current limitation of unary predicates is a crucial step in a successful integration. Secondly, I plan on studying state-of-the art constraint programming techniques in more detail and add the findings to the system. I would also like to compare the Leapfrog Triejoin-based system to existing constraint programming systems, like Gecode (Gecode team 2006).

3 Zipping trees for modular search

3.1 Problem description

Prolog's nondeterminism has long inspired functional programming (FP) researchers. Over the years, many FP researchers have proposed solutions to make Prolog's rigid depth-first search more flexible (Spivey 2009; Seres et al. 1999). These solutions do not take the limitations of Prolog's system architecture (the WAM) into account. Hence, none of the FP work done has had any impact on Prolog programmers.

In earlier work, we have argued the need for a modular alternative to Prolog's standard control of inflexible DFS (see below). So, how can we incorporate all the FP work and achieve both modularity and compositionality?

3.2 Goal of the research

Our primary goal is to investigate alternatives to Prolog's depth-first search in a functional setting, but in contrast to earlier work, take into account the limitations of the Prolog system architecture at every step.

Secondary, we will provide an elegant implementation of modular search based on delimited continuations in Prolog. The implementation should also be provably correct, and in contrast to our earlier implementations make minimal use of mutable variables.

3.3 Results accomplished

We have presented a new *zip* operator enabling a highly modular solution for adding custom control orthogonally to logic, thereby overcoming Prolog's inflexible depth-first search. Starting from a functional model involving algebras, qualified types, free monad transformers, we have succeeded in deriving a Prolog implementation requiring only the primitives for delimited continuations presented in earlier work (discussed below).

3.4 Current status

The paper is still in draft. Both our Haskell and Prolog code is available at `http://users.ugent.be/~bdsouter/mergesearch.html`. Additionally, Coq correctness proofs can be found on the same webpage.

4 Delimited continuations for Prolog

4.1 Problem description

Prolog is a versatile language that essentially consists of Horn clauses extended with mostly simple builtins. To encode frequently occurring patterns, Prolog's rich meta-programming and program transforming capabilities come to the rescue. Examples of these are definite clause grammars (DCGs), structured state threading, logical loops and compiling control (Bruynooghe et al. 1989).

However these non-local program transformations may not be ideal for defining new language features. The effort of defining a transformation is proportional to the number of features in the language. Secondly, they are fragile with respect to language evolution. When new features are added to the language, the transformations need to be amended. When we introduce a new feature, we may need to transform the entire system. These issues hinder the development and adoption of new programming language features defined using non-local program transformations.

Delimited continuations (DCs) are well-known in the functional programming world. We wish to introduce this technique in logic programming and explore their interaction with other parts of Prolog. Given there are many applications of delimited continuations, we need to investigate which of them make sense in Prolog.

While we are not aware of any prior implementation of delimited continuations in Prolog, there are several noteworthy related works. The implementation of BinProlog (Tarau 2012) is based on explicit continuation passing and also provides a coroutine-like feature, *logic engines* (De Meuter and Roman 2011). Demoen and Nguyen (2008) describe an implementation of coroutining in which environments of certain (declared) predicates are put on the heap instead of on the local stack. It is fairly clear that delimited continuations are more general. Apart from the common name "coroutine", attributed variable coroutines (Holzbaur 1992; Le Houitouze 1990; Neumerkel 1990; Demoen 2002) share very little with coroutines based on delimited continuations.

4.2 Goal of the research

We wish to introduce the well-known functional technique of delimited continuations in the logic programming world. We investigate how these continuations could enable the definition of new high-level language features at the program level, e.g. in libraries, rather than at the meta-level.

Apart from specifying the semantics of delimited continuations in a meta-interpreter, we will demonstrate how to support them in a WAM-based system, hProlog, as well as in the ZIP-based SWI-Prolog. Concretely, we will work on the following points:

- show an implementation of delimited continuations in the context of the WAM and the ZIP;
- the idea behind this particular implementation should be easily adaptable to other Prolog implementations;
- demonstrate the power of the resulting built-ins in a series of examples;
- compare the constructs to similar ones in BinProlog and Haskell;
- explore their interaction with other parts of Prolog, in particular the cut, if-then-else constructs, backtracking, reactivation and exception handling.

In a way DCs are both very primitive and very abstract features intended for advanced programmers such as library or system developers. They do not have well-defined concrete applications, but are instead intended to implement various high-level control abstractions on top of. These abstractions do have well-defined applications. Typically those higher-level control abstractions have a more concrete and easy to understand semantics. This makes them usable by less advanced programmers like library users.

In this sense DCs have no concrete expected use. It is up to advanced programmers to use their own creativity to use them wherever they come in handy. Our paper will show various such examples and points to related work. In fact, the examples we will show are about the smallest meaningful ones that could be shown.

4.3 Results accomplished

We have introduced a design of delimited continuations for Prolog enabling many useful applications. Alongside, we have described the implementation in both the WAM (hProlog) and the ZIP (SWI-Prolog). The ideas behind the implementation can be easily transferred to other Prolog systems.

Three new predicates are available to the user:

- the continuation is delimited by `reset/3`;
- and captured by `shift/1`;
- `call_continuation/1` is used for calling the continuation.

The primitives do not turn continuations into first-class Prolog citizens, however they are useful in a wide range of applications. The implementations are independent of most of the rest of the system and therefore lightweight. Performance is suitable for the applications.

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

- The *transformation*-based approach can be thought of as partially evaluating a meta-interpreter for a given program. 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 (Tarau 2012) 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 have compared the three implementation approaches on artificial benchmarks. For reasons of space, we only discuss a single representative benchmark here: shifting a delimited continuation. Calling a previously shifted continuation shows the same behavior. We use three different sizes of continuations: 5,000, 10,000 and 20,000 chunks.

Table 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 benchmark 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 the

	Native		Transformed		Binarization
Depth	hProlog	SWI-Prolog	hProlog	SWI-Prolog	BinProlog
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

Table 1. *Benchmark results for shifting continuations.*

native implementation is 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.

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.

In addition to describing the implementation, we show how to implement various forms of coroutines. We do not mean the Prolog variety of coroutines, but their more general meaning of subroutines that can be suspended and resumed at certain locations. We elaborate on iterators, iteratees and transducers. Furthermore, catch/throw can be implemented in terms of reset/shift. Finally, we show how to implement both definite clause grammars and implicit state in terms of the primitives.

4.4 Current status

The technical report has lead to an article that has been conditionally accepted at ICLP. Delimited continuations have been implemented in hProlog and SWI-Prolog. We have successfully used the implementation of delimited continuations in subsequent articles.

5 Tor: Modular search with hookable disjunction

5.1 Problem description

Prolog programs have a natural depth-first procedural semantics. Although this depth-first search is a convenient choice of control for many, notably smaller, programs, it is unpractical for many others. For these programs, we need the ability to modify the search method. For example, the following code shows a straightforward way of imposing a depth limit to a labelling:

```
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)
  ).
```

This approach has several disadvantages. Logic and control are intermingled. Therefore, if we which to modify the control, we need to modify working code. This phenomenon is also known as the copy-paste-modify antipattern. The problem is that syntactically logic and control in Prolog are tightly coupled.

Secondly, the same heuristic is implemented over and over in different settings (different applications, different labeling predicates, ...). It is obvious that this approach is suboptimal.

Furthermore, as soon as we write more complex labeling code that spans different predicates or multiple invocations of the same predicate, the complexity increases drastically. Given the expertise required to combine labeling code with various search heuristics is non-trivial, fewer combinations will be explored. In the end, suboptimal solutions are obtained.

5.2 Goal of the research

We explore how to specify alternatives to Prolog's rigid depth-first search in a modular way. Acceptable approaches should meet the following goals:

- have the freedom to replace one search heuristic for another, without touching the existing logic;
- offer a library of reusable heuristics, and, in addition, allow the user to add others;
- enable a flexible composition of search heuristics;
- realize all of the above goals in an efficient way.

5.3 Results accomplished

We have presented TOR, a new approach to adding control in an orthogonal matter. Our solution is library-based, lightweight and easily portable across Prolog systems. Search methods can be composed with minimal overhead.

In addition to the search methods we supply as a library, users can easily add their own heuristics. The approach is useful for general Prolog programs with a large search space, and particularly suitable for Constraint Logic Programming.

5.4 Current status

Tor is available as an SWI-Prolog library at `http://www.swi-prolog.org/pack/list?p=tor`. Our article has received positive reviews and we have reworked it accordingly. The article is to appear in Science of Computer Programming.

Following the publication of a previous version of this article (Schrijvers et al. 2012), the editors of the Newsletter of the Association of Logic Programming, have asked us to write an article for their March 2013 issue. I was happy to do so.

References

Bruynooghe, M., de Schreye, D., and Krekels, B. 1989. Compiling control. *J. Log. Program. 6,* 1-2, 135–162.

Carlsson, M. 1984. On implementing Prolog in functional programming. *New Generation Computing 2*, 347–359.

De Meuter, W. and Roman, G.-C., Eds. 2011. *Coordination Models and Languages.* LNCS, vol. 6721. Springer.

Demoen, B. 2002. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Comp. Sc., KU Leuven, Belgium. Oct.

Demoen, B. and Nguyen, P.-L. 2008. Two WAM implementations of action rules. In *Logic Programming.* LNCS, vol. 5366. Springer, 621–635.

Desouter, B. and Schrijvers, T. 2013a. Contribution to the newsletter of the ALP – Tor: Modular search with hookable disjunction. Published in the Newsletter of the ALP.

Desouter, B. and Schrijvers, T. 2013b. Integrating Datalog and constraint solving. Submitted to CICLOPS.

Duba, B., Harper, R., and MacQueen, D. 1991. Typing first-class continuations in ML. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* POPL '91. ACM, 163–173.

Gecode team. 2006. Gecode: Generic constraint development environment.

Hinze, R. 1998. Prological Features In A Functional Setting Axioms And Implementations. In *Third Fuji Int. Symp. on Functional and Logic Programming.* WSPC, 98–122.

Holzbaur, C. 1992. Meta-structures vs. Attributed Variables in the Context of Extensible Unification. In *Programming Language Implementation and Logic Programming.* LNCS, vol. 631. Springer-Verlag, 260–268.

Le Houitouze, S. 1990. A New Data Structure for Implementing Extensions to Prolog. In *Programming Language Implementation and Logic Programming.* LNCS, vol. 456. Springer-Verlag, 136–150.

Neumerkel, U. 1990. Extensible unification by metastructures. In *META'90.* 352–364.

Schrijvers, T., Demoen, B., and Desouter, B. 2013. Delimited continuations in Prolog: Semantics, use and implementation in the WAM. Report CW 631, Dept. of Computer Science, KU Leuven.

Schrijvers, T., Demoen, B., Desouter, B., and Wielemaker, J. 2013. Delimited continuations for Prolog. Conditionally accepted at ICLP.

Schrijvers, T., Demoen, B., Triska, M., and Desouter, B. 2013. Tor: Modular search with hookable disjunction. To appear in Science of Computer Programming.

Schrijvers, T., Desouter, B., and Demoen, B. 2013. Zipping trees for modular search – from functional specification to logic programming implementation. Submitted to ICFP.

Schrijvers, T., Triska, M., and Demoen, B. 2012. Tor: Extensible search with hookable disjunction. In *Principles and Practice of Declarative Programming, 14th International ACM SIGPLAN Symposium, Proceedings*, A. King, Ed. ACM.

Schulte, C. and Stuckey, P. J. 2008. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst. 31,* 1, 2:1–2:43.

Seres, S., Spivey, M., and Hoare, T. 1999. Algebra of logic programming. In *International Conference on Logic Programming.* Palgrave MacMillan, 184–199.

Spivey, J. M. 2009. Algebras for combinatorial search. *J. Funct. Program. 19,* 3-4, 469–487.

Tarau, P. 2012. The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. *TPLP 12,* 1-2, 97–126.

Vardi, M. Y. 2000. Constraint satisfaction and database theory: a tutorial. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* PODS '00. ACM, 76–85.

Veldhuizen, T. L. 2012. Leapfrog triejoin: a worst-case optimal join algorithm.