Towards a Transformation from Prolog to Constraint Prolog Programs

Yu Fu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA

January 28, 2015

Abstract

Constraint Logic Programming (CLP) has great potential in the field of software analysis and testing; however, there is barely any research on how to transform programs written in C or Java into programs written in CLP. Combined with current progress on translating C programs into Prolog programs, this research project proposed and implemented a Prolog-to-CLP translator to exploit its "untapped" potentials; This research also attempted to build a type inference system to facilitate the construction of the translator.

1 Introduction

Constraint Logic Programming (CLP) can be viewed as an integration of constraint satisfaction to the concept of logic programming. Just like logic programs, CLP programs are highly declarative, with rules and facts to describe a certain situation; it also depends on the interpreter to automatically query the information base to prove a specified goal. However, in CLP the variables are not necessarily restricted to specific values, as they may also be within a constrained domain. As a result, the result of a CLP execution could be assigning target variables a set of possible values rather than a single one. CLP works by firstly collecting all the constraints into the *constraint store*, where they are all presumed to be satisfiable; at the end of one execution, *labeling* is often used to check the satisfiability or partial satisfiability of the variables within the *constraint store* and to find a satisfying assignment to all the variables in the goal.

CLP could be applied into the field of software testing and program analysis. To begin with, we propose to use CLP to improve the current con-colic testing for more efficient coverage of all the execution paths in a program. Specifically, the original program shall be transformed to the corresponding CLP program, then the CLP goals would be queried until all the possible execution paths are uncovered, and finally concrete test cases with respect to each particular execution path would be generated by deploying a constraint solver upon the path constraints. By taking advantage of the inherent backtracking mechanisms offered by CLP, we don't need to bother implementing a full-fledged con-colic testing system; Moreover, by using the APIs provided by the CLP interpreter, it is more convenient to monitor and tune the execution details during the test process; Last but not the least, such testing tool can be quite versatile as long as programs in other languages are transformed into CLP in advance.

We also see the possibility of extending CLP to the field of program analysis. One of the most dominant techniques in the field – abstract interpretation is flawed, although significantly lowering the complexity of program states, because the huge gap between the abstract semantics and the original program often leads to unsatisfactory result. As pointed out by Ponsini, abstract interpretation often fails to consider the constraints between variables when mapping thee into abstract domains, especially when handling conditional statements and non-linear expressions. However, by using CLP as the intermediate language for abstract interpretation, we can take advantage of its more uniformed syntax and an array of existing CLP analyzers to generate a safer approximation of the semantics of the original program, while maintaining the constraints between variables in the original program.

One major obstacle when trying to integrate CLP into software analysis and testing is to translate a program written in a common programming language such as C and Java into a program in CLP, which can be treated as a two-step process: first translating a C/Java program into the corresponding Prolog program, then translating the derived Prolog program into a CLP program. Considering that there has already been work on the transformation from C into Prolog programs, our object is to propose and implement a Prolog-to-CLP translator to attempt its usage in software testing.

There is not an official Prolog distribution to stick to, so I have chosen SWI-Prolog, one of the most popular and robust open-source Prolog distribution with CLP support as well. Specifically, current stable SWI-Prolog (version 6.6.6) provides the following three libraries for extended ability to process CLP: library(clpb)–Constraint Logic Programming over Boolean Variables, library(clpfd)–Constraint Logic Programming over Finite Domains, and library(clpqr)–Constraint Logic Programming over Rationals and Reals. Although these libraries are similar to some extent, there are still differences w.r.t. specific pre-defined predicates, hereby we choose SWI-Prolog with library(clpfd) as our target CLP program because it is sufficient for most software testing needs while keeping up an acceptable performance.

Outline The remainder of this article is organized as follows. Section 2 gives account of previous work. Section 3 reviews the approaches in the transformation of Prolog into CLP programs. Our new and exciting results are described in Section 4. Finally, Section 5 laid out the future challenges to resolve.

2 Previous work

There has been a myriad of research with regard to the optimization of Prolog programs. In the [6], the authors reviewed some of the major techniques often used for the transformation of logic programs, which is based on the use of two elementary transformation rules: *unfolding* and *folding* originally introduced for the transformation of functional rules [2]. Loosely, unfolding can be viewed as replacing a goal in the body of a clause by its definition, while folding corresponds to recognizing that goals in the body of a clause are an instance of a definition, unfolding and folding are inverse as an unfolding step followed by the corresponding folding step would produce the initial program. By introducing a set of theorems, the authors justified that such unfolding/folding based transformation is indeed semantics-preserving. Finally, considering the inverses within transformation rules, in order to avoid producing futile transformations (namely, equal to the initial programs), the authors concluded some transformation strategies to guide the transformation towards producing more efficient programs. Partial evaluation, another major program transformation method, is also based on repetitive unfolding procedures.

Although folding and unfolding has long been the mainstream in the realm of logic program transformation, it does not necessarily fit our purposes, mainly due to the fact that Prolog and CLP are different in a lot of ways. Considering that there are many Prolog implementations, we choose SWI-Prolog because it is well supported by many external libraries and extensions, including library(clpfd), which we use as our target constraint Prolog programs. The library(clpfd) was developed as an extension to provide SWI-Prolog with the ability to model and solve various combinatorial problems such as planning, scheduling and allocation tasks. Based on the conventional SWI-Prolog grammar, new predicates for the definition and resolution of finite domain constraints are introduced. Specifically, library(clpfd) use the following set of arithmetic constraints: $Exp1\# >= Exp2$, $Exp1\# =< Exp2$, $Exp1\# = Exp2$, $Exp1\# = Exp2$, $Exp1\# > Exp2$, whose meanings are rather self-explanatory. Another often used constraint predicate is $?Varin + Domain$, which means that Var is an element of $Domain$, which can be a single integer, an integer domain bounded by lower and upper limit, or the union of two domains. As for the enumeration of constraints, library(clpfd) deploys the $labeling(+Options, +Vars)$ predicate, which systematically tries out values for the finite domain variables Var until all of them are ground, while *Options* contain a list of options that allow for extra control over the search process. There are still some new predicates introduced by library(clpfd) but these are the most basic ones.

One critical step in our transformation to constraint Prolog programs is to add domain declarations for finite domain variables; it is not hard to distinguish between variables and atoms in Prolog thanks to the predicates $var(@Term)$ and $nonvar(@Term)$, yet how to determine whether the variable would bound to a finite domain or not is a challenge due to the fact that vanilla Prolog is a typeless language. As a result, there is no existent predicates in the original Prolog program to help us determine the type of each variable: for unbounded variables, they could be of any type, so it is futile to try getting its type; for bounded variables, we can use $integer(@Term)$ to tell whether it is bound to an integer; the tricky part is those variables that are implicitly bound, which requires a closer examination rather than the nave $integer(@Term)$ predicate to tell which type it is bound to. Obviously it will not work (dynamic type checking error) if we simply add domain declarations for each variable in the program regardless of their type, so we need to apply type inference in the process of program transformation to locate all the variables that are (or going to) bound to an finite domain.

One of the first contributions in this field is a polymorphic type system proposed by Mycroft and O'Keefe [5], here *polymorphic* denotes the ability to include type variables in the type structures. In this Prolog system, the user need to declare types as restrictions upon the arguments of predicates and terms, and a static type checker would examine if these restrictions are followed. A recent effort was modeled after the Mycroft-O'Keefe type system to gradually introduce types in Prolog (Specifically, SWI-Prolog and YAP) using a type-checking library[7], which achieved a somewhat "on-demand" type system, i.e. runtime type checks are only performed when typed predicates (i.e. predicates whose type signature has already been provided) are invoked, therefore simplifying the migration from untyped code to typed code. Another branch of research, however, focuses on building a type inference system for existed Prolog code, one example was the bottom-up polymorphic type system developed by Barbuti and Giacobazzi [1], which opted for a fixed-point bottom-up abstract interpretation technique to infer types for Horn-Clause based logic programs. Although this system also supports user defined type signatures, they are not necessary as the system is able to infer types for predicates.

The most recent endeavor in the area is the Gradual Polymorphic Type System with Subtyping for Prolog proposed by Spyros Hadjichristodoulou [3], which works as a preprocessor of the original source code to provide both type checking and inference functionalities for XSB-Prolog programs. It ported the type-checking library from [7] to work on XSB-Prolog, allowing user to provide type signatures and to invoke the correspondent type-checking library with a complier flag. Its type-inference was also based on the approach of [2] while adopting the same notations for declaring types in [5], which would infer predicates with no pre-defined type signatures. Compared with previous type systems, Hadjichristodoulou's system extended the Mycroft-O'Keefe to allow for the definition of predicates with different types with the introduction of simple-fixed subtyping rules between primitive types that each program can have: for instance, by concluding integer, atom and float as the subtypes of atomic, the user definition of fact $p(20)$ and $p(value)$ will conclude the type $p(atomic)$ for $p/1$ instead of failing; moreover, instead of the "cut-off" method to avoid infinite growth of a predicate type in [1], Hadjichristodoulou suggested using $unify\_with\_occurs\_check$ and concluded that the predicate is ill-typed when its two clauses give types that cannot be unified the occurs check.

3 Methodology

One crucial task is to implement the transformation from Prolog code into Constraint Logic Prolog code, i.e. SWI-Prolog code to SWI-Prolog code under the library(cfd). Up till now, we haven't yet incorporated a type inference system to aid transformation, therefore its capabilities are limited. For our initial trial, our methodology is demonstrated in the pseudocode below:

Algorithm 1 Transformation from Prolog to CLP

```
1: procedure TRANSFORM($P$)                ▷ $P$ is the original Prolog program
2:     for each clause p$_i$ in $P$ do
3:         transform_clause p$_i$ to c$_i$  ▷ c$_i$ is the corresponding CLP clause
4:         add clause c$_i$ to $C$
5:     end for
6:     Return $C$                          ▷ $C$ is the transformed CLP program
7: end procedure
end
```

As shown in algorithm 1, the main idea of transformation is quite straightforward: we first read in each clause of the original Prolog program with a simple iteration of the *read_file*/3 predicate came as the API provided by SWI-Prolog until it has reached the end of the input stream (also supported by the $at_end_of_stream$/1 predicate from SWI-Prolog), then transform each clause of the original program into the corresponding CLP clause before building up the new CLP program. Because of the grammar difference between Prolog and SWI-Prolog, the most difficult part lies within the transformation of each individual clauses:

Algorithm 2 Transformation of Prolog clause to CLP clause

```
1: procedure TRANSFORM_CLAUSE($p$)  ▷ $p$ is the original Prolog clause
2:     $read\_clause(p, Variablelist, \_)$
3:     $sortout(Variablelist)$
4:     $insert\_definition(p, Variablelist, p')$
5:     $convert\_expressions(p', c)$
6:     Return $c$                    ▷ $c$ is the transformed CLP clause
7: end procedure
end
```

As further demonstrated in algorithm 2, the transformation of each Prolog clause can be divided into three steps: first, it is necessary to identify the list of variables in the original clause, which could be extracted using the *read_clause*/3 predicate provided by SWI-Prolog; after we have sorted out the variable list, i.e. get rid of repetitive ones, we add the finite domain declarations for each variable in the beginning of the clause body:

$$Head(V_1, ..., V_n) : -V_1\, in\, inf..sup, ..., V_n\, in\, inf..sup, Body. \tag{1}$$

Note that here $Head$ and $Body$ both symbolize the corresponding part within each Prolog clause, while $V_1, V_2, ..., V_n$ stand for the processed variable list extracted from *read_clause*/3 . We find that here all the variables are sloppily treated as singular finite domain variables, which is often the case, and would lead to the problem demonstrated in the section 4.

The *convert_expression* procedure is rather self-explanatory: all the arithmetic expressions would be transformed to its constraint logic programming format, for instance, $Exp1 =< Exp2$ would be converted to $Exp1\# =< Exp2$, and $Exp1 :=: Exp2$ would be converted to $Exp1\# :=: Exp2$; the other parts in the clause remain the same.

At the same time, the insert_definition part is quite tricky, as a result of the multiple possible cases of the variable list extracted from the original Prolog clause, which is described in algorithm 3.

Algorithm 3 Transformation of Prolog clause to CLP clause

```
 1: procedure INSERT_DEFINITION($p$, $Variablelist$, $p'$)
 2:     if $Variablelist == null$ then                ▷ The clause is grounded
 3:         $p' \leftarrow p$
 4:     else if $p == q$ then      ▷ No need to insert Variablelist into a fact
 5:     else
 6:         Extract the body of clause $p$
 7:         Iteratively appends domain declaration to the body of $p'$
 8:         Repeat step 7 until $Variablelist$ is empty
 9:         Construct $p'$ with the head of $p$
10:     end if
11:     Return $p'$                          ▷ $p$ is the transformed CLP clause
12: end procedure
end
```

4 Results

We implemented the Prolog-to-CLP translator using Prolog, and tested our translator using the holmer logic programming test benchmark [4], which contains 26 programs, grouped into "small" and "large". There is no problem in applying the preliminary translator upon these logic programs, however, in many cases the generated constraint logic programs doesn't work properly. Picking one program in the benchmark, "fast_mu.pl", an optimized prover of a theorem of Hofstadter's "mu-math", the code snippet of the original program and the transformed program "fast_mu_clp.pl"is shown here.

```
 1 % fast_mu.pl snippet
 2 main :- theorem([m,u,i,i,u]).
 3
 4 theorem(G) :-
 5   length(G, GL1),
 6   GL is GL1 - 1,
 7   derive([m,i], G, 1, GL, Derivation, 0).
 8 ...
```

```
 1 % fast_mu_clp.pl snippet
 2 :- use_module(library(clpfd)).
 3
 4 main:-theorem([m,u,i,i,u]).
 5
 6 theorem(_G2812):-(_G2812 in inf..sup,_G2815 in inf..sup,_G2820
       in inf..sup,_G2833 in inf..sup,
 7   length(_G2812,_G2815),
 8   (_G2820#=_G2815-1,
 9   derive([m,i],_G2812,1,_G2820,_G2833,0))).
10 ...
```

We would get a type error when running the "fast_mu_clp.pl" program, this is because the variable G (i.e. _G2812 because the variable name is lost during the transformation process) would be bound to the list $[m, u, i, i, u]$ once it is invoked by the main predicate on line 4, however on line 6 in the transformed program, it is treated as a variable bound to a finite domain, further inspection of other test cases demonstrated that all the other malfunctions in transformed program is induced by the sloppy treatment of all the variables as finite-domain variables. The only solution to this problem, as far as I can see is to employ type inference when inserting domain declarations for variables so that type clashes can be avoided.

5 Challenges

The most pressing challenge in this project is to implement a type inference system to aid our program transformation in SWI-Prolog. Such a type inference could also benefit programmers and analysts to better understand existing Prolog programs.

We would also like to provide a formalized description of the translation process, and preferably with a rigorous proof that the semantics of the original program is well-preserved in this process; otherwise, the software analysis based on the transformed program would fail to provide any insight on the original program.

References

[1] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of computer programming*, 19(3):281–313, 1992.

[2] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.

[3] S. Hadjichristodoulou. A gradual polymorphic type system with subtyping for prolog. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 17. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[4] B. K. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, W. R. Bush, A. M. Despain, J. M. Pendleton, and T. Dobry. Fast prolog with an extended general purpose architecture. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 282–291. ACM, 1990.

[5] A. Mycroft and R. A. O'Keefe. A polymorphic type system for prolog. *Artificial intelligence*, 23(3):295–307, 1984.

[6] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19:261–320, 1994.

[7] T. Schrijvers, V. S. Costa, J. Wielemaker, and B. Demoen. Towards typed prolog. In *Logic Programming*, pages 693–697. Springer, 2008.