$\mathcal{TOY}$: A Multiparadigm Declarative System*

F.J. López Fraguas and J. Sánchez Hernández

Dep. Sistemas Informáticos y Programación, Univ. Complutense de Madrid
Fac. Matemáticas, Av. Complutense s/n, 28040 Madrid, Spain
email: {fraguas,jaime}@sip.ucm.es Phone: +34 91 3944429

Abstract. $\mathcal{TOY}$ is the concrete implementation of CRWL, a wide theoretical framework for declarative programming whose basis is a constructor based rewriting logic with lazy non-deterministic functions as the core notion. Other aspects of CRWL supported by $\mathcal{TOY}$ are: polymorphic types; HO features; equality and disequality constraints over terms and linear constraints over real numbers; goal solving by needed narrowing combined with constraint solving. The implementation is based on a compilation of $\mathcal{TOY}$ programs into Prolog.

1 Introduction

$\mathcal{TOY}$ is a system for multiparadigm declarative programming which encompasses (and extends in some interesting ways) functional programming (FP), logic programming (LP) and constraint programming. The system has been made publicly available (at `http://mozart.sip.ucm.es/toy`), and we have developed for it a variety of interesting programs and programming methodologies [AHL+96,CL98b,CL98a].

As with many other narrowing-based FP+LP proposals (see [Han94] for a survey), the starting point of $\mathcal{TOY}$ can be described as follows: rewrite systems can be seen as functional programs, and rewriting performs evaluation. The use of narrowing instead of rewriting results in goal solving capabilities, turning the rewrite system into a functional logic program. When seen as rewrite systems, $\mathcal{TOY}$ programs have the following characteristics:

1. They follow a *constructor discipline*. Rules use linear constructor-made patterns in left-hand sides.

2. They can be *non-terminating* and *non-confluent*. Therefore $\mathcal{TOY}$ programs serve to compute non-deterministic lazy functions, which turn out to be a useful tool for programming. As a particular source of non-confluence, $\mathcal{TOY}$ allows extra variables to appear in right-hand sides of rules.

3. They use *constrained conditional rules* for defining functions. $\mathcal{TOY}$ can manage *equality* and *disequality constraints* over constructor terms, and *linear constraints* over real numbers.

4. They may be *higher order*. $\mathcal{TOY}$'s approach to HO features is based on an *intensional* view of functions: functions, when partially applied, behave as

* The authors have been partially supported by the Spanish CICYT (project TIC 98-0445-C03-02 'TREND') and the ESPRIT Working Group 22457 (CCL-II).

P. Narendran and M. Rusinowitch (Eds.): RTA'99, LNCS 1631, pp. 244–247, 1999.
© Springer-Verlag Berlin Heidelberg 1999

data constructors, and can be used to form *HO patterns* which, in particular, can appear in left-hand sides of rules and also in answers.

5. They are *polymorphically typed*. Types are inferred (optionally declared) according to Hindley-Milner system.

6. As goal solving mechanism, $\mathcal{TOY}$ uses a suitable combination of lazy narrowing (with a sophisticated strategy, called *demand driven* [LLR93] or *needed narrowing* [AEH94]) and constraint solving. Alternative solutions for a given goal are obtained by backtracking.

2 $\mathcal{TOY}$ in the FLP context

$\mathcal{TOY}$ implements a wide theoretical framework (called CRWL) for declarative programming. The first order untyped core of the framework can be found in [GHL+98]. HO features were included in [GHR97]. Polymorphic types (and also algebraic types, not yet implemented in $\mathcal{TOY}$) are addressed in [AR97a,AR97b], and further extended with constraints in [ALR98a,ALR98b]. The concrete lazy narrowing strategy adopted by $\mathcal{TOY}$ is studied in [LLR93,AEH94], and formally justified within the CRWL-framework in [LS98].

$\mathcal{TOY}$ is an evolution of BABEL [MR92], a functional logic language with a much more restrictive class of programs (confluent, no constraints, more limited HO features). Among other proposals for FLP with publicly available implementations, the most related one is *Curry* [Han98], a still in progress initiative for developing a 'standard' FLP language. $\mathcal{TOY}$ and *Curry* share many characteristics, but there are still remarkable differences:

* *Curry* 's operational model, which combines lazy narrowing and residuation, was proposed and formally justified [Han97] for certain kinds of unconditional confluent programs. For practical reasons, *Curry* now accepts conditional rules and non-deterministic functions, but they lie outside its formal foundations.

* *Curry* contemplates constraints as a special kind of expressions, but in fact only equality constraints are considered. Constraints are also outside the formal foundations of *Curry*.

* *Curry* does not contemplate HO patterns nor HO logical computations.

3 Programming in $\mathcal{TOY}$

The table below contains a small program showing some of $\mathcal{TOY}$'s abilities, together with some goals and the corresponding answers. $\mathcal{TOY}$ borrows from Haskell most of its syntax, with the exception that variables begin with upper-case letters while constructor and function symbols use lower-case.

The goal *(a)* in the table sorts a list, in a pure functional computation. Goal *(b)* sorts a list with a variable as one of its elements; the three given answers include linear constraints over such variable. The answer for goal *(c)* involves a syntactic disequality. In goal *(d)* `F` is a higher order logic variable; the obtained values for this variable are higher order patterns (`permut`, `sort`,...).

```
% Non-deterministic choice of one of two values
infixr 40 //                    X // Y = X                    X // Y = Y

% Non-deterministic insertion of an element into a list
insert X [] = [X]           insert X [Y|Ys] = [X,Y|Ys] // [Y|insert X Ys]

% Non-deterministic generation of permutations of a list
permut [] = []              permut [X|Xs] = insert X (permut Xs)

% Tests if a list of numbers is sorted
sorted [] = true            sorted [X] = true
sorted [X,Y|Ys] = sorted [Y|Ys] <== X <= Y

% Lazy 'generate-and-test' permutation sort. 'check' calls 'sorted' which demands
% its argument, which is lazily non-deterministically generated by 'permut'.
% As soon as the test fails, 'permut' stops the generation and tries another
% alternative for the last choice it made
sort Xs = check (permut Xs)          check Xs = Xs <== sorted Xs == true
```

Goal	*Answers*
(a) `sort [4,2,5,1,3] == L`	`L == [1,2,3,4,5]; no more solutions`
(b) `sort [3,2,X] == L`	`L == [X,2,3] {X=<2}; L == [2,X,3] {X>=2,X=<3};` `L == [2,3,X] {X>=3}; no more solutions`
(c) `sort [3,2,1] /= L`	`L /= [1,2,3] ; no more solutions`
(d) `F [2,1,3] == [1,2,3]`	`F == permut; F == sort; ...`

4 Implementation of $\mathcal{TOY}$

At its outer level, $\mathcal{TOY}$ behaves as a small command interpreter to be executed within a Sicstus Prolog session. There are commands for compiling, running and editing programs, solving goals, showing type information, ...

The core of $\mathcal{TOY}$'s implementation is a process of *compilation to Prolog*: any $\mathcal{TOY}$ program is translated into a set of clauses that reproduce, when executed with Prolog, the expected behaviour of the source program under $\mathcal{TOY}$'s own operational model. Common to all $\mathcal{TOY}$ programs are the clauses for *constraint solving*. For *equality and disequality constraints* (over constructor terms), reduction of arguments is interleaved with occur-check and check for constructor clashes. Disequalities with the form $X/= t$, where t is a constructor term, are in solved form and kept in a *store*, which must be 'awoken' if in a later step X becomes bound. For solving a *linear constraint* $e \diamond e'$ (with $\diamond \in \{<, >, =<, >=, ==, /=\}$), e and e' are reduced to normal forms t and t' and then the Sicstus linear constraint solver is invoked to solve $t \diamond t'$. When a computation is finished, all the stored constraints are conveniently projected over the set of relevant variables for taking part in the answer.

Some other clauses are heavily dependent on the source program. The most important ones are those which control the computation of *head normal forms* for function calls. They must use the rules of the source $\mathcal{TOY}$ program so that $\mathcal{TOY}$'s demand driven strategy is reflected. For this purpose, we built the *def-*

initional tree (see [LLR93]) of each function, from which the Prolog code is extracted.

Although its performance is not very impressive, $\mathcal{TOY}$ easily supports the development of medium size programs. The largest $\mathcal{TOY}$ program of which we are aware (a partial evaluator for functional logic programs) contains about 200 function definitions and 1500 lines of $\mathcal{TOY}$ code.

Acknowledgements:
We thank all members of the Declarative Programming Group at the UCM for their help when developing $\mathcal{TOY}$. Very special thanks merits Rafa Caballero.

References

AEH94. S. Antoy, R. Echahed, M. Hanus. A needed narrowing strategy. *Proc. POPL'94*, 268–279, 1994.

AHL+96. P. Arenas, T. Hortalá, F.J. López, E. Ullán. Real constraints within a functional logic language. *Proc. APPIA-GULP-PRODE'96*, 451–462, 1996.

ALR98a. P. Arenas, F.J. López, M. Rodríguez. Embedding multiset constraints into a lazy functional logic language. *Proc. PLILP'98*, Springer LNCS 1490, 429–444, 1998.

ALR98b. P. Arenas, F.J. López, M. Rodríguez. Functional plus logic programming with built-in and symbolic constraints, 1998. *Submitted.*

AR97a. P. Arenas, M. Rodríguez. A semantic framework for functional logic programming with algebraic polymorphic types. *Proc. TAPSOFT'97*, Springer LNCS 1214, 453–464, 1997.

AR97b. P. Arenas, M. Rodríguez. A lazy narrowing calculus for functional logic programming with algebraic polymorphic types. *Proc. ILPS'97*, MIT Press, 53–69, 1997.

CL98a. R. Caballero, F.J. López. A functional logic alternative to monads. *Proc. of Workshop on Component-Based Software Development in Computer Logic*, 87–100, 1998.

CL98b. R. Caballero, F.J. López. Parsing with non-deterministic functions. *Proc. APPIA-GULP-PRODE'98*, 1–16, 1998.

GHL+98. J.C. González, T. Hortala, F.J. López, M. Rodríguez. An approach to declarative programming based on a rewriting logic. *To appear in JLP*, 1998. Earlier version in *ESOP'96*.

GHR97. J.C. González, M.T. Hortalá, M. Rodríguez. A higher order rewriting logic for functional logic programming. *ICLP'97*, MIT Press, 153–167, 1997.

Han94. M. Hanus. The integration of functions into logic programming, *JLP*, 19 & 20, 583–628, 1994.

Han97. M. Hanus. A unified computation model for functional and logic programming, *Proc. POPL'97*, 80-93, 1997.

Han98. M. Hanus (ed.). Curry: An integrated functional logic language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1998.

LS98. F.J. López, J. Sánchez. An efficient narrowing strategy by means of disequality constraints. Tech. Rep. 98/84, Dep. SIP, UCM Madrid, 1998.

LLR93. R. Loogen, F.J. López, M. Rodríguez. A demand driven computation strategy for lazy narrowing. *Proc. PLILP'3*, Springer LNCS 714, 184–200, 1993.

MR92. J.J. Moreno, M. Rodríguez. Logic programming with functions and predicates: The language BABEL. *JLP*, 12, 191–223, 1992.