8th Int'l Symp. on Prog. Lang., Implementations, Logics, and Programs (PLILP'96)
Aachen, Germany, Sept. 1996, Springer LNCS Vol. 1140, pages 473-474.

Needed Narrowing in Prolog

(Extended Abstract)

Sergio Antoy
Portland State University

We discuss an implementation of *needed narrowing* deployed in an extension [4] of the Gödel compiler [6], which translates Gödel source code into Prolog source code. Our implementation is high-level, portable, and similar to, but more efficient than, [5] that in turn is an improvement of [7].

Needed narrowing [3] is a sound, complete, and optimal strategy for semantic unification in inductively sequential rewrite systems. Inductive sequentiality [1] amounts to the existence of a definitional tree $\mathcal{T}$ for each operation f, i.e., a set of patterns partially ordered by subsumption with the following properties up to renaming of variables.

- [root property] The minimum element, referred to as the *root*, of $\mathcal{T}$ is $f(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are distinct variables.
- [leaves property] The maximal elements, referred to as the *leaves*, of $\mathcal{T}$ are all and only (variants of) the left hand sides of the rules defining f. Non-maximal elements of $\mathcal{T}$ are referred to as *branches*.
- [parent property] If π is a pattern of $\mathcal{T}$ different from the root, there exists in $\mathcal{T}$ a unique pattern π' strictly preceding π such that there exists no other pattern strictly between π and π'. π' is referred to as the *parent* of π and π as a *child* of π'.
- [induction property] All the children of a same parent differ from each other only at the position of a variable, referred to as *inductive*, of their parent.

There exist operations with no definitional tree, and operations with more than one definitional tree, examples are in [1]. The existence of a definitional tree of a function f is decidable and simple to decide in most practical situations.

Our implementation of needed narrowing maps each operation f into a family of Prolog predicates $f_0, f_1, \ldots$ such that if $f_i(u_1, \ldots, u_n, u)$ succeeds, then u is a minimal head normal form of $f(u_1, \ldots, u_n)$. Like all modern approaches to the implementation of efficient narrowing, the predicates $f_0, f_1, \ldots$ are generated by a traversal of a definitional tree $\mathcal{T}$ of f. Clauses are generated when a each node of $\mathcal{T}$ is visited and depend on whether the node is a *branch* or a *leaf*. The latter case further considers whether the corresponding rule right hand side is

This work has been supported in part by the National Science Foundation under grant CCR-9406751. Author's address: Dept. of Computer Science, P.O. Box 751, Portland, OR 97207, antoy@cs.pdx.edu.

a variable, a constructor-rooted term, or an operation-rooted term. Finally, half a dozen optimizations, some of which were originally proposed in [5], are applied to the generated code. A full account of the translation of $\mathcal{T}$ into $f_0, f_1, \ldots$ is in [2].

The semantic unification of terms t and u is computed by narrowing the equation $t \approx u$ to *true*, where "$\approx$" is defined by the equality rules of each sort [3]. Since these rules are inductively sequential, we obtain the Prolog predicates defining equality as for any other operation. We apply to this code a set of optimizations specialized for the relatively simple rules of "$\approx$". We use a definition of equality, referred to as *semi-strict*, that is more general than [5,7], since throughout our implementation variables are substituted by constructor terms only — a property that also holds for Gödel, whose compiler has been extended with our implementation.

Our implementation differs from [5] in that we adopt a less strict notion of equality, which reduces the size of the search space in some cases; we perform more optimizations, which make better use of the built-in unification; and we take better advantage of mode information, which avoids the creation of some choice points and the execution of some unnecessary predicate calls.

We use the five equations proposed in [5, Sect. 7] to benchmark our implementation. The following table shows the computation time for finding the first solution of an equation as percent of the time required by Hanus's code. The comparison with several other implementations of narrowing in Prolog can be inferred using the benchmarks in [5], where it is shown that Hanus's code is the fastest.

Equation	E_1	E_2	E_3	E_4	E_5	Aver.
% time	64	44	68	44	41	50

The benchmark shows that our code is twice as fast as [5]. The amount of memory allocated by the two methods for computing the first solution of each equation is the same.

References

1. S. Antoy. Definitional trees. In *Proc. of the 4th Intl. Conf. on Algebraic and Logic programming*, pages 143–157. Springer LNCS 632, 1992.
2. S. Antoy. Needed narrowing in Prolog. Technical report TR 96-2, Portland State University, Portland, OR, May 1996. Full version of this abstract accessible via `http://www.cs.pdx.edu/~antoy`.
3. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.
4. B. J. Barry. Needed narrowing as the computational strategy of evaluable functions in an extension of Gödel. Master's thesis, Portland State University, June 1996.
5. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.
6. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1993.
7. R. Loogen, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *PLILP'93*, pages 184–200, Tallinn, Estonia, August 1993. Springer LNCS 714.