Herbrand Constraints in HAL

Bart Demoen, María García de la Banda, Warwick Harvey, Kim Marriott, David Overton, and Peter J. Stuckey

1 Department of Computer Science, Catholic University Leuven, Belgium
Bart.Demoen@cs.kuleuven.ac.be
2 School of Computer Science & Software Engineering, Monash University, Australia
{maria,wharvey,mariott,dmo}@mail.csse.monash.edu.au
3 Department of Computer Science & Software Engineering, University of Melbourne, Australia
pjs@cs.mu.oz.au

Abstract. Mercury is a logic programming language that is considerably faster than traditional Prolog implementations, but lacks support for full unification. HAL is a new constraint logic programming language specifically designed to support the construction of and experimentation with constraint solvers, and which compiles to Mercury. In this paper we describe the HAL Herbrand constraint solver and show how by using PARMA bindings, rather than the standard WAM representation, we can implement a solver that is compatible with Mercury’s term representation. This allows HAL to make use of Mercury’s more efficient procedures for handling ground terms, and thus achieve Mercury-like efficiency while supporting full unification. An important feature of HAL is its support for user-extensible dynamic scheduling since this facilitates the creation of propagation-based constraint solvers. We have therefore designed the HAL Herbrand constraint solver to support dynamic scheduling. We provide experiments to illustrate the efficiency of the resulting system, and systematically compare the effect of different declarations such as type, mode and determinism on the resulting code.

1 Introduction

The logic programming language Mercury [11] is considerably faster than traditional Prolog implementations for two main reasons. First, Mercury requires the programmer to provide type, mode and determinism declarations and information from these is used to generate efficient target code. Types allow a compact representation for terms, modes guide reordering of literals and multivariant specialization, and determinism is used to remove the overhead of unnecessary choice point creation. The second main reason for Mercury’s efficiency is that variables can only be ground (i.e., bound to a ground term) or new (i.e., first time seen by the compiler and thus unbound and unaliased). Since neither aliased variables nor partially instantiated structures are allowed, Mercury does not need to support full unification; only assignment, construction, deconstruction and equality testing for ground terms are required. Furthermore, it does not need to
perform trailing, a technique that allows an execution to continue computation from a previous program state by logging information about prior states during forward computation and using it to restore the states again during backtracking. Trailing usually means recording the state of unbound variables right before they become aliased or bound. Since Mercury’s new variables have no run-time representation they do not need to be trailed.

This paper investigates whether it is possible to have Mercury-like efficiency, yet still support true logical variables. In order to do so we describe our experiences with HAL, a new constraint logic programming language that compiles to Mercury so as to leverage from Mercury’s sophisticated compilation techniques. Like Mercury, HAL requires the programmer to provide type, mode and determinism declarations. Unlike Mercury, HAL was specifically designed to support the construction of and experimentation with constraint solvers.

In particular, HAL includes a built-in Herbrand constraint solver that provides full unification (without the occurs check), thus supporting logical variables. The Herbrand solver uses PARMA bindings rather than the standard variable representation used in the WAM. PARMA bindings represent equivalence of variables by keeping all equivalent variables in a cycle, as opposed to WAM bindings which implement a union-find style equivalence class. The use of PARMA bindings allows the solver to use essentially the same term representation for ground terms as does Mercury (see Section 4). This is important because it allows the HAL compiler to replace calls to the Herbrand constraint solver by calls to Mercury’s more efficient term manipulation routines whenever ground terms are being manipulated.

An important feature of HAL is its use of type classes to distinguish between solver and non-solver types (i.e., types with an associated solver and types without) and for the hierarchical organisation of constraint solvers. Type classes allow a clean separation between a constraint solver’s interface and its implementation, thus supporting experimentation with different solvers. We detail how HAL’s Herbrand constraint solver fits into this hierarchy.

Another important feature of HAL is its support for user-extensible dynamic scheduling, that is intended to support communication between solvers and construction of efficient propagation-based solvers. We have therefore designed the HAL Herbrand constraint solver to support dynamic scheduling. Here we detail how this has been achieved with a PARMA-binding based solver. Again type classes allow us to distinguish between solvers that support dynamic scheduling and those that do not.

The HAL programmer may specify for a particular constructor type \( t \) whether \( t \) requires a Herbrand constraint solver (i.e. must support full unification) and, if so, whether this solver should support dynamic scheduling. The HAL compiler will then automatically generate an appropriate instance of the Herbrand solver for \( t \). By requiring that constructor types that need a solver must be specified, HAL can simplify the representation, analysis and compilation of constructors types that do not need a solver.

---

4 Actually, as long as the term is “sufficiently” instantiated.