The Vienna Abstract Machine

Andreas Krall    Ulrich Neumerkel

Institut für Praktische Informatik
Abteilung für Programmiersprachen und Übersetzerbau
Technische Universität Wien
andi@vip.UUCP  ulrich@vip.UUCP

Abstract

The Vienna Abstract Machine (VAM) is a Prolog machine developed at the TU Wien. In contrast to the standard implementation technique (Warren Abstract Machine – WAM), an inference in VAM is performed by unifying the goal and head immediately, instead of bypassing arguments through a register interface. We present two implementations for VAM: VAM2P and VAM1P. VAM2P is well suited for an intermediate code emulator (e.g. direct threaded code) which uses two instruction pointers for both goal code and head code. During an inference VAM2P fetches one instruction from the goal code, and one instruction from the head code and executes the combined instruction. More optimization is therefore possible, since information about the calling goal and the head of the clause is available at the same time. VAM performs cheap shallow backtracking, needs less dereferencing and trailing and implements a faster cut. In a Prolog with the occur-check, VAM omits many unnecessary operations. VAM1P is designed for native code compilation. It combines instructions at compile time and supports several optimizations, such as fast last-call optimization. In this paper we present the VAM in detail and compare it with existing machines.

1 Introduction

Five years ago, we began on research in the area of implementation of logic programming languages. A small, slow and portable interpreter [Ge84] and a fast compiler based on the WAM (Warren Abstract Machine [Wa83]) for a commercial Prolog System [Pi84] were developed. With this experience the VIP research project [Op85] was started. Our project developed new interpreter and compiler implementation techniques [Kr87], extensions for meta programming and constraints [Ne88,Ne90], multi user implementations of Prolog with a shared database, database systems [Kü88] etc. One of the results of the project was the design and implementation of the VAM (Vienna Abstract Machine).

In order to outline the major differences between VAM and WAM, we will first present an abstract machine for restricted clauses in Chapter 2. This simplistic model is generalized in Ch. 3 to the basic VAM, focusing on the emulator implementation VAM2P and its basic optimization schemes. Further aspects of VAM are dealt with in Ch. 4. In particular: meta-call, garbage collection, occur-check, delay mechanisms and constraints. Finally, more sophisticated optimizations are presented: the native code model VAM1P and improved calling sequences for a hybrid between VAM2P and VAM1P. Ch. 5 gives a brief comparison with WAM and describes future work on VAM currently under investigation.
clause(Head,Goals) :- head(Head), body(Goals).

body(true) :- [c-nogoal].
body(Goals) :- goallist(Goals), [c-lastcall], {Goals \= true}.

goallist(Goal) :- goal(Goal), {Goal \= (_,_)}.
goallist((GoalA,GoalB)) :- goallist(GoalA), [c-call], goallist(GoalB).

head(Head) :- [F/A], {functuniv(Head,F,A,L)}, argumentlist(h,L).
goal(Goal) :- [c-goal,F/A], {functuniv(Goal,F,A,L)}, argumentlist(g,L).

argument(X,Const) :- [X-const,Const], {const(Const)}.
argument(X,Str) :- [X-struct,F/A], {functuniv(Str,F,A,L), A>O}, argumentlist(X,L).

argumentlist(X,[]) :- [].
argumentlist(X,[E|Es]) :- argument(X,E), argumentlist(X,Es).

functuniv(Funct,F,A,L) :- functor(Funct,F,A), Funct =.. [FIL].

Figure 1: Clause representation in VAM for ground clauses

2 A simplified model for ground clauses

Initially, we restrict clauses to those containing no variables at all. While this is not a realistic subset of Prolog for practical applications, it serves to clarify the fundamental differences between WAM and VAM. Later on, we will relax this restriction to programs, where variables are allowed in clauses, provided they will unify with other variables or constants only.

2.1 Representation of clauses

The representation of clauses in VAM intermediate code is very close to their syntactic representation. By and large, terms are translated to a flat prefix code. In a clause three different kinds of codes are used:

control codes, (c-Any) are used to embrace goals. A goal starts with c-goal ( ) and either ends with c-call if another goal is thereafter or ends with c-lastcall if it is the last goal in a clause. If the clause is a fact we have no goal at all denoted by c-nogoal.

head codes, (h-Any) are used to encode terms in the arguments of a clause's head. The arguments are translated into flat prefix code.

goal codes, (g-Any) encode terms in goals. The structure is the same as for head codes.

In Fig. 1 the complete (bijective) mapping between ground clauses and VAM code is defined by a Definite Clause Grammar (DCG).

2.2 Specification of VAM

VAM instructions differ fundamentally from WAM instructions. They can be understood only by their combination at runtime. The real instruction set of VAM is the set of all valid combinations of instructions. Taking the translation of Prolog clauses to VAM code and a simple meta-interpreter as input, the VAM could probably be derived automatically by partial evaluation (deduction)—being in the style of [Ku87]. However, the abstract interpreter as well as the complete VAM was designed by hand.

In Fig. 2 an abstract interpreter for VAM code is given. The specification describes the process of unification and (determinate) control in detail, but—similar to [BB83]—it