The YAIL: An Intermediate Language for the Native Compilation of Prolog Programs

Ricardo Lopes, Vítor Santos Costa
{rslopes,vsc}@ncc.up.pt

LIACC, Universidade do Porto,
Rua do Campo Alegre, 823, 4150 Porto, Portugal

September 9, 1996

A major goal for any Prolog implementor is to achieve the best performance for applications. Advanced compilation is essential for high performance. The last few years have seen improvements in Prolog compilation technology resulting in a return to the idea of native code execution. We present work on designing a novel native code compiler for Prolog. The compiler uses an intermediate language optimised for novel architectures, such as the ALPHA and performs several optimisations to obtain good quality code.

1. Introduction

A major goal for any Prolog implementor is to achieve the best performance for applications. Performance in logic programs can be obtained by using techniques such as advanced compilation techniques, by extensions to Prolog such as co-routining, constraints or tabulation, or by using parallel execution.

Advanced compilation is essential for high performance. Compiler-based systems appeared rather early in the history of logic programming. The first Prolog compiler, the DEC-10 Prolog system, was designed to generate DEC-10 assembly code which that could then be executed by the target machine. The search for portability and simplicity in the implementation in Prolog systems lead to a different approach, where compilers generate code for abstract machines that are then are emulated at run-time. Examples include Quintus Prolog [8], YAP [5], or SICStus Prolog [1], among many others. Most of these systems implement variants of David H. D. Warren's WAM [12] as their abstract machine.

The last few years have seen improvements in Prolog compilation technology resulting in a return to the idea of native code execution [11]. There were two major motivations for skipping emulation in recent systems:

- Overhead Reduction: Native code compilation eliminates the overhead intrinsic to the emulation of abstract machine instructions, which can take a significant percentage of execution time.

- Flexibility: native code computation allows for easier specialisation, whereas the WAM would require new abstract machine instructions for specialised cases. Note that reducing the granularity of WAM instructions is not a good idea, because it would increase overheads.

Native code generation can lead to a significant improvement in performance, at a cost. Portability is harder. The compiler will become much more complex, and usually slower. Whereas a Prolog emulator is a relatively compact piece of code, native code size can grow quite substantially, especially when all possible optimisations are applied. This can lead to large compiled binaries, and ultimately to a reduction in performance due to worse program locality.

In this report we present work on designing a novel native code compiler for Prolog. Our motivation in designing this system is to improve performance of a pre-existing system, YAP, by using native code compilation for user-defined predicates. Our initial goal is to reach the same or better performance than what is achieved by other native code systems, such as Aquarius or native SICStus Prolog. We aim at this work becoming freely available to the logic programming community

As in the case of Prolog by BIM or of SICStus Prolog [7] by we decided to start from a pre-existing system. YAP already provides access to a large library of built-in predicates and a compiler, which we would need to rewrite for a novel system. Moreover, by maintaining both emulated and native code, the user has the attitude to choose what is the best alternative for his or her individual modules. The drawback is that the implementation must support both modes, hence complicating the implementation of native code.

The paper is organised as follows. We first discuss our main design decisions. Next, we describe the YAIL, the intermediate abstract machine we use for the system. We then describe the compilation steps currently used in the system, and conclude with preliminary performance results and with some conclusions and suggestions for future work.

2. Designing the Native Code System

Two major implementation issues must be addressed in native code implementations. First, although most native code systems will use the WAM data representation, it is clear that the WAM instructions are too coarse-grained for a native code system, and that a finer intermediate language must be used. Second, there is a question of what should be the target language and system supported. One alternative is to generate assembly code. This should return the best possible execution, but makes portability harder. A different alternative is to generate code for a high-level language, such as C, and trust the abilities of modern optimising compilers.

2.1. Intermediate Languages

There are several approaches to the problem of intermediate languages for native code systems. In the WAMCC [3], the WAM is used as an intermediate language and compiled code is generated directly from here. This results in a clean implementation, but makes it difficult to obtain the best performance. In Aquarius [10] or PARMA [9] the WAM is avoided and code is used for an intermediate representation directly. Last, in SICStus Prolog [7] a WAM-like abstract machine is used in the first step, and a lower-level intermediate representation is used for native code generation. The advantages of using this extra step is that the original system already performs several optimisations for generating the WAM-like code. These optimisations can be used directly in the native code system, simplifying its implementation. We decided to use the SICStus approach in YAP. The native code system is an extra step, after WAM-like compilation.

Considering the actual intermediate language, our major goal was to take the best advantage of the underlying hardware. As in most related systems, we followed a RISC-like approach of keeping a set of simple instructions that can be directly translated to assembly code, plus some higher-level instructions that can be implemented either as separated routines or inline. Last, we tried to support novel features in modern architectures that can be emulated in previous architectures, such as `conditional move` operations.

2.2. Compiling the Intermediate Languages

A second question was whether to use C or an assembly language compiler. Notwithstanding the good results obtained with KLIC [2] or Mercury [4], study of Prolog implementations indicates that the best results for native code Prolog systems so far have been obtained with assembly based implementations. Performance of C based implementations is not yet to par with what one can obtain from careful generation of native code. We therefore decided to concentrate on generating assembly code.

An interesting solution would be using an intermediate representation, such as GCC's RTL. This approach allows us to use several steps in the GCC compiler for further optimisation. The main disadvantage is that RTL GCC is a very complex language, subject to major changes at the whim of the GNU CC developers. This means that in practice this solution will be more complex than direct generation of native code. For simplicity we decided to use assembly in the first experiment, and consider RTL for future implementations.

We decided to use the ALPHA RISC architecture as our target architecture. The ALPHA is probably the cleanest architecture currently available, and is an ideal RISC target. A second interesting advantage is that although previous logic systems have been designed for a 32 bit architecture, the ALPHA architecture has 64-bit data registers and a 64 bit address space, which is now being followed by other architectures, such as SPARC and MIPS. For Prolog implementers this has important advantages, such as simplifying the tag scheme. Last, the ALPHA architecture is not yet supported by any other native code system, making the YAIL the first system to do so.

3. YAIL - Yet Another Intermediate Language

The YAIL, "Yet Another Intermediate Language" is an intermediate language between assembly code generation and the WAM instruction set. It aims at fulfilling two goals. Firstly, to isolate the dependencies of the compiler from the target architecture. Secondly, to create a representation that could improve optimisations and transformations difficult (or impossible) in the original WAM. The second factor was the major factor in the current implementation.

The instructions were designed towards independence, so that reordering could be easily achieved. The main characteristics of the ALPHA architecture influenced the design, so that mapping could be implemented easily and efficiently. On the other hand, the design tried to avoid excessive dependency on characteristics from the ALPHA architecture. An important consideration throughout was easy translation to architectural registers. High performance in modern machines depends on avoiding unnecessary memory access overheads. Last, one major goal was minimising the work in compiling from the YAAM (the YAP dialect for the WAM) to YAIL.

3.1. The YAIL Instruction Set

The YAIL instruction set follows Aquarius approach by including simple and complex instructions. We do not have "kernel" instructions, to allow the further steps to decide whether to expand or to call sub-routines. Instructions are mostly represented as groups of simple instructions, with a few complex instructions to support common operations such as trailing and unification. Note that only two simple instructions can access memory in the YAIL: `load` and `store`. All other simple instructions operate on registers.

Instructions have been organised into four categories. They are:

- **Memory Access Instructions** (see table 1). These are just two instructions, one for loading and the other for storing registers.

Instruction	**Semantics**
`Load(Ra,Rb,disp)`	Ra = *(Rb+disp)
`Store(Ra,Rb,disp)`	*(Rb+disp)=Ra

Table 1: Memory Access Instructions.

- **Register Operation Instructions** (see table 2). These include most general purpose instructions that can be directly translated into Assembly.

Instruction	**Semantics**
`test(cond,Ra, label)`	if cond(Ra) jump label
`compare(cond,Ra,Rb,Rc)`	Rc=(Ra cond Rb)
`comparei(cond,Ra n,Rc)`	Rc=(Ra cond n)
`move_cond(cond,Ra,Rb,Rc)`	if cond(Ra) Rc=Rb
`move_condi(cond,Ra,n,Rc)`	if cond(Ra) Rc=n
`set(Ra,Rb,n)`	Ra=Rb+n (n com 16 bits)
`add(Ra,Rb,Rc)`	Ra=Rb+Rc
`addv(Ra,Rb,n)`	Ra=Rb+n
`mul(Ra,Rb,Rc)`	Ra=Rb*Rc
`mulv(Ra,Rb,n)`	Ra=Rb*n
`shift(Ra,Rb,n)`	Ra=Rb$\gg$ n

Table 2: Simple Instructions that operate on Registers.

The `test`, `compare`, `comparei`, `move_cond` and `move_condi` require an extra parameter indicating which flag to test. Table 3 describes the test conditions.

Conditions for `test` and `move_cond`	**Conditions for `compare`**
`Equal to zero`	`EQual`
`Greater or Equal than zero`	`Less or Equal`
`Greater Than zero`	`Less Than`
`Low Bit Clear`	`Unsigned Less or Equal`
`Low Bit Set`	`Unsigned Less Than`
`Less Than zero`	
`Less or Equal to zero`	
`Not Equal to zero`	

Table 3: Available Flags for Comparison Instructions

- **Complex Instructions** that operate on registers (see table 4). Although all these instructions could be organised as a simpler set of instructions, compacting them simplifies several optimisations. One example is dereferencing variables.

Instruction	**Semantics**
`deref( Ra,Rb)`	Ra=deref(Rb)
`failz(Ra)`	if (!Ra) goto fail
`failnz(Ra)`	if (Ra) goto fail
`trail(Ra)`	trail(Ra)
`trailGlobal (Ra)`	trailGlobal(Ra)
`trailLocal(Ra)`	trailLocal(Ra)
`unify(Ra,Rb)`	unifycells(Ra,Rb)
`unifyGround(Ra,Rb)`	unifyground(Ra,Rb)
`push(Ra)`	push(Ra)
`pop(Ra)`	pop(Ra)

Table 4: Complex Instructions.

Note that some of these instructions, such as `deref`, `trail` and friends, `push` and `pop` can access memory. The unification operation is one of the most complex and in some cases may require calling an unification procedure.

- **Tag manipulation instructions** (see table 5). The major goal here is to obtain portability, as different architectures require different tag schemes. Usually these instructions can be implemented as simpler instructions.

Instruction	**Semantics**
`IsVar(Ra,Rb)`	Return 0 in Ra if Rb is not a variable
`IsAppl(Ra,Rb)`	Return 0 in Ra if Rb is not a compund term
`IsPair(Ra,Rb)`	Return 0 in Ra if Rb is not a pair
`RepAppl(Ra,Rb)`	untag Rb into Ra
`RepPair(Ra,Rb)`	untag Rb into Ra
`AbsAppl(Ra,Rb)`	Ra is Rb tagged as compound term
`AbsPair(Ra,Rb)`	Ra is Rb tagged as a pair

Table 5: Tag Support Instructions.

Note that the design follows a RISC-like (and WAM-like) architecture in trying to keep often used variables in registers.

3.2. Register Organisation in YAIL

The YAIL supports an infinite number of registers. It is the task of later steps in the compilation process to restrict the number of registers to the ones available in the implementation. This is the current organisation:

- `R0`, zero valued register;
- `R1` - `R9`, general purpose registers for control of execution (see table 6);
- `R10` - `R13`, YAIL temporary registers;
- `R14` - {} , WAM temporary registers `X[i]`;

YAIL	YAAM
R1	S
R2	H
R3	Y
R4	ENV
R5	B

YAIL	YAAM
R6	HB
R7	TR
R8	CP
R9	P

Table 6: Equivalence between YAIL and YAAM registers

Table 6 shows the equivalence between registers in the YAIL and YAAM/WAM registers. Temporary registers are used as follows:

- `R10` - used when writing structures to control when branch back to read-mode code.
- `R11`,`R12` - used to receive output of dereferencing.
- `R13` - used in comparisons.

4. Compilation

Native YAP uses two intermediate languages for Prolog compilation before using the YAIL. First it compiles Prolog to YAAM code, a WAM approximation developed for YAP with some improvements. Second it converts the YAAM code to YAAM 2-streams. This is the YAAM code divided in two area codes, read code and write code. The read code includes the YAAM instructions which execute in read mode. On the write code it includes the YAAM instructions executing in write mode. The YAAM 2-streams has advantages over YAAM because there is a significant reduction on jumps, wich are very expensive on RISC machines. For example, consider the case in which we have one `get_struct` followed by n `unify's`. In the YAAM the `get` instruction set the `Mode` register to write or read depending if the argument is var or nonvar. All the `unify's` instructions that fallow must test that register to decide in which mode they are going to execute. In the YAAM 2-streams the `get` instruction starts by dereferenciating the argument in read code, then it jumps to write code if it concludes to be var, otherwise it continues on read code. The `unify's` that fallow the `get` instruction will execute without the need of any test in boths cases

After converting the Prolog code to YAAM 2-streams, the latter is converted to YAIL. Many unnecessary sequences of register-to-register moves are removed at this stage, but some optimisations are only possible by using a specialised phases that we discuss next.

4.1. Optimising YAIL Code

The techniques that we have used to improve YAIL code are applicable to many languages. A transitive closure is performed on a sequence of eight transformations, i.e., they are applied repeatedly until there are no more changes in code (see figure 1). Each transformation was carefully studied so the resulting code is better, and so the closure operation terminates. By better code we mean code that is either faster, or smaller, or both.

Next, we describe the eight operations used to improve the code. For each operation we present a small example. For simplicity in those examples we use the instruction `jump(label)` that is a short name for the YAIL instruction `test(equal,R0,label)`. Remember that `R0` is always zero.

1. **Label elimination** (figure 2). All `label` instructions that are not used by other instructions can be eliminated. Also, when two `label` instructions appear together the second one can be eliminated after changing all it references to the other `label`.

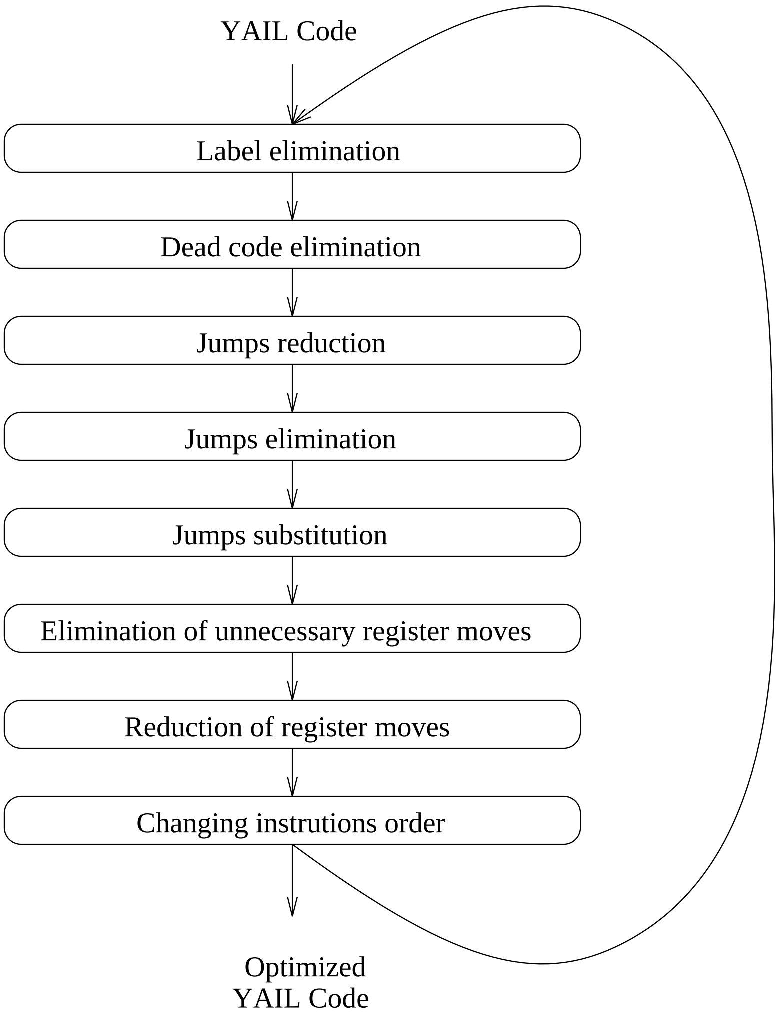

Figure 1: YAIL transformations

2. **Dead code elimination**(figure 3). Every non `label` instruction that follows a `jump`, `proceed`, `execute` or `failz R0` can be eliminated. Also, every instruction that do not change the execution state can be removed. Examples of these instructions are: `failnz R0`, `set Rx,Rx,0`, `add Rx,Rx,R0`.

3. **Jump reduction** (figure 4). Replace a `jump` instruction whose destination is another `jump` or a `fail` instruction by a new `jump` identical to the original one except that the destination label is replaced by the destination label of the `jump` or by `fail`.

4. **Jump elimination:** (figure 5) a `jump` or `test` instruction followed by the `label` instruction to where it points can be removed.

5. **Jump substitution** (figure 6). Replaces the `jump` instruction by the code it points if that block of code is shorter than a pre-set variable.

```
<...>                                 <...>
labely:                               labelx:
labelx:                  --->         <...>
<...>                                 jump labelx
jump labely
```

Figure 2: Label elimination

```
<...>                                 <...>
jump labelx                           jump labelx
<ins. different of label>             labely:
labely:                               <...>
<...>                    --->         <...>
labelw:                               labelw:
failnz R0                             <...>
<...>                                 <...>
```

Figure 3: Dead code elimination.

```
<...>                                 <...>
labely:                               <...>
jump labelx                           jump labelx
<...>                    --->         <...>
jump labely                           jump labelx
<...>                                 <...>
```

Figure 4: Jump reduction.

```
<...>                                 <...>
jump labelx              --->         <...>
labelx:                               labelx:
<...>                                 <...>
```

Figure 5: Jump elimination.

```
<...>                                 <...>
jump labelx                           proceed
<...>                    --->         <...>
labelx:                               labelx:
proceed                               proceed
<...>                                 <...>
```

Figure 6: Jump substitution.

6. **Elimination of unnecessary register moves** (figure 7). One instruction that puts a value in register `Rn` can be eliminated if it is followed by a block of instructions that do not use `Rn` plus another instruction that assembles another value to `Rn`.

```
<...>                                      <...>
set Rn,Ra,n                  --->          <...>
<ins. that do not use Rn>                  <no change>
set Rn,Rb,m                                set Rn,Rb,m
<...>                                      <...>
```

Figure 7: Elimination of unnecessary register moves.

7. **Reduction of register moves** (figure 8). One instruction that does a register move Rw to Rx can be eliminated if is followed by a block of instructions where Rx can be substituted by Rw, plus another instruction where another value is given to Rx.

```
<...>                                      <...>
set Rx,Rw,n1                               <...>
set Ry,Rx,m                   --->         set Ry,Rw,m+n1
set Rx,Rx,n2                               set Rx,Rw,n1+n2
<...>                                      <...>
```

Figure 8: Reduction of register moves.

8. **Change instructions order** (figure 9).

   Most instructions can be reordered to concentrate the biggest number of `set` instructions so that some of these can be eliminated using the preceding step.

```
<...>                                      <...>
set Rx,Ry,n                   --->         store Rw,Ry,n+m
store Rw,Rx,m                              set Rx,Ry,n
<...>                                      <...>
```

Figure 9: Changing instructions order.

None of those preceding steps work on their own, and in fact they where built to work together. The first five steps are used to reduce size code, eliminating code that is never used, as well as reducing jumps whenever possible. The last three phases are used to eliminate unnecessary moves between registers, using as key idea the fact that some instructions use a register plus a integer in their operation.

So that one can understand better the power of the optimisation that can be performed, we present one small extract of code that is very often found when using real programs in Prolog (see figure 10). Note that letters are used in register to make the example more legible, and that `CELLSIZE`

```
set   R_S,R_H,0              \  End code of instruction
set   R_H,R_H,2*CELLSIZE     /  unify_last_list_write
addv  R_T,R_Z,value_atom_1   \
store R_T,R_S,0               > unify_atom_write atom_1
set   R_S,R_S,CELLSIZE       /
addv  R_T,R_Z,value_atom_[]  \
store R_T,R_S,0               > unify_atom_write []
set   R_S,R_S,CELLSIZE       /
```

Figure 10: Non optimised block of code.

is used to represent the size of the word in target machine. On the YAAM/WAM code, presented on the right, no more improvement could be done. In YAIL code, using the fact that the access of memory is done using a register plus a displacement, two instructions can be removed, which is very good considering the whole code has only eight instructions (see figure 11).

```
addv  R_T,R_Z,value_atom_1
store R_T,R_H,0
addv  R_T,R_Z,value_atom_[]
store R_T,R_H,CELLSIZE
set   R_S,R_H,2*CELLSIZE
set   R_H,R_H,2*CELLSIZE
```

Figure 11: Optimised block of code.

4.2. Code generation to Alpha

After optimising the YAIL code, we are almost ready to start the code generation to Alpha. But before that, it is necessary to make register allocation, because in YAIL code it is assumed that an unlimited number of registers are available. We have created a simple scheme to allocate the Alpha registers, and it has proved to be more than adequate for this first version of Native YAP. The YAIL registers `CP, B, HB` are used as Alpha registers on native code, and as memory positions in YAP emulator. The others YAIL general purpose registers (see table 6) are always used as Alpha registers. The YAIL temporary registers are only used in native code and are always used as Alpha registers. For the WAM temporary registers `X[i]`, we use an region in memory where all that registers are represented. The start of this area is hold on an Alpha register Rn . To access to `X[i]` we use a load operation with Rn as argument plus a displacement of i times the word size (8 in Alpha). The registers `X[i]` could not be used as Alpha registers because the interleaving between the YAP and the native code would spoil the contents of those registers. As more instructions become executed in native code, avoiding the transition between the emulator and the native code, more Alpha registers can be used, and thus archive better performance.

After allocating the Alpha registers, the process of generating Alpha assembler code from YAIL is very simple. The instructions from tables 1 and 2 can be immediately transformed into one Alpha instruction. The instructions related with TAGS (table 5) implementation are also very simple, and are transformed into one or two Alpha instructions. As with the complex instructions (table 4), its transformation depends upon the instruction. For example, in Native YAP the `fail_X`, `trail_X`, `pop` and `push` are transformed directly into a small block of Alpha instructions. The `unify_X` are transformed in a subroutine call to a YAP function under the same name. The `deref` instruction is treated differently. In the optimisation process the `deref` instruction is forced to stay together to the instructions that use its results. This way when dereferencing a term the jump can be done immediately to the correct place whether the results is a variable or any other kind of term. In figure 12 is presented the deref cycle as implemented on Native Yap. On right is the case where code for X variable comes first. On left is the opposite.

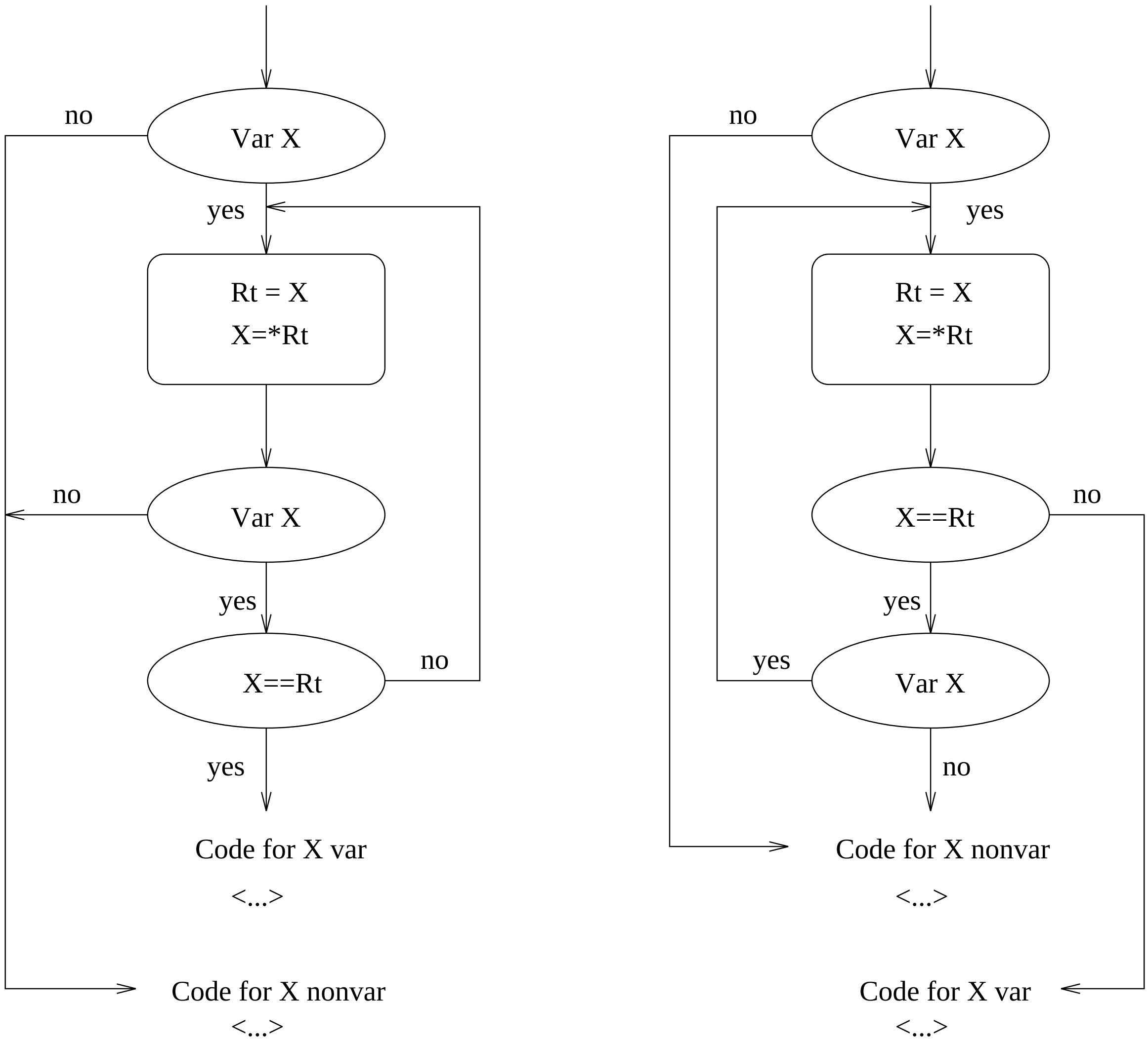

Figure 12: Deref cyle.

To identify the locations where the code of clauses begin, it are also generated several labels. On the beginning of code for each clause is created a unique label that identifies that clause. It is also created one label for each possible returning point within a clause. The code for each clause

Benchmark	Lines	Description
unification	8	Evaluate unification of a compound term
nreverse	10	Naive reverse of a 30-element list.
tak	15	Recursive integer arithmetic.
qsort	19	Quicksort of a 50-element list.
queens	45	Solve the queens problem
crypt	64	Solve a simple cryptarithmetic puzzle.
query	68	Query a static database (with integer arithmetic).
poly_10	86	Symbolically raise a polynomial to the tenth power.
magic	150	Solve the magic cube puzzle

Table 7: Benchmark descriptions

would look like the following code :

```
start_clause_code
    ...
    call ()
return_point1:
    ...
    call ()
return_point2:
    ...
    proceed
```

Note that the label `start_clause_code` is used to identify the location to where the execution must pass when we pretend to execute the clause in question. The labels `return_point` are used to identify the point to where execution must continue after returning from the call to other clauses.

5. Early Results

This section presents first results of Native YAP performance and compares them to YAP and YAPC (see table 8). YAPC is Native YAP generating `C` as output. We start by comparing YAPC and Native YAP to allow us to access the difference in performance by using C and assembler as our output. Almost all benchmarks presented are well known in the Prolog community (see table 7).

From table 7 we can see that Native YAP archives the best results, however, one can still ask why the Native YAP could not archive even better performance. In this first version of Native YAP the only code that is transformed to assembler is the clause code. The indexing instructions plus the connection instructions (`try,retry...`), are still executed by the emulator. So, when a program runs, its execution is divided by the emulator and the native code. This results in a major overhead, since the interleaving of native code and the emulator forces the contents of the cache to flushed, resulting in performance loss. More speed can be obtained as more instructions are executed in native code, avoiding at maximum the transition between the emulator and the native code.

The difference between the speedup results can be explained by the fact that some benchmarks spend a lot more time calling certain built-in predicates. Since the built-in predicates are the same in Native YAP and YAP, there can not be much of improvement. What other benchmarks, like nreverse, mainly do is to manipulate lists, requiring less calls to YAP, and therefore archive better performance.

Benchmark	**Yap**	**YapC**	**NYap**
unification	1	1.06	1.69
nreverse	1	1.44	1.74
tak	1	1.00	1.17
qsort	1	1.28	1.69
queens	1	1.30	1.44
crypt	1	1.20	1.20
query	1	1.10	1.10
poly_10	1	1.33	1.62
magic	1	1.02	1.67
Average	1	1.19	1.48

Table 8: Speedup of YAPC and Native YAP over YAP

The fact that YAPC is slower than Native YAP can be explained by the fact that for some YAIL instructions the compiler generate superfluous assembler instructions. For example, most of the time, when the compiler catches one `if` it tends to generate two jumps even when one would be sufficient. Yet another source for inefficiency is register usage. In C code we can't use so many registers as we can in assembler, because the compiler can also be using the registers without our knowledge.

6. Conclusions

An intermediate machine was developed to support the compilation of Prolog to native code. The YAIL integrates many of the recent ideas on intermediate languages for logic programming execution into a framework designed to support both easy translation from YAAM, and easy compilation into ALPHA assembly code. A first prototype of the YAIL compiler has been developed. Early results are promising, indicating a significant performance improvement over the emulator-based implementation of YAP, even when generating code for clauses only.

We next plan to support full procedure code, and to refine the YAIL instruction set through performance evaluation and perform further optimisations. We also plan to the use YAIL as target for sophisticated Prolog compilers, such as the optimisations being developed within the Proloppe project [6]. This might require changes to the YAIL architecture, such as the introduction of pragmas [11].

Acknowledgments

The authors would like to acknowledge and thank for the contribution and support that Fernando Silva and Luís Damas gave to this work. This work has been partly supported by JNICT under the PROLOPPE project (contract Praxis/3/3.1/TIT/24/94). Ricardo Lopes thanks PRAXIS (Grant Praxis/2094/94) for supporting his research.

References

[1] Mats Carlsson. Internals of Sicstus Prolog version 0.6. Internal Report, Gigalips Project, November 1987.

[2] Takashi Chikayama, Tetsuro Fujise, and Hiroshi Yashiro. A portable and reasonably efficient implementation of KL1. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, page 833, Budapest, Hungary, 1993. The MIT Press.

[3] Philippe Codognet and Daniel Diaz. `wamcc`: Compiling Prolog to C. In *12th International Conference on Logic Programming*. The MIT Press, 1995.

[4] Thomas Conway, Ferugus Henderson, and Zoltan Somgyi. Code generation for Mercury. In *ILPS95*, pages 245–256, 1995.

[5] L. Damas, V. Santos Costa, R Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1989.

[6] Michel Ferreira. Um Compilador de Prolog com Expansão de Código. Technical note, LIACC, Universidade do Porto, 1996.

[7] R. C. Haygood. Native code compilation in SICStus Prolog. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*. MIT Press, June 1994.

[8] *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[9] A. Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 174–185. MIT Press, June 1990.

[10] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[11] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *The Journal of Logic Programming*, 19/20, May/July 1994.

[12] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.