Appendix A

User manual for the Aquarius Prolog compiler

1. Introduction

The Aquarius Prolog compiler reads clauses and directives from stdin and outputs Prolog-readable compiled code to stdout as one fact per instruction. The output is assembly code for the Berkeley Abstract Machine (BAM). Directives hold starting from the next predicate that is input. Clauses do not have to be contiguous in the input stream, however, the whole stream is read before compilation starts.

This manual is organized into ten sections. Section 2 documents the compiler's directives. Section 3 gives the compiler's options. Section 4 gives a short overview of the dataflow analysis done by the compiler. Section 5 gives the type declarations accepted by the compiler. Section 6 summarizes the differences between Aquarius Prolog and the Edinburgh standard. Section 7 gives an example showing how to use the compiler. Section 8 describes the method used to compile specialized entry points to increase the efficiency of built-ins. Section 9 describes the assembly language interface. Section 10 describes how to define BAM assembly macros.

2. Directives

The directives recognized by the Aquarius compiler are given in Table 1.

3. Options

The Aquarius compiler's options are given in three categories: high-level (these options control actions of the compiler at the Prolog level), architecture-dependent (these options are constant for a particular architecture), and low-level (mainly useful for debugging purposes). The default options are set for the VLSI-BAM processor. The options are given in Tables 2, 3, and 4.

4. Dataflow analysis

Dataflow analysis is enabled with the `analyze` option. It generates ground, nonvar, recursively dereferenced and uninitialized variable types which are merged with the programmer's types. Both uninitialized memory and uninitialized register types are generated. Entry declarations (given by `entry` directives) are used to drive the analysis. Predicates of arity zero are always used as entry declarations. The quality of the generated types is such that compilation time, execution time, and code size are all significantly reduced. Therefore it is recommended always to compile with analysis. The whole program is kept in memory during the analysis.

All `mode`, `entry`, and `op` directives are executed before the analysis starts. Other directives are executed after the analysis and before compilation. The directives `default` and `clear` interfere with dataflow analysis, so they should be given only when the `analyze` option is disabled.

4.1. Dataflow analysis and dynamic code

The compiler makes the distinction between static and dynamic code. Static code is completely known at compile-time and is subject to analysis. Dynamic code is created at run-time by the built-in predicates `assert/1`, `retract/1`, and their cousins. It is not analyzed. There are two cases to consider:

(1) A dynamic predicate calls a static predicate. In this case, there must be an entry declaration giving the worst-case type of the call for each static predicate that might be called by a dynamic predicate.

Leaving out this declaration may result in incorrect compilation.

(2) A static predicate calls a dynamic predicate. The analyzer will assume worst-case types for the dynamic predicate unless it has a type declaration.

The most common uses of dynamic code are as databases of facts, or as rules that only call a limited set of static predicates. For these uses, there is no problem in integrating analyzed static code with dynamic code.

4.2. Dataflow analysis and the `call/1` built-in

The `call/1` built-in predicate can call any predicate in the program with any modes, and it is not possible in general to determine these predicates and their modes at compile-time. However, most programs that use `call/1` will call one of a known set of predicates or will call a dynamic predicate. There are three cases to consider:

(1) If the set of predicates that may be arguments of `call/1` is known by the programmer, then these predicates should be given entry declarations with worst-case modes. (This case can be written more efficiently by writing a new predicate that directly calls one of the set, and avoids calling `call/1`.)

(2) If the predicates that may be arguments of `call/1` are dynamic, then analysis is correct without entry declarations. This is true because dynamic predicates are not analyzed.

(3) If any predicate in the program may be an argument of `call/1` and nothing is known about the modes then analysis is useless and it should not be done.

5. Types

The Aquarius compiler accepts type declarations for a predicate. Using types results in a significant improvement in code quality. Types are represented as `(Head:-Formula)` where `Head` contains only variables and `Formula` is a logical conjunction. Almost any Prolog test can be used in a type formula. Possible type formulas are given in Table 5. This representation for types is simple, yet powerful enough to represent much important information in a compact way. The representation generalizes the declarations of Dec-10 Prolog. For example, the Dec-10 declaration:

```
:- mode(concat(+,+,-))
```

is expressed here as:

```
:- mode((concat(A,B,C):-nonvar(A),nonvar(B),var(C))).
```

6. Differences with Edinburgh Prolog

Aquarius Prolog recognizes new type-checking built-ins which are not part of the Edinburgh Prolog standard as embodied by C-Prolog. The new built-ins and their definitions in standard Prolog are given in Table 6.

7. An example of the compiler's use

The following example shows how the compiler is used:

```
% /hprg2/Bam/Compiler/compiler  % Run the compiler.
                               % Code is entered directly.
:-mode((a(A):-nonvar(A))).     % Enter the type.
a(a).                          % Enter a simple two-fact predicate.
a(b).
^D                             % End-of-file.
                               % The output follows:

% Cputime between start and finish is 1.383

procedure(a/1).
    deref(r(0),r(0)).
    hash(atomic,r(0),2,l(a/1,1)).
    fail.
label(l(a/1,1)).
    pragma(hash_length(2)).
    pair(a,l(a/1,3)).
    pair(b,l(a/1,4)).
label(l(a/1,3)).
label(l(a/1,4)).
    return.
```

8. Entry specialization for more efficient built-ins

The directive `modal_entry(Head,EntryTree)` adds a discrimination tree of entry points for the predicate `Head`. This directive is used by the system to implement more efficient built-ins. It is not normally needed by programmers, although they can take advantage of it for other predicates. The compiler uses the discrimination tree to choose the most efficient entry point for each call of a predicate depending on the type formula that is true at the predicate's call. The syntax of the discrimination tree in modal_entry is:

```
tree(entry(EntryHead)).
tree(mode(Formula,TrueTree,FalseTree)) :-
    tree(TrueTree), tree(FalseTree).
```

`EntryHead` is the entry point that replaces `Head` and `Formula` is a type formula. Compilation of a the predicate `Head` proceeds by following a path down the discrimination tree. If the formula valid when `Head` is called implies `Formula` then the `TrueTree` is followed. Otherwise the `FalseTree` is followed. Tree traversal stops when an entry point `entry(EntryHead)` is encountered. At that point the original call is replaced by `EntryHead`.

9. Interfacing with BAM assembly language routines

Prolog predicates can efficiently call routines written in BAM assembly code (the compiler's output) or in the target machine's assembly language (for example, VLSI-BAM, MIPS, or MC68020 assembly code). The interface with both low-level languages is provided through the five-argument type declaration. This declaration has the following form:

```
:- mode(Head, Require, Before, After, Survive).
```

`Head` is the head of the predicate. `Require` is the required type formula, i.e. the formula made true by the compiler. All uninitialized variable types (both uninitialized memory and uninitialized register) must be part of the required formula. `Before` is the type formula known to be valid before the call. `After` is the type formula known to be valid after the call. `Survive` is the register survive flag. If the flag is `y` then the predicate must not alter the values of any argument registers (except those used to return a result). It must save and restore any argument registers it needs. The predicate is called with a `simple_call` instruction and must return with a `simple_return` instruction (or its equivalent in

VLSI-BAM processor assembly). A simple call may not be nested. It is more efficient than a standard call because it does not need an environment frame around it in the calling routine.

If the survive flag is `n` then the predicate is assumed to invalidate all argument register values. In this case the argument registers are available as scratch registers and the calling routine will create an environment frame.

Efficient parameter passing is implemented by using uninitialized variables. These are of two kinds: uninitialized memory and uninitialized register variables. An uninitialized memory variable is a pointer to an empty memory cell. Binding to it is a store to memory. An uninitialized register variable is an empty register. Binding to it is a move to the register. No trailing or dereferencing is needed in either case.

Declaring an argument to have a uninitialized register type means that the output of the routine is stored in the corresponding argument register. Similarly, an uninitialized memory type requires the output to be stored to the location pointed to by the argument register. Inputs and outputs must be put in separate registers.

10. Defining BAM assembly language macros

It is possible to define macros in the Prolog source that are expanded into BAM assembly instructions. The advantages of macros are that they do not have call-return overhead, that unnecessary shuffling of data between registers is avoided, and that the full range of low-level compiler optimizations is performed on them. A macro definition has the following form:

```
:- macro((Head :- Body)).
```

where `Head` is the head of the predicate that will be expanded and `Body` is a series of BAM instructions. For example:

```
:- mode(quad(A,B), uninit_reg(B), true, deref(B), y).
:- macro((quad(A,B) :- add(A,A,X), add(X,X,B))).
```

The macro definition is preceded by a mode declaration telling that the second argument is the output.

Macro definitions must obey the following rules:

(1) All legal BAM instructions and addressing modes are allowed in the macro definition including user instructions, except as noted below. User instructions are never generated by the compiler, but they are recognized and optimized in macro definitions. Labels are given as ground terms or as Prolog variables. The latter are given unique ground values by the compiler. Registers are given as user registers (e.g. `r(h)` and `r(t2)`) or as Prolog variables (e.g. `X` and `Y`). The latter are allocated by the compiler. Do not use numbered registers (`r(0)`, `r(1)`, ...).

(2) The macro definition must be preceded by a mode declaration. The exit modes must be valid upon exiting the macro. All head arguments that return results must be of uninitialized register type.

(3) The macro may not alter any of the head arguments except those returning a result.

(4) The second argument of the `deref(X,Y)` instruction must be a new variable, i.e. it must not have a value upon entering the macro. Failing to obey this constraint will lead to incorrect behavior on backtracking.

(5) It is not recommended to create choice points inside macros since it is not known how many registers are live.

Table 1 – Compiler directives

Directive	Action
`:- help.`	Print a summary of these directives.
`:- default.`	Set the default options for the VLSI-BAM processor and clear all type declarations and modal entries.
`:- mips.`	Ensure compatibility with the MIPS processor. This directive should occur only once in a file. It sets the option align(1), disables the option split_integer, and sets all other options to their default values. It clears all type declarations and modal entries.
`:- vlsi_plm.`	Ensure compatibility with the re-microcoded VLSI-PLM. This directive should occur only once in a file. It sets the options high_reg(6) and align(1), disables the option split_integer, and sets all other options to their default values. It clears all type declarations and modal entries. Trail checks and shifts are compiled differently.
`:- clear.`	Clear all type declarations and modal entries.
`:- option(Options).`	Add the options in `Options` to the current options. `Options` may be a single option or a list of options.
`:- notoption(Options).`	Remove the options in `Options` from the current options. `Options` may be a single option or a list of options.
`:- printoption.`	Print a list of the currently active options.
`:- mode((Head:-Formula)).`	Type declaration for a predicate. The type information is remembered until new types are given for that predicate or until all type information is cleared. This declaration is not used as a starting point for dataflow analysis. However, the types generated by dataflow analysis are used to supplement the declaration, and an error message is given if there is a contradiction.
`:- entry((Head:-Formula)).`	Type declaration for a predicate—same as above. This declaration is also used as a starting point in dataflow analysis.
`:- mode(H,R,B,A,S).`	Detailed type declaration for a predicate. This declaration is useful for interfacing with assembly language. H is the head, R is the required type formula (made true by the compiler before each call), B is the before type formula (assumed true before each call), A is the after type formula (assumed true after each call), S is the survive flag (y/n depending on whether the call lets registers survive). The after type formula is used by dataflow analysis to improve the generated types.
`:- entry(H,R,B,A,S).`	Detailed type declaration for a predicate—same as above. This declaration is also used as a starting point in dataflow analysis.
`:- modal_entry(H,T).`	Optional discrimination tree of efficient entry points for the predicate H. The tree T contains type formulas used to replace each call of the predicate by a more efficient entry point.
`:- macro((Head:-Body)).`	Macro definition. The head is expanded into a sequence of BAM assembly instructions.
`:- include(FileName).`	Insert the text of the file `FileName`. This directive may be nested up to the system limit of simultaneous open files.
`:- pass(Anything).`	Pass the input ‘‘`:- pass(Anything).`’’ unaltered to the output in Prolog-readable form.
`:- version.`	Print the creation date of this version of the compiler.
`:- op(A,B,C).`	Declare an operator in Prolog. Pass the input ‘‘`:- op(A,B,C).`’’ unaltered to the output in Prolog-readable form.

Table 2 – High-level compiler options

Option	Default	Description
`select_limit(L)`	L=1	Perform selection for up to L arguments. Selection is done according to the enrichment heuristic. See Chapter 4 section 6.2.
`analyze`	off	Perform dataflow analysis for all predicates in the input stream. This option enables analysis of the entire input stream, no matter where it occurs in the stream. The starting points for analysis are the entry declarations and all predicates of arity zero. The types obtained from the analysis are merged with the programmer's types. The predicates are then compiled with the merged types.
`compile`	on	Compile the input. When this option is disabled, the entry types generated by the dataflow analyzer for the source predicates are output as valid Prolog-readable type declarations.
`factor`	on	Do factoring source transformation. With this transformation similar compound terms in adjacent heads are only unified once. Often this gives faster code.
`comment`	on	Give information about what the compiler is doing.
`same_number_solutions`	on	Keep the same number of solutions on backtracking as standard Prolog. Relaxing the semantics by removing this option results in better code in some cases.
`same_order_solutions`	on	Keep the same order of solutions on backtracking as standard Prolog. Relaxing the semantics by removing this option results in better code in some cases.
`depth_limit(D)`	D=2	Nesting depth limit on unification goals. Unifications deeper than this limit are transformed to remain within this limit. This transformation is used because compilation time and code size for deeply nested unifications would otherwise increase as the square of the size of the unification.
`short_block(S)`	S=6	Threshold on basic block length for shuffle optimization.

Table 3 – Architecture-dependent compiler options

Option	Default	Description
`low_reg(L)`	L=0	Lowest numbered machine register.
`high_reg(H)`	H=100	Highest numbered machine register. In the VLSI-BAM processor, registers higher than `r(15)` are mapped into memory.
`low_perm(P)`	P=0	Lowest numbered permanent variable.
`hash_size(H)`	H=5	Minimum size of a hash table.
`align(K)`	K=2	Align all compound terms to start on a multiple of K.
`uni`	on	Generate unify_atomic instruction to unify with an atomic term.
`split_integer`	on	Use separate tags for negative and nonnegative integers.

Table 4 – Low-level compiler options

Option	Default	Description
`system(X)`	quintus	The system running the compiler (other value: cprolog).
`write`	on	Write the object code when compilation is complete.
`peep`	on	Do peephole optimization.
`stats(S)`	off	Print timing statistics during compilation. S is one of the following atoms, or a list of them: `t` (top level of compilation), `c` (compilation of a single procedure), `p` (peephole optimization), `s` (selection algorithm—extraction of determinism), `d` (deterministic code generation).
`debug`	off	Print debugging messages during compilation.

Table 5 – Type formulas

Type	Meaning
`nonvar(A)`	A is a nonvariable term, i.e. its main functor is instantiated. Nothing is implied about its arguments.
`ground(A)`	A is a ground term, i.e. it contains no unbound variables.
`var(A)`	A is an unbound variable.
`uninit(A)`	A is an uninitialized memory variable. At the Prolog level, this means that A is an unbound variable known not to be aliased to another variable. In the implementation, A is a pointer to an empty memory cell. Binding to this variable is a simple store, without dereferencing or trailing.
`uninit_reg(A)`	A is an uninitialized register variable. At the Prolog level, this has the same meaning as an uninitialized memory variable. In the implementation, A is an empty machine register. This type increases the efficiency of parameter passing by returning a value directly in a register. It is useful for interfacing with assembly language.
`deref(A)`	A is dereferenced.
`rderef(A)`	A is recursively dereferenced, i.e. A is dereferenced and all subterms of A are recursively dereferenced.
`structure(A)`	A is a structure.
`list(A)`	A is a list, i.e. a cons cell or nil.
`cons(A)`	A is a cons cell, i.e. a non-nil list.
`compound(A)`	A is a structure or a cons cell.
`functor(A,F,N)`	A is the structure F with arity N.
`atom(A)`	A is an atom.
`atomic(A)`	A is atomic, i.e. a number or an atom.
`simple(A)`	A is atomic or an unbound variable.
`integer(A)`	A is an integer.
`float(A)`	A is a floating point number.
`number(A)`	A is an integer or a float.
`negative(A)`	A is a negative integer.
`nonnegative(A)`	A is a nonnegative integer.
`A>0`	A is a positive integer.
`A==x`	A is the atom `x`.
`true`	Nothing is known about the type.
`fail`	This means ‘‘execution can never reach this point.’’
`(F1,F2)`	This means ‘‘F1 and F2,’’ where F1 and F2 are type formulas.
`(F1;F2)`	This means ‘‘F1 or F2,’’ where F1 and F2 are type formulas.
`not(F)`	This means ‘‘not F,’’ where F is a type formula.

Table 6 – New type-checking predicates in Aquarius Prolog

Predicate	Prolog Definition
`nil(A)`	`:- nonvar(A), A=[].`
`cons(A)`	`:- nonvar(A), A=[_|_].`
`list(A)`	`:- nonvar(A), (A=[] ; A=[_|_]).`
`compound(A)`	`:- nonvar(A), \+atomic(A).`
`structure(A)`	`:- nonvar(A), \+atomic(A), \+A=[_|_].`
`ground(A)`	`:- nonvar(A), functor(A, _, N), ground(N, A).`
`simple(A)`	`:- (var(A) ; atomic(A)).`
`negative(A)`	`:- integer(A), A<0.`
`nonnegative(A)`	`:- integer(A), A>=0.`
`is_list(A)`	`:- (var(A), ! ; A=[] ; A=[_|B], is_list(B)).`
`is_partial_list(A)`	`:- (var(A), ! ; A=[_|B], is_partial_list(B)).`
`is_proper_list(A)`	`:- (var(A),!,fail;A=[];A=[_|B],is_proper_list(B)).`

The following clauses are part of the definition:

```
ground(N, _) :- N=:=0.
ground(N, A) :- N=\=0, arg(N, A, X), ground(X), N1 is N-1, ground(N1, A).
```