TM-0811

Optimization of GHC Programs

by
K. Ueda & M. Morita

October, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456 3191 - 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Optimization of GHC Programs

(Extended Abstract)

Kazunori Ueda[†] *and Masao Morita*[‡]

[†] Institute for New Generation Computer Technology

[‡] Mitsubishi Research Institute

September 20, 1989

Abstract. Concurrent processes can be used both for programming computation and for programming storage. Previous implementations of (Flat) GHC, however, have been tuned for computation-intensive programs, and perform poorly for storage-intensive programs and demand-driven programs. This paper proposes an optimization technique for programs in which processes are almost always suspended. The technique is based on a mode system which is powerful enough to analyze bidirectional communication and streams of streams. The proposed technique is expected to expand the application areas of concurrent logic languages.

1. Motivations

Guarded Horn Clauses (GHC) [Ueda 1986] is a simple concurrent logic language born from the research on parallelism in logic programming. Its subset, Flat GHC [Ueda and Furukawa 1988], can be viewed naturally as a process description language in which the static property of a process, namely the relationship between input and output information, is expressed in terms of its logical reading and in which the dynamic property, namely the causality between input and output information, is specified using the *guard* construct. Readers who are unfamiliar with GHC and concurrent logic programming are referred to [Shapiro 1987] and [Ueda 1989].

A prominent feature of Flat GHC and other concurrent logic languages viewed as process description languages is that they use *unification* (or its restricted form, *matching*) for interprocess communication. Externally, a process is viewed as an entity that observes and generates substitutions. Internally, the behavior of a process is defined in terms of other processes using guarded clauses. A guarded clause making up a program can be regarded as a conditional rewrite rule of a process, whose guard specifies what substitution should be observed before performing the rewriting. A substitution is generated by spawning a unification process whose behavior is language-defined.

Concurrent logic languages employ the notion of streams, implemented as lists, for interprocess communication. Unlike most concurrent languages, a sequence of messages communicated is just a data structure manipulated by unification, and this contributes much to the simplicity and the flexibility of the languages. Unidirectional (data-driven) communication, bidirectional (demand-driven) communication, and even a stream of streams can be programmed quite easily.

It has been claimed, however, that unification is too inefficient for interprocess communication. Upon unification, a straightforward implementation should determine the direction of dataflow and also check against the possibility of failure. These operations are considered as overheads in that they are not needed in other concurrent languages. The overheads should be more serious in parallel implementations. Another argument against unification for interprocess communication is that its straightforward implementation performs dynamic memory allocation (*cons*), which necessitates some sort of garbage collection. These considerations motivated us to explore the possibility of static analysis of complex dataflow.

Another motivation comes from our hope to expand the application areas of concurrent logic languages. So far, concurrent logic languages have mainly been used for writing computation-intensive programs in which processes do not suspend frequently. However, those languages could

```
nt_node([],                 _, _, L,R) :- true | L=[], R=[].
nt_node([search(K,V)|Cs],K, V1,L,R) :- true | V=V1, nt_node(Cs,K,V1,L,R).
nt_node([search(K,V)|Cs],K1,V1,L,R) :- K<K1 |
        L=[search(K,V)|L1], nt_node(Cs,K1,V1,L1,R).
nt_node([search(K,V)|Cs],K1,V1,L,R) :- K>K1 |
        R=[search(K,V)|R1], nt_node(Cs,K1,V1,L,R1).
nt_node([update(K,V)|Cs],K, _, L,R) :- true | nt_node(Cs,K,V,L,R).
nt_node([update(K,V)|Cs],K1,V1,L,R) :- K<K1 |
        L=[update(K,V)|L1], nt_node(Cs,K1,V1,L1,R).
nt_node([update(K,V)|Cs],K1,V1,L,R) :- K>K1 |
        R=[update(K,V)|R1], nt_node(Cs,K1,V1,L,R1).

t_node([]               ) :- true | true.
t_node([search(_,V)|Cs]) :- true | V=undefined, t_node(Cs).
t_node([update(K,V)|Cs]) :- true | nt_node(Cs,K,V,L,R), t_node(L), t_node(R).
```

Program 1. A program defining binary trees of processes

be used also for programming storage such as dynamic data structures using processes as building blocks. For instance, given Program 1, a process `t_node(S)` acts as a binary tree database that accepts `search` and `update` commands.

Processes in storage-intensive programs are almost always dormant and should respond quickly to incoming messages which may not arrive successively. However, currently available implementations such as [Kimura and Chikayama 1987] and [Morita et al. 1987], which are tuned for computation-intensive programs, perform poorly for storage-intensive programs because of their heavy process switching overhead. New implementation techniques that optimize the latency rather than the throughput of interprocess communication are badly needed for executing those programs efficiently.

2. Mode System and Mode Analysis

The first step towards the optimization of interprocess communication is to analyze what forms of communication will take place when a program is executed. This section presents a mode system that generalizes our previous system [Morita and Ueda 1989] (that classifies the arguments of a predicate simply into input and output) to handle complex dataflow.

The basic idea is as follows. We assume that interprocess communication in Flat GHC is cooperative rather than competitive; that is, we assume that when several occurrences of the same variable (each occurring in some goal) have been generated in the course of execution, exactly one of them is an *output* occurrence which can determine its top-level function symbol and all the others are *input* occurrences. We also assume that the mode of an occurrence p of a variable in a goal a can depend on and only on the predicate symbol of a and the principal function symbols of all terms in which p occurs. For example, consider commands '`search(Key,Value)`' and '`update(Key,Value)`' used in Program 1. The mode of the second argument of a command can depend (and actually depends) on the command name, but cannot on the value of the first argument `Key`. Most GHC programs written so far are written, or can be easily rewritten, on these assumptions.

2.1 Mode System

Let *Pred* be a set of predicate symbols, *Fun* a set of function symbols (we do not distinguish between constant and function symbols), *Atom* a set of atoms, and *Term* a set of terms. For each $p \in Pred$ with the arity n_p, let N_p be the set $\{1, \ldots, n_p\}$. N_f is defined similarly for each $f \in Fun$. Furthermore, we define the set of *paths* P_t (for terms) and P_a (for atoms) as follows:

$$P_t = (\sum_{f \in Fun} N_f)^*, \quad P_a = (\sum_{p \in Pred} N_p) \times P_t.$$

An element of P_t can be denoted $\langle f_1, j_1 \rangle \ldots \langle f_n, j_n \rangle$, and an element of P_a can be denoted $\langle p, i \rangle p'$, where $p' \in P_t$. They are intended to specify a subterm of a term or an atom, respectively. That is, with each term t we associate a function $\overline{t} : P_t \rightarrow Term$ for obtaining its subterms, which is defined as follows:

$$\begin{cases} \overline{t}(\epsilon) = t; \\ \overline{t}(\langle f, j \rangle p') = \begin{cases} \overline{t_j}(p'), & \text{if } t \text{ is of the form } f(t_1, \ldots, t_n); \\ \bot, & \text{otherwise.} \end{cases} \end{cases}$$

A function for obtaining a subterm of an atom is defined similarly.

Finally, we define the set of *modes* M as

$$M = P_a \rightarrow \{\text{in}, \text{out}\},$$

where we assume in $\neq$ out for the codomain. Let $m \in M$ be a mode, $a \in Atom$ a goal at some stage of computation, and $p \in P_a$ a path such that $\overline{a}(p)$ is a variable. Informally, $m(p) =$ out means that the principal function symbol of $\overline{a}(p)$ is determined only by the goal a through this occurrence, and $m(p) =$ in means that the principal function symbol of $\overline{a}(p)$ is not determined through this occurrence.

Our mode system is based on the assumption that programmers obey the following conventions:

(1) A program can be given a mode m. This means that the mode of an argument of a predicate is uniquely given, but the mode of an argument of a function can depend on the context in which the argument occurs. The exception is the predefined predicate '=' for unification, whose different occurrences (calls) in a program can have different modes.

(2) Let a be a goal at some stage of computation, and $p \in P_a$ a path such that $\overline{a}(p)$ is a variable and $m(p) =$ in. Then the variable $\overline{a}(p)$ will not be rewritten to another term through this occurrence.

(3) Let a be a goal at some stage of computation, and $p \in P_a$ a path such that $\overline{a}(p)$ is a variable and $m(p) =$ out. Then the goal a will not suspend on $\overline{a}(p)$ because of this occurrence.

(4) A unification body goal is effectively an assignment to an uninstantiated variable.

(5) Of all occurrences of a variable generated in the course of execution, only one of them tries to determine its top-level function symbol.

2.2 Mode Analysis

The purpose of mode analysis is to find a feasible mode of a program. A feasible mode of a program is a mode that satisfies all the mode constraints imposed by the program. A program for which the mode analysis succeeds is guaranteed to enjoy the properties listed in Section 2.1.

To simplify the analysis, we first normalize the program using the method described in [Ueda and Furukawa 1988]. The obtained program has the following properties:

(1) No unification goals exist in guards.

(2) The set of unification goals in the body of a clause is of the form $v_1 = t_1, \ldots, v_n = t_n$, where

- v_i's are distinct variables occurring in the head of the clause,
- $v_1, \ldots, v_n$ do not occur in $t_1, \ldots, t_n$ or other goals in the body, and
- if some t_i is a variable, it occurs in the head.

For instance, Program 1 is in a normal form. Furthermore, to cope with the overloading of the predicate '=' in a monomorphic system, we assume that all its occurrences in a program are virtually indexed as '$=_1$', '$=_2$', ...

The constraints on the mode m of a program are as follows:

(1) If a predicate q examines a path p, $m(p) = \text{in}$. Here, a predicate q is said to *examine* p if

(1a) there is a clause head h such that $\overline{h}(p)$ is a non-variable,

(1b) there is a clause head h and a prefix p' of p such that $\overline{h}(p')$ is a variable occurring more than once in h, or

(1c) there is a clause head h and a prefix p' of p such that $\overline{h}(p')$ is a variable occurring in a guard goal. (Condition (1c) can be weakened depending on the guard goal.)

(2) The two arguments of a unification body goal $t_1 =_k t_2$ has inverse modes, that is,

$$\forall p \in P_t \big(m(\langle =_k, 1\rangle p) \neq m(\langle =_k, 2\rangle p)\big).$$

(3) If a subterm $\overline{a}(p)$ of a body goal a is a non-variable, $m(p) = \text{in}$.

(4) Let v be a variable occurring n times in some clause, where we do not count the second and the subsequent occurrences in the head and all the occurrences in guard goals. Let the $i(\leq n)$th occurrence be at the path p_i of an atom a_i (head or body goal) with the predicate symbol q_i. For each $i(\leq n)$, we define $m_i \in M$ as follows:

$$\begin{cases} \forall p \in P_a \big(m_i(p) = m(p)\big), & \text{if } a_i \text{ is a body goal;} \\ \forall p \in P_a \big(m_i(p) \neq m(p)\big), & \text{if } a_i \text{ is the clause head;} \end{cases}$$

Then, we impose the constraint

$$\forall p \in P_t \exists i \leq n \big(m_i(p_i p) = \text{out} \wedge \forall j \leq n (j \neq i \rightarrow m_j(p_j p) = \text{in})\big).$$

Intuitively, this says that each function symbol in a possible instance of v will be determined by exactly one of the occurrences of v. Note that i can depend on p in the above constraint. The reason why we introduce m_i's is that it enables us to treat all the occurrences of a variable in a uniform way; an input occurrence of a variable in a clause head considered a source of information from inside the clause.

For example, consider Program 2. Let $t_i(p)$ denote $m(\langle \texttt{test}, i\rangle p)$ and $s_i(p)$ denote $m(\langle \texttt{stack}, i\rangle p)$, for $i = 1, 2$. Let '.' denote the function symbol of a non-empty list. Constraints we can obtain from the predicate `test` include

$$t_1(\epsilon) = \text{in},\ t_2(\epsilon) = \text{out},\ t_2(\langle ., 1\rangle) = \text{out},$$
$$t_2(\langle ., 2\rangle) = \text{out},\ t_2(\langle ., 2\rangle\langle ., 1\rangle) = \text{out},$$
$$\forall p \in P_t \big(t_2(\langle ., 2\rangle\langle ., 2\rangle p) = t_2(p)\big),$$
$$t_2(\langle ., 1\rangle\langle \texttt{push}, 1\rangle) = \text{out},\ t_2(\langle ., 2\rangle\langle ., 1\rangle\langle \texttt{pop}, 1\rangle) = \text{in},$$

and those we can obtain from `stack` include

$$s_1(\epsilon) = \text{in},\ s_1(\langle ., 1\rangle) = \text{in},\ \forall p \in P_t \big(s_1(\langle ., 2\rangle p) = s_1(p)\big),$$
$$s_2(\epsilon) = \text{in},\ \forall p \in P_t \big(s_2(\langle ., 2\rangle p) = s_2(p)\big),$$
$$\forall p \in P_t \big(s_2(\langle ., 1\rangle p) = s_1(\langle ., 1\rangle\langle \texttt{push}, 1\rangle p)\big),$$
$$\forall p \in P_t \big(s_2(\langle ., 1\rangle p) \neq s_1(\langle ., 1\rangle\langle \texttt{pop}, 1\rangle p)\big).$$

There are many feasible modes for Program 2 which differ at paths of no interest; the above constraints are what all feasible modes satisfy.

Note that the concrete values of $s_1(\langle ., 1\rangle\langle \texttt{push}, 1\rangle)$, $s_1(\langle ., 1\rangle\langle \texttt{pop}, 1\rangle)$, and $s_2(\langle ., 1\rangle)$ cannot be determined solely by `stack`; they are determined only by supplying a context in which the predicate

```
test(M,S) :- M=:=0 | S=[].
test(M,S) :- M=\=0 | S=[push(M),pop(N)|S1], N1:=N-1, test(N1,S1).

stack([],          _     ) :- true | true.
stack([push(X)|S],D      ) :- true | stack(S,[X|D]).
stack([pop(X)|S], [Y|D1]) :- true | X=Y, stack(S,D1).
```

Program 2. A stack program and its driver

`stack` is used. For example, if the goal `test(10,S)` is used for driving the goal `stack(S,[])`, $s_1(\langle ., 1\rangle(\mathtt{push}, 1))$ and $s_2(\langle ., 1\rangle)$ are constrained to 'in', and $s_1(\langle ., 1\rangle(\mathtt{pop}, 1))$ is constrained to 'out'.

3. Process- vs. Message-Oriented Scheduling

The mode analysis technique in Section 2 has two major applications: One is the optimization of conventional implementations based on what we call *process-oriented scheduling*, and the other is a new implementation scheme based on *message-oriented scheduling* [Morita and Ueda 1989]. This paper deals with the latter. Although we focus on multiprocessing within one processor, we believe that our technique can be utilized also in parallel implementations.

In conventional, process-oriented scheduling, a scheduler tries to reduce the number of process switching. Once a process starts or resumes execution, it runs as long as possible (unless it is swapped out) before another process in a process queue gains control. A stream connecting processes act as a buffer whose contents are processed at once whenever possible. Process-oriented scheduling can be rephrased as *throughput-oriented* scheduling.

Message-oriented scheduling is at the other extreme. Whenever a process sends a message to another, it does not buffer the message but transfers control to the receiver process so that the receiver may consume the message immediately. (For simplicity, suppose for a while that interprocess communication is one-to-one, which is the case with Program 1.) The receiver process should be ready to receive and handle the message immediately. To this end, message-oriented scheduling always makes the consumer of a stream run ahead of its producer and makes the former suspend, while process-oriented scheduling would run the producer ahead of the consumer. The mode analysis enables the identification of the producer and the consumer of a stream. Message-oriented scheduling can be rephrased as *response-oriented* scheduling, because quicker responses can be expected in bidirectional communication.

For example, consider a process that simply copies the contents of the input stream to the output stream:

```
p([A|X1],Y) :- true | Y=[A|Y1], p(X1,Y1).
```

Of the two body goals, process-oriented scheduling first buffers the datum `A` by executing `Y=[A|Y1]` and then executes `p(X1,Y1)` efficiently with the aid of last-call optimization. In contrast, message-oriented scheduling first executes `p(X1,Y1)`, thus restoring the dormant state of the process, and then executes `Y=[A|Y1]` as message passing. The possible source of efficiency is the efficient transfer of control and data which does not use a process queue or a data buffer. To achieve this, we implement a stream not as a list but as a special cell (called a *communication cell*) pointing to the code and the environment (i.e., the process record) of the consumer process. A message to be transferred is placed on a hardware register called a *communication register*.

A process-oriented implementation often caches (part of) a process record on hardware registers, but this should not be done in a message-oriented implementation in which process switching takes place frequently.

Two questions arise here. One is how a compiler can distinguish between variables representing streams and those representing ordinary data; and the other is how to cope with communication

that is not one-to-one. Due to space limitations, regarding the first question we only note that a type system similar to the mode system in Section 2 can be employed to determine whether each path of interest is used as a stream or not.

How to cope with various forms of communication is more intriguing. First, a stream may have two or more consumers or no consumer at all. Second, a process may consume two or more streams in various ways.

The easiest way to implement stream communication with two or more consumers or with no consumer is to transform it into one-to-one communication. For example, when a process commits to the following clause,

```
consumer([kill|X]) :- true | true.
```

a dummy stream is created which eats up the rest of the messages in X. When there are two or more consumers initially or when a single consumer splits into two or more, a process for distributing messages is created. There should be more efficient ways of handling these cases; however, our primary concern here is to implement one-to-one communication as efficiently as possible. If message-oriented scheduling turns out to be inefficient for one-to-many communication, we could just use an ordinary implementation of lists for that.

Implementation of many-to-one communication seems more important, since it is ubiquitous in concurrent programming in GHC. We should consider two cases: *non-selective message receiving* and *selective message receiving*.

By non-selective message receiving we mean message receiving found in a nondeterministic merge program:

```
merge([A|X1],Y,Z) :- true | Z=[A|Z1], merge(X1,Y,Z1).
merge(X,[A|Y1],Z) :- true | Z=[A|Z1], merge(X,Y1,Z1).
```

Non-selective message receiving can be implemented exactly in the same way as one-to-one communication. The communication cells of different input streams point to different codes for processing incoming messages, and messages in one input stream are processed independently of the other input stream.

By selective message receiving we mean message receiving found in order-preserving merge of two streams of integers:

```
merge([A|X1],[B|Y1],Z) :- A< B | Z=[A|Z1], merge(X1,[B|Y1],Z1).
merge([A|X1],[B|Y1],Z) :- A>=B | Z=[B|Z1], merge([A|X1],Y1,Z1).
```

Two numbers, one from each input stream, are necessary for the first commitment. Suppose the first number arrives at the first stream. Then the process records it and waits for another number to arrive at the second stream. However, the second number may arrive at the first stream again. In that event, the process should buffer that number for later use.

Another example that requires buffering is the append program:

```
append([],    Y,Z) :- true | Z=Y.
append([A|X1],Y,Z) :- true | Z=[A|Z1], append(X1,Y,Z1).
```

Messages arriving at the second input stream must be buffered until the first input stream is closed. In either example, it is the responsibility of a process, rather than of a stream, to buffer messages that cannot be processed immediately.

In general, there are two cases where a process must buffer incoming messages. One case arises with selective message receiving as discussed above. The other case arises even without many-to-one communication. Consider the following clause:

```
p([a|X1],Y,Z) :- true | Y=[b|Y1], Z=[c|Z1], p(X1,Y1,Z1).
```

This clause says that a process `p(A,B,C)` may send two messages (say b_1 and c_1) in response to an incoming message (say a_1). Sending b_1 may cause another message (say a_2) to arrive at this process before c_1 is sent. In this case, the message a_2 should be buffered and processed after c_1 is sent, because otherwise the order of messages on the stream `C` would be reversed.

Fortunately, when only one message is sent in response to an incoming message, there is no need of buffering after commitment. To generalize, suppose a process should send n messages in response to an incoming message and hasn't received another message in response to the first $n-1$ messages. Then, the last message can be sent without preparing for buffering, and moreover, the control need not be returned to the process after the message has been handled by the receiver. This could be called *last-send optimization*, which is analogous to the last-call optimization of Prolog [Warren 1980].

4. Preliminary Evaluation

We are designing an abstract machine instruction set for message-oriented scheduling [Morita and Ueda 1989]. Initial performance evaluation using hand-compiled intermediate codes (which were translated into native codes of VAX11/780) was quite encouraging. Using Program 1, we measured the processing time of 800 `search` commands given to a binary process tree with 800 nonterminal nodes, and compared the result with the numbers on a native-code, process-oriented implementation on VAX11/780, GHC/V [Morita et al. 1987]:

Message-oriented:	0.75 sec.
Process-oriented, batch:	1.04 sec.
Process-oriented, interactive:	2.09 sec.

'Batch' means that 800 commands were given at a time and 'interactive' means that each command was issued after receiving the result of the previous command. The way commands were given made no difference in message-oriented scheduling.

For this program, message-oriented scheduling was more efficient than process-oriented scheduling even when all the commands were given at a time. In addition, message-oriented scheduling does not perform *cons* upon message sending. It is noteworthy that a binary tree program in C using records and pointers took 0.31 sec. for the same data on the same machine.

We also observed that message-oriented scheduling much improved the performance of a demand-driven program. The statistics obtained from data-driven and demand-driven prime number generators to compute 168 primes up to 1000 are as follows:

	data-driven	demand-driven
Message-oriented:	0.83 sec.	1.38 sec.
Process-oriented:	1.23 sec.	4.96 sec.

GHC/V, employing 32-bit words, ran naive reverse at 33kRPS (kilo-reductions per second), and this number improved to 53kRPS by optimization based on the mode analysis [Morita and Ueda 1989]. On the other hand, our message-oriented implementation, employing 64-bit words, ran naive reverse at 55kRPS and consumed less memory (i.e., linear size). It is interesting to see how a naive reverse program runs under message-oriented scheduling.

5. Conclusion

We have proposed a new implementation technique of Flat GHC that contrasts sharply with previous techniques. Although our primary goal was to optimize storage-intensive programs and demand-driven programs, the proposed technique worked quite well also for computation-intensive

programs which did not use one-to-many communication. The technique avoids *conses* for interprocess communication except when buffering is essential, which is another important aspect of the technique. We believe that our technique can be utilized also in parallel implementations, though much work has to be done to demonstrate it. One of our next goals is to implement distributed dynamic data structures efficiently.

The technique is based on a mode system which is simple and yet powerful enough to analyze most programs. The mode analysis is based on constraint solving rather than on abstract interpretation. The analysis of dataflow is important particularly on parallel implementations, whether the scheduling is process-oriented or message-oriented. Some concurrent languages such as Strand [Foster and Taylor 1989] introduces an assignment primitive instead of unification to generate bindings. However, without compile-time analysis, an assignment goal must still check if the left-hand side is a variable and if some process is suspending on that variable. Our mode analysis can be utilized for eliminating both [Morita and Ueda 1989]. Compile-time analysis is important also for reducing the size of native codes and making the use of native codes more realistic.

Acknowledgments

We are indebted to Koichi Furukawa and Kenji Horiuchi for valuable comments and suggestions.

References

[Foster and Taylor 1989] Foster, I. and Taylor, S. Strand: A Practical Parallel Programming Language. To be presented at the 1989 North American Conf. on Logic Programming, 1989.

[Kimura and Chikayama 1987] Kimura, Y. and Chikayama, T. An Abstract KL1 Machine Instruction Set. In *Proc. 1987 Symp. on Logic Programming*, IEEE Computer Society, 1987, pp. 468-477.

[Morita et al. 1987] Morita, M., Yoshimitsu, H., Dasai, T. and Ueda, K. GHC Compiler on a General-Purpose Computer. In *Proc. 35th Annual Convention IPS Japan*, 1987, pp. 759–760 (in Japanese).

[Morita and Ueda 1989] Morita, M. and Ueda, K. Optimization of GHC Programs. In *Proc. the Logic Programming Conference '89*, ICOT, 1989, pp. 203–214 (in Japanese).

[Shapiro 1987] Shapiro, E. Y. (ed.) *Concurrent Prolog: Collected Papers*, Vol. 1–2, The MIT Press, 1987.

[Ueda 1986] Ueda, K. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. ICOT Tech. Report TR-208, ICOT, 1986. Also in *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, 1988, pp. 441–456.

[Ueda 1989] Ueda, K. Parallelism in Logic Programming. In *Information Processing 89*, Ritter, G. X. (ed.), North-Holland, 1989, pp. 957–964.

[Ueda and Furukawa 1988] Ueda, K. and Furukawa, K. Transformation Rules for GHC Programs. In *Proc. Int. Conf. on FGCS'88*, ICOT, 1988, pp. 582–591.

[Warren 1980] Warren, D. H. An Improved Prolog Implementation Which Optimises Tail Recursion. In *Proc. Logic Programming Workshop*, Tärnlund, S. -Å. (ed.), Debrecen, Hungary, 1980, pp. 1–11.