Proceedings of the Seventh International Conference on Logic Programming, The MIT Press, Cambridge, MA, 1990, pp. 3–17.

A New Implementation Technique for Flat GHC

Kazunori Ueda
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan
ueda@icot.or.jp

Masao Morita
Mitsubishi Research Institute
3-6, Otemachi 2-chome, Chiyoda-ku, Tokyo 100, Japan

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 (such as programs implementing reconfigurable data structures using processes and streams) and demand-driven programs. This paper proposes an optimization technique for programs in which processes are almost always suspended. The technique compiles unification for data transfer into message passing. Instead of reducing the number of process switching operations, the technique optimizes the cost of each process switching operation and reduces the number of *cons* operations for data buffering. The technique is based on a mode system which is powerful enough to analyze bidirectional communication and streams of streams. The mode system is based on mode constraints imposed by individual program clauses rather than on global dataflow analysis. Benchmark results show that the proposed technique well improves the performance of storage-intensive programs and demand-driven programs compared with a conventional native-code implementation. It also improves the performance of some computation-intensive programs. Although many problems remain to be solved particularly with regard to parallelization, we expect that the proposed technique will expand the application areas of concurrent logic languages.

1. Motivations

Guarded Horn Clauses (GHC) [8][9] is a simple concurrent logic language born from the research on parallelism in logic programming. Its subset, Flat GHC [11], can be viewed naturally as a process description language in which the static property of a process (implemented by a multiset of body goals), 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 [7] and [10].

A prominent feature of Flat GHC and other concurrent logic languages viewed as process description languages is that they use unification (or its restricted forms such as matching) for interprocess communication. Externally, a process is viewed as an abstract entity that observes and generates substitutions. Internally, the behavior of an individual body goal, a multiset of which implements a process, is defined in terms of other goals using guarded clauses. Each of the guarded clauses making up a program can be regarded as a conditional rewrite rule of goals. It is the guard part of a clause that specifies what substitution should be observed before performing the rewriting. A substitution is generated by spawning a unification body goal 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, which contributes much to the simplicity and the flexibility of the languages. Unidirectional (data-driven) communication, bidirectional (demand-driven) communication, and furthermore, streams 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, which can be costly particularly in parallel implementations, are not needed in other concurrent languages. Another argument against interprocess communication by unification is that its straightforward implementation performs dynamic memory allocation (*cons*) that necessitates some sort of garbage collection. These considerations motivated us to explore the possibility of static analysis of complex dataflow, which is one of the two major topics of the paper.

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 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)` (for *terminal node*; `nt_node` for *non-terminal* node) acts as a binary tree database that accepts `search` and `update` commands through the stream `S`.

Processes in storage-intensive programs are almost always dormant but should respond quickly to incoming messages which may not arrive successively. However, currently available implementations such as [4] and [5], 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. This is another major topic of the paper.

```
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

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 [6] (which classified the arguments of a predicate simply into input and output) to handle complex dataflow. This generalization is very important because the flexibility of unification-based interprocess communication is the primary *raison d'être* of concurrent logic languages.

The purpose of our mode system is to infer "which goal will determine which part of a data structure, if each part is to be determined at all," rather than to infer the instantiation states of the arguments of goals as in [1].

Because our mode system is intended for static analysis, it is impossible to analyze the dataflow of all meaningful Flat GHC programs. We chose to assume that programmers obey the following conventions:

(1) Interprocess communication is *cooperative* rather than *competitive*; that is, 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 the *output* occurrence which can determine its top-level function symbol and all the others are *input* occurrences.

(2) The mode of an occurrence of a variable in a goal g can depend on and only on the predicate symbol of g and the principal function symbols of all terms containing that occurrence. 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 function occurs. For example, consider the commands `search(K,V)` and `update(K,V)` used in

Program 1. The modes of the second arguments `V` can depend (and actually depend) on the command names, but cannot on the values of the first arguments `K`. The exception to the above rule is the predefined predicate '`=`' for unification, whose different occurrences (calls) in a program can have different modes.

The introduction of a mode system into Flat GHC is effectively the subsetting of Flat GHC; the resulting language could be called *Moded Flat GHC*. So a question arises as to whether this subsetting is serious for GHC programming. Fortunately, most GHC programs written so far are written, or can be easily rewritten, following these conventions. One reason for this is that GHC provides no means to recover from the failure of output unification.

Some programs we have seen use the 'stop signal' technique; in those programs, several processes $p_1, \ldots, p_n$ share a variable v and agree upon a constant c that v will be bound to, and when some p_i finds that other processes need no longer to work, it notifies them by binding v to c. All of the p_i's are possible producers of the binding, though the failure of unification cannot happen. One way to conform those programs to the above conventions is to use an n-ary arbiter process which the p_i's can ask (via distinct variables) to bind v to c.

Finally, we claim that the above conventions also serve as a guideline for good GHC programming.

2.1 Mode System

As usual, we first fix the vocabulary with which programs are written and executed. Let *Pred* be the set of predicate symbols, *Fun* the set of function symbols (we do not distinguish between constant and function symbols), *Atom* the set of atoms, and *Term* the set of terms. For each $p \in Pred$ with the arity n_p, let N_p be the set $\{1, 2, \ldots, n_p\}$. N_f is defined similarly for each $f \in Fun$. Furthermore, we define the sets 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 $p' \in P_t$. The empty sequence in P_t will be denoted ϵ. Paths are intended to specify subterms of terms and atoms. That is, with each term t we associate a function $\overline{t} : P_t \to Term$ for obtaining its subterms, which is defined as follows:

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

The function for obtaining subterms of an atom is defined similarly.

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

$$M = P_a \to \{in, out\},$$

where we assume $in \neq out$ for the codomain.

The intended meaning of a mode $m \in M$ is as follows. Consider a process implemented by a goal $a_0 \in Atom$, and assume that the unification goals spawned so far by this process or other processes running in parallel have instantiated a_0 to a. Note that a records the information communicated with the outside. Let $p \in P_a$ be a path such that $\overline{a}(p)$ is a variable. Then,

(1) $m(p) = in$ means that the variable $\overline{a}(p)$ will not be rewritten to another term (possibly being a variable) through this occurrence, and
(2) $m(p) = out$ means that the process will not suspend on $\overline{a}(p)$ because of this occurrence.

2.2 Mode Analysis

The purpose of mode analysis is to find a feasible mode of a program, namely a mode that satisfies all the mode constraints (listed below) imposed by the program.

To simplify the analysis, we first normalize (the unification goals in) the program using the method described in [11]. 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 '=', we assume that all its occurrences in a program are virtually indexed as '$=_1$', '$=_2$', ...

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

(1) If some clause examines a path $p \in P_a$, $m(p) = in$. Here, a clause with the head h is said to *examine* p if
   (1a) $\overline{h}(p)$ is a non-variable, or
   (1b) there is a prefix p' of p (that is, a path $p' \in P_a$ such that form some $p'' \in P_t$, $p'p'' = p$) such that $\overline{h}(p')$ is a variable occurring more than once in h, or
   (1c) there is 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. For example, suppose $\overline{h}(p')$ is the variable `X`. Then a guard goal `X>5` for integer comparison needs to constrain the value of $m(p')$ but not the value of $m(p)$ unless $p' = p$.)
(2) The two arguments of a unification body goal $t_1 =_k t_2$ have exactly 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, $m(p) = in$.

(4) Let v be a variable occurring n times in some clause, where we do not count the second or the subsequent occurrences in the head or any of the occurrences in the guard goals. That is, at least $n-1$ occurrences are those of body goals and the remaining one, if any, represents the occurrence(s) in the head. Let the ith occurrence be at the path p_i of an atom a_i (that is either a head or a body goal). 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

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

Intuitively, this says that each function symbol occurring 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. When $n = 2$, which is usually the case, the above constraint is simplified to

$$\forall p \in P_t\big(m_1(p_1 p) \neq m_2(p_2 p)\big).$$

Motivations of Constraint (4) are appropriate here. Assume a goal g commits to a clause C. Constraint (4) states how the variables in the body goals of C should be instantiated from then on. A variable v occurring in the head of C is regarded as a communication channel between the body goals of C and other goals running in parallel. Multiple occurrences of the same variable in the head are for equality checking *before* commitment, and the only thing that matters *after* commitment is whether a variable occurring in the body occurs also in the head and conveys information to and from outside. This is why only one of the occurrences of v in the head is taken into account.

The reason why we introduce the m_i's is that it enables us to treat all the occurrences of a variable in a uniform way. An input (output) occurrence of a variable in the clause head is considered a source (sink) of information from inside the clause, respectively, and this is why we invert the mode of the clause head in considering Constraint (4).

2.3 An Example and Discussions

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

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

```
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

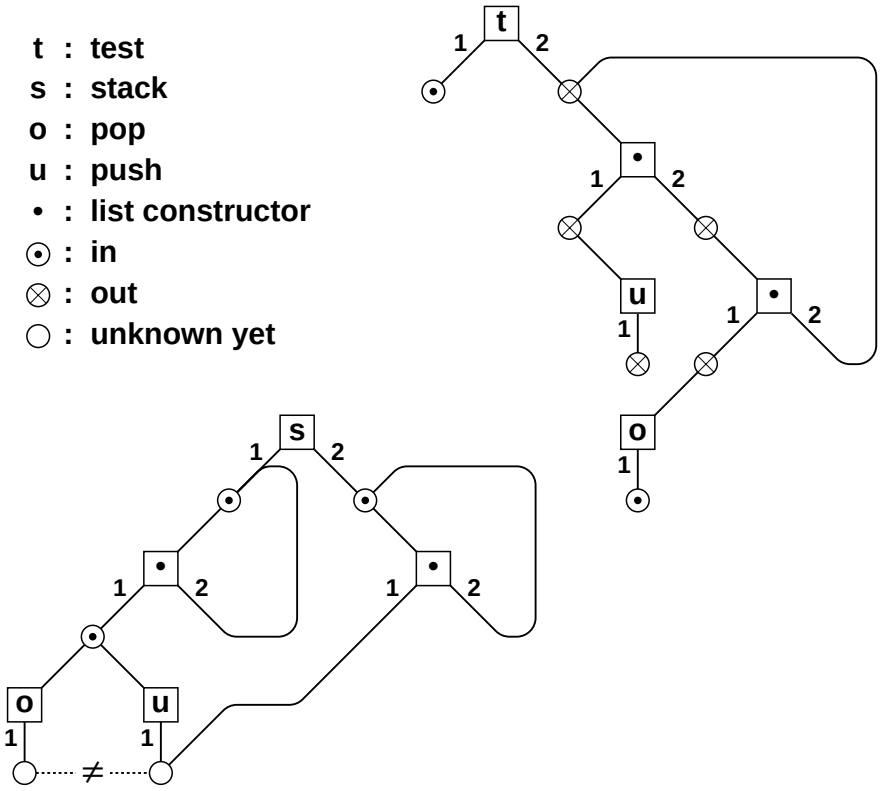

Figure 1. Mode constraints obtained separately from `test` and `stack`

and those we can obtain from `stack` include

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

Figure 1 illustrates these constraints using directed graphs.

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

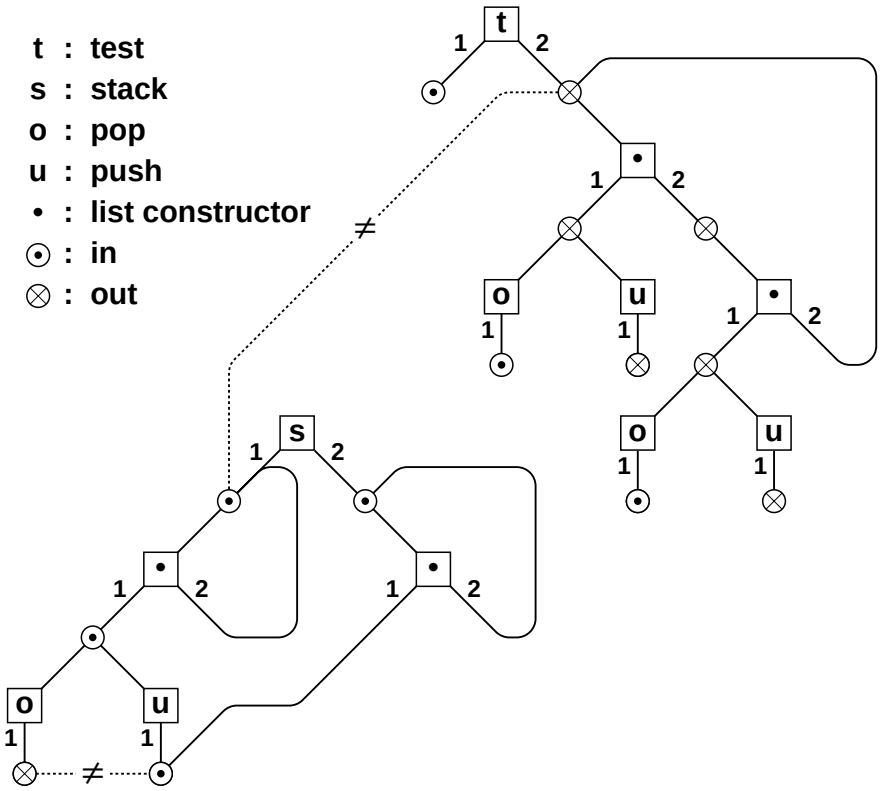

Figure 2. Mode constraints after merging

some other clause contains the body goals `test(10,S)` and `stack(S,[])` and `S` does not occur elsewhere in the clause, these two occurrences of `S` are constrained to exactly inverse modes. Hence $s_1(\langle .,1\rangle\langle \texttt{push},1\rangle)$ and $s_2(\langle .,1\rangle)$ are constrained to *in*, and $s_1(\langle .,1\rangle\langle \texttt{pop},1\rangle)$ is constrained to *out*. Figure 2 illustrates the directed graphs with these additional constraints. The two figures indicate that the operation of merging independently obtained constraints is very closely related to the unification of rational trees.

The mode of a predicate may not be uniquely determined even if a complete context is provided. However, all that we need is the information relevant to code generation. For instance, the value of $t_2(\langle f,i\rangle q)$ is not constrained at all for any $f \neq$ '.', $i \in N_f$, and $q \in P_t$, but this causes no problem. Note that the analysis requires as part of the context information the modes of top-level goals and of predefined processes such as I/O processes.

It is easy to see that a program for which the mode analysis succeeds is guaranteed to follow the conventions listed in the beginning of Section 2 and the intended meaning (Section 2.1) of the obtained mode; the analysis is sound in this sense. As an important corollary, a program for which the mode analysis succeeds is guaranteed not to fail except due to occur check.

The above analysis is based on mode constraints locally imposed by individual program clauses rather than on global dataflow analysis using iterative

abstract interpretation. This means that it is well amenable to separate compilation of large programs, though the code for unification whose mode cannot be determined locally should be supplied at link time. Alternatively, one could *declare* the modes of global predicates of modules so that more object code can be determined at compile time. In this case, mode analysis at link time acts as mode checking that checks the consistency of the declared constraints. Thus the constraint-based mode system provides us with a unified framework for mode declaration, checking and inference.

3. A Message-Oriented Implementation Technique

The mode system in Section 2 has two major applications to implementation: 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* [6]. This paper focuses on multiprocessing within one processor, though we believe that the techniques we propose here can be utilized also in parallel implementations.

Since the rest of this section deals with message-oriented scheduling, here we briefly discuss the optimization of process-oriented implementation. In process-oriented scheduling, mode information enables us to compile a body unification goal into assignment to a variable. Furthermore, in some cases we can easily guarantee that the variable has been fully dereferenced and that *no* goals are suspending on that variable [6]. That is, when starting or resuming the execution of a goal g, we can replace (some of) the output occurrences of variables $v_1, \ldots, v_n$ in g by fresh variables $u_1, \ldots, u_n$ and delay the unification between the v_i's and the u_i's until all the subgoals of g have succeeded, suspended, or been swapped out. This is an interesting application of anti-substitution [8].

3.1 Process- vs. Message-Oriented Scheduling

In conventional, process-oriented scheduling, a scheduler tries to reduce the number of process switching. Once a goal starts or resumes execution, its subgoals run as long as possible (unless they are swapped out) before another goal in a goal queue gains control. A stream connecting goals acts 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 goal sends a message to another, it does not buffer the message but transfers control to the receiver goal 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 should be ready to receive and handle the message. To this end, message-oriented scheduling tries to run the consumer of a stream ahead of its producer and to make the consumer suspend, while process-oriented scheduling would try to run the producer ahead of the consumer. 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.

3.2 A Simple Example

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 executes `Y=[A|Y1]` to buffer the datum `A`, and then executes `p(X1,Y1)` efficiently with the aid of last-call optimization [12].

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; that is, it transfers control and the datum `A` together to the consumer of the stream `Y`. The possible source of efficiency is the efficient transfer of control and data which does not use a goal queue or a data buffer. To achieve this, we implement a stream not as a list but as a special two-word cell (called a *communication cell*) pointing to the code (the resumption address) and the environment (the goal record) of the consumer goal. These two entries are initialized by executing the consumer goal of `Y` prior to the interprocess communication. A message to be transferred is placed on a hardware register called a *communication register*. We could apply the same implementation scheme to non-stream data structures also, but in this paper we choose not to do so because the overhead of creating a communication cell will pay only when it is used many times.

In fact, the execution of `p(X1,Y1)` involves *no* operation (until the next message arrives at the input stream), because

(1) the goal record can be inherited from the parent goal,
(2) the first argument recorded by the goal record continues to point to the same communication cell,
(3) the second argument also continues to point to the same communication cell, and
(4) the goal can be immediately suspended at the same instruction at the beginning of the code of `p`.

So the only things to be done in the body turn out to be

(1) to let the 'current' goal record be the one pointed to by the communication cell for the second argument, and
(2) to transfer control to the code pointed to by that communication cell.

Note here that the message `A` need not be loaded to the communication register because it had already been loaded when control was transferred to this clause. The above clause is thus compiled into a very efficient code. More complex cases will be discussed in Section 3.3.

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

3.3 Message-Oriented Scheduling in General Cases

One question that arises when generalizing the above scheme is how a compiler can distinguish between variables representing streams and those representing non-stream data. Due to space limitations, we only note that a constraint-based type system similar to the mode system in Section 2 can be employed for this purpose. The type system will infer constraints on a feasible typing function $t : P_a \rightarrow \{stream, nonstream\}$, where

(1) $t(p) = stream$ means that only the constructors of (empty and non-empty) streams can appear at p, and
(2) $t(p) = nonstream$ means that the constructors of streams cannot appear at p.

The type system proposed in [13] could be used also, though that system is based on type checking rather than type inference.

Another question is how to cope with communication that is not one-to-one. A stream may have two or more consumers or no consumer at all (one-to-many/zero communication), and a goal may consume two or more streams in various ways (many-to-one communication).

The easiest way to implement one-to-many/zero communication is to transform it into one-to-one communication. For example, when a goal commits to the following clause,

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

a dummy process is created which eats up 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 may be more efficient ways of handling these cases, but we do not consider them here since our primary concern is to implement one-to-one communication as efficiently as possible. One possibility is to use an ordinary implementation of lists for one-to-many/zero communication.

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 the receiving of a message that can be handled immediately; an example is 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 resumption addresses for handling incoming messages, and messages in one input stream are handled independently of messages in the other input stream.

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

```
omerge([A|X1],[B|Y1],Z) :- A< B |
        Z=[A|Z1], omerge(X1,[B|Y1],Z1).
omerge([A|X1],[B|Y1],Z) :- A>=B |
        Z=[B|Z1], omerge([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 `omerge` goal 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 `omerge` goal should buffer that number for later use. Buffered messages, if any, must be used first whenever a process is ready to accept new ones.

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; then they must be sent through the output stream `Z`. In either example, it is the responsibility of a receiver goal, rather than of a stream, to buffer incoming messages that cannot be handled immediately.

In general, buffering is required for those streams through which messages not ready to be handled may possibly be sent. Selective message receiving discussed above is one possible reason for this, but the need of buffering can arise without many-to-one communication also.

First, a receiver goal may suspend upon the *content* of a message when it is sent before sufficiently instantiated. This can happen in the last six clauses of `nt_node` in Program 1. In that event, subsequent messages must be buffered until the message in question has been handled.

Second, it is not always possible to run the consumer of a stream ahead of the producer; consider two goals `g1(X,Y)` and `g2(Y,X)` where `g1` consumes `X` and `g2` consumes `Y`. We must execute these goals so that no messages are lost. One solution to this example is

(1) first to make sure that `g1` buffers incoming messages from `X`,
(2) then to run `g2` until it suspends, and
(3) finally to run `g1`.

Third, a message sent by a goal g may possibly arrive at g itself or may cause another message to be sent back to g. Consider the following clause:

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

When g commits to this clause upon receiving a message $\mathtt{a}_1$, it will send two messages $\mathtt{b}_1$ and $\mathtt{c}_1$ (the suffixes are for distinguishing between different messages with the same content). Under message-oriented scheduling, however, sending $\mathtt{b}_1$ may cause another message $\mathtt{a}_2$ to arrive at g before $\mathtt{c}_1$ is sent. The goal g should therefore buffer incoming messages until $\mathtt{c}_1$ is sent, because otherwise the order of messages on the stream `Z` would be reversed. Fortunately, buffering for this reason is not needed when only one message is sent in response to each incoming message. To generalize, suppose a goal

- should send n messages in response to an incoming message and
- hasn't received any 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 goal 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 [12].

3.4 Preliminary Evaluation

We are designing an abstract machine instruction set for message-oriented scheduling. Initial performance evaluation using hand-compiled intermediate codes (which were mechanically translated into native codes of VAX11/780) was quite encouraging. First, using Program 1, we measured the processing time of 800 `search` commands given to a binary process tree with 721 non-terminal nodes, and compared the result with the numbers on a native-code, process-oriented implementation on VAX11/780, GHC/V [5]:

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. The reason seems to be that message-oriented scheduling does not perform *cons* for each message. It is noteworthy that a binary tree program in C using records, pointers, and iteration took 0.31 sec. for the same data on the same machine. Another point to note is that the message-oriented object code knows when each communication cell can be explicitly deallocated.

Second, we measured how much message-oriented scheduling 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.

Third, we tried a typical benchmark program, naive reverse. 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 [6]. Our message-oriented implementation, employing 64-bit words, ran naive reverse at 55kRPS and improved the space complexity (down to linear space). It is interesting to see how a naive reverse program runs under message-oriented scheduling.

Unfortunately, not all programs we tried were made more efficient. An 8-queens program, which made heavy use of one-to-many communication, ran about 3 times slower than on GHC/V, where we employed the naive scheme described in Section 3 to implement one-to-many communication. However, we expect that the conventional representation and scheduling schemes and the proposed ones can naturally co-exist in a single implementation, since our preliminary implementation was actually obtained by modifying GHC/V.

4. Conclusion and Related Works

We have proposed a new implementation technique of Flat GHC that contrasts sharply with previous techniques. A significance of this work is that the use of Flat GHC processes for programming dynamic, mutable data structures was shown to be more realistic as one might expect. 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 *cons*es 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. Although it could be used just as a tool for program analysis, it was designed to be used as a language construct that could be included in a subset of Flat GHC. The mode system helps both optimization and the static detection of program errors. Furthermore, it will make the use of native codes more realistic.

Our system could be understood in the framework of abstract interpretation, though we believe the current constraint-based presentation is simple and comprehensive. It is worth noting that the assumptions on our programming conventions enabled us to compute modes in a way similar to the unification of rational trees. Iterative computation of fixpoints, which is often used in abstract interpretation to capture the properties of recursive programs, is thus avoided.

Some concurrent logic languages such as Strand [2] introduce an assignment primitive (v `:=` t) instead of unification to generate bindings. However, without compile-time mode analysis, an assignment goal must still check if the left-hand side is a variable. In our framework, the assignment primitive can be considered as identical to unification except that it implicitly declares that $m(\langle \texttt{:=}, 1\rangle) = \mathit{out}$ and $m(\langle \texttt{:=}, 2\rangle) = \mathit{in}$.

Concurrent languages Doc [3] and $\mathcal{A}'\mathcal{UM}$ [14] attempt to simplify the implementation of concurrent logic languages by allowing each variable to occur only twice and letting programmers distinguish between input and output occurrences using annotations. Again, these annotations can be regarded as mode declarations, and the static or dynamic checking of whether the declarations are consistent is still needed. They may contribute to readability

and/or ease of compilation, but they are optional because they can be inferred in principle.

Acknowledgments

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

References

[1] Debray, S. A., Static Inference of Modes and Data Dependencies in Logic Programs. *ACM TOPLAS*, Vol. 11, No. 3 (1989), pp. 418–450.

[2] Foster, I. and Taylor, S., Strand: A Practical Parallel Programming Language. In *Proc. 1989 North American Conf. on Logic Programming*, Lusk, E. L. and Overbeek, R. A. (eds.), MIT Press, 1989, pp. 497–512.

[3] Hirata, M., Programming Language Doc and Its Self-Description or, $X = X$ is Considered Harmful. In *Proc. 3rd Conf. of Japan Society of Software Science and Technology*, 1986, pp. 69–72.

[4] 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.

[5] 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).

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

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

[8] Ueda, K., *Guarded Horn Clauses*. Doctoral thesis, Faculty of Engineering, Univ. of Tokyo, 1986.

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

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

[11] Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 582–591.

[12] 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.

[13] Yardeni, E. and Shapiro, E., A Type System for Logic Programs. In [7], Vol. 2, pp. 211–244.

[14] Yoshida, K. and Chikayama, T., $\mathcal{A}'\mathcal{UM}$ — A Stream-Based Concurrent Object-Oriented Language, in *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, 1988, pp. 638–649.