New Generation Computing, Vol. 13, No. 1 (1994), pp. 3–43.

Moded Flat GHC and Its Message-Oriented Implementation Technique

Kazunori Ueda†
Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

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 clauses rather than on global dataflow analysis, enabling separate analysis of program modules. The introduction of a mode system into Flat GHC effectively subsets Flat GHC; the resulting language is called *Moded Flat GHC*. Moded Flat GHC programs enjoy two important properties under certain conditions: (1) reduction of a goal clause retains the well-modedness of the clause, and (2) when execution terminates, all the variables in an initial goal clause will be bound to ground terms. Practically, the computational complexity of all-at-once mode analysis can be made almost linear with respect to the size n of the program and the complexity of the data structures used, and the complexity of separate analysis is higher only by $O(\log n)$ times. Mode analysis provides useful information for debugging as well.

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. We expect that the

† Author's current address: Department of Information and Computer Science, Waseda University, 4-1, Okubo 3-chome, Shinjuku-ku, Tokyo 169, Japan. E-mail: ueda@cfi.waseda.ac.jp.

proposed technique will expand the application areas of concurrent logic languages.

Keywords: Concurrent logic programming, Moded Flat GHC, Static mode system, Constraint-based program analysis, Message-oriented implementation.

1. Introduction

Guarded Horn Clauses (GHC) [16][17][19] is a simple concurrent logic language born from the research on parallelism in logic programming. Its subset, Flat GHC [19] [21], 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 (Flat) GHC and concurrent logic programming are referred to [14], [15], [18] and [19].

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 individual body goals, 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 (also referred to as *program clauses*) can be regarded as a conditional rewrite rule of goals in a goal clause. It is the guard part of a clause that specifies what substitution should be observed before the rewriting. A substitution is generated by spawning a unification body goal whose behavior is language-defined.

Flat GHC is different from GHC in that guard goals are restricted to

- unification goals and
- calls to *test predicates*, namely predicates defined by clauses with empty bodies.

However, this restriction is not really restrictive as a process description language [20]. We have found that it is more natural to distinguish between test predicates that define rewrite conditions and non-test predicates that define processes; they have very different characteristics. Thus, in this paper, we make a clearer distinction by assuming that predicates (other than the predefined predicate '=' for unification) are divided into:

- test predicates which are defined by clauses with empty bodies and can be called only from clause guards, and
- non-test predicates which are defined by clauses with possibly non-empty bodies and can be called only from clause bodies.

Concurrent logic languages employ the notion of streams, implemented as lists, for interprocess communication. Unlike most concurrent languages,

```
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(K,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.

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. A mode system will be introduced into Flat GHC for this purpose. The resulting language seems to be an important subset of Flat GHC and hence will be called *Moded Flat GHC*. This topic will be discussed in detail in Section 2.

Another motivation comes from our desire to expand the application areas of concurrent logic languages. So far, concurrent logic languages have been used mainly 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 those described in [9] and [10], 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, which will be discussed in Section 3.

This paper is a revised and extended version of an earlier paper [22]. In particular, the mode system first proposed in [22] is described here in much more detail, with proofs of important properties and a study of the cost of mode analysis.

2. Moded Flat GHC

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 for Flat GHC programs that generalizes our previous system [11] (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 [4]. The mode system is very different also from that of PARLOG [2] and of DEC-10 Prolog [5], as will be discussed in Section 2.7.

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. In other words, a goal clause, whether it is the one given initially or is derived by repeated, concurrent reductions, has exactly one output occurrence for each variable in the clause.

(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. (We call those symbols *ancestor* symbols.) 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 predicate '`=`' is said to be *overloaded* in this sense.

The introduction of a mode system into Flat GHC is effectively the subsetting of Flat GHC; the resulting language is called *Moded Flat GHC*. So a question arises as to whether this subsetting seriously affects 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 unification body goals.

On Convention (1), we must note that 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. When some p_i finds that the 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 Convention (1) is to use an n-ary arbiter process which the p_i's can ask (via distinct variables) to bind v to c. Such an arbiter process can be implemented efficiently by applying the message-oriented implementation technique (described in Section 3) to many-to-one non-stream communication, though the detail is beyond the scope of this paper.

We have seldom seen programs excluded by Convention (2). However, an example can be contrived from Program 1 by replacing all the occurrences of the terms `search(K,V)` and `update(K,V)` by the three-element lists `[search,K,V]` and `[update,K,V]`, respectively. With this change, the command names `search` and `update` are not ancestor symbols of `K` and `V` any more, and the modes of the arguments of the two commands become indistinguishable. Thus Convention (2) encourages programmers to use meaningful function symbols only and discourages the excessive use of list structures.

2.1 The Mode System

As usual, we first fix the vocabulary with which programs are written and executed. Let *Pred*, *Fun*, and *Var* be disjoint sets of predicate, function, and variable symbols, respectively (we do not distinguish between constant and function symbols). Let *Atom* be the set of atoms, and *Term* the set of terms, over *Pred*, *Fun*, and *Var*. 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_{Term} (for terms) and P_{Atom} (for atoms) using disjoint union as follows:

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

An element of P_{Term} can be written as a string $\langle f_1, j_1 \rangle \ldots \langle f_n, j_n \rangle$, and an element of P_{Atom} can be written as $\langle p, i \rangle q$, where $q \in P_{Term}$. (As usual, concatenation of strings and string elements is represented by juxtaposition.)

The empty sequence in P_{Term} will be written as ϵ. Paths are intended to specify variables or function symbols constructing terms, atoms and possible instances thereof. That is, with each term t we associate a function $\widetilde{t} : P_{Term} \to Var \cup Fun \cup \{\bot\}$ ($\bot$ standing for 'undefined') for accessing its constituent symbols by means of paths, which is defined as follows:

$$\begin{cases} \widetilde{t}(\epsilon) = \begin{cases} f, & \text{if } t \text{ is of the form } f(t_1, \ldots, t_n); \\ t, & \text{otherwise (i.e., if } t \text{ is a variable);} \end{cases} \\ \widetilde{t}(\langle f, j \rangle q) = \begin{cases} \widetilde{t_j}(q), & \text{if } t \text{ is of the form } f(t_1, \ldots, t_n) \text{ and } j \in N_f; \\ \bot, & \text{otherwise.} \end{cases} \end{cases}$$

The function for accessing constituent symbols of an atom is defined similarly. In our setting, each element of a path, being of a dependent type, is indexed explicitly by a predicate or function symbol. This is because we must be able to specify constituent symbols of all possible instances of terms or atoms.

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

$$M = P_{Atom} \to \{in, out\},$$

where we assume $in \neq out$ for the codomain. This means that a mode assigns either of *in* or *out* to every possible path of every possible instance of every possible goal.

Some miscellaneous definitions for notational convenience:

- For a mode $m \in M$ and a path $p \in P_{Atom}$, m/p denotes a function, called a *submode* of m, from P_{Term} to $\{in, out\}$, such that

$$\forall q \in P_{Term}\big((m/p)(q) = m(pq)\big).$$

- The *inverse* of a mode m, denoted $\overline{m}$, is defined as a function such that

$$\forall p \in P_{Atom}\big(\overline{m}(p) \neq m(p)\big).$$

  Submodes of a submode and the inverse of a submode are defined similarly.
- By IN we denote a submode such that

$$\forall q \in P_{Term}\big(IN(q) = in\big),$$

  and by OUT we denote $\overline{IN}$.

2.2 Mode Analysis

The purpose of mode analysis is to find a mode $m \in M$ that satisfies all the constraints (listed below) syntactically imposed by the pair of a program P (a set of program clauses) and a goal clause G to activate P. When such a mode exists, the pair of P and G is said to be *well-moded*, and the mode m is called a *well-moding* of the pair. Well-modedness is defined in the same

manner for each of P and G, or more generally, for a set of (program and/or goal) clauses which we will call *program fragments* henceforth.

A program fragment may fail to have any well-moding, in which case it is considered to be non-well-moded. This happens when, for some path p, $m(p)$ is constrained to both *in* and *out*, or to $\overline{m}(p)$. A well-moded program fragment usually has many well-modings. This is because the paths of a goal which are not examined or instantiated can be given either mode values. So we choose to express modes implicitly in terms of mode constraints. Constraint-based representation requires an appropriate constraint satisfaction algorithm that reduces a set of constraints and checks if the set is consistent or not. However, the design of a constraint satisfaction algorithm is a separate issue and will be discussed in Sections 2.4 and 2.5.

To simplify the analysis, we assume that programs and initial goal clauses (from which execution commences) satisfy the following *normal form* conditions:

(i) No unification goals exist in guards.
(ii) The multiset of unification goals in the body of a clause is of the form $v_1 = t_1, \ldots, v_n = t_n$, where
- the v_i's are distinct variables occurring in the head of the clause,
- the v_i's do not occur in any of the t_j's 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. A clause not in a normal form is first normalized in the way described in Appendix. For goal clauses, Condition (i) is simply irrelevant, and Condition (ii) means that they contain no unification goals. Because of Condition (i), by unification goals we always mean unification *body* goals henceforth. Strictly speaking, Condition (ii) is not essential for the analysis, but it may turn a non-well-moded program into a well-moded one without changing its behavior. Note that a goal clause obtained as a result of reduction may not enjoy Condition (ii), because the reduction of a non-unification goal may spawn unification goals.

Furthermore, to cope with the overloading of the predicate '=', we assume that all its occurrences in a program are virtually indexed as '$=_1$', '$=_2$', ... This treatment is necessary because it is very common in a Flat GHC program that different unification body goals have different modes. Fortunately, this treatment does not complicate the analysis because the predicate '=' has a fixed meaning given by the language and imposes a strong constraint on the possible modes of its arguments (see Constraint (BU) below). Note that unification goals spawned upon reduction inherit the indices of the unification predicates in the program.

Now we list the constraints* imposed by each clause C in a program

* The names of the constraints (HF), (HV), (GV), (BU), (BF), and (BV) stand for *head function*, *head variable*, *guard variable*, *body unification*, *body function*, and *body variable*, respectively.

fragment. Recall that C is either of the form h :- G | B (guarded clause) or the form :- B (goal clause), where G and B are multisets of atoms. While we apply all of these constraints to clauses defining *non-test* predicates, we apply only (HF), (HV), and (GV) to clauses defining test predicates.** To goal clauses, only (BU), (BF), and (BV) are applicable because they have no guards.

(HF) $\forall p \in P_{Atom}\big(\widetilde{h}(p) \in Fun \;\Rightarrow\; m(p) = in\big)$
(if the symbol at p in h is a function symbol, $m(p) = in$),

(HV) $\forall p, p' \in P_{Atom}\big(\widetilde{h}(p) \in Var \wedge p \neq p' \wedge \widetilde{h}(p) = \widetilde{h}(p') \;\Rightarrow\; m/p = IN\big)$
(if the symbol at p in h is a variable occurring elsewhere in h, then $\forall q \in P_{Term}\big(m(pq) = in\big)$, namely $m/p = IN$),

(GV) $\forall p, p' \in P_{Atom}\, \forall a \in G\big(\widetilde{h}(p) \in Var \wedge \widetilde{h}(p) = \widetilde{a}(p')$
$\Rightarrow\; \forall q \in P_{Term}\big(m(p'q) = in \;\Rightarrow\; m(pq) = in\big)\big)$
(if the same variable occurs both at p in h and at p' in G, then $\forall q \in P_{Term}\big(m(p'q) = in \;\Rightarrow\; m(pq) = in\big)$),

(BU) $\forall k > 0\, \forall t_1, t_2 \in Term\big((t_1 =_k t_2) \in B \;\Rightarrow\; m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle}\big)$
(the two arguments of a unification body goal have exactly inverse submodes),

(BF) $\forall p \in P_{Atom} \forall a \in B\big(\widetilde{a}(p) \in Fun \;\Rightarrow\; m(p) = in\big)$
(if the symbol at p in a body goal is a function symbol, $m(p) = in$),

(BV) Let v be a variable occurring exactly $n\,(\geq 1)$ times in h and B at $p_1, \ldots, p_n$, of which the occurrences in h are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

$$\begin{cases} \mathcal{R}\big(\{m/p_1, \ldots, m/p_n\}\big), & \text{if } k = 0; \\ \mathcal{R}\big(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}\big), & \text{if } k > 0; \end{cases}$$

where the unary predicate $\mathcal{R}$ over finite *multisets* of submodes is a 'cooperativeness' relation defined as

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{Term}\, \exists s \in S\big(s(q) = out \wedge \forall s' \in S \backslash \{s\}\, \big(s'(q) = in\big)\big). \;\blacksquare$$

Note that Constraint (BV) ignores the second and the subsequent occurrences of v in h and any of the occurrences in G. The other occurrences of v are called *channel occurrences*. Note also that s can depend on q in the above definition of $\mathcal{R}$. Intuitively, Constraint (BV) means that each function symbol occurring in a possible instance of v will be determined by exactly one of the channel occurrences of v.

The relation $\mathcal{R}$ enjoys the following properties:

(P1) $\mathcal{R}\big(\{s\}\big) \;\Leftrightarrow\; s = OUT,$

(P2) $\mathcal{R}\big(\{s_1, s_2\}\big) \;\Leftrightarrow\; s_1 = \overline{s_2},$

(P3) $\mathcal{R}\big(\{IN\} \cup S\big) \;\Leftrightarrow\; \mathcal{R}\big(S\big),$

** Since (BU) and (BF) are simply not applicable to test predicates, the point here is that (BV) is not applied to test predicates.

$$\text{(P4)} \qquad \mathcal{R}(\{OUT\} \cup S) \;\Leftrightarrow\; \forall s' \in S(s' = IN),$$

$$\text{(P5)} \qquad \mathcal{R}(\{s, s\} \cup S) \;\Leftrightarrow\; s = IN \wedge \mathcal{R}(S),$$

$$\text{(P6)} \qquad \mathcal{R}(\{\overline{s}, s\} \cup S) \;\Leftrightarrow\; \forall s' \in S(s' = IN),$$

$$\text{(P7)} \quad \mathcal{R}(\{\overline{s}\} \cup S_1) \wedge \mathcal{R}(\{s\} \cup S_2) \;\Rightarrow\; \mathcal{R}(S_1 \cup S_2),$$

$$\text{(P8)} \qquad \mathcal{R}(\textstyle\bigcup_{1 \le i \le n}\{s_i\}) \;\Rightarrow\; \mathcal{R}(\bigcup_{1 \le i \le n}\{s_i/q\}), \quad q \in P_{Term}.$$

Proofs are all straightforward. Property (P7) is reminiscent of Robinson's resolution principle.

Properties (P1) and (P2) say that Constraint (BV) becomes much simpler when v has at most two channel occurrences. If it has exactly two channel occurrences at p_1 and p_2, which is usually the case, Constraint (BV) is equivalent to $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$, depending on whether one of the occurrences is in the head or the both occur in the body. This means that the variable v is used for one-to-one communication. When v has only one channel occurrence at p, Constraint (BV) is equivalent to $m/p = IN$ or $m/p = OUT$, depending on whether the occurrence is in the head or the body.

2.3 Rationale of the Constraints

Rationale of the six mode constraints are appropriate here. We consider non-test predicates first. Test predicates will be considered later, since they are quite different from non-test predicates.

In concurrent logic programming, a body goal (or more precisely, the process implemented by a body goal), defined by a non-test predicate, can be considered an information processing device with inlets and outlets of information that we call *terminals*. A variable is considered a one-to-n ($n \ge 0$) communication channel connecting its occurrences, and each channel occurrence of a variable is considered to be plugged into a terminal of a goal. A variable occurring both in the head and in the body of a program clause is considered a channel that conveys information between a goal (which the head matches) and its subgoals. A function symbol is considered an unconnected plug that acts as the source or the absorber of atomic information, depending on whether it occurs in the body or the head. While channels and terminals of electric devices usually have array structures, those in our setting have nested structures. That is, a variable that connects the terminals at p_1, ..., p_n also connects the terminals at $p_1 q$, ..., $p_n q$, for all $q \in P_{Term}$.

A terminal of a goal always has its counterpart. The counterpart of a terminal at p on the caller side of a non-unification goal is the one at the same path on the callee side, and the counterpart of a terminal at $\langle =_k, 1\rangle q$ in the first argument of a unification goal is the one at $\langle =_k, 2\rangle q$ in the second argument. Reduction of a goal is considered the removal of the pairs of corresponding terminals whose connection has been established.

The mode constraints are concerned with the direction of information flow (i) in channels and (ii) at terminals. The two underlying principles are:

(i) When a channel (a variable) connects n terminals of which at most one

is in the head, exactly one of the terminals is the outlet of information and the others are inlets. In other words, when a variable has n channel occurrences in a clause, it is used for one-to-$(n-1)$ communication.

(ii) Of the two corresponding terminals of a goal, exactly one is the outlet of information and the other is an inlet.

Constraint (BV) comes from Principle (i). An input (output) occurrence of a variable in the head of a clause is considered an outlet (inlet) of information from inside the clause, respectively, and this is why we invert the mode of the clause head in considering Constraint (BV). Constraint (BV) takes into account only one of the occurrences of v in the head. 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 the variable occurs also in the body and conveys information to the body goals.

Constraints (HF) and (HV) come from Principle (ii). For non-unification goals, $m(p)$ should be constrained to in when some clause may examine the value of the path p, because the examination must be done at the *outlet* of information on the *callee* side of a goal. Hence Constraints (HF) and (HV). The strong constraint imposed by Constraint (HV) is due to the semantics of Flat GHC: when a variable occurs twice or more in a clause head, these occurrences must receive identical, though arbitrary, terms from the caller.

Constraint (BU) is exactly the application of Principle (ii) to unification body goals. Any value fed through some path $\langle =_k, i \rangle q$ in one of its arguments will come out through the corresponding path $\langle =_k, 3-i \rangle q$ in the other argument.

Constraint (BF) also comes from Principle (ii). A non-variable symbol on the caller side of a goal must appear only at the inlet of information, because the information will go out from the corresponding outlet.

Now we consider Constraint (GV) on guard goals, which are defined by test predicates. The idea here is that the paths of a goal g whose value may be examined by the guard goals of a clause called by g are constrained to in. An alternative to the consequent of Constraint (GV) would be to use a stronger form $m/p' = m/p$ instead of $\forall q \in P_{Term}\big(m(p'q) = in \Rightarrow m(pq) = in\big)$. However, test predicates should be generic in terms of modes, and our choice avoids the unnecessary propagation of constraints from non-test predicates to test predicates.

Almost all implementations of Flat GHC restricts guard goals to calls to *predefined* test predicates. Those predicates are regarded as defined virtually by a set of clauses like

```
6>5 :- true | true,
```

for which the mode constraints are considered.

When guard goals exist, Constraint (BV) could be weakened. Suppose a variable v occurs both in the head of a clause C and its guard goals. Then the guard goals may guarantee that, when a goal commits to C, the variable v has been bound to a non-variable term. For instance, when C is of the form

```
p(X, ...) :- X>0 | ... ,
```

`X` will have been bound to a constant upon commitment. In such a case, constraints on the paths $\langle \mathtt{p}, 1\rangle q$ ($q \in P_{Term} \setminus \{\epsilon\}$) are superfluous, and hence the quantified variable in the definition of $\mathcal{R}$ need not range over those paths.

In general, let T_v be any set of terms which includes all the possible values of v upon commitment to C, and P_{T_v} be the set

$$P_{Term} \setminus \{\, p \in P_{Term} \mid \forall t \in T_v (\widetilde{t}(p) = \bot) \,\}.$$

Informally, P_{T_v} excludes (some of) the irrelevant paths. Using P_{T_v}, we can slightly change the definition of $\mathcal{R}$ to

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{T_v}\ \exists s \in S \big(s(q) = out \ \wedge\ \forall s' \in S \setminus \{s\}\ \big(s'(q) = in\big)\big).$$

Note that P_{T_v} is equal to P_{Term} if $T_v = Term$, namely if no information is available on the possible values of v. However, if it can be inferred that a set of constants can be used as T_v (as in the example of `X>0`), P_{T_v} becomes a singleton set $\{\epsilon\}$. By finding a small T_v, a mode analyzer can avoid imposing unnecessary mode constraints and rejecting meaningful programs. However, the basic properties of well-moded programs we will discuss in Section 2.6 do not depend on what T_v is used by the analyzer, as long as the analyzer uses a correct one.

2.4 Managing and Solving Mode Constraints

This section shows how the mode constraints of a simple program can be represented and manipulated using graphs and unification. General cases not covered by the example will be addressed in the next section.

Let us consider Program 2, a simple stack program and its driver.*** The predicate `terminate` is used for processing remaining stack elements when the stack is being terminated. In this particular case, `terminate` just discards all the elements because the stack is used for storing ground terms (integers). However, when the stack stores streams connected to some other processes, the streams in the stack should be closed upon termination of the stack so that the receiver processes can terminate themselves.

The program uses three predefined predicates: `=:=`, `=\=`, and `subtract`, so we need to know the mode constraints they impose. We assume that they accept integers, but not integer expressions. Then the mode constraints they impose are:

$$m(\langle \mathtt{=:=}, i\rangle) = m(\langle \mathtt{=\backslash=}, i\rangle) = in, \quad \text{for } i = 1, 2,$$
$$m(\langle \mathtt{subtract}, 1\rangle) = m(\langle \mathtt{subtract}, 2\rangle) = in, \quad m(\langle \mathtt{subtract}, 3\rangle) = out.$$

*** The second argument of `stack` for storing elements does not use list constructors. This is because we need to distinguish between constructors for streams and those for non-stream data structures later in Section 3. Which list constructors are used for the second argument does not affect our mode analysis at all.

```
drive(M,S) :- M=:=0 | S=1[].                                          (i)
drive(M,S) :- M=\=0 |
    S=2[push(M),pop(N)|S1], subtract(N,1,N1), drive(N1,S1).           (ii)

stack([],          D       ) :- true | terminate(D).                  (iii)
stack([push(X)|S],D        ) :- true | stack(S,p(X,D)).               (iv)
stack([pop(X)|S], p(Y,D1)) :- true | X=3Y, stack(S,D1).               (v)

terminate(D) :- true | true.                                          (vi)
```

Program 2. A stack program and its driver.

Let '.' denote the function symbol of a non-empty list. The constraints we can obtain directly from the predicate `drive` are:

(1) $\forall q \in P_{Term}\big(m(\langle \texttt{=:=},1\rangle q) = in \;\Rightarrow\; m(\langle \texttt{drive},1\rangle q) = in\big)$ by (GV) applied to `M` in (i),

(2) $m/\langle \texttt{=}_1,1\rangle = \overline{m/\langle \texttt{=}_1,2\rangle}$ by (BU) applied to $\texttt{=}_1$ in (i),

(3) $m(\langle \texttt{=}_1,2\rangle) = in$ by (BF) applied to `[]` in (i),

(4) $m/\langle \texttt{drive},1\rangle = IN$ by (BV) applied to `M` in (i),

(5) $m/\langle \texttt{drive},2\rangle = m/\langle \texttt{=}_1,1\rangle$ by (BV) applied to `S` in (i),

(6) $\forall q \in P_{Term}\big(m(\langle \texttt{=\=},1\rangle q) = in \;\Rightarrow\; m(\langle \texttt{drive},1\rangle q) = in\big)$ by (GV) applied to `M` in (ii),

(7) $m/\langle \texttt{=}_2,1\rangle = \overline{m/\langle \texttt{=}_2,2\rangle}$ by (BU) applied to $\texttt{=}_2$ in (ii),

(8) $m(\langle \texttt{=}_2,2\rangle) = in$ by (BF) applied to the outer . in (ii),

(9) $m(\langle \texttt{=}_2,2\rangle\langle .,1\rangle) = in$ by (BF) applied to `push` in (ii),

(10) $m(\langle \texttt{=}_2,2\rangle\langle .,2\rangle) = in$ by (BF) applied to the inner . in (ii),

(11) $m(\langle \texttt{=}_2,2\rangle\langle .,2\rangle\langle .,1\rangle) = in$ by (BF) applied to `pop` in (ii),

(12) $m(\langle \texttt{subtract},2\rangle) = in$ by (BF) applied to `1` in (ii),

(13) $m/\langle \texttt{drive},1\rangle = m/\langle \texttt{=}_2,2\rangle\langle .,1\rangle\langle \texttt{push},1\rangle$ by (BV) applied to `M` in (ii),

(14) $m/\langle \texttt{drive},2\rangle = m/\langle \texttt{=}_2,1\rangle$ by (BV) applied to `S` in (ii),

(15) $m/\langle \texttt{=}_2,2\rangle\langle .,2\rangle\langle .,1\rangle\langle \texttt{pop},1\rangle = \overline{m/\langle \texttt{subtract},1\rangle}$ by (BV) applied to `N` in (ii),

(16) $m/\langle \texttt{subtract},3\rangle = \overline{m/\langle \texttt{drive},1\rangle}$ by (BV) applied to `N1` in (ii),

(17) $m/\langle \texttt{=}_2,2\rangle\langle .,2\rangle\langle .,2\rangle = \overline{m/\langle \texttt{drive},2\rangle}$ by (BV) applied to `S1` in (ii).

By eliminating constraints on the predicate '`=`' and other predefined predicates, we obtain the following constraints on $m/\langle \texttt{drive},1\rangle$ and $m/\langle \texttt{drive},2\rangle$:

$m/\langle \texttt{drive},1\rangle = IN$, $m(\langle \texttt{drive},2\rangle) = out$,
$m(\langle \texttt{drive},2\rangle\langle .,1\rangle) = out$, $m/\langle \texttt{drive},2\rangle\langle .,1\rangle\langle \texttt{push},1\rangle = \overline{m/\langle \texttt{drive},1\rangle}$,

$m(\langle \texttt{drive},2\rangle\langle .,2\rangle) = out,\ m(\langle \texttt{drive},2\rangle\langle .,2\rangle\langle .,1\rangle) = out,$
$m(\langle \texttt{drive},2\rangle\langle .,2\rangle\langle .,1\rangle\langle \texttt{pop},1\rangle) = in,$
$m/\langle \texttt{drive},2\rangle\langle .,2\rangle\langle .,2\rangle = m/\langle \texttt{drive},2\rangle.$

For instance, $m(\langle \texttt{drive},2\rangle) = out$ follows from (2), (3) and (5); $m/\langle \texttt{drive}, 2\rangle\langle .,2\rangle\langle .,2\rangle = m/\langle \texttt{drive},2\rangle$ follows from (7), (14) and (17).

If the system knows that `M=:=0` and `M=\=0` will not succeed for non-constant values of `M`, the weakened version of Constraint (BV) (Section 2.3) gives us $m(\langle \texttt{drive},1\rangle) = in$ instead of $m/\langle \texttt{drive},1\rangle = IN$ in (4), which is exactly the same as the constraint deducible from Constraint (GV).

Similarly, we can obtain the following constraints from `stack`:

$m(\langle \texttt{stack},1\rangle) = in,\ m(\langle \texttt{stack},1\rangle\langle .,1\rangle) = in,$
$m/\langle \texttt{stack},1\rangle\langle .,1\rangle\langle \texttt{push},1\rangle = m/\langle \texttt{stack},2\rangle\langle \texttt{p},1\rangle,$
$m/\langle \texttt{stack},1\rangle\langle .,1\rangle\langle \texttt{pop},1\rangle = \overline{m/\langle \texttt{stack},2\rangle\langle \texttt{p},1\rangle},$
$m/\langle \texttt{stack},1\rangle\langle .,2\rangle = m/\langle \texttt{stack},1\rangle,$
$m(\langle \texttt{stack},2\rangle) = in,\ m/\langle \texttt{stack},2\rangle\langle \texttt{p},2\rangle = m/\langle \texttt{stack},2\rangle,$
$m/\langle \texttt{terminate},1\rangle = m/\langle \texttt{stack},2\rangle.$

The conjunction of all these constraints forms a kind of feature structure that can be represented as a (directed, possibly cyclic, and possibly unconnected) feature graph [12] as illustrated in Figure 1. We call it a *mode graph*.

In our setting, each feature of the resulting structure is the pair of a predicate or function symbol and its argument position, which is shown in Figure 1 by the label on an edge. A sequence of features forms a path both in the sense of our mode system and in the graph-theoretic sense. A node is possibly labeled with a mode value to which any paths terminating with that node is constrained.

An edge with a bullet is a 'mode inversion' edge: When a path passes an odd number of bulleted edges, that path is said to be *inverted*, and the mode value of the path should be understood to be inverted. Thus the number of bulleted edges on a path determines the *polarity* of the path. A universally quantified constraint of the form $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$ is represented by a shared node with two (or more) incoming paths with possibly different polarities. When the polarities of the two incoming paths are different, the shared node stands for complementary submodes; otherwise the node stands for identical submodes.

A universally quantified constraint of the form $m/p = IN$ or $m/p = OUT$ is represented using a node labeled 'ground'. A node labeled 'ground' indicates that all paths ending with or passing beyond that node are constrained to *in* or *out*, depending on whether the number of bulleted edges passed before reaching the node is even or odd.

A mode m and its mode graph (as in Figure 1) can be regarded as having omitted the information on the mode values of whole atoms (rather than their arguments) which are always *in*; predicate symbols of goals are always determined by the callers.

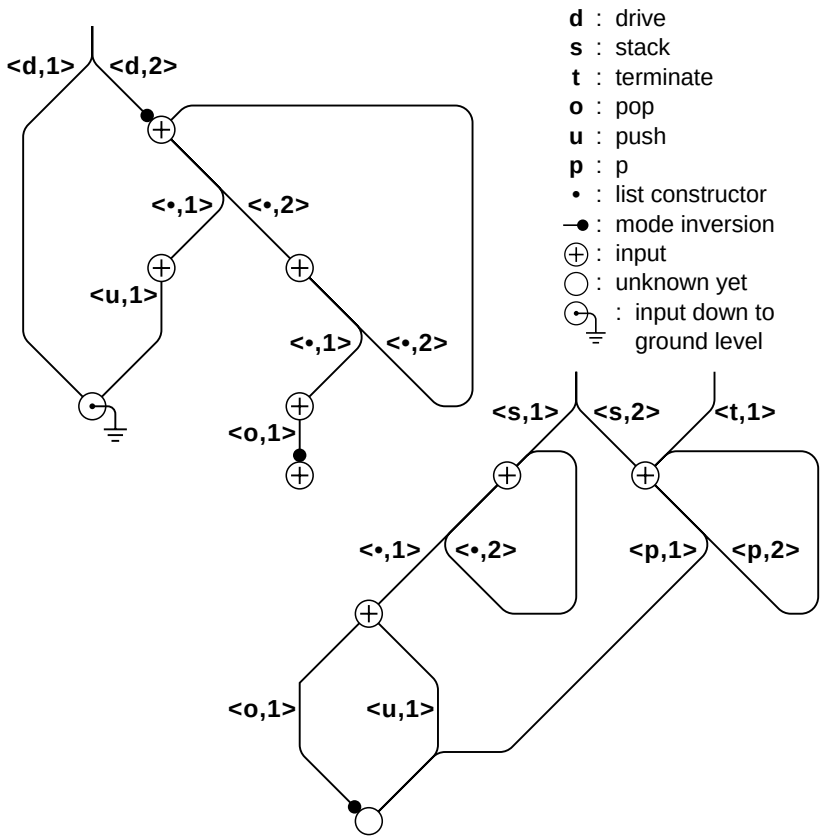

Figure 1. Mode constraints obtained separately from `drive` and `stack`.

Given the mode graph of m and a path p, the node at p and all nodes and edges reachable from that node forms a subgraph representing the submode m/p. We call it the *submode graph* of m/p, and the node at p the *root* of the submode graph.

As the mode of `drive` in Figure 1 indicates, our framework does not necessarily require that different elements of a stream must have the same mode constraint.

The concrete values of $m(\langle \texttt{stack},1\rangle\langle .,1\rangle\langle \texttt{push},1\rangle)$, $m(\langle \texttt{stack},1\rangle\langle .,1\rangle \langle \texttt{pop},1\rangle)$, and $m(\langle \texttt{stack},2\rangle\langle \texttt{p},1\rangle)$ cannot be determined solely by `stack`; they are determined only by supplying a context in which the predicate `stack` is used or by giving the definition of `terminate`. For example, if a goal clause or some other program clause contains the body goals `drive(10,S)` and `stack(S,none)` and `S` does not occur elsewhere in the clause, the two submodes, $m/\langle \texttt{drive},2\rangle$ and $m/\langle \texttt{stack},1\rangle$, are constrained to exactly inverse submodes. Hence $m(\langle \texttt{stack},1\rangle\langle .,1\rangle\langle \texttt{push},1\rangle)$ and $m(\langle \texttt{stack},2\rangle\langle \texttt{p},1\rangle)$ are constrained to *in*, and $m(\langle \texttt{stack},1\rangle\langle .,1\rangle\langle \texttt{pop},1\rangle)$ is constrained to *out*. Figure 2 illustrates the mode graph with these additional constraints.

The predicate `terminate` imposes a strong mode constraint by Constraint (BV):

$$m/\langle \texttt{terminate},1\rangle = IN.$$

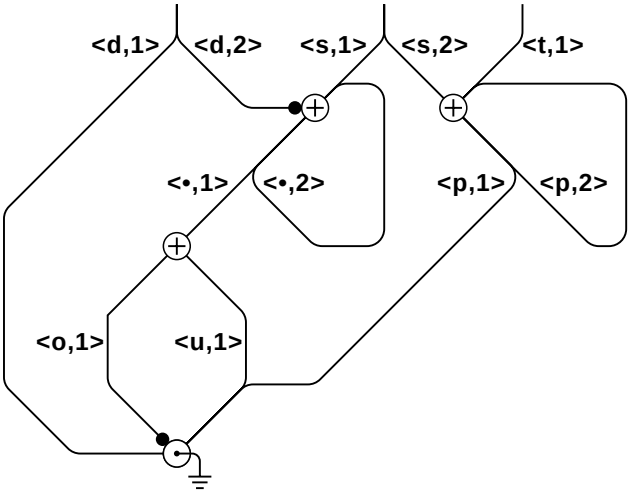

Figure 2. Mode constraints after merging a new constraint.

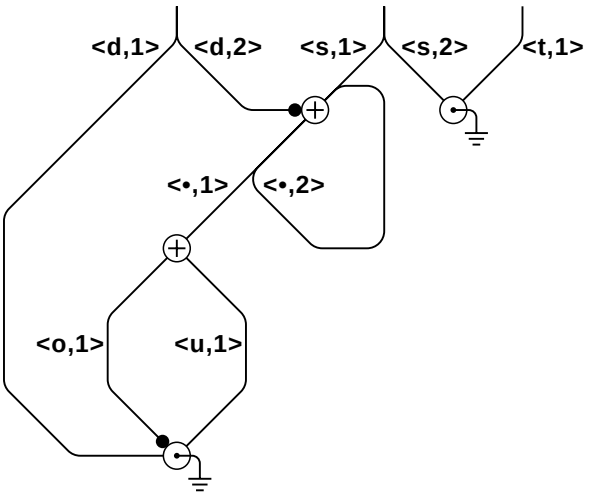

Figure 3. Mode constraints after merging another constraint.

Merging this constraint into the mode graph in Figure 2 results in the graph in Figure 3.

The mode of a predicate may not be uniquely determined even when a complete context is provided. However, all that we need is the information relevant to code generation. For instance, the value of $m(\langle \texttt{drive}, 2\rangle\langle f, i\rangle q)$ is not constrained at all for any function symbol f other than '.', $i \in N_f$, and for any $q \in P_{Term}$, but this causes no problem.

Predicates defined by mutual recursion can be handled exactly in the same way and without any additional mechanism. The path-based mode system can deal with streams of streams also. Consider the clause

```
create_stacks([S|Ss]) :- true |
        stack(S,none), create_stacks(Ss).
```

From this clause, we obtain the constraints $m/\langle \texttt{create\_stacks}, 1\rangle\langle ., 1\rangle = m/\langle \texttt{stack}, 1\rangle$ and $m/\langle \texttt{create\_stacks}, 1\rangle\langle ., 2\rangle = m/\langle \texttt{create\_stacks}, 1\rangle$. Hence all the elements of a stream occurring as the argument of `create_stacks` have a common submode which is identical to the submode of a stream occurring as the first argument of `stack`.

When a well-moded program is activated by a top-level goal clause, it must be checked first if the mode constraints imposed by the goal clause are consistent with the mode of the program.

2.5 Cost of the Analysis

In this section, we discuss the time complexity of mode analysis, starting with simple yet important cases.

For the time being, we focus on non-test predicates and assume the following:

(i) no clause has guard goals, and
(ii) no clause contains three or more channel occurrences of a variable.

These assumptions guarantee that the mode constraints can be expressed as a set of *primitive constraints* of the following forms for some p, p_1 and p_2:

(1) $m(p) = in$ (or $m(p) = out$),

(2) $m/p_1 = m/p_2$ (or $m/p_1 = \overline{m/p_2}$),

(3) $m/p = IN$ (or $m/p = OUT$).

The first form can be represented in a mode graph by labeling the node at p as in. The second can be represented by the sharing of a node. The third can be represented by labeling the node at p as 'ground'. The forms in the parentheses can be represented using inverted paths. Hence, mode graphs explained in Section 2.4 suffice for the class of programs we are considering. Note also that each of these primitive constraints can itself be represented as a very simple mode graph. Figures 1 to 3 can be regarded as the results of merging many simple mode graphs representing primitive constraints.

2.5.1 Merging Two Mode Graphs

As Figures 1 to 3 indicate, the merging of two sets of mode constraints represented as mode graphs is very close to the unification of feature graphs [12] which is in turn very close to the unification of rational terms [8].****

We have a difference, however: The mode values of corresponding paths must be unified as well as the graph structures, taking into account both the polarities of the paths and the labels of the nodes at the ends of the paths.

**** A rational term is a possibly infinite term which has a finite set of subterms [3].

In particular, nodes labeled 'ground' must be dealt with properly. First, unification of two one-node submode graphs with 'ground' nodes simply succeeds or fails, depending on the polarities of the paths leading to those 'ground' nodes. Second, a one-node submode graph $\mathcal{G}_1$ with a 'ground' node and a submode graph $\mathcal{G}_2$ whose root is not labeled 'ground' can be unified by (re-)labeling all the nodes in $\mathcal{G}_2$ as 'ground', where it is of course necessary to check if the previous labeling (if any) of those nodes is consistent with the new labeling. Outgoing arcs from a node newly labeled as 'ground' can be removed. A node that has thus lost all its incoming arcs becomes a garbage node, whose outgoing arcs can be removed in the same manner. Note that some of the nodes in $\mathcal{G}_2$ may be shared by other paths; such nodes do not become garbage and must be (re-)labeled properly.

On the time complexity, we first note that a practical algorithm for the unification of two feature graphs achieves almost linear time complexity with respect to the size l of the graphs to be unified [1][8]. That is, the time complexity is $\mathrm{O}(l \cdot \alpha(l))$, where α is the inverse of the Ackermann function. The unification algorithm represents the sharing of a node using invisible pointers (which are implicit in Figures 1–3) managed by the union-find algorithm. Access to a shared node may involve the dereferencing of those invisible pointers, which is why the algorithm is almost but not precisely linear.

Therefore, if we focus on the merging of two graph structures and ignore the handling of node values, the cost is almost linear with respect to the size of the graphs. The size of the graphs to be merged depends on the *complexity* of the data structures used in the program (in terms of the size of their recursive definition) but not on the *size* of the data structures formed. For instance, streams of streams are a more complex data structure than streams of constants.

The size l of the mode graph depends also on the size of the program, because larger programs will use more predicate symbols. However, the complexity of the most complex data structure in a program may not necessarily grow as the size of the program grows. We will thus restate the time complexity of the same algorithm using more parameters:

(1) the numbers w_1, w_2 of top-level features (i.e., features of the form $\langle p, i \rangle$, where $p \in \mathit{Pred}$ and $i \in n_p$) occurring in the two mode graphs,
(2) the number c of top-level features occurring in the both mode graphs, and
(3) the maximum size d of the submode graphs whose roots are at the paths of the form $\langle p, i \rangle$ ($p \in \mathit{Pred}$, $i \in n_p$).

Note that w_i reflects the size of the program and d reflects the maximum complexity of the data structures. By using hash tables (for random access) as well as linked lists (for sequential access) for representing the sets of top-level features, the average complexity of merging two mode graphs becomes $O\big(\min(w_1, w_2) + cd \cdot \alpha(l)\big)$. The term $\min(w_1, w_2)$ reflects the cost of merging the sets of top-level features, while the term $cd \cdot \alpha(l)$ reflects the cost of merging submode graphs with common top-level features. The term $\alpha(l)$ reflects the cost of dereferencing invisible pointers, but this can be considered virtually

constant because α grows extremely slowly.

The difference between the merging of mode graphs and the unification of feature graphs does not affect the time complexity. First, the label of a node and the polarity of an arc can be handled in constant time each time an arc is followed. Second, the handling of a node labeled 'ground' does not increase the time complexity, either. The unification of a graph with a single node labeled 'ground' and another graph is essentially the depth-first search of the latter graph which may involve implicit dereferencing, but this can be done within the above-mentioned time complexity.

2.5.2 Analyzing a Whole Program

Now we are in a position to consider the time complexity of the mode analysis of a whole program. We are still assuming the two assumptions at the beginning of Section 2.5.

We note that the number of primitive constraints imposed is $O(n)$, where n is the size of the program in terms of the number of symbols. This is because (i) for each function or variable symbol in a clause, each of Constraints (HF), (HV), (BF), and (BV) imposes at most one primitive constraint, and (ii) Constraint (BU) applies to each unification goal in the body. We also note that at most two top-level features occur in each primitive constraint.

As in the merging of two mode graphs, let us consider two aspects separately, namely the merging of these $O(n)$ sets of top-level features, and the merging of submode graphs with common top-level features. First, the time complexity of merging the $O(n)$ sets of top-level features one by one is $O(n)$, because merging two sets of sizes w_1 and w_2 costs $O(\min(w_1, w_2))$ time and $\min(w_1, w_2) \leq 2$. However, for large programs, we may want to compute the mode graphs of individual program modules separately and merge them later in an *arbitrary* order, because in incremental program development, we don't want to analyze a large program from scratch. In the event that the sets of top-level features are merged in an order in which the two sets to be merged are always almost equal, the cost can become $O(n \log n)$.

Second, we consider the cost of merging submode graphs with common top-level features. Suppose some top-level feature $\langle p, i \rangle$ occurs in k primitive constraints. Then the merging of submode graphs rooted at $\langle p, i \rangle$ will occur $k - 1$ times, irrespective of the order in which the primitive constraints are merged. Since at most two top-level features occur in each primitive constraint, the merging of submode graphs is performed $O(n)$ times in total. The time complexity of one merging operation is $O(d \cdot \alpha(n))$, where d is the size of the largest submode graph to be merged. The term $\alpha(n)$ reflects the fact (proof omitted) that the number of invisible pointers involved is $O(n)$. So the total cost of submode graph merging is $O(nd \cdot \alpha(n))$.

To summarize, when the primitive mode constraints are merged all at once, the cost of the analysis is $O(nd \cdot \alpha(n))$, and when the primitive mode constraints are merged in an arbitrary order, the cost is $O(n \log n + nd \cdot \alpha(n))$. In the latter case, however, the contribution of the term $n \log n$ to the total cost is expected to be much smaller than that of $nd \cdot \alpha(n)$.

For the class of programs being considered, the total correctness of the above analysis algorithm is almost immediate from the total correctness of the underlying unification algorithm for feature graphs [1].

2.5.3 General Cases

How can we cope with clauses with guard goals and/or variables with three or more channel occurrences?

When clauses may have guard goals, Constraint (GV) must be considered as well. We have not fully studied the general and efficient treatment of guard goals, but in practice, it suffices to consider the cases where

- guard goals are all calls to predefined test predicates,
- the size of the mode graph of each predefined test predicate is $O(d)$ (and therefore finite), and
- the mode graphs have been obtained before the analysis of non-test predicates.

When the above three conditions are met, Constraint (GV) on a variable v occurring at p_h in the head and at p_g in the guard of a clause can be implemented by (1) making a copy of the submode graph of m/p_g and (2) merging the obtained copy and the current submode graph of m/p_h. The copying operation is to avoid the propagation of constraints to test predicates, but it can be done in $O(d)$ time. (Dereferencing of invisible pointers is unnecessary here, because the invisible pointers can be eliminated from the mode graphs beforehand.) In addition, with Constraint (GV), the number of top-level features to be merged is still $O(n)$. So Constraint (GV) does not increase the time complexity obtained in Section 2.5.2.

When some variable has three or more channel occurrences in a clause, the constraint associated with that variable cannot be solved immediately using unification. In many cases, however, such a constraint can be reduced to a unary or binary constraint (constraint involving only one or two paths, respectively) with the aid of other constraints, in which event the generate-and-test search of well-moding is not needed. We anticipate that this is almost always the case in practice. For example, a predicate `p` that may spawn sibling processes `q` sharing the same input data would have a clause of the form

```
p(X, ...) :- ... | q(X, ...), p(X, ...).
```

From this clause, we can immediately infer $m/\langle \mathtt{q}, 1\rangle = IN$, using the implicit constraint that the mode of the predicate `p` is common to all calls to it. The submode $m/\langle \mathtt{p}, 1\rangle$ is not constrained at all.

The best strategy of constraint solving is thus to solve unary and binary constraints first, delaying non-binary constraints. Unfortunately, some programs still require generate-and-test search or a more powerful set of reduction rules to solve non-binary constraints. An example is the following set

of clauses, which is non-well-moded:

```
p1 :- true |       r(X), s(X), t(X).
p2 :- true | q(X),       s(X), t(X).
p3 :- true | q(X), r(X),       t(X).
p4 :- true | q(X), r(X), s(X).
```

However, such powerful constraint solving seems to be necessary only for the analysis of pathological programs and hence will not justify the implementation effort. A much more promising way is to avoid generate-and-test search completely by letting programmers *declare*, in some form or other, the modes of the paths where variables with three or more channel occurrences may occur. The declared mode constraints are used for reducing constraints associated with these variables. This is a reasonable solution, because (1) only a small number of variables have three or more channel occurrences in a clause and (2) those variables almost always have simple dataflow that is easy to declare.

2.6 Well-Moded Programs Do Not Go Wrong

Here we first present and prove the basic theorem concerning one-step reduction of a goal clause, namely the execution of a unification goal or the replacement of a non-unification goal by the body goals of a program clause.

Lemma 1. Let m be a well-moding of a clause C, and let $t_1 =_k t_2$ be a unification (body) goal in C. Then there exists an i such that (i) $m(\langle =_k, i\rangle) = \mathit{out}$ and (ii) t_i is a variable.

Proof. (i) is immediate from Constraint (BU), and (ii) is immediate from (i) and the contrapositive of Constraint (BF). ∎

Let v be a variable and t a term. We say that the *extended occur check* for unification between v and t *fails* if t is v or t contains v.

Theorem 1. Let m be a well-moding of a program P and a goal clause G. Suppose G is reduced by one step into a goal clause G', where the reduced goal $g \in G$ is *not* a unification goal for which the extended occur check fails. Then m is a well-moding of P and G' as well.

Proof. We have two cases. For notational convenience, we identify a goal clause with the multiset of the body goals it contains.

Case (i): The reduction reduces a non-unification goal g using a clause $C \in P$ (renamed using fresh variables) of the form h :- ... | B. The synchronization rule of GHC states that there is a substitution θ such that $g = h\theta$. Note that $G' = G \setminus \{g\} \cup B\theta$, where '$\setminus$' and '$\cup$' are multiset difference and union, respectively.

We must consider Constraint (BV) imposed by the variable symbols occurring in $g \in G$ and Constraint (BF) imposed by the occurrences of function symbols introduced to $B\theta$ ($\in G'$) by θ. (These occurrences originate in the

occurrences of the same function symbols in g.) Constraints imposed by the other variable symbols in G' and constraints imposed by the other occurrences of function symbols in G' (which are already in $G \setminus \{g\}$ and/or B) are exactly the same as those before reduction. Hence we consider each symbol in g, namely $\widetilde{g}(p)$ ($p \in P_{Atom}$) such that $\widetilde{g}(p) \neq \bot$:

(a) $\widetilde{g}(p)$ is a function symbol f. Then either (1) $\widetilde{h}(p) = f$, or (2) there exist $p' \in P_{Atom}$ and $q \in P_{Term}$ such that $p = p'q$ and $\widetilde{h}(p')$ is a variable (say v). In Case (1), the occurrence disappears upon reduction and Constraint (BF) becomes inapplicable. So we need only to consider Case (2), which may introduce new occurrences of f to $B\theta$. Suppose v occurs n (≥ 0) times in B at $r_1, \ldots, r_n$, and let g_j be the goal to which the occurrence at r_j belongs. This means $\widetilde{g_i\theta}(r_iq) = f$ holds in G' for $i = 1, \ldots, n$. For m to be a well-moding of G', $m(r_iq) = in$ must hold because of Constraint (BF). However, this can be derived as follows:

(1) $m(p) = in$ by Constraint (BF) applied to f in G,
(2) $\mathcal{R}(\{\overline{m/p'}\} \cup \bigcup_{1 \leq i \leq n}\{m/r_i\})$ by Constraint (BV) applied to v in C,
(3) $\mathcal{R}(\{\overline{m/p}\} \cup \bigcup_{1 \leq i \leq n}\{m/r_iq\})$ by (2) and Property (P8),
(4) $m(r_iq) = in, \quad 1 \leq i \leq n$ by (1) and (3).

(b) $\widetilde{g}(p)$ is a variable w occurring l (≥ 1) times in g at $p_1(= p), p_2, \ldots, p_l$, and m (≥ 0) times in $G \setminus \{g\}$ at $p_{l+1}, \ldots, p_{l+m}$. Because there is a substitution θ such that $g = h\theta$, for each p_i ($1 \leq i \leq l$), there exist paths $p_i' \in P_{Atom}$ and $q_i \in P_{Term}$ such that $p_i = p_i'q_i$ and $\widetilde{h}(p_i')$ is a variable (say v_i). Suppose v_i occurs n_i (≥ 0) times in B at $r_{i1}, \ldots, r_{in_i}$. Then,

(1) $\mathcal{R}(\bigcup_{1 \leq i \leq l+m}\{m/p_i\})$ by Constraint (BV) applied to w in G,
(2) $\mathcal{R}(\{\overline{m/p_i}\} \cup \bigcup_{1 \leq j \leq n_i}\{m/r_{ij}q_i\}), \quad 1 \leq i \leq l$
by Constraint (BV) applied to v_i in C, and Property (P8),
(3) $\mathcal{R}(\bigcup_{1 \leq i \leq l}\bigcup_{1 \leq j \leq n_i}\{m/r_{ij}q_i\} \cup \bigcup_{l+1 \leq i \leq l+m}\{m/p_i\})$
by (1), (2), and Property (P7).

Here, we first consider the case where none of the v_i's has two or more occurrences in h. Then w will occur $n_1 + \cdots + n_l$ times in $B\theta$ at $r_{ij}q_i$ ($1 \leq i \leq l$, $1 \leq j \leq n_i$), and Property (3) above is exactly Constraint (BV) applied to w in G'.
Next, we consider the case where some v_k occurs twice or more in h. Let K be the set of such k's. In this case, w may occur less than $n_1 + \cdots + n_l$ times, because two occurrences of w in g may be received by different occurrences of the same variable in h and introduced to $B\theta$ not independently. However,

(4) $m/p_k' = IN, \quad k \in K$ by Constraint (HV) applied to v_k in C,
(5) $\mathcal{R}(\{\overline{m/p_k'}\} \cup \bigcup_{1 \leq j \leq n_k}\{m/r_{kj}\}), \quad k \in K$

by Constraint (BV) applied to v_k in C,

(6) $m/r_{kj} = IN, \quad k \in K,\ 1 \le j \le n_k$ by (4), (5), and Property (P4),

(7) $m/r_{kj}q_k = IN, \quad k \in K,\ 1 \le j \le n_k$ by (6).

This means that Constraint (BV) applied to w in G' can ignore occurrences of w introduced to $B\theta$ by the v_k's ($k \in K$), because in general, $\mathcal{R}(S \cup \{IN\}) \Leftrightarrow \mathcal{R}(S)$ holds (Property (P3)). On the other hand,

(8) $\mathcal{R}(\bigcup_{1 \le i \le l,\, i \notin K} \bigcup_{1 \le j \le n_i} \{m/r_{ij}q_i\} \cup \bigcup_{l+1 \le i \le l+m} \{m/p_i\})$
by (3), (7), and Property (P3).

This is exactly Constraint (BV) applied to w in G', with the occurrences of w introduced by the v_k's ($k \in K$) ignored.

Case (ii): The reduction executes a unification goal $t_1 =_k t_2$. By Lemma 1, there exists an i such that t_i is a variable and $m(\langle =_k, i\rangle) = out$. Without loss of generality, we can assume $i = 1$ (that is, unification is always assignment to the *left-hand* side variable). By the assumption of the extended occur check, the term t_2 is not, or does not contain, the variable t_1. Hence $G' = (G \setminus \{t_1 =_k t_2\})\theta$, where $\theta = \{t_1 \leftarrow t_2\}$. Suppose the variable t_1 occurs n (≥ 0) times elsewhere in G at $r_1, \ldots, r_n$. We consider the symbols in t_2, namely the $\widetilde{t_2}(q)$'s ($q \in P_{Term}$) such that $\widetilde{t_2}(q) \neq \bot$:

(a) $\widetilde{t_2}(q)$ is a function symbol f. Then

(1) $m(\langle =_k, 2\rangle q) = in$ by Constraint (BF),

(2) $m(\langle =_k, 1\rangle q) = out$ by (1) and Constraint (BU),

(3) $m(r_i q) = in, \quad 1 \le i \le n$
by (2) and Constraint (BV) applied to t_1 in G.

By executing $t_1 =_k t_2$, f is made to occur newly at $r_1 q, \ldots, r_n q$ in G'. However, as shown above, m satisfies Constraint (BF) imposed by these occurrences.

(b) $\widetilde{t_2}(q)$ is a variable w ($\neq t_1$) occurring l (≥ 1) times in $t_1 =_k t_2$ at $p_1 (= \langle =_k, 2\rangle q)$, $p_2, \ldots, p_l$ and m (≥ 0) times in $G \setminus \{t_1 =_k t_2\}$ at $p_{l+1}, \ldots, p_{l+m}$. Let q_i be such that $\langle =_k, 2\rangle q_i = p_i$, for $1 \le i \le l$. Then

(1) $\mathcal{R}(\bigcup_{1 \le i \le l+m} \{m/p_i\})$ by Constraint (BV) applied to w in G,

(2) $\mathcal{R}(\{m/\langle =_k, 1\rangle q_i\} \cup \bigcup_{1 \le j \le n} \{m/r_j q_i\}), \quad 1 \le i \le l$
by Constraint (BV) applied to t_1 in G, and Property (P8),

(3) $m/\langle =_k, 1\rangle q_i = \overline{m/p_i}, \quad 1 \le i \le l$ by Constraint (BU) applied to $=_k$,

(4) $\mathcal{R}(\bigcup_{1 \le i \le l} \bigcup_{1 \le j \le n} \{m/r_j q_i\} \cup \bigcup_{l+1 \le j \le l+m} \{m/p_j\})$
by (1), (2), (3), and Property (P7).

However, Property (4) is exactly Constraint (BV) applied to w in G'. ∎

Note that Theorem 1 does not hold when the reduced goal g is of the form $v = v$. Consider the case:

G : `:- p(A,A), q(A),`
P : `{p(X,Y) :- true | X=Y}.`

The program P imposes the constraint $m/\langle \mathtt{p},1\rangle = \overline{m/\langle \mathtt{p},2\rangle}$ which, combined with Constraint (BV) applied to the goal clause G, gives us $m/\langle \mathtt{q},1\rangle = IN$. However, by reducing `p(A,A)` and then `A=A`, the goal clause becomes

```
:- q(A),
```

which violates the constraint $m/\langle \mathtt{q},1\rangle = IN$. The problem here is that the goal of the form $v = v$ eliminates the output occurrence of v in the goal clause without providing any value to the occurrences of v outside the goal $v = v$. Fortunately, unification goals of the form $v = v$ are not used in any practical programs. Moreover, we can easily detect them by slightly extending the occur check (if it is implemented at all).

Alternatively, goals of the form $v = v$ can be allowed in Theorem 1 if a variable is disallowed to have more than two channel occurrences in a clause. This alternative, however, does not work in Theorem 2 below.

Theorem 1, together with Constraint (BV) applied to goal clauses, guarantees that well-moded programs follow Convention (1) shown in the beginning of Section 2 unless the extended occur check fails.

Note that the use of the weakened version of Constraint (BV) shown in Section 2.3 does not affect Theorem 1.

From Lemma 1 and Theorem 1, a unification goal in a goal clause (derived as a result of reduction) is always unification between a variable and a term, from which the following important corollary follows:

Corollary 1. If the pair of a program and a goal clause is well-moded and the extended occur check does not fail, the pair does not cause unification failure (failure of unification body goals).

The exclusion of unification of the form $v = v$ allows us to obtain another strong theorem:

Theorem 2. Let m be a well-moding of a program P and a goal clause G. Assume the execution of G has succeeded (that is, G has been reduced to an empty multiset of goals) without causing the failure of the extended occur check. Then, in that execution, a unification goal of the form $v =_k t$ such that $m(\langle =_k, 1\rangle) = out$, or a unification goal of the form $t =_k v$ such that $m(\langle =_k, 2\rangle) = out$, must have been executed, for any variable v occurring in G.

Proof. By Constraint (BV), v has exactly one output occurrence in G. Let g be the goal containing that occurrence, and p the path such that $\widetilde{g}(p) = v$ and $m(p) = out$. Without loss of generality, we can assume that $m(\langle =_k, 1\rangle) = out$ holds for all unification goals in P and G. Since the theorem vacuously holds if G is an empty clause, we can assume that G is non-empty and has succeeded. So we consider the first reduction from G to some goal clause G', and classify it into the following four cases:

(i) The goal g is of the form $v =_k t$ (that is, $p = \langle =_k, 1\rangle$), and the reduction reduces g.

(ii) The goal g is of the form $w =_k t$ such that $p = \langle =_k, 2\rangle q$ for some q (that is, t contains an output occurrence of v), and the reduction reduces g.

Suppose w occurs n (≥ 0) times in $G \backslash \{w =_k t\}$ at $r_1, \ldots, r_n$. Then

(1) $m(\langle =_k, 2 \rangle q) = out$ by assumption,
(2) $m(\langle =_k, 1 \rangle q) = in$ by (1) and Constraint (BU),
(3) $\mathcal{R}(\{m/\langle =_k, 1 \rangle, m/r_1, \ldots, m/r_n\})$ by Constraint (BV) applied to w in G,
(4) $\exists k \leq n\big(m(r_k q) = out\big)$ by (2) and (3),
(5) $n > 0$ by (4).

Hence G' retains at least one occurrence of v, because those n occurrences of w are rewritten to t by the reduction.

(iii) The goal g is a non-unification goal, and the reduction reduces g using a clause h :- ... | B. Then there must be paths $p' \in P_{Atom}$ and $q \in P_{Term}$ such that $p = p'q$ and $\widetilde{h}(p')$ is a variable (say w). Suppose w occurs n (≥ 0) times in B at $r_1, \ldots, r_n$. Then

(1) $m(p'q) = out$ by assumption,
(2) $\mathcal{R}(\{\overline{m/p'}, m/r_1, \ldots, m/r_n\})$ by Constraint (BV) applied to w in h :- ... | B,
(3) $\exists k \leq n\big(m(r_k q) = out\big)$ by (1) and (2),
(4) $n > 0$ by (3).

Hence G' retains at least one occurrence of v, because v occurs newly in G' at $r_1 q, \ldots, r_n q$.

(iv) The reduction reduces a goal g' other than g. Note that g' does not contain an output occurrence of v because g contains it. The variable v will not be rewritten by the reduction, because only a unification goal of the form $v =_k t$ which has an output occurrence of v can rewrite v. So G' contains an instance of g which has an output occurrence of v.

To summarize, by the reduction from G to G', either

(a) a unification goal of the form $v =_k t$ was executed, or
(b) G' retained an output occurrence of v.

In Case (b), the above argument applies also to the reduction of G' because, by Theorem 1, m is a well-moding of G' as well. However, by assumption, G was finally reduced into an empty multiset of goals. So Case (a) must have eventually happened. ∎

The following corollary follows from Theorem 2:

Corollary 2. Under the assumptions of Theorem 2, the product of all substitutions generated by unification (body) goals maps all the variables in G to ground terms.

Proof. We again assume that $m(\langle =_k, 1 \rangle) = out$ for all unification goals in P. By Theorem 2, for any variable v occurring in G, a goal of the form $v =_k t$ such that $m(\langle =_k, 1 \rangle) = out$ must have been executed, rewriting v to t. There are two cases:

(a) t is ground. Then the corollary is immediate.
(b) t contains one or more variables $v_1, \ldots, v_n$. Let G' be the goal clause whose reduction executed the goal $v{=}_k t$. (G' is equal to G iff $v{=}_k t$ was the first goal executed.) By Theorem 1, m is a well-moding of G'. So the corollary holds for v and G if it holds for $v_1, \ldots, v_n$ and G'.

However, in a successful execution of G, in which only a finite number of unification goals are executed, the recursion in Case (b) cannot happen infinitely many times. Hence the product of all substitutions generated by executing unification goals must have finally mapped v to a ground term. ∎

Thus, under the extended occur check, the groundness property follows from the termination property.

One may suspect that the following pair of clauses forms a counterexample of Corollary 2:

```
        :- p(A).
p(X) :- true | X=f(Y).
```

However, our mode system excludes the pair, because the goal clause imposes the constraint $m/\langle \mathtt{p}, 1\rangle = OUT$, while the program clause imposes the constraint $m/\langle \mathtt{p}, 1\rangle\langle \mathtt{f}, 1\rangle = IN$. To make the above example well-moded, we must supply a goal that instantiates `Y`, for instance in the following manner:

```
          :- p(A), q(A).
   p(X) :- true | X=f(Y).
q(f(Y)) :- true | Y=g.
```

2.7 Discussions

The mode analysis described above 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 inherently 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 program 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, mode checking and mode inference.

The mode system described here is quite different from the mode system of PARLOG. Basically, PARLOG modes are for moving output unification to clause bodies when compiling into Kernel PARLOG. Declaring an argument as input does *not* mean that the principal function symbol will not be determined by a callee, and the program

```
mode p(?).                (‘?’ stands for input)
p(X) :- true : X=5.
```

is a correct PARLOG program which corresponds to a GHC program

```
p(X) :- true | X=5.
```

The modes of DEC-10 Prolog are very different also; they are the declaration of the time-of-call instantiation states of arguments. On the other hand, our mode system is concerned with eventual rather than time-of-call properties, though output paths in a well-moded program are guaranteed to be uninstantiated at the time of call.

Mode analysis is useful also for debugging purposes. In GHC programming, a variable with only one channel occurrence very often indicates a program error, unless its value is checked in the guard of the clause. This can happen, for instance, when a programmer misspells variable names:

```
p(X0, ...) :- ... | q(X0,X1), p(X1, ...).   (X0 must have been X0)
```

In this case, the two variables `X0` and `X0` impose strong constraints, namely $m/\langle \mathtt{p}, 1\rangle = IN$ and $m/\langle \mathtt{q}, 1\rangle = OUT$. They are very different from the constraint $m/\langle \mathtt{p}, 1\rangle = m/\langle \mathtt{q}, 1\rangle$ obtained from the correct clause, which are highly likely to be inconsistent with constraints obtained from other clauses.

Also, programmers often forget to close streams after use:

```
p([],     Ys) :- true | true.   (The body must have been Ys=[])
p([X|Xs1],Ys) :- ... | Ys=[...|Ys1], p(Xs1,Ys1).
```

This is detected as a mode error, because the (erroneous) base-case clause imposes the constraint $m/\langle \mathtt{p}, 2\rangle = IN$, while the recursive clause says $m(\langle \mathtt{p}, 2\rangle) = out$.

Note that the proofs in Section 2.6 do not use Constraints (HF) or (GV). These constraints need not be imposed if the purpose of the mode system is just to guarantee the basic properties proved in Section 2.6. However, they are well-motivated as we argued in Section 2.3, and can be useful for finding bugs and reducing non-binary constraints.

Implementation details of the mode system are beyond the scope of this paper, but we give one remark here. A user-friendly mode analyzer should not simply say "no" for non-well-moded programs. The system should tell *why* the program is non-well-moded. It must be helpful if the system finds a (nearly) minimal set of clauses whose mode constraints are inconsistent.

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 technique based on *message-oriented scheduling* [11]. This paper focuses on multiprocessing within one processor, though the techniques we propose here can be utilized also in parallel implementations [23].

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 unification body goal into assignment to a variable (assuming that the extended occur check always succeeds).

Furthermore, in some cases we can easily guarantee that the variable has been fully dereferenced and that *no* goals are suspending on that variable [11]. Consider an n-ary predicate `p` whose recursive clause C is of the form

```
p(...,X0) :- ... | X0=[...|X], p(...,X).
```

The clause imposes the constraint $m(\langle \mathtt{p}, n\rangle) = \mathit{out}$ and hence the last argument of any call to `p` must be a variable. Assume that a goal `p(...,A)` is about to be executed. Instead of `p(...,A)`, suppose we execute `p(...,A')`, a goal whose last argument is replaced by a fresh variable `A'`. The clause C will reduce that goal to `A'=[...|A'']` and `p(...,A'')`, namely assignment to a fresh variable and a recursive call having another fresh variable as the last argument. Thus, by starting execution with a fresh output variable, unification body goals executed during repetitive tail recursion are guaranteed to be assignments to fresh variables on which no goals are suspending. The connection between `A` and `A'` must be established eventually, but it can be done by executing the goal `A=A'` only once when the repetitive tail recursion has succeeded, suspended, or has been swapped out (that is, a recursive subgoal is put into an appropriate goal queue in order to execute another goal in the same or another queue).

In general, 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 [16].

3.1 Process- vs. Message-Oriented Scheduling

In conventional, process-oriented scheduling, a scheduler tries to reduce the number of process switching operations. Once a goal starts or resumes execution, its subgoals run as long as possible (unless they are swapped out) before another goal in some goal queue gains control. A stream connecting goals acts as a buffer whose contents are processed all 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. (We 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([],    Y) :- true | Y=[].
p([A|X1],Y) :- true | Y=[A|Y1], p(X1,Y1).
```

Of the two body goals of the recursive clause, 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 [24].

In contrast, message-oriented scheduling executes `Y=[A|Y1]` as message passing; that is, it transfers both control and the datum `A` 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. Furthermore, it turns out that the tail-recursive goal `p(X1,Y1)` is compiled into no operations, as will be discussed soon.

To achieve the efficient transfer of control and data, we implement a stream not as a linked data structure 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 (Figure 4). A communication cell is allocated, and its entries are initialized, by executing the consumer goal of `Y` prior to the inter-process communication. A message to be transferred is placed on a hardware register called a *communication register*. For a goal p to send a message m to a goal q through a stream s, the goal p first loads m on the communication register, and then executes the code of q pointed to by the first field of the communication cell for s, using the goal record for q pointed to by the second field of the communication cell for s as the environment. We could apply the same implementation technique to non-stream data structures also, but this paper focuses on stream communication in which larger advantage can be expected from the repetitive use of the same communication cells.

In the recursive clause, the execution of `p(X1,Y1)` involves *no* operations (until the next message arrives at the input stream). This is because

- its goal record can be inherited from the parent goal,
- the first argument recorded in the inherited goal record continues to point to the same communication cell,
- the second argument also continues to point to the same communication cell, and
- the goal will be immediately suspended at the same instruction at the beginning of the code of `p`.

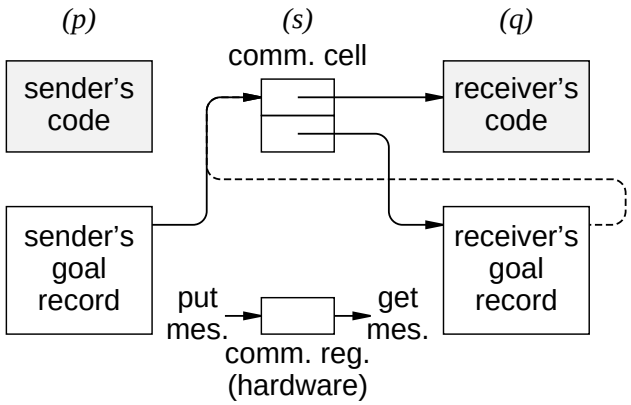

Figure 4. Immediate message send.

Because no linked data structure is formed, the end of a stream is represented as a special message (say *eos*), not as the constructor of empty lists.

Thus the execution of a process defined by the predicate `p` will proceed as follows: First of all, `p` creates a communication cell for the input stream and suspends until a message is sent. As soon as a message is sent through the input stream, `p` resumes execution and first checks if the message is *eos* or not. If it is not *eos*, the body of the recursive clause will

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

Control need not be returned to the recursive clause because no operations remain to be done for the current message. When the next message arrives later, the execution of `p`'s code is resumed from the address from where execution was resumed for the current message. Moreover, the message `A` need not be loaded to the communication register because it has already been loaded when control is transferred to this clause. The body of the recursive clause is thus compiled into a very efficient code.

If the message is *eos*, the body of the base-case clause will send another *eos* to the output stream. It will also deallocate the current goal record.

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 and caching operations would incur large overhead.

3.3 Message-Oriented Scheduling in General Cases

A number of questions may arise when generalizing the above implementation technique: How can a compiler distinguish between variables representing streams and those representing non-stream data? How to cope with communication that is not one-to-one? Shouldn't a stream sometimes act as a buffer?

This section will discuss basic ideas on these issues. Detailed description at the intermediate code level is given in [23], which deals with parallel execution as well.

3.3.1 Identifying Stream Communication

A compiler must be able to distinguish between variables representing streams and those representing non-stream data. 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 well-typing function $\tau : P_{Atom} \to \{stream, nonstream\}$, where

- $\tau(p) = stream$ means that only the constructors of empty and non-empty streams (`[]` and '`.`') can occur at p, and
- $\tau(p) = nonstream$ means that the constructors of streams cannot occur at p.

For a type τ and a path $p \in P_{Atom}$, $\tau/p : P_{Term} \to \{stream, nonstream\}$ denotes a function such that $\forall q \in P_{Term}\big((\tau/p)(q) = \tau(pq)\big)$. The function τ should always satisfy the constraint

$$\forall p \in P_{Atom}\big(\tau(p) = stream \;\Rightarrow\; \tau(p\langle .,2\rangle) = stream\big),$$

to enable the special treatment of repetitive communication with streams. In addition, τ should satisfy the following type constraints syntactically imposed by each clause C in a program fragment, where C is either of the form h `:-` G `|` B or the form `:-` B:

$$\begin{aligned}
(\mathrm{HBF}_\tau)\;\; & \forall p \in P_{Atom} \forall a \in \{h\} \cup B\big((\widetilde{a}(p) \in \{\texttt{[]}, .\} \;\Rightarrow\; \tau(p) = stream) \\
& \qquad \wedge\; (\widetilde{a}(p) \in Fun \backslash \{\texttt{[]}, .\} \;\Rightarrow\; \tau(p) = nonstream)\big) \\
(\mathrm{HBV}_\tau)\;\; & \forall p, p' \in P_{Atom}\, \forall a, a' \in \{h\} \cup B\big(\widetilde{a}(p) \in Var \wedge \widetilde{a}(p) = \widetilde{a'}(p') \;\Rightarrow\; \tau/p = \tau/p'\big) \\
(\mathrm{GV}_\tau)\;\; & \forall p, p' \in P_{Atom}\, \forall a \in G\big(\widetilde{h}(p) \in Var \wedge \widetilde{h}(p) = \widetilde{a}(p') \\
& \qquad \Rightarrow\; \forall q \in P_{Term}\big(m(p'q) = in \;\Rightarrow\; \tau(pq) = \tau(p'q)\big)\big) \\
(\mathrm{BU}_\tau)\;\; & \forall k > 0\, \forall t_1, t_2 \in Term\big((t_1 \texttt{=}_k t_2) \in B \;\Rightarrow\; \tau/\langle \texttt{=}_k, 1\rangle = \tau/\langle \texttt{=}_k, 2\rangle\big)
\end{aligned}$$

Constraint (GV_τ), which depends on the mode system, does not constrain the type values of the paths that will not be examined by guard goals.

As an example, we consider Program 2 again. We assume that the type constraints for the predefined predicates are:

$$\begin{aligned}
& \tau(\langle \texttt{=:=}, i\rangle) = \tau(\langle \texttt{=\textbackslash=}, i\rangle) = nonstream, \quad \text{for } i = 1,2, \\
& \tau(\langle \texttt{subtract}, i\rangle) = nonstream, \quad \text{for } i = 1,2,3.
\end{aligned}$$

Then, the constraints obtained from the clauses in Program 2 are:

$$\begin{aligned}
& \tau(\langle \texttt{drive}, 1\rangle) = nonstream,\; \tau(\langle \texttt{drive}, 2\rangle) = stream, \\
& \tau(\langle \texttt{drive}, 2\rangle\langle .,1\rangle) = nonstream,\; \tau(\langle \texttt{drive}, 2\rangle\langle .,1\rangle\langle \texttt{push}, 1\rangle) = nonstream, \\
& \tau(\langle \texttt{drive}, 2\rangle\langle .,2\rangle) = stream,\; \tau(\langle \texttt{drive}, 2\rangle\langle .,2\rangle\langle .,1\rangle) = nonstream,
\end{aligned}$$

$\tau(\langle \texttt{drive},2\rangle\langle .,2\rangle\langle .,1\rangle\langle \texttt{pop},1\rangle) = \mathit{nonstream}$,
$\tau/\langle \texttt{drive},2\rangle\langle .,2\rangle\langle .,2\rangle = \tau/\langle \texttt{drive},2\rangle$,
$\tau/\langle \texttt{=}_1,1\rangle = \tau/\langle \texttt{=}_1,2\rangle = \tau/\langle \texttt{drive},2\rangle$, $\tau/\langle \texttt{=}_2,1\rangle = \tau/\langle \texttt{=}_2,2\rangle = \tau/\langle \texttt{drive},2\rangle$,
$\tau(\langle \texttt{stack},1\rangle) = \mathit{stream}$, $\tau(\langle \texttt{stack},1\rangle\langle .,1\rangle) = \mathit{nonstream}$,
$\tau/\langle \texttt{stack},1\rangle\langle .,1\rangle\langle \texttt{push},1\rangle = \tau/\langle \texttt{stack},2\rangle\langle \texttt{p},1\rangle$,
$\tau/\langle \texttt{stack},1\rangle\langle .,1\rangle\langle \texttt{pop},1\rangle = \tau/\langle \texttt{stack},2\rangle\langle \texttt{p},1\rangle$,
$\tau/\langle \texttt{stack},1\rangle\langle .,2\rangle = \tau/\langle \texttt{stack},1\rangle$,
$\tau(\langle \texttt{stack},2\rangle) = \mathit{nonstream}$, $\tau/\langle \texttt{stack},2\rangle\langle \texttt{p},2\rangle = \tau/\langle \texttt{stack},2\rangle$,
$\tau/\langle \texttt{terminate},1\rangle = \tau/\langle \texttt{stack},2\rangle$,
$\tau/\langle \texttt{=}_3,1\rangle = \tau/\langle \texttt{=}_3,2\rangle = \tau/\langle \texttt{stack},2\rangle\langle \texttt{p},1\rangle$.

The basic theorem of our typing scheme is analogous to Theorem 1:

Theorem 3. Let τ be a well-typing of a program P and a goal clause G. Suppose G is reduced by one step into a goal clause G', where the reduced goal $g \in G$ is *not* a unification goal for which the extended occur check fails. Then τ is a well-typing of P and G' as well.

We omit the proof, which is analogous to, and simpler than, the proof of Theorem 1.

The type system proposed in [25] could be used also, though that system is based on type checking rather than type inference.

3.3.2 One-To-Many/Zero Communication

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). This subsection discusses one-to-many/zero communication and the next subsection discusses many-to-one communication.

An obvious way to implement one-to-many/zero communication is to transform it into repetitive one-to-one communication. For example, when a goal commits to the clause

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

a dummy process is created which eats up the messages in `X`.

When a stream has two or more consumers initially or when a single consumer splits into two or more, a process for distributing messages is created. Consider the clause

```
p :- true | q(Xs), r(Xs), s(Xs).
```

If the variable `Xs` is for stream communication from `q` to `r` and `s`, the above clause is compiled into:

```
p :- true | q(Xs), r(Ys), s(Zs), distribute(Xs,Ys,Zs).
distribute([]    ,Ys,Zs) :- true | Ys=[], Zs=[].
distribute([X|Xs],Ys,Zs) :- true |
    Ys=[X|Ys1], Zs=[X|Zs1], distribute(Xs,Ys1,Zs1).
```

Here we have assumed that the element `X` distributed in the second clause of `distribute` is not, or does not contain, a stream. This is what the type system must guarantee. If `X` is a stream or contain streams, the second clause of `distribute` must spawn another distributor for the variable `X` occurring three times in the clause.

There may be more efficient ways of handling one-to-many/zero communication, but we will not go into details since our primary concern is to implement one-to-one communication as efficiently as possible. One possibility is to use an ordinary, linked-list implementation of streams for one-to-many/zero communication.

3.3.3 Many-To-One Communication

Implementation of many-to-one communication, namely implementation of processes with more than one input stream, is important since many-to-one communication 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 an indeterminate 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 streams.

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 is sent through the first stream. Then the `omerge` goal records it and waits for another number to be sent through the second stream. However, the second number may be sent through 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.

Figure 5 shows a straightforward implementation of buffered message sending. A communication cell is made to point to the code for buffering and the descriptor of a buffer, and the previous contents of the communication

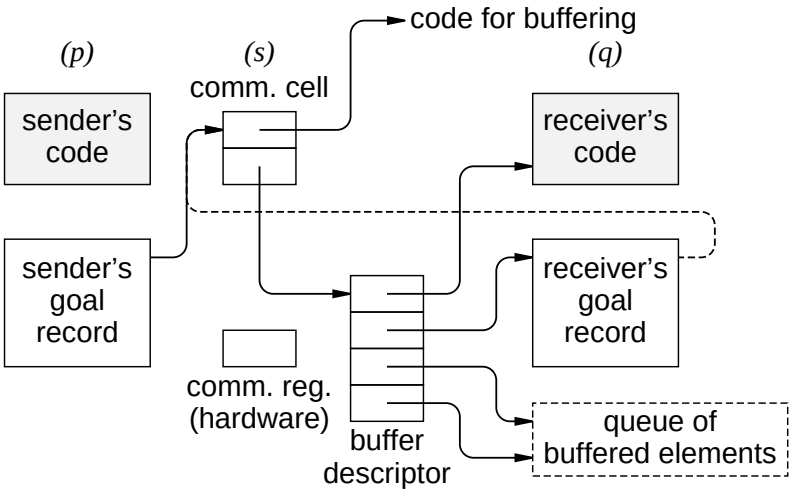

Figure 5. Buffered message send.

cell are saved in the buffer descriptor. The code will put the content of the communication register into the buffer pointed to by the descriptor. Note that whether the receiver buffers a message or not makes no difference to the sender's protocol. Although buffered message sending in message-oriented scheduling is more costly than buffered communication in process-oriented scheduling, most messages can be sent without buffering and no overhead is incurred for those messages.

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 sent through the second input stream must be buffered until the first input stream is closed; then they must be forwarded to 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.

3.3.4 When Buffering Is Required

In general, the buffering of messages is required for streams through which messages not ready to be handled may possibly be sent. Selective message receiving discussed in Section 3.3.3 is one possible reason for this, but the need of buffering can arise for other reasons also:

(i) *Suspension on non-stream data.* A receiver goal may suspend upon non-stream data. In particular, it may suspend upon the *content* of a message which has been 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.

(ii) *The producer of a stream running ahead of the consumer.* 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)`, of which `g1` consumes `X` and `g2` consumes `Y`. We must execute these goals in such a way 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`.

(iii) *Circular process structure.* A message sent by a process p may possibly arrive at p itself or may cause another message to be sent back to p. Suppose p is going to send two messages $\mathtt{b}_1$ and $\mathtt{c}_1$ in response to an incoming message $\mathtt{a}_1$, using the following clause:

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

(The suffixes here are to distinguish between different messages with the same content.) We assume $\mathtt{b}_1$ is sent first. Control will be transferred to $\mathtt{b}_1$'s receiver, which may cause subsequent message sends in its turn. However, control will eventually return to the above clause, which is still taking care of p's response to $\mathtt{a}_1$, when all those message sends have been done and all newly spawned non-unification goals have suspended. Then the message $\mathtt{c}_1$ will be sent.

The problem here is that the sending of $\mathtt{b}_1$ may cause another message $\mathtt{a}_2$ to arrive at p before $\mathtt{c}_1$ is sent. If $\mathtt{a}_2$ is received and $\mathtt{b}_2$ and $\mathtt{c}_2$ are sent in response before $\mathtt{c}_1$ is sent, the order of messages on the second output stream is inverted. The process p should therefore buffer messages sent through the input stream until $\mathtt{c}_1$ is sent.

Fortunately, we do not have to prepare for message inversion when only one message is sent in response to each incoming message. To generalize, suppose

(a) a process should send n messages in response to an incoming message but hasn't received any message in response to the first $n-1$ messages, and
(b) the tail-recursive goal for responding to the next incoming message requires no operations to be performed after the nth message send (as in the example of Section 3.2).

Then, the last message can be sent without preparing for buffering. Moreover, 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 last-call optimization of Prolog [24]. The example in Section 3.2 is the case where $n = 1$ and hence last-send optimization is enabled regardless of the process structure. When Condition (a) does not hold, buffering must be continued during the last message send, and the buffered incoming messages must be processed after sending the last message. Condition (b) is usually satisfied by recursive clauses in which tail-recursive goals take the rest of the input/output streams of the

parent goals. A recursive goal may involve some operations (the setting of non-stream arguments, for example), but they do not disable last-call optimization as long as they can be moved before the last message send by instruction reordering.

3.4 Preliminary Evaluation

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 nonterminal nodes, and compared the result with the numbers on a native-code, process-oriented implementation on VAX11/780, GHC/V [10]:

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 (hence no suspension), and 'interactive' means that each command was issued after receiving the result of the previous command (hence suspension happened every time a command passed a tree node). 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 recursion took 0.31 seconds 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 the technique described in the beginning of Section 3. Our message-oriented implementation, employing 64-bit words, ran naive reverse at 55kRPS and improved the space complexity (from $O(n^2)$ to $O(n)$, n being the input size). It is interesting to see how a naive reverse program runs under message-oriented scheduling [23].

Unfortunately, not all programs we tried were made more efficient. An 8-queens program, which made heavy use of one-to-many communication naively implemented as repetitive one-to-one communication (Section 3.3.2), ran about 3 times slower than on GHC/V. However, we expect that the process-oriented and the message-oriented implementation techniques can naturally co-exist in a single implementation, since our preliminary implementation was actually obtained by modifying GHC/V.

4. Conclusion and Related Work

We have proposed a new implementation technique of Flat GHC that contrasts sharply with previous techniques. One of the contributions 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. Our recent work has shown that the technique can be utilized also in fine-grain parallel implementations [23].

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 Flat GHC. The resulting language is called Moded Flat GHC. Well-moded programs enjoy desirable properties as proved in Section 2.6. The mode system helps both optimization and the static detection of program errors. Furthermore, it will make the use of native codes more realistic. Native-code implementation is a less realistic choice for most (concurrent) logic programming languages because in those languages, the codes must prepare for many possible exceptional situations.

Our mode system could be understood in the framework of abstract interpretation, though we believe the constraint-based presentation is simpler and more comprehensive for our system. The assumptions on our programming conventions (Section 2) enabled us to compute modes in a way similar to the unification of rational trees. Iterative computation of fixpoints, which is used in abstract interpretation to capture the properties of recursive programs, is thus avoided.

Some concurrent logic languages such as Strand [6] 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 $m(\langle :=, 1\rangle) = out$ and (consequently) $m(\langle :=, 2\rangle) = in$.

Concurrent languages Doc [7], $\mathcal{A}'\mathcal{UM}$ [26] and Janus [13] 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, the consistency of which needs to be checked statically or dynamically. The annotations 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, Kenji Horiuchi, Mark Korsloot, Evan Tick, Ehud Shapiro, and anonymous referees for valuable discussions, comments and suggestions.

References

[1] Aït-Kaci, H. and Nasr, R., LOGIN: A Logic Programming Language with Built-In Inheritance. *J. Logic Programming*, Vol. 3, No. 3 (1986), pp. 185–215.

[2] Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.

[3] Colmerauer, A., Prolog and Infinite Trees. In *Logic Programming*, Clark, K. L. and Tärnlund, S. -Å. (eds.), Academic Press, London, 1982, pp. 231–251.

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

[5] Bowen, D. L. (ed.), Byrd, L., Pereira, F. C. N., Pereira, L. M. and Warren, D. H. D., *DECsystem-10 Prolog User's Manual.* Dept. of Artificial Intelligence, Univ. of Edinburgh, 1983.

[6] 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, Cambridge, MA, 1989, pp. 497–512.

[7] 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.

[8] Jaffar, J., Efficient Unification over Infinite Terms. *New Generation Computing*, Vol. 2, No. 3 (1984), pp. 207–219.

[9] 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.

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

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

[12] Pereira, F. C. N., Grammars and Logics of Partial Information. In *Proc. Fourth Int. Conf. on Logic Programming*, Lassez, J. -L. (ed.), MIT Press, 1987, pp. 989–1013.

[13] Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conference on Logic Programming*, Debray, S. and Hermenegildo, M. (eds.), MIT Press, 1990, pp. 431–446.

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

[15] Shapiro, E., The Family of Concurrent Logic Programming Languages. *Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.

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

[17] 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, Amsterdam, 1988, pp. 441–456.

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

[19] Ueda, K., Designing A Concurrent Programming Language. In *Proc. InfoJapan'90*, Information Processing Society of Japan, 1990, pp. 87–94.

[20] Ueda, K. and Chikayama, T., Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.

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

[22] Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, Warren, D. H. D. and Szeredi, P. (eds.), MIT Press, 1990, pp. 3–17.

[23] Ueda, K. and Morita, M., Message-Oriented Parallel Implementation of Moded Flat GHC. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992*, ICOT, Tokyo, 1992, pp. 799–808. Revised version in *New Generation Computing*, Vol. 11, Nos. 3–4 (1993), pp. 323–341.

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

[25] Yardeni, E. and Shapiro, E., A Type System for Logic Programs. In [14], Vol. 2, pp. 211–244.

[26] 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, Tokyo, 1988, pp. 638–649. Also in *New Generation Computing*, Vol. 7, Nos. 2–3 (1990), pp. 127–157.

Appendix. Normalization of Clauses [21]

We first consider the normalization of program clauses. Let the program clause to be normalized be

$$h \text{ :- } G_U \cup G_N \mid B_U \cup B_N,$$

where h is the clause head, G_U the multiset of unification guard goals, G_N the multiset of non-unification guard goals, B_U the multiset of unification body goals, and B_N the multiset of non-unification body goals. If $G_U \cup B_U$ is unsolvable, the clause is not normalizable. We do not define the well-modedness of a program with an unnormalizable clause.

Otherwise, we first 'execute' G_U and the clause is changed into

$$h\sigma \text{ :- } G_N\sigma \mid B_U\sigma \cup B_N\sigma,$$

where σ is an idempotent mgu of the simultaneous equations G_U.

Then we 'execute' the unification body goals $B_U\sigma$. Let θ be an idempotent mgu (that can be represented as a set of bindings of the form $v \leftarrow t$) of $B_U\sigma$ which satisfies the condition

$$\forall (v \leftarrow t) \in \theta \left(v \in \mathcal{V}_{h\sigma} \wedge t \in \mathit{Var} \Rightarrow t \in \mathcal{V}_{h\sigma} \right),$$

where $\mathcal{V}_a$ stands for the set of variables occurring in the atom a. In other words, no variable-to-variable binding in θ can replace a variable in $h\sigma$ by a variable not in $h\sigma$. The mgu θ with this property can be constructed from an arbitrary idempotent mgu of $B_U\sigma$ [21].

Let $\eta|_a$ be the restriction of a substitution η to an atom a, which is defined as

$$\eta|_a = \left\{ (v \leftarrow t) \in \eta \;\middle|\; v \in \mathcal{V}_a \right\}.$$

Also, when η is idempotent and is of the form $\bigcup_{i=1}^{n} \{v_i \leftarrow t_i\}$, by $\overline{\eta}$ we denote the set of equations $\bigcup_{i=1}^{n} \{v_i = t_i\}$, which obviously has an idempotent mgu η. Then $\overline{\overline{\theta|_{h\sigma}}}$ stands for the output unification goals of the clause. So the normalized clause is

$$h\sigma \text{ :- } G_N\sigma \mid \overline{\overline{\theta|_{h\sigma}}} \cup B_N\sigma\theta.$$

Note that θ need not be applied to the guard part, because the guard part has been reduced when unification body goals are executed.

To normalize a goal clause of the form

$$\text{:- } B_U \cup B_N,$$

we simply 'execute' B_U. If B_U is unsolvable, the clause is not normalizable. Otherwise, the normalized clause is

$$\text{:- } B_N\theta,$$

where θ is an idempotent mgu of B_U.