ICOT Technical Report: TR-468

TR-468

Parallel Unification and Meta-Interpreters in GHC

by
H. Fujita

March, 1989

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

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Parallel Unification and Meta-Interpreters in GHC

Hiroshi Fujita

Institute for New Generation Computer Technology,
1-4-28 Mita, Minato-ku, Tokyo, 108, Japan
fujita%icot.jp@relay.cs.net

ABSTRACT

This paper presents a new unification program in GHC. The program is designed for obtaining the maximal efficiency by making the best of parallelism inherent in the unification problem. It employs *stream parallelism* together with a useful programming technique called *short circuit* for detecting stable properties of networks. Efficient meta-interpreters in which an object variable is represented by a ground term are developed by incorporating this unification program. Superiority of the parallel unification program over more sequential ones becomes more evident as it is used in larger applications.

1. Introduction

It is widely accepted that logic programming is congenial to meta-programming. The experience in Prolog seem to support this thesis. The only shortcomming of meta-programming is inefficiency of execution due to layers of interpretation, which, however, can be remedied by applying partial evaluation.

Is meta-programming successful in concurrent logic languages such as GHC too? This question is the primary motivation of the research reported in this paper.

It may be difficult, within GHC, to obtain the efficiency competitive to that obtainable in Prolog. One big problem is concerning representation of a variable from different levels of a meta-program. The overall performance of a meta-program strongly depends on how suitably variables are represented and how easily they can be manipulated by the meta-program, for a great deal of execution time and space is spent on handling variables. In particular, unification plays a central role in every logic program, whether the language used is Prolog or GHC, and its efficiency dominates the total performance of the program.

In a meta program written in Prolog, an object variable can be represented by a logical variable of the meta-program without any serious problem of confusion in many cases. This makes unification of object terms containing object variables very efficient, since

the unification can be performed by the underlying system. In addition to this, the `var` and the `nonvar` predicates in Prolog may be used effectively to determine whether an object variable is bound or not. However, from the semantical viewpoint, this way of handling object variables is inadequate, for it is not guaranteed that the above method always works correctly.

Moreover, the above method can no longer be applied to a meta-program within GHC. For instance, although the `wait` predicate in GHC, which corresponds to the `nonvar` predicate, can be used as a guard goal in order to make sure that a variable is instantiated, there is no counterpart of the `wait` predicate, which would correspond to the `var` predicate.

Therefore, we should adopt the orthodox method, such as those in [Llo88], to handle object variables in meta-programs properly. In this paper, the *ground representation* is employed. Then, the problem is how to obtain an efficient implementation of a meta-program in the ground representation. Here, I claim that the implementation of an efficient unification program is the key to solving this problem.

Section 2 presents the parallel unification program. Correcness is proved of an abstracted algorithm for the program and its complexity is analysed in Section 3. Section 4 presents a GHC meta-interpreter which employs the unification program. Section 5 gives the performance results. Section 6 concludes the paper.

2. The Parallel Unification Program

A number of unification algorithms have been developed with working implementations in the literature. In particular, very efficient algorithms of unification are known, such as the one in [MM82], which are especially suitable for sequential implementation. However, it may still be possibile to develop more efficient implementation than conventional ones by extracting maximum parallelism inherent in the unification problem.

In this section, a parallel unification program is developed in GHC.

(a) *functional specification*

The unification goal, `unify(X,Y,Res)`, given two terms, `X` and `Y`, returns `Res`, which is a set of bindings as a *mgu* of `X` and `Y` when they are unifiable, `fail` otherwise.

(b) *programming paradigm: generate-and-test plus stream parallelism*

If the input terms are composit terms, more than one binding may be generated in parallel. A collection of bindings generated may be inconsistent since they will be given

independently of one another. Therefore the `unify` goal should be divided into two processes as follows:

```
unify(X,Y,Res) :- true | generate(X,Y,S), test(S,Res).
```

where `S` is a stream of bindings.

(c) *feedback loop*

It may be possible that two bindings, say $v = t_1$ and $v = t_2$, for an identical variable, v, are passed onto the `test` process. In this case a new unification problem, $t_1 = t_2$, arises and a new set of bindings may be generated as the solution for this subproblem. Hence the `unify` clause should be modified to:

```
unify(X,Y,Res) :- true |
   generate(X,Y,S1), merge(S1,S2,S3), test(S3,S2,Res).
```

where `S2` is the stream of the bindings generated within the `test` process. The bindings given in `S2`, as those in `S1`, may be inconsistent. Therefore `S2` must be fed back to the `test` process. Thus the `merge(S1,S2,S3)` process is inserted to obtain `S3` which becomes the total input of the `test` process.

(d) *how to terminate in failure*

Every unification subprocess under the `generate` or the `test` process is a candidate where failure may be detected. Failure detected in any process implies total failure of the initial unification goal. Hence, once failure is detected in some process, it is desired that every process spawned under the initial goal is killed immediately.

(e) *how to terminate in success*

When all the processes terminate with no failure detected at any process, it is known that the initial unification goal has succeeded. At this point, important questions arise. How can one be assured that every process always terminates when the given terms are unifiable? Is not there any possibility of an infinite process? Does not the feedback loop lead the processes to deadlock which cannot be escaped from?

(f) *signaling system used for smart termination*

In order to solve the above problems of termination, let us introduce a unique mechanism called *signaling system*. The use of signaling system is considered as a basic programming technique in concurrent programming, which is indeed as useful and indispensable as streams.

Fig. 2.1 shows the `unify/3` code in the actual implementation, where two kinds of signaling systems are employed: an *abort signal* and a *short circuit*.

```
unify(X,Y,Res) :- true |
    unify(X,Y,S1,sw(S,success),Abort),
    merge(S1,S2,InS,Abort),
    sift(InS,OutS,S2,Abort),
    unify_result(S,Abort,OutS,Res).

    unify_result(_,abort(all),_,Res) :- true | Res=fail.
    unify_result(success,Abort,OutS,Res) :- true | Res=OutS,
                            Abort=abort(filters).
```

Fig. 2.1 The `unify/3` code in the actual implementation

An abort signal is commonly used for detecting some special event which may occur among cooperating processes, and for forcing processes to quit when the event does occur. In Fig. 2.1, the variable, `Abort`, will be instantiated to the `abort(all)` signal by some subprocess when failure of unification is detected. Usually it is assumed that an abort signal is issued at most once by a single process, and all the other processes are forced to observe this signal as early as possible. However, it is not necessarily harmful to allow for more than one process to activate an identical signal line (multiple write on a shared variable), if no signal of different types are issued on it (no different values are assigned to the shared variable).

A short circuit is commonly used for detecting a stable property of a network of cooperating processes [Tak83, SWKS88]. A pair of variables called a *switch* is delivered to each process in the network. The terminals (variables) of a switch may be shared by neighboring processes. The cooperating processes are then virtually lined in a chain according to the connection by the shared variables. A switch will be closed or shorted (two variables are unified) when some condition is met in the process holding the switch. When all the switches held by the processes on the chain are closed, terminals that occur at either side of the chain are shortcircuit. This makes it possible to let a process, which is holding terminals at both sides of the chain, know the fact that every condition has been met in the responsible process on the chain. In Fig. 2.1, the switch, `sw(S,success)`, is given to the `unify/5` process. When this switch is shortcircuit, `S` is instantiated to `success` and the `unify_result` process becomes able to bind `Res` to `OutS`.

It is obvious that the nondeterministic choice made by the `unify_result` according to the two signals, `Abort` and `S`, if issued correctly, will never lead the program to any incorrect result, since the two results, success or failure, of unification are exclusive to one another.

The `Abort` signal is in fact used for multipurpose: to detect the failure, to force the unnecessary processes to quit after the failure is detected in some process, and to force the

```
unify(_,_,_,_,abort(_)) :- true | true.
unify(X,X,OutS,sw(A,Z),_) :- true | OutS=[], A=Z.
unify(V,T,OutS,SW,_) :- V=var(_), V\=T |
          OutS=[bind(V,T,T,Tf,SW)].
unify(T,V,OutS,SW,_) :- V=var(_), V\=T |
          OutS=[bind(V,T,T,Tf,SW)].
unify(X,Y,OutS,SW,Abort) :- X\=var(_), Y\=var(_) |
   functor(X,F,N), functor(Y,G,M),
   unify_functor(F/N,G/M,X=Y,OutS,SW,Abort).

   unify_functor(_,_,_,_,_,_,abort(_)) :- true | true.
   unify_functor(F_N,G_M,_,_,_,Abort) :- F_N\=G_M |
                              Abort=abort(all).
   unify_functor(F/N,F/N,X_Y,OutS,SW,Abort) :- true |
      unify_arg(N,X_Y,OutS,SW,Abort).

      unify_arg(_,_,_,_,_,abort(_)) :- true | true.
      unify_arg(N,X=Y,OutS,sw(A,Z),Abort) :-
                N>0 | N1:=N-1, arg(N,X,X1), arg(N,Y,Y1),
         unify(X1,Y1,S1,sw(A,M),Abort),
         merge(S1,S2,OutS,Abort),
         unify_arg(N1,X=Y,S2,sw(M,Z),Abort).
      unify_arg(0,_,OutS,sw(A,Z),_) :- true | OutS=[], A=Z.
```

Fig. 2.2 The unify/5 code and its subprocedures

processes in the feedback loop, which may be in deadlock, to quit. The abort(filters) signal issued by the unify_result process is for the last purpose above. More details on this mechanism will be explained later.

(g) *the* unify/5 *process as the generator*

Fig. 2.2 shows the unify/5 code and its subprocedures.

In what follows, an object variable is represented by a ground term of the form, var(V), where V is an identifier assigned to each distinct variable.

The unify(X,Y,OutS,sw(A,Z),Abort) process closes the switch, sw(A,Z), if the input terms, X and Y, are equal. If either X or Y is a variable, and let V be the variable and T be the other term, then the unify/5 process enters a quintuple of the form, bind(V,T,T,Tf,SW), into its output stream, OutS, as the binding information for the variable, V. The third element of the quintuple, which is initially set to T, is a place where the intermediate value of T is put as the dereference operation proceeds incrementally. The fourth element of the quintuple, which remains unbound until the unification

terminates in success, is to be the final value of the variable, V.

It should be noted that the creation of the binding information for a variable may not immediately mean the successful termination of the unification subproblem for this particular variable, since it may give rise to creation of further subproblems at some time in the `sift` process. Therefore the switch, `SW` (= `sw(A,Z)`), for which the `unify/5` process should have been responsible, is thrown out enclosed with the binding information, thereby shifting the responsibility onto some other process. A switch, which is released from its holder process and is transfered from process to process as a data, is called an *embedded switch*. This trick is in fact the key to the successful termination of the parallel unification program.

If `X` and `Y` are not equal and none of them is a variable, the two terms are decomposed into subterms. (An atomic term is assumed to be a composed term of a null-ary function symbol.) The `unify_functor` process determines whether the given terms are unifiable or not by comparing their primary function symbol and arity. If the comparison results in failure, the `abort(all)` signal is broadcasted through the signaling system, `Abort`. If the comparison results in success, the `unify_arg` process is spawned, which in turn spawns a `unify/5` process for each pair of the subterms. The switch, `sw(A,Z)`, held by the initial `unify/5` is split to the subswitches, each of which is delivered to the new `unify/5` subprocesses in the manner that these subprocesses comprise a chain. The output streams of quintuples from these subprocesses are merged into a single stream.

(h) *the* `sift` *and the* `filter` *processes as the tester*

Fig. 2.3 shows the `sift` codes and the `filter` codes.

The `sift(InS,OutS,OutS2,Abort)` process, observing a quintuple in the input stream, `InS`, passes it onto the main output stream, `OutS`. At the same time, the `sift` process spawns the `filter` process and assigns the input quintuple to it. `OutS2` is the suboutput stream for collecting quintuples which may be created by the `unify/5` subprocesses spawned from within the `filter` processes. Also at the same time, the embedded switch, `sw(A,Z)`, in the quintuple is closed. This is because the quintuple that reaches the `sift` process must already have undergone all the necessary trials performed by the `filter` processes.

The `filter(InS,V,VT,VTm,VTf,OutS,OutS2,Abort)` process, holding the variable, `V`, its initial value, `VT`, intermediate value, `VTm`, and the final value, `VTf`, looks at the input stream of quintuples, `InS`, and passes only appropriate quintuples onto the main output stream, `OutS`.

```
sift(_,_,_,abort(all)) :- true | true.
sift(_,OutS,_,abort(filters)) :- true | OutS=[].
sift([bind(V,T,Tm,Tf,sw(A,Z))|InS],OutS,OutS2,Abort) :-
     true | A=Z,
   sift_final(V,Tf,OutS,OutS1),
   filter(InS,V,T,Tm,Tf,InS1,OutS21,Abort),
   merge(OutS21,OutS22,OutS2,Abort),
   sift(InS1,OutS1,OutS22,Abort).

   sift_final(V,V,OutS,OutS1) :- true | OutS=OutS1.
   sift_final(V,T,OutS,OutS1) :- V\=T | OutS=[V=T|OutS1].

filter(_,_,_,_,_,_,_,_,abort(all)) :- true | true.
filter(_,_,_,Tm,Tf,_,_,abort(filters)) :- true | Tf=Tm.
filter([bind(V,VT1,_,VTf1,SW)|InS],
       V,VT,VTm,VTf,OutS,OutS2,Abort) :- true |
   VTf1=VTf,
   unify(VT1,VT,OutS21,SW,Abort),
   merge(OutS21,OutS22,OutS2,Abort),
   filter(InS,V,VT,VTm,VTf,OutS,OutS22,Abort).
filter([bind(U,V,_,UTf,SW)|InS],
       V,VT,VTm,VTf,OutS,OutS2,Abort) :- true |
   UTf=VTf,
   OutS=[bind(U,VT,VTm,VTf,SW)|OutS1],
   filter(InS,V,VT,VTm,VTf,OutS1,OutS22,Abort).
filter([bind(U,UT,UTm,UTf,SW)|InS],
       V,VT,VTm,VTf,OutS,OutS2,Abort) :- U\=V, UTm\=V |
   deref(UTm,V=VTf,UTm1,Abort),
   deref(VTm,U=UTf,VTm1,Abort),
   OutS=[bind(U,UT,UTm1,UTf,SW)|OutS1],
   filter(InS,V,VT,VTm1,VTf,OutS1,OutS2,Abort).
```

Fig. 2.3 Codes for `sift` and `filter`

If the input quintuple is `bind(V,VT1,VTm1,VTf1,SW)`, `VTf1` and `VTf` are unified by the builtin unification, and the new `unify/5` process is spawned to unify `VT1` and `VT`. Also the embedded switch, `SW`, in the quintuple must be inherited to the new `unify/5` process.

If the input quintuple is `bind(U,V,UTm,UTf,SW)`, `UTf` and `VTf` are unified by the builtin unification, and the modified quintuple, `bind(U,VT,VTm,VTf,SW)`, is given as the output.

If the input quintuple is `bind(U,UT,UTm,UTf,SW)` such that neither `U` nor `UT` is identical to `V`, then the modified quintuple, `bind(U,UT,UTm1,UTf,SW)`, is given as the output. At the same time, the two bindings, `V`←`VTf` and `U`←`UTf`, are applied to each other's

intermediate value, UTm and VTm, obtaining the new ones, UTm1 and VTm1, by deref procedure.

The sift_final process is just for eliminating redundant bindings such as $v \leftarrow v$ that may pass through the sift process.

(i) *deadlock detection and elimination*

Now, let us turn to the problem of deadlock detection and elimination.

How and when processes come to deadlock? It occurs after all the quintuples passed through the sift process. After that, no unify/5 process will remain, whereas the sift process and several filter and merge processes may remain in the feedback loop waiting, in vain, an input that will no longer be supplied by any process.

This deadlock does not mean failure of the initial unification goal in any sence. On the contrary, this deadlock is in fact possible only when the unification must succeed. The unification problem is finitely solvable under the assumption taken in this paper that no terms are given which need *occurs check*. Hence, there should be some way to detect completion of unification and to eliminate the deadlock. But how?

The exhaution of quintuples in the feedback loop can be detected by observing that all the switches embedded in the quintuples are closed. The switches other than the embedded ones, which are also the subswitches split from the parent switch in the initial unify/5, should have already been closed within the unify/5 processes that terminated in success. Hence, the unify_result process can be assured of successful termination of the unification when it observes the success signal, thereby issuing the abort(filters) signal to force every process that remains in deadlock to quit.

On observing the abort(filters) signal, every remaining filter process unifies VTm and VTf by the builtin unification, then terminates; the sift process closes the main output stream, OutS, then terminates; and every remaining merge process simply terminates.

Note that some deref processes may still be performing their task after the success signal is broadcasted. These deref processes must not of cource be killed at this point. The detection of the termination of the deref processes, if required, is possible by using another short circuit. However, it is not absolutely necessary since they will eventually terminate in any event.

Due to limited space, the deref and the merge codes are ommitted.

3. Correctness and Complexity

The correctness of the unification program might be stated as follows:

Proposition 3.1. *(Correctness of the parallel unification program)*

(i) The unification program always terminates.

(ii) If X and Y are unifiable, `unify(`X`,`Y`,`θ`)` *gives $\theta = [v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n]$ as a most general unifier of X and Y. If X and Y are not unifiable,* `unify(`X`,`Y`,`θ`)` *gives $\theta = fail$.*

The paper is not presenting the direct proof of Proposition 3.1 in a way of verifying GHC codes line by line, for there is not, at present, any formal method suitable to do that. Instead, let us investigate on several properties essential to the parallel unification program on the basis of more abstract setting.

(a) *abstract algorithm*

First of all, it will be convenient to address a parallel unification algorithm, of which implementation has already been presented, in a more abstract form.

Fig. 3.1 shows Algorithm 1. In the algorithm, Rule 1 and 2 are called *decomposition*, and Rule 6,7 and 8 are called *dereference*.

Algorithm 1 might be found basically the same as the one in [MM82]. However, they are quite different in that in our algorithm, Rule 1-8 may be performed not only nondeterministically but also in parallel. More precisely, even decomposition (*term reduction* in their terminology) and dereference (*variable elimination*) may be done in parallel. In particular, dereference is no longer an atomic or indivisible operation here.

Nevertheless, the following theorem for the correctness of Algorithm 1 is proved similarly along the line of [MM82].

Theorem 3.2. *Given a pair of terms t_1 and t_2,*

(i) Algorithm 1 always terminates, no matter which choices are made.

(ii) If Algorithm 1 terminates with failure, t_1 and t_2 have no unifier. If Algorithm 1 terminates with success, t_1 and t_2 are unifiable and θ is a mgu of them.

Before presenting the proof of Theorem 3.2, several lemmas need to be given.

Lemma 3.3. *(Decomposition preserves mgu)*
Let E be a set of equations. If Rule 1 is applicable, then E have no unifier. If Rule 2 is applicable, and let E' be obtained by applying Rule 2, then E' gives the same mgu as E, if it exists.

Algorithm 1 (Parallel nondeterministic unification)

Given a pair of terms, t_1 and t_2;

PROCEDURE

E: a set of equations := $\{t_1 = t_2\}$;

S: a set of quadruples := $\{\}$;

Repeat applying the following rules in parallel and nondeterministically;

1) if $\exists\ s = t \in E \quad (s = f^n(\ldots) \ \wedge\ t = g^m(\ldots) \ \wedge\ f^n \neq g^m)$,
   then exit with *fail*;
2) if $\exists\ s = t \in E \quad (s = f^n(s_1, \ldots, s_n) \ \wedge\ t = f^n(t_1, \ldots, t_n) \ \wedge\ n \geq 0)$,
   then remove $s = t$ from E, and enter $s_i = t_i (1 \leq i \leq n)$ into E;
3) if $\exists\ v = t \in E \quad (v = t \ \wedge\ v : variable)$,
   then remove $v = t$ from E;
4) if $\exists\ v = t \in E \quad (v \neq t \ \wedge\ v : variable)$,
   then remove $v = t$ from E, and enter $\langle v, t, t, \bigcirc_v \rangle$ into S;
5) if $\exists\ t = v \in E \quad (v \neq t \ \wedge\ v : variable)$,
   then remove $t = v$ from E, and enter $\langle v, t, t, \bigcirc_v \rangle$ into S;
6) if $\exists \langle u, s, s^\circ_u, \bigcirc_u \rangle, \langle v, t, t^\circ_v, \bigcirc_v \rangle \ \in S \quad u = v$,
   then equate $\bigcirc_u$ to $\bigcirc_v$, remove $\langle u, s, s^\circ_u, \bigcirc_u \rangle$ from S, and
   enter $s = t$ into E;
7) if $\exists \langle u, s, s^\circ_u, \bigcirc_u \rangle, \langle v, t, t^\circ_v, \bigcirc_v \rangle \ \in S \quad s = v$,
   then equate $\bigcirc_u$ to $\bigcirc_v$, remove $\langle u, s, s^\circ_u, \bigcirc_u \rangle$ from S, and
   enter $\langle u, t, t^\circ_v, \bigcirc_v \rangle$ into S;
8) if $\exists \langle u, s, s^\circ_u, \bigcirc_u \rangle, \langle v, t, t^\circ_v, \bigcirc_v \rangle \ \in S \quad (u \neq v \wedge s \neq v \wedge t \neq u)$,
   then remove $\langle u, s, s^\circ_u, \bigcirc_u \rangle$ and $\langle v, t, t^\circ_v, \bigcirc_v \rangle$ from S, and
   enter $\langle u, s, s^\circ_u \{v \leftarrow t^\circ_v\}, \bigcirc_u \rangle$ and $\langle v, t, t^\circ_v \{u \leftarrow s^\circ_u\}, \bigcirc_v \rangle$ into S;

until a condition is met such that none of Rule 1-7 is applicable and no application of Rule 8 results in any change to S;

For $\forall\ \langle v, t, t^\circ_v, \bigcirc_v \rangle \in S$, do equate $\bigcirc_v$ to t°_v;

Exit with $\theta = \{\ v \leftarrow \bigcirc_v \mid\ \langle v, t, t^\circ_v, \bigcirc_v \rangle \in S\ \}$;

where t°_v is the intermediate value for v and $\bigcirc_v$ is the meta-variable associated to each distinct variable, v.

Fig. 3.1 An abstract algorithm of parallel unification

Proof is ommitted (readers are refered to Theorem 2.1 in [MM82].)

A pair of quadruples, $\langle u, s, s_u^\circ, \bigcirc_u \rangle$ and $\langle v, t, t_v^\circ, \bigcirc_v \rangle$, in S such that $u = v$, recognozed in Rule 6, is called a *critical pair*. A pair of quadruples, $\langle u, s, s_u^\circ, \bigcirc_u \rangle$ and $\langle v, t, t_v^\circ, \bigcirc_v \rangle$, in S such that $s = v$, recognized in Rule 7, is called a *connected pair*. A pair of quadruples, $\langle u, s, s_u^\circ, \bigcirc_u \rangle$ and $\langle v, t, t_v^\circ, \bigcirc_v \rangle$, in S such that $u \neq v \wedge s \neq v \wedge t \neq u$, recognized in Rule 8, is called a *regular pair*. Note that no pair can be both critical and connected in S due to Rule 3.

LEMMA 3.4. *(Dereference preserves mgu)*
Let E be a set of equations, and let S be a set of quadruples, $\{\langle v, t_v, t_v^\circ, \bigcirc_v \rangle\}$. If Rule 6 is applicable, and let E' and S' be obtained by applying Rule 6, then E' and S' gives the same mgu as E and S, if it exists. If Rule 7 (8) is applicable, and let S' be obtained by applying Rule 7 (8), then S' gives the same mgu as S, if it exists.

PROOF.
Case Rule 6: If a pair, $\langle u, s, s_u^\circ, \bigcirc_u \rangle$ and $\langle v, t, t_v^\circ, \bigcirc_v \rangle$, is recognized in S by Rule 6, the pair is a critical pair and $u = v$. Then, if there exists for E and S a mgu θ such that $u\theta = v\theta = s\theta$ and $v\theta = t\theta$ so does for E' and S' after Rule 6 is applied, and vice versa.
Case Rule 7: If a pair, $\langle u, s, s_u^\circ, \bigcirc_u \rangle$ and $\langle v, t, t_v^\circ, \bigcirc_v \rangle$, is recognized in S by Rule 7, the pair is a connected pair and w.l.o.g. we can assume $s = v$. Then, if there exists for S a mgu θ such that $u\theta = s\theta = v\theta$ and $v\theta = t\theta$ so does for S' after Rule 7 is applied, and vice versa.
Case Rule 8: If a pair, $\langle u, s, s_u^\circ, \bigcirc_u \rangle$ and $\langle v, t, t_v^\circ, \bigcirc_v \rangle$, is recognized in S by Rule 8, the pair is a regular pair and $u \neq v \wedge s \neq v \wedge t \neq u$. Then, if there exists for S a mgu θ such that $u\theta = s\theta$ and $v\theta = t\theta$ so does for S' after Rule 8 is applied, and vice versa. □

A set of quadruples, S, is said to be in *solved form* iff it satisfies the following conditions:
(i) neither critical pair nor connected pair is in S;
(ii) no variable, v, that occurs as the leftmost element of some quadruple, $\langle v, t, t_v^\circ, \bigcirc_v \rangle$, occurs in any intermediate value, t_u°, in S.

LEMMA 3.5. *Let S be a set of quadruples, $\{\langle v, t, t_v^\circ, \bigcirc_v \rangle\}$, and S be in solved form. There exists a mgu θ such that $v\theta = t\theta$ for every quadruple in S iff does so a mgu σ such that $\bigcirc_v\sigma = t_v^\circ\sigma$ in S.*

Proof is ommitted.

A term, t, is said to be *meta-ground*, if no meta-variable occurs in t. A term, $t^\bullet$, is called a *meta-instance* of another term, t°, if there exists a substitution, $\sigma = \{\bigcirc_i \leftarrow s_i\}$, for a set of meta-variables, $\{\bigcirc_i\}$, such that $t^\bullet = t^\circ\sigma$.

LEMMA 3.6. *Let T be a set of pairs, $\{\langle \bigcirc_i, t_i^\circ \rangle\}$, of meta-variables, $\bigcirc_i$, and terms, t_i°. If all of the following condition are met:*

(i) any meta-variable that occurs as the left member of some pair in T does occur as the left member of no other pair in T;

(ii) every right mebmer, t_i°, in T is not a meta-variable;

(iii) any meta-variable that occurs in some right member t_i° in T does occur as the left member of some pair in T;

then, for every t_i°, there exists the term, $\{t_i^\bullet\}$, such that every $t_i^\bullet$ is meta-ground and is a meta-instance of t_i°.

PROOF. *The thesis is easily proved by structural induction on terms.* □

Now we can prove Theorem 3.2.

PROOF. (of Theorem 3.2)

(i) Let us define a function F mapping any pair of a set of equations, E, and a set of quadruples, $S = \{(v, t, t_v^\circ, \bigcirc_v)\}$, into a quadruple of natural numbers, (n_1, n_2, n_3, n_4). The first number, n_1, is the sum of the number of critical pairs and connected pairs in S. The second number n_2, is the total number of occurrences of function symbols in E. The third number, n_3, is equal to $2|E| + |S|$. The fourth number, n_4, is the total number of variables (but meta-variables) that occur in t_v° in S. Let us define a total ordering on such quadruples as follows:

$$(n_1', n_2', n_3', n_4') < (n_1, n_2, n_3, n_4) \text{ if } n_1' < n_1$$
$$\text{or } n_1' = n_1 \text{ and } n_2' < n_2$$
$$\text{or } n_1' = n_1 \text{ and } n_2' = n_2 \text{ and } n_3' < n_3$$
$$\text{or } n_1' = n_1 \text{ and } n_2' = n_2 \text{ and } n_3' = n_3 \text{ and } n_4' < n_4$$

With the above ordering, N^4 becomes a well-founded set. Thus, if we prove that any Rule of Algorithm 1 transforms a pair, $\langle E, S \rangle$, in a pair, $\langle E', S' \rangle$ such that $F(\langle E', S' \rangle) < F(\langle E, S \rangle)$, we have proved the termination. In fact, Rule 1, if applied, always decreases n_2 and n_3. Rule 2 can possibly increase n_3, but it surely decreases n_2. Rule 3 always decreases n_3. Rule 4 and 5 always decrease n_3 and possibly decrease n_2. Rule 6 can possibly increase n_2 and increase n_3, but it surely decrease n_1. Rule 7 always decreases n_1. Rule 8 always decreases n_4 as long as it causes a change in S.

(ii) If Algorithm 1 terminates with failure, the thesis immediately follows from Lemma 3.3. If Algorithm 1 terminates with success, Rule 3,4 and 5 clearly do not change the set of unifiers, while for Rule 2,6,7 and 8 this fact is stated in Lemma 3.3 and 3.4. Finally, E is empty and S is in solved form. In fact, if Rule 6 and 7 cannot be applied, it

means that neither critical pair nor connected pair is in S. If Rule 8 cannot be applied, it means that no variable, v, which occurs as the leftmost member of some quadruple, $(v, t.t_v^\circ, \bigcirc_v)$, occurs in any t_u° in S. After equating every $\bigcirc_{v_i}$ to corresponding $t_{v_i}^\circ$ in S, Lemma 3.5 and 3.6 follws that $\theta = \sigma\{\bigcirc_{v_i} \leftarrow t_{v_i}^\circ\} = \sigma\{\bigcirc_{v_i} \leftarrow t_{v_i}^\bullet\}$ and θ is a mgu of S hence of initial $E - \{t_1 = t_2\}$. □

(b) *deadlock freedom*

The parallel unification program introduces the enevitable deadlock due to the feedback loop of a stream. However, this deadlock can be eliminated by utilizing some stable properties in the process network and by employing appropriate signaling systems as in the way described in the previous section. Nevertheless, deadlock freedom must be proved of the overall behavior of the program in order to prove the total correctness of the program.

(c) *idempotent mgu*

The mgu, θ, given by Algorithm 1 and the program is in fact *idempotent*, that is, $\theta\theta = \theta$ in the sense of substitution composition.

(d) *non-redundant mgu*

The mgu, θ, given by Algorithm 1 and the program is in fact *non-redundant*, that is, no binding such as $v \leftarrow v$ for any variable, v, occurs in θ.

(e) *complexity*

Finally, it should be worth noting that in the parallel unification program, at most one `filter` process is created for a distinct variable appearing in the input terms. Moreover, all the pairs of quintuples for distinct variables are checked once and only once.

This makes the parallel unification program, in which pipelined processing is performed by the chain of `filters`, to take only $O(n)$ steps in checking all the pairs of quintuples, whereas the more sequential one, in which the checks are performed in the naive manner, takes $O(n^2)$ steps, hence, the estimated factor of efficiency increase is $O(n)$, where n is the number of occurrences of variables in the input terms.

4. A Flat-GHC Meta-interpreter

This section presents a Flat-GHC meta-interpreter incorporating the parallel unification program. The unification in the guard part becomes rather complicated and the program needs to be modified according to the synchronization rules of GHC [Ued86]. For

```
exec(Q,Res) :- true |
   solve_body(Q,top,S3,S1,sw(T,terminate),Abort),
   merge(S1,S2,InS,Abort),
   sift(InS,OutS,S2,S3,Abort),
   exec_result(T,Abort,OutS,Q,Res).

   exec_result(_,abort(all),_,_,Res) :- true | Res=fail.
   exec_result(terminate,Abort,OutS,Q,Res) :- true |
      Abort=abort(filters), apply_subst(Q,OutS,Res).

solve_body(_,_,_,_,_,_,abort(_)) :- true | true.
solve_body(true,_,_,OutS,sw(A,Z),_) :- true | OutS=[], A=Z.
solve_body(X=Y,_,_,OutS,SW,Abort) :- true |
   unify(X,Y,OutS,SW,Abort).
solve_body((Q1,Q2),Path,InS,OutS,sw(A,Z),Abort) :- true |
   solve_body(Q1,1(Path),InS,OutS1,sw(A,M),Abort),
   merge(OutS1,OutS2,OutS,Abort),
   solve_body(Q2,r(Path),InS,OutS2,sw(M,Z),Abort).
solve_body(Q,Path,InS,OutS,SW,Abort) :-
      Q\=true, Q\=(_=_), Q\=(_,_) |
   clauses(Q,Clauses),
   resolve(Clauses,0,Q,Path,InS,Winner,AbortC),
   commit(Winner,AbortC,InS,OutS,SW,Abort).
```

Fig. 4.1 The Flat-GHC meta-interpreter (body part)

brevity's sake, we assume that unification, `_=_`, is the only builtin predicate both within the guard part and the body part of a program clause.

Fig. 4.1 shows codes for solving the top level goal and the body part of a program clause.

The top level goal and its descendants created in the cource of reductions can be considered as a single unification task as a whole, since in a GHC execution only a single solution is necessary and sufficient in the sense of logic. A GHC goal (process) may eventually be reduced to a set of unification goals, hence a goal can be considered as a generator of bindings. Collecting all the bindings generated by these processes, a single chain of filters for the bindings is constructed to obtain the consistent set of bindings as the final solution.

Fig. 4.2 shows codes for resolving a goal by the program clauses and for the commit operation.

In principle, every clause whose head is of the same function symbol and arity as the given goal is tried in parallel to resolve the goal. Every clause becomes a candidate for

```
resolve([Cl|Clauses],N,Q,Path,InS,Winner,AbortC) :-
      true | N1:=N+1,
   copy(Cl,Path-N1,(H:-G|B)),
   unify_guard(H,Q,InS,SH,sw(S,M),AbortG),
   merge(InS,S3,S0,AbortG),
   solve_guard(G,S0,SG,sw(M,satisfied),AbortG),
   merge(SH,SG,S1,AbortG),
   merge(S1,OutS2,S2,AbortG),
   sift(S2,OutS,OutS2,S3,AbortG),
   resolve_result(S,AbortG,Cand1,me(OutS,B,Path-N1),AbortC),
   resolve(Clauses,N1,Q,Path,InS,Cand2,AbortC),
   tournament(Cand1,Cand2,Winner,AbortC).
resolve([],_,_,_,_,_,_) :- true | true.

   resolve_result(_,AbortG,_,_,abort) :- true |
                    AbortG=abort(all).
   resolve_result(_,abort(_),_,_,_) :- true | true.
   resolve_result(satisfied,AbortG,Cand,Me,_) :- true |
            AbortG=abort(filters), Cand=Me.

   tournament(_,_,_,abort) :- true | true.
   tournament(X,_,Winner,_) :- wait(X) | Winner=X.
   tournament(_,Y,Winner,_) :- wait(Y) | Winner=Y.

commit(_,AbortC,_,_,_,abort(_)) :- true | AbortC=abort.
commit(me(S,B,NewPath),AbortC,InS,OutS,SW,AbortB) :- true |
   AbortC=abort,
   apply_subst(B,S,BS),
   bodyvar(BS,NewB),
   solve_body(NewB,NewPath,InS,OutS,SW,AbortB).
```

Fig. 4.2 The Flat-GHC meta-intepreter (resolve and commit)

the reduction when its head is unified with the goal and all the guard goals are satisfied. However, only one candidate clause can be used for the actual resolution. To determine the clause for the goal to commit, a tournament is performed among the candidate clauses. When the final winner of the tournament is determined, the abort signal is issued to discard all the processes created for other candidate clauses. At the same time, the goal is reduced to the body of the committed clause, after applying the substitution resulting from the head unification and marking every variable in the body as a body variable.

Fig. 4.3 shows codes for the unification in the guard part.

```
unify_guard(_,_,_,_,_,abort(_)) :- true | true.
unify_guard(X,X,_,OutS,sw(A,Z),_) :- true | OutS=[], A=Z.
unify_guard(var('_'),_,_,OutS,sw(A,Z),_) :- true | OutS=[], A=Z.
unify_guard(_,var('_'),_,OutS,sw(A,Z),_) :- true | OutS=[], A=Z.
unify_guard(V,U,InS,OutS,SW,Abort) :-
      V=var(X), X\='_', X\=new(_), U\=var(Y), Y\='_', Y\=new(_) |
   apply_subst(V,InS,V1),
   apply_subst(U,InS,U1),
   unify_guard(V1,U1,InS,OutS,SW,Abort).
unify_guard(V,T,InS,OutS,SW,Abort) :-
      V=var(X), X\='_', X\=new(_), T\=var(_) |
   apply_subst(V,InS,V1),
   unify_guard(V1,T,InS,OutS,SW,Abort).
unify_guard(T,V,InS,OutS,SW,Abort) :-
      V=var(X), X\='_', X\=new(_), T\=var(_) |
   apply_subst(V,InS,V1),
   unify_guard(V1,T,InS,OutS,SW,Abort).
unify_guard(V,T,InS,OutS,SW,_) :-
      V=var(new(_)), T\=var('_'), V\=T |
   OutS=[bind(V,T,T,Tf,SW)].
unify_guard(T,V,InS,OutS,SW,_) :-
      V=var(new(_)), T\=var('_'), V\=T |
   OutS=[bind(V,T,T,Tf,SW)].
unify_guard(X,Y,InS,OutS,SW,Abort) :- X\=var(_), Y\=var(_) |
   functor(X,F,N), functor(Y,G,M),
   unify_guard_functor(F/N,G/M,X=Y,InS,OutS,SW,Abort).
```

Fig. 4.3 The Flat-GHC meta-interpreter (unify in guard)

For the unification in the guard part, special attention must be paid when either of the input terms is a variable, v_b (body variable), coming from the top level goal or the body part of a program clause. If the other term is a variable, v_g (guard variable), coming from the guard part of a program clause, a binding, $v_g \leftarrow v_b$, must simply be generated. If the other term, t, is not a variable, then v_b is rewritten to v_b' by applying the substitution given from the body part, and the unification is retried with the rewritten v_b' and t. If the other term is also a body variable, v_b1, both v_b and v_b1 are rewritten to v_b' and v_b1' respectively, then the unification is retried with the rewritten terms.

The intended meaning of all these modifications is to guarantee success of the guard part without instantiating any body variable. The result of the unification in the guard part is a set of bindings only for variables local to the program clause.

Due to limited space, codes for the rest of the predicates are ommitted.

A GHC program is given as follows:

```
clauses(append(_,_,_),Clauses) :- true | Clauses=[
  (append([],var(y),var(z)) :- true | var(z)=var(y)),
  (append([var(h)|var(x)],var(y),var(z)) :- true |
     var(z)=[var(h)|var(z1)],
     append(var(x),var(y),var(z1)))].
```

Finally, a sample execution looks like:

```
?- ghc exec(append([1,2],[3],var(z)),A).

A = append([1,2],[3],[1,2,3])
```

5. Performance Results

The performance results of the parallel unification program is given in comparison with a more conventional sequential implementation.

The efficiency may not increase very much (or even decreases) when the input terms are very simple and the number of variable occurrences is small. However, the more complex terms are given the more efficiency increase will be observed.

Table 5.1 shows the performance results of the several small test cases, of the unification program in comparison with the sequential one, and Table 5.2 shows the performance results of more complex test cases.

Table 5.3 shows the performance results of a simple Prolog meta-interpreter which is written in GHC in a straightforward manner using the unification program. Also, the performance results of the Flat-GHC meta-interpreter is shown in Table 5.4.

As observed, both the number of reductions and the number of suspensions taken by the parallel unification program are almost always larger than those taken by the sequential one. (The factor is three times at the worst case.)

However, the number of cycles taken by the parallel unification program is almost always smaller than those taken by the sequential one. (The factor is one hundredths at the best case!)

These observations support the expectation of the characteristic of the performance made above.

Assuming some ideal parallel architecture, the number of cycles can be considered as the number of execution steps per processor. This also means the net execution time.

Table 5.1 Performance results of the small test cases

Test No.		1	2	3	4	5	6
The sequential unifier	No. of red.	2	5	6	28	22	46
	No. of susp.	0	1	2	9	7	14
	No. of cycles	2	4	6	22	17	34
The parallel unifier	No. of red.	5	8	10	35	33	58
	No. of susp.	3	4	7	16	16	27
	No. of cycles	4	5	7	15	15	21

(red. as reductions, susp. as suspensions)

test 1) $unify(a, a, \theta) \Rightarrow \theta = []$
test 2) $unify(a, b, \theta) \Rightarrow \theta = fail$
test 3) $unify(v_1, v_2, \theta) \Rightarrow \theta = [v_1 \leftarrow v_2]$
test 4) $unify((v_1, v_2), (v_3, v_4), \theta) \Rightarrow \theta = [v_2 \leftarrow v_4, v_3 \leftarrow v_3]$
test 5) $unify((v_1, v_2), (v_2, v_1), \theta) \Rightarrow \theta = [v_2 \leftarrow v_1]$
test 6) $unify((v_1, v_2, v_3), (v_3, v_1, v_2), \theta) \Rightarrow \theta = [v_1 \leftarrow v_2, v_3 \leftarrow v_2]$

Table 5.2 Performance results of the complex test cases

test No.		1	2	3	4	5
The sequential unifier	No. of red.	34	126	454	1686	6454
	No. of susp.	13	53	205	797	3133
	No. of cycles	26	78	230	726	12486
The parallel unifier	No. of red.	41	131	423	1447	5239
	No. of susp.	20	66	230	846	3230
	No. of cycles	15	29	51	89	159

test n) $unify(VT^n, CT^n, \theta) \Rightarrow \theta = [v_1 \leftarrow t_1, \ldots, v_{2^n} \leftarrow c_{2^n}]$
where VT^n (CT^n) is a balanced binary tree of 2^n distinct variables (constants) as leaves.

Hence, it can be said that order of magnitude speedup is attained by using the parallel unification program.

Incidentally, substancial amount of reductions and suspensions are due to the `merge` processes. If the `merge` process is available as an efficient builtin predicate, the total execution cost of the parallel unification program could be further reduced.

7. Conclusion

The paper has presented a new parallel unification program in GHC, which is designed for obtaining the maximal efficiency by making the best of parallelism inherent in the unification problem. Concerning this, [DKM84] states that "parallelism cannot significantly improve on the best sequential solutions for unification". However, the performance results show that the efficiency has increased significantly for the parallel unification program compared to more sequential ones.

Table 5.3 Performance results of a simple Prolog meta-interpreter in GHC

test No.		1	2	3	4
Equipped with sequential unifier	No. of red.	5718	10858	18518	29215
	No. of susp.	2351	4505	7729	12245
	No. of cycles	693	1008	1383	1817
Equipped with parallel unifier	No. of red.	9396	22199	45513	84108
	No. of susp.	4287	10089	20612	37891
	No. of cycles	210	284	370	468

test n) $bagof(append(v_1, v_2, [1, 2, \ldots, n+2]), A)$

Table 5.4 Performance Results of the Flat-GHC meta-interpreter

test No.		1	2	3	4
Equipped with sequential unifier	No. of red.	5049	9764	17453	29212
	No. of susp.	2028	3994	7224	12186
	No. of cycles	478	633	796	967
Equipped with parallel unifier	No. of red.	2058	2923	3975	5242
	No. of susp.	851	1213	1655	2189
	No. of cycles	203	254	312	374

test n) $exec(append([1, 2, \ldots, n+2], v_1, v_2), A)$

The parallel unification program is also unique in that it employs *stream parallelism* together with a useful programming technique called *short circuit*, for detecting stable properties of networks, very effectively. Concerning correctness of the program, more direct and rigorous proof should be given, for instance along the line of [Apt86].

The paper also has presented the Flat-GHC meta-interpreter which incorporates the parallel unification program. I admit that the meta-interpreter may be of no practical significance, if used in the form as it is. However, the meta-interpreter will be easily extended to include so-called *flavors* such as proof tree construction and reduction steps counting. In particular, the meta-interpreter is intended to be a model for developing more sophisticated ones with *reflective functions* such as variables management and load balancing [Tan88].

I also admit that the Flat-GHC interpreter is not deadlock-free. That is, the interpreter itself will come to deadlock if the object program running on it does. The problem of detection and elimination of deadlock for a meta-program would be an interesting research theme.

Another promising research theme is concerning further optimization by using partial evaluation or other compilation techniques. Some results have been obtained [Fuj88],

however, I think that substantial effort is still be required, within the state of the art techniques, to achieve more significant improvement.

References

[Apt86] K. R. Apt, "Correctness Proofs of Distributed Termination Algorithms", *ACM Trans. on Programming Languages and Systems*, Vol. 8, No. 3, pp. 388-405, 1986.

[DKM84] C. Dwork, P. C. Kanellakis and J. C. Mitchell, "On the sequential nature of unification", *J. Logic Programming*, Vol. 1, pp. 35-50, 1984.

[Fos88] I. Foster, "Parallel implementation of PARLOG", *Proc. of the 1988 International Conference on Parallel Processing*, pp.9-16, 1988.

[Fuj88] H. Fujita, "FGHC Partial Evaluator as a General Purpose Parallel Compiler", ICOT TR-386, 1988.

[KYKS88] S. Kliger, E. Yardeni, K. Kahn and E. Shapiro, "The Language FCP(:,?)", *Proc. of the International Conference on Fifth Generation Computer Systems*, pp. 763-773, 1988.

[Llo88] J. W. Lloyd, "Directions for Meta-Programming", *Proc. of the International Conference on Fifth Generation Computer Systems*, pp. 609-617, 1988.

[MC82] J. Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes", *ACM Trans. on Programming Languages and Systems*, Vol. 4, No. 1, pp. 37-43, 1982.

[MM82] A. Martelli and U. Montanari, "An Efficient Unification Algorithm", *ACM Trans. on Programming Languages and Systems*, Vol. 4, No. 2, pp. 258-282, 1982.

[MW83] Z. Manna and R. Waldinger, "Deductive synthesis of the unification algorithm", In A. W. Biermann and G. Guiho (eds.), *Computer Program Synthesis Methodologies*, Reidel Publishing Company, pp. 251-307, 1983.

[SS86] L. S. Sterling and E. Y. Shapiro, The Art of Prolog, *MIT Press*, 1986.

[SWKS88] V. A. Saraswat, D. Weinbaum, K. Kahn and E. Shapiro, "Detecting stable properties of networks in concurrent logic programming languages", *Proc. of the Seventh Annual ACM Symposium on Principle of Distributed Computing*, pp. 210-222, 1988.

[Tak83] A. Takeuchi, "How to solve it in Concurrent Prolog", Unpublished note, ICOT, 1983.

[Tan88] J. Tanaka, "Meta-interpreters and reflective Operations in GHC", *Proc. of the International Conference on Fifth Generation Computer Systems*, pp. 774-783, 1988.

[Ued86] K. Ueda, "Guarded Horn Clauses", In E. Wada (ed.), *Proc. Logic Programming '85*, LNCS-221, Springer-Verlag, pp. 168-179, 1986.