4U-8

Implementing "Safe" GHC the easy way by Compilation to Guard-free form

Martin Nilsson and Hidehiko Tanaka *
The University of Tokyo

Abstract

This paper describes compilation of the parallel logic programming language GHC, into a parallel logic programming language without guard goals. The advantage with this compilation is that a language without guard goals is much easier to implement. A compiler from Flat GHC written in Prolog becomes only a few pages long.

1 Introduction

It has been shown [Ued 86] that GHC is a very nice language for programming parallel computers. However, there are some points of GHC which makes it hard to implement efficiently on existing computer architectures. Such a feature is the use of arbitrary goals in the guard part of clauses. This becomes hard to implement efficiently because guard goals may not export variable bindings outside the guard, but they may bind variables which only occur in the guard.

Another implementational problem with GHC is the management overhead of processes which is hidden under the surface of the implementation. If GHC is given a query G, H, consisting of two goals G and H, GHC in effect requires that if execution of the goal G fails, the process H should be stopped, and vice versa.

In this paper we will show how (in fact, more than) Flat GHC can be compiled to a similar language FLENG [Nil 86]. This method should be useful also for compilation to the language Oc [Hir 86]. A similar approach has been used by Codish [Cod 86] for compiling concurrent Prolog into Flat Concurrent Prolog.

2 GHC and FLENG

A GHC clause has the form

$$H :- G_1 G_2 \ldots G_m | B_1 B_2 \ldots Bn$$

Where H, G_i, B_j are atoms. The interpretation of this clause is that the head H and the guard goals $G_1 \ldots G_m$ are tests ("if"), and that the body $B_1 \ldots B_n$ is an action ("then").

All goals $G_1 \ldots G_m B_1 \ldots B_n$ are supposed to return truth values to tell the result of their execution. The guard and body are both considered as conjunctions and the implementation is supposed to transmit failure of goals to failure of the entire conjunction.

A FLENG clause has a form similar to a GHC clause, but there are no guard goals. Only the head is used for testing.

The goals in the body of a FLENG clause, as opposed to GHC goals, may only return information about their state by binding shared variables. This means that a FLENG body is executed by just "forking" the body goals and execute them independently.

3 Compilation of the Guard

Compared with GHC, FLENG's testing ability is restricted to head matching. This means that we can test for equality in FLENG, but we cannot test properties like "greater than."

We get around this problem by changing our programming methodology: Instead of testing data in the called clause, we test data in the caller. We pass the result of this test as an argument of the call. Then, depending on this argument, head matching alone will be able to judge whether we should commit to the clause or not.

Given a GHC clause

$$H(x,y) :- Test(x) | B(x,y).$$

the FLENG translation could be, for instance:

$$\begin{aligned} H(x,y) &:- Test(t,x), H'(t,x,y). \\ H'(TRUE,x,y) &:- B(x,y). \end{aligned}$$

Here we assume that the predicate *Test* also is rewritten (compiled) so that it returns its result explicitly by a parameter t. There is a complication: We have assumed that the predicate *Test* does not try to instantiate the variable x. This is usually true in normal (Flat) GHC programs, but it is not generally true. A

*The Tanaka Lab., Dept. of Electr. Engineering, Univ. of Tokyo, Bunkyo-ku, Hongo 7-3-1, TOKYO 113

guard goal in GHC may treat variables in argument positions in one of three ways:

- It might try to both use the variable's value, if it is bound, and also try to bind it.
- It might only try to use the variable's value ("input")
- It might only try to bind the variable ("output")

In Flat GHC, the only example of the first case is the unification primitive "=" . All other system predicates' arguments are either purely input or output. This seems in fact to be the normal case for guard goals. This is lucky for us, since compilation of the first case is much harder than the second two cases.

We require for compilation to FLENG that the arguments of guard goal predicates are declared to be either input or output. We will not allow the first kind of system primitive, except for unification. Clearly, this is a strict subset of GHC, but it is still considerably larger than Flat GHC. (It could possibly be called "Safe GHC.") The procedure is to replace a GHC guard goal

$$P(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

where $x_1, \ldots, x_n$ are input terms, and $y_1, \ldots, y_m$ are output terms, with the equivalent GHC goals

$$P(x_1, \ldots, x_n, z_1, \ldots, z_m), z_1 = y_1, \ldots, z_m = y_m$$

where $z_1, \ldots, z_m$ are fresh variables. This first step isolates the risk of exporting variable bindings to the unification system predicate. As the second step, we compile this guard to FLENG. Here we have to be careful only when we compile unification, so that unification indeed always suspends when the GHC program does.

The trick in compiling unification is to remember that variables occuring in the head of the clause and variables in guard goal output positions (z_i in the example above) may not be bound.

4 Compilation of Commit

The commit operator in GHC makes sure that only one clause in a set of clauses with successful guards will be used.

In our implementation, guards are executed in an *and*-parallel way, and the first guard to succeed will bind a shared variable to a number, which is unique for that clause. If another guard succeeds later, it will not be able to bind that variable. The shared variable can thus function as a mutual exclusion mechanism. The execution of the body can start at the moment the shared variable gets bound.

We can now give the general pattern of a compilation from GHC to FLENG. If a given GHC predicate is

$$P(x,\ldots) \;:-\; Test_1(x,\ldots)|B_1(x,\ldots).$$
$$\vdots$$

the whole definition of P in FLENG becomes:

$$\begin{aligned}
P(x,\ldots) \;&:-\; Guard_1(x,\ldots,n),\ldots \\
Guard_1(x,\ldots,n) \;&:-\; Test_1(t,x,\ldots), \\
&\quad\; Commit(t,1,n), \\
&\quad\; Body_1(n,x,\ldots). \\
&\vdots \\
Body(1,x,\ldots) \;&:-\; B_1(x,\ldots). \\
&\vdots \\
Commit(TRUE,m,n) \;&:-\; m = n.
\end{aligned}$$

Note here that the compilation of the *Test* goals from GHC to FLENG guarantees that these predicates will not interfere with each other.

5 Results and Discussion

We have briefly described a method for compiling a subset of GHC into FLENG.

GHC programs probably become most efficient if they are compiled directly from GHC to machine code, or executed directly by hardware. On the other hand, FLENG is easier to implement than GHC, especially on a very low level. A computer tailored for FLENG may thus become simpler and faster than one for GHC. This could compensate for an efficiency loss in (non-optimal) compilation. It is also worth noting that guards in GHC programs tend to be quite simple, and that GHC clauses often can be read directly as FLENG clauses. Such programs compiled will not display any efficiency loss at all.

6 References

[Cod 86] Codish,M.: *Compiling Or-parallelism into And-parallelism.* In Proc. 3rd Int. Conf. on Logic Progr., London. July 1986.

[Hir 86] Hirata,M.: Letter to the Editor. SIGPLAN Notices, ACM. March 1986.

[Nil 86] Nilsson,M.: *FLENG Prolog - Turning supercomputers into Prolog machines.* In Proc. Logic Progr. Conf. '86, Tokyo. June 1986.

[Ued 86] Ueda,K.: *Guarded Horn Clauses.* D. Eng. Thesis, University of Tokyo. March 1986.