

A Lingua Franca for Concurrent Logic Programming

Hamish Taylor

1. Introduction

Among concurrent logic programming or CLP languages like Concurrent Prolog [16] two of the most prominent are GHC [21] and Parlog [6]. GHC and Parlog have similar execution models but complementary virtues as will be argued. Hence it could be desirable to find some common denominator of both languages, which is sufficiently expressive to be worth programming in its own right, and yet is sufficiently close to both languages to be inter-translatable with them. Implementations of this language would support programming in GHC and Parlog as well as in its own common denominator style. If this lingua franca was simpler to implement than either GHC and Parlog, then the benefits of a more streamlined implementation could be realised as well. This paper describes such a lingua franca.

2. GHC and Parlog

A GHC or Parlog program is a set of relations $\{R_1, \dots, R_n\}$. Each R_i is made up of guarded Horn clauses of the same name and arity. In Edinburgh Prolog syntax where $H, G_1, \dots, G_m, B_1, \dots, B_n$ are atomic formulae (unitary Prolog goals) and $m \geq 1$ and $n \geq 1$ each clause has the following form

$$H \text{ :- } G_1 \text{ and } \dots \text{ and } G_m \mid B_1 \text{ and } \dots \text{ and } B_n$$

The clause's head H gives its relation name and arity, and the G_i s and B_j s are its guard and body goals, separated by the commitment operator \mid . The meta-symbol **and** signifies a conjunction operator. The primitive *true*, which always succeeds, fills empty guards or bodies. The clause's declarative reading is

$$H \text{ is true if } G_1 \text{ and } \dots \text{ and } G_m \text{ and } B_1 \text{ and } \dots \text{ and } B_n \text{ are true}$$

and is a place filler in GHC for the parallel conjunction operator $\text{"},"$, and in Parlog for either the parallel $\text{"},"$ or the sequential conjunction operator "\&" . One or more clauses form an ordered relation

$$C_1 \text{ or } \dots \text{ or } C_n.$$

where each C_i is a guarded Horn clause, **or** is a meta-symbol acting as a place filler for a clause search operator, and the symbol "." terminates the relation. In GHC **or** only stands for the parallel search operator "." whereas in Parlog it

stands for either the parallel "." or the sequential clause search operator ";". These operators are right associative with "." binding more tightly than ";", allowing nesting in Parlog. Parlog clauses for the same relation \mathbf{R} of arity $n \geq 0$ are preceded by a single mode declaration

mode $\mathbf{R}(M_1, \dots, M_n)$.

where \mathbf{R} is the clause head's principal functor and each M_i is either the input argument symbol ? or the output argument symbol ^. All the head arguments of Parlog clauses for the same relation are classified as input or output by this declaration. GHC clause head arguments are always input arguments.

The operational semantics of GHC can be redescribed in such a way that its correspondences with Parlog come out. From this point of view to execute a GHC and Parlog program is to refute a conjunction of goals

G_1 and ... and G_n

or *resolvent* where $n \geq 1$ by input resolution. Ignoring for the moment Parlog's sequential operators for clause search and goal conjunction, each goal G_i is solved in parallel either as a primitive by being satisfied and then removed from the resolvent or as a user defined goal as follows. The goal is matched on all input arguments of clause heads for that relation by determining in parallel whether relevant goal arguments can be unified with input arguments of the head of a fresh copy of the clause without binding goal variables or sharing them with each other. An algorithm for explicit input matching using a primitive is given later. Other clauses which may satisfy the head matching requirements, if the goal is instantiated further, are suspended upon relevant goal variables. They have their input head arguments unified with corresponding goal arguments, if and when the variables upon which they are suspended are bound. In parallel with head matching, guard goals for each clause are solved in parallel. If one or more clauses' head matching requirements and guard goals for that relation are satisfied, one of these is selected non-deterministically. Parallel clause searches are terminated and the goal is reduced to the goals in the selected clause's body by replacing it with them in the resolvent. If a guard goal or head match fails for each clause of the goal's relation, the goal fails. Resolution succeeds when all goals have been solved so that the resolvent is empty, or fails when a goal in a resolvent obtained from the original goals fails.

Parlog execution is subject to the following special conditions:

- output unification on head arguments

The author is with the Department of Computer Science, Heriot-Watt University, Edinburgh, Scotland
GHC is actually defined without assuming finite failure, but effective implementation make assuming finite failure a practical necessity.
See chapter 4 especially section 4.10 [21] and [14] p.352-353.

- sequential clause search
- sequential goal conjunction

After reduction, the output arguments of Parlog clause heads, as specified in the mode declaration, are unified with their corresponding goal arguments in parallel with body goal evaluation. Where Parlog clauses have sequential search restrictions on clauses, then a goal only initiates guard goal satisfaction and head matching with all clauses up to the next unencountered sequential clause search operator in order from left to right. Only where a guard goal or a head match fails in each such clause, are the heads and guards of clauses up to the next unencountered clause search operator evaluated. If Parlog goals are conjoined by the sequential operator "&", then only when the goal or conjunction on the left hand side of the operator has been satisfied, is satisfaction attempted on the right hand side goal or conjunction.

GHC execution is subject to the following special conditions:

- synchronisation rule
- sequencing rule

The synchronisation rule states that trying to bind a calling argument variable in the passive part (head and guard) prior to commitment should cause the unification to suspend. It guarantees safety for guard evaluation, namely that a guard whose clause does not figure in the reduction of a goal will not instantiate the goal. The sequencing rule states that the clause body may be executed before commitment, so long as the attempt to bind a passive part variable prior to commitment causes the unification to suspend. It can be simply satisfied by delaying execution of body goals until commitment.

GHC and Parlog are distinctive among CLP languages [17] in performing guard evaluation in a single binding environment. This avoids big runtime overheads faced by other CLP languages such as Concurrent Prolog [16] of supporting multiple binding environments. Flattening CLP languages by banning user defined goals from clause guards overcomes this problem but only by sacrificing significant expressive power [3].

3. A Lingua Franca of GHC and Parlog

In what follows a lingua franca of GHC and Parlog is characterised and justified. Its main feature is that each Parlog and GHC clause can be expressed by a lingua franca clause which has the same satisfaction conditions. The lingua franca is a common language variation of both GHC and Parlog with similar syntax and semantics. Like both GHC and Parlog a lingua franca program is a set of relations $\langle \mathbf{R}_1, \dots, \mathbf{R}_n \rangle$. Each relation \mathbf{R}_i is composed of guarded Horn clauses

H :- G₁ and ... and G_m | B₁ and ... and B_n

where **and** is a meta-symbol signifying the parallel conjunction operator "," and **m** ≥ 1 and **n** ≥ 1. Like GHC the lingua franca has no sequential conjunction operator. These clauses C_i form ordered relations

C₁ or ... or C_n.

In a relation each C_i is a guarded Horn clause, and "." terminates the relation. Like Parlog **or** acts as a meta-symbol place filler for either a sequential ";" or a parallel clause search operator ".". Lingua franca relations are modeless in the GHC fashion with each clause head argument occurring in an input matching role. Lingua franca goals are executed by and-parallel input resolution in the common way described earlier for GHC and Parlog except that sequential clause search operators constrain or-parallel execution in the Parlog style. A full meta-interpreter for the lingua franca is given later.

The lingua franca's operational semantics differs from GHC's in allowing sequential search and from Parlog's in disallowing sequential conjunctions. It differs from both in sequencing head matching before guard evaluation. Instead of requiring compile time analysis of guards to ensure guard safety like Parlog or requiring a run-time safety test like GHC, the lingua franca imposes no mandatory requirement of safety. However, it supplies language primitives for supporting a run-time safety test anywhere safe guard evaluation must be guaranteed as we shall see. All standard GHC and Parlog primitives are supported by the lingua franca as well as three new primitives =>/2, *satisfy*/2, and *ward*/3 needed for translating GHC and Parlog to the lingua franca. The point of developing such a lingua franca is made by reviewing the factors responsible for the differences between GHC and Parlog.

4. The Control Emphasis

Parlog thematises the control of logic programs and is full of constructs to allow the programmer to synchronise and sequence execution of parts of a logic program. Parlog supports both sequential and parallel operators for clause search and goal conjunction. Thus the following clauses for *process*/1 recursively process a list of query/response *message(Q, R)* elements

```
mode process(?).
process([]).
process( [ message(Q, R) | B ] ) :- valid_query(Q) |
                                  execute(Q, R) & process(B) ;
process([message(_, R)|B])      :- true |
                                  R = no & process(B).
```

The second clause's guard tests whether the query part Q is a valid goal. If it is, $execute(Q, R)$ unifies R with the response. Otherwise the message query is ignored, and the reply *no* given. The sequential conjunction operator in the second and third clause bodies ensures that each element is executed before processing continues on the next element. The second clause's sequential search operator ensures that only failure of the $valid_query/1$ guard test allows the third clause to be tried. Both types of sequential operator enhance the control features of Parlog. They can be used to sequence calls to input/output primitives, to help to control the extent to which computation is demand driven, and to control the granularity of parallelism. However, both sequential operators weaken the complementarity between Parlog's operational and declarative semantics needed for Parlog's status as a logic programming language. The declarative reading of Parlog's or-sequential operator is to signify the negation of prior clauses' guards in subsequent clause guards. It cannot be soundly realised operationally by implementing such negation merely as the failure of prior guards without imposing extra restrictions on them determined by compile time analysis [6] p.89-92. These restrictions raise the problem of precisely identifying unsafe guards which is discussed later. Furthermore the sequentiality of Parlog's and-sequential operator affects the termination properties of a Parlog program, but does not contribute to its declarative meaning. Swapping sequential for parallel conjunction operators can deadlock Parlog goals which used to succeed. Thus with A unbound the goal $valid(A), A = truth$ succeeds over the relation

```
mode valid(?).  
valid(truth).
```

whereas the sequential variant $valid(A) \ \& \ A = truth$ suspends for ever. Although the operator swap does not change the declarative meaning of the query, it makes the query no longer satisfiable.

The lack of emphasis on clean semantic properties for Parlog allowed it to be defined so that certain kinds of Parlog programs could be written, those with unsafe guards, which could not be validly executed. Unsafe guards can sabotage the successful reduction of a goal by relevant clauses, by enabling irrelevant guard evaluations, whose clause does not figure in the reduction of a goal, to bind calling arguments.

```
mode choice(?, ?).  
choice(A, life)  :- way(A) | true.  
choice(B, faith) :- true   | B = true.  
  
mode way(^).  
way(C) :- true | C = godless.
```

Making the call $choice(X, Y)$, $Y = faith$ over the program above, makes the following things possible

- X in the first top level goal is shared with the first argument A of the first clause for *choice/2*
- X in the first clause's guard is bound to *godless* by satisfaction of *way(X)*
- Y in the second top level goal is bound to *faith*
- the first top level goal's head match on the first clause fails

all before the first top level goal's match on the second clause head succeeds and instantiates B to *godless*. If all this happens, then the top levels goals are reduced to the body of the second clause for *choice/2* - *godless* = *true* - which fails. Thus the examination of a subsequently rejected clause has resulted in an unsafe guard goal *way(X)* binding X to *godless*, which has interfered with the top level goals being able to succeed over the second clause. In order to try to rectify this kind of difficulty, Gregory introduced a compile time safety check to try to weed out unsafely guarded clauses from Parlog programs. Unfortunately analysing the safety of a Parlog program at compile time is an undecidable matter in general [6] p.121-132. Thus no algorithm exists for selecting out all and only those clauses in Parlog programs whose guards are unsafe.

The control emphasis of Parlog also allows Parlog to use primitives like *var/1* which have time of call semantics [6] p.82. Thus the conjunction of goals called with A unbound

$\text{var}(A), A = \text{true}$

succeeds or fails depending upon whether the first primitive is executed before the second. Like unsafe guards, this time of call behaviour makes it harder to prove correctness properties of programs, because whether a goal succeeds or fails depends not upon the meaning of the goal but upon non-deterministic properties of the implementation. Time of call properties are not confined to primitives like *var/1*. They apply even to key Parlog primitives like the two argument meta-call *call/2*. This primitive, which always succeeds, initiates the attempt to satisfy its first argument and unifies its second argument with a constant expressing the result. However, if the satisfiable goal *call(2 = 1, _)* is transformed to

$\text{call}(X = 1, _), X = 2$

a new conjunction is obtained, whose success or failure depends on the time of execution of each conjunct. Preservation of execution conditions under this transformation of a program clause by replacing an occurrence of some term T in the guard/body by a fresh variable X and conjoining the goal $X = T$ in parallel with the rest of the guard/body is called *anti-substitutability* [21] 4.7.2. Clearly, some Parlog clauses containing *call/2* lack this property. Thus Parlog's semantic drawbacks are that

- sequentiality of "&" affects goal termination but has no declarative meaning
- unsafe guards cannot be precisely identified to prevent them from causing invalid execution
- realising declarative meaning of ";" by negation as failure also raises safety identification problem

- primitives with time of call semantics make computation outcomes non-deterministic

These drawbacks exist in addition to the basic drawback of all CLP languages of precluding complete searches through embracing the *don't care* non-determinism of the committed choice mechanism. However, these extra semantic deficiencies have been bought at the significant price of enhancing control in Parlog and it is these features which make it so apt for systems programming as will be argued.

5. The Semantic Emphasis

GHC approaches CLP language design from an opposite point of view to Parlog. It thematises semantic rather than control issues in CLP language programs. Sequential conjunction and search operators are excluded from the language to remove their execution order constraints from marring the declarative reading of guarded clauses and the termination properties of programs. The rule of synchronisation is introduced to ensure that guards are guaranteed to be safe. Primitives like *var/1*, *==/2* and meta-calls are excluded to eschew time of call effects and to allow all GHC clauses to possess *anti-substitutability*. Lastly by localising the generation of output bindings to the unification primitive *=/2*, it becomes simpler to reason about the flow of bindings in GHC execution. These measures produce a simple and powerful language open to the application of various transformation techniques [5, 21], and amenable to formal analysis [9, 14, 20].

However, this emphasis of GHC on the clarity and simplicity of its semantics weakens its ability to be used as a general purpose programming tool. By excluding primitives like *var/1*, GHC cannot use the condition of whether a variable is currently unbound to decide whether to commit to a clause or not, although it can delay commitment until that variable is bound. This precludes GHC from defining unification within itself, and also precludes it from being able to define a wide range of related unification functions within itself. The ability to program variations on unification is crucial to a logic programming language's ability to program meta-interpreters of languages and language flavours different from itself. Experience with Prolog has demonstrated that Prolog's ability to support a wide variety of meta-interpreters of related languages on top of itself is a major part of the reason for its success [18]. By losing access to the meta-programming of unification, GHC has hamstrung its capacity to mirror Prolog's wide meta-programming capability.

Systems programming in CLP languages requires the ability to handle failures, exceptions, and run-time errors of sub-computations in a modular fashion which localises their effects. It also requires the ability to control and interact with them as tasks - to suspend, resume or abort them. Meta-interpreters with flavours or extra control features are needed to do this. They can either be explicitly programmed or implemented by control meta-calls [2]. Operating systems also need dynamic and programmable control over scheduling and resource allocation to sub-computations

which only a sophisticated control meta-call can achieve [4]. However, as GHC refuses to support Parlog meta-calls and refuses to support primitives needed for programming unification, GHC is limited in its ability to support flavoured meta-interpreters. The result is an impoverished systems programming capability.

GHC's eschewal of sequential search operators makes programming in GHC more difficult. For example, lack of a sequential search operator makes it more difficult to control the grain of or-parallelism by making it harder to control the order of clause examination. Furthermore a clause choice between a user-defined condition *test/I* holding and its not holding is more conveniently expressed by sequential search

```
goal(In, Out) :- test(In) | process(In, Out) ;
goal(In, Out) :- true    | transform(In, Out).
```

than by using negation as failure *not/I* [6] p.89-92.

```
goal(In, Out) :- test(In)      | process(In, Out).
goal(In, Out) :- not(test(In)) | transform(In, Out).
```

This avoids the condition *test(In)* being evaluated twice when it does not hold. All other well-known CLP languages like FCP, Parlog, CP and P-Prolog support sequential search in some form.

Because GHC lacks a sequential conjunction operator, GHC programs are limited in being able to use the *short circuit* technique to test if a set of goals has all been satisfied. Using this technique the conjunction

```
alpha(A, D), beta(A, B), gamma(C, D).
```

is transformed to give each goal two extra arguments in a chain.

```
alpha(A, D, X1, X2), beta(A, B, X2, X3), gamma(C, D, X3, X4).
```

Just before each goal succeeds, it unifies its last two arguments. The test for joint termination is done by binding one end *X1* to a constant *done* and waiting until the chain's other end *X4* is bound to *done*. Because GHC goals are executed in and-parallel, GHC programs cannot use Parlog's simple expedient of sequencing the execution of the goal with the unification of the local chain ends.

```
mode alpha(?, ^, ^, ^).
alpha(A, D, X1, X2) :- alpha(A, D) & X1 = X2.
```

The design of KL1-B, the basic machine language for ICOT's parallel inference machine [15], has already been influenced by these considerations. KL1-B [10] is based upon Flat GHC but has been extended by pragmas and a meta-call. The price is the loss of GHC's nice semantic properties.

Neither can GHC generally use its sequencing rule

$$\text{alpha}(A, D, X1, X2) \text{ :- alpha}(A, D) \mid X1 = X2.$$

to achieve the same effect, because the first goal in the sequence cannot violate the safety of its guard. If *alpha/2*'s second argument accepted input but tried to instantiate it further then the goal would suspend on that attempted binding. To use the short circuit in a fully general way GHC programs must be extended to test for the termination of each primitive call liable to bind variables in the relevant program! For each unification $A = B$ in the body of the GHC clauses defining *alpha/2* this means replacing it with the goal *unify*($A, B, X1, X2$) defined as follows [17] p.462-463.

$$\text{unify}(A, B, X1, X2) \text{ :- true} \mid A = B, \text{ match}(A, B, X1, X2).$$
$$\text{match}(A, A, X1, X2) \text{ :- true} \mid X1 = X2.$$

The local part of the short circuit is only closed when *match/4* detects that the unification has been done. Plainly the overheads of this transformation mark a gulf of expressiveness between GHC and Parlog.

This paper shows that a choice does not have to be made between having a CLP language with clean semantics like GHC or one with a good systems programming capability like Parlog. It is possible to benefit from the respective virtues of each language on a single implementation by rewriting each of them in a lingua franca and emulating this language directly.

6. The Common Linguistic Denominator

A lingua franca of GHC and Parlog must abstract from their differences and yet enable both languages to be translated to the lingua franca and back again. It will be more expressive than either, because neither GHC nor Parlog is expressive enough to translate away all the other language's clauses. Parlog cannot support GHC's run-time suspension test and GHC disallows the action of many Parlog primitives. Besides restrictions on primitives GHC has four main features which distinguish it from Parlog.

- no mode declarations
- purely parallel search and conjunction operators
- clause bodies are executable before commitment so long as trying to instantiate a body variable shared with the head or guard causes the unification to suspend until commitment
- attempts by guard goals to write upon calling argument variables before commitment suspend

6.1. Mode Declarations

GHC input matches on each head argument and explicitly represents output bindings with unifications in the bodies of clauses. Parlog can be easily transformed to obey the same rule [6] p.64-69. Output mode Parlog head arguments can be removed and replaced by a fresh variable, and an extra unification goal between it and the original head argument can be added to the clause body. After this change all Parlog head arguments can be treated as input arguments and mode declarations dispensed with. In the same way the lingua franca can translate away mode declarations from Parlog relations without supporting them itself.

6.2. Passive Part Concurrency

Both GHC and Parlog allow head matching to proceed in parallel with guard evaluation. The lingua franca would be made simpler, if it required instead that head matching be completed before guard evaluation begins. This would eliminate implementation overheads in providing safe storage for variables shared between head arguments and user defined guard goals, which are accessed during execution of the guard goal before the relevant binding for the variable is supplied from the goal by a head match. It would also enable indexing tests for head matching to be used as the sole means for determining early whether a clause is not suitable for reducing a goal. This would avoid early evaluation of guards of clauses, whose heads don't yet match the goal, to detect whether the guard fails.

Where clause guards are empty (signified by *true*), no transformation is needed in translating GHC or Parlog to the lingua franca to compensate for the lingua franca's sequencing of head matching and guard evaluation. Furthermore where no guard goal can fail before the head match succeeds, no transformation is necessary either. However, in other cases the head match has to be performed in parallel with guard evaluation. Otherwise suspension in the head match could delay indefinitely discovery that guard goal execution will fail. Thus where the guard is non-empty and it is not known that the guard will not fail before the head match succeeds, the head match is decomposed into an extra one way unification in the guard in the fashion for translating Parlog to Kernel Parlog [6] p.65-66.

A variant of Gregory's method uses a different one-way unification primitive $\Rightarrow/2$. It tries to unify its arguments and suspends on relevant left hand argument variables if successful unification would have to bind one or share two of them. The whole head match can be realised by forming an aggregate term out of each head argument, which does not occur as a unique variable in the head, and one-way unifying it with a similar aggregate of their distinct variable replacements. Thus the guarded clause

```
compare([A|B], [C|D]) :- test(A, C) | compare(B, D).
```

is translated to

$$\text{compare}(E, F) \text{ :- } (E, F) \Rightarrow ([A|B], [C|D]), \text{test}(A, C) \mid \text{compare}(B, D).$$

where the new clause head has distinct fresh variables for arguments. This approach gives a simpler representation for head matching. Separate goals are not created to match each head argument and repeated variables as with Gregory's method. However, multi-processing implementations may want to implement $\Rightarrow/2$ using several simple matching goals in Gregory's fashion to avoid creating contention for exclusive access to variables by tying all these variables together into one primitive process. A sequential algorithm given in figure 1 specifies the behaviour required of this one-way unification operation.

FIGURE 1

The algorithm returns *success*, *failure* or *suspend* on execution. It is executed each time the $\Rightarrow/2$ process is run until it returns *success* or *failure*. When it returns *suspend*, the $\Rightarrow/2$ process should be suspended on *at least* all the left hand side variables detected to be bound by the Unify Procedure. Early detection of failure requires it to be suspended on the other variables as well. It is awoken and made runnable when any of these variables gets bound. As lists can be represented by two argument structures, they are not handled separately. The whole *Algorithm* is executed atomically. Faster algorithms would circumvent using full unification to test for non-unifiability or for determining all variables to suspend on in order to avoid obtaining exclusive access to left hand variables during execution.

6.3. And Sequential Operators

Unlike GHC, Parlog supports and-sequential operators. However, the synchronised satisfaction of Parlog goals can be achieved indirectly. Sequentially conjoined goals can be removed from Parlog clauses using meta-calls and synchronisation flags linking the meta-calls [6] p. 98-99, 141-142. The absence of and-sequential operators in GHC and the possibility of eliminating sequential conjunction operators in Parlog in this way establishes another common feature between GHC and Parlog which can be copied by the lingua franca. It need only support and-parallel conjunctions. And-sequential operators can be eliminated by transforming them away from the Parlog source. This would help lingua franca programs extract and-parallelism from applications in a GHC-like programming style and make these programs less prone to the sequential style of Prolog programming dogging many Parlog programs.

Rather than using Gregory's method which employs Parlog's three argument control meta-call, a simpler two argument meta-call *satisfy/2* is introduced instead. It attempts to satisfy its first argument. If it succeeds, it binds its second argument to some simple term and succeeds itself. If the attempted satisfaction fails, it fails. It is different from the Parlog meta-call *call/2* which never fails, although the meta-call *satisfy/2* can be defined using *call/2*. However, *satisfy/2* does not violate anti-substitutability unlike *call/2* as was seen earlier. By introducing a new primitive *ground/1*, which suspends until its argument is ground, *satisfy/2* can be used to define an auxiliary relation *wait/3* as follows:

$$\text{wait}(\text{Goal}, \text{Control}, \text{Flag}) \text{ :- } \text{ground}(\text{Control}) \mid \text{satisfy}(\text{Goal}, \text{Flag}).$$

wait/3 suspends until its second argument is ground, then it executes its first argument goal. It fails, if this goal fails. If it succeeds, it signals this fact by binding its third argument to a constant. *wait/3* provides a simple means of realising sequential execution by controlling the execution order of and-parallel goals with synchronisation flags. Thus the mixed conjunction

$$a(A, B) \text{ :- } (\text{one}(A), \text{two}(B)) \ \& \ \text{three}((A,B)).$$

can be translated as follows:

$$a(A, B) \text{ :- } \text{wait}(\text{one}(A), [], D), \text{wait}(\text{two}(B), [], E), \text{wait}(\text{three}((A,B)), [D, E], _).$$

Control variables D and E delay execution of *three((A,B))* until *one(A)* and *two(B)* are satisfied.

FIGURE 2

The general form of this method for translating away sequential conjunctions applies to any Prolog syntax structure not containing the reserved predicate *wait/3*. Over the pure Prolog program given in figure 2, the query

$$\mid \text{?- } p(\text{Goals}, \text{New_goals}, [], _, \text{parallel}).$$

where *Goals* is a conjunction of Parlog goals unifies *New_goals* with its lingua franca translation.

6.4. Or-Sequential Search

GHC only supports or-parallel clause search, although the option of allowing sequential search by means of an *otherwise* predicate was considered in GHC's design at one stage [21] 3.4.7. On the other hand Parlog allows clauses to be searched in parallel or in sequence. Gregory has shown that it is possible to eliminate or-sequential clause search using the three argument Parlog meta-call [6] p.140-141. However, using Gregory's rather complex method would mean abandoning the principle of representing each Parlog or GHC clause with a lingua franca clause translation of the same relation name and arity. Fortunately, it is unnecessary, because sequential search is easy to support directly in CLP implementations [1, 7].

6.5. Rule of Sequencing

The option on executing the bodies of GHC clauses before commitment subject to GHC's rule of sequencing [21] may seem to be appropriate for highly parallel data-flow architectures, but is likely to result in a lot of redundant computation on less parallel architectures at the expense of what could be more profitably executed. Furthermore, by allowing such premature computation almost no GHC program can be guaranteed to terminate without importing extra fairness assumptions [14] p.353-354. To avoid these semantic difficulties and in keeping with extant implementations of GHC [8, 12], no special suspension mechanism is proposed for supporting GHC's rule of sequencing by the lingua franca. The same rule obeyed by Parlog of strict sequencing of evaluation will be obeyed by the lingua franca and only when the clause has committed will evaluation of the body commence.

6.6. Rule of Synchronisation

GHC's rule of synchronisation preserves guard safety by requiring that if a unification would bind or share calling argument variables in an ancestor guard, that unification should suspend. Thus over the clauses

```
ask(A) :- tell(A) | true.
```

```
tell(A) :- true | A = yes.
```

the call *ask(A)* should suspend, because the unification *A = yes* has to suspend in order not to instantiate a calling argument variable in an ancestor guard.

Two of the main approaches canvassed for implementing the guard suspension test are :

- Ueda's Pointer Colouring Scheme [12, 21]
- Miyazaki's Guard Numbering Scheme [11]

Both schemes concentrate upon suspending unifications in the bodies of GHC clauses being used to evaluate guard goals in the event of the unification attempting to write upon a calling argument variable. Either scheme threatens to impose a suspension test overhead upon every explicit unification. However, most GHC clauses either only match input against head arguments or have only primitives with input arguments for guards. Only in evaluating the remaining cases is it necessary to test for a user defined or primitive guard goal with output arguments writing upon a calling argument variable. A more demand driven scheme could separate out the suspension producing conditions from GHC clauses, in order to localise the requirement to suspend to evaluation of an extra primitive goal in the responsible guard. This would impose no general overhead upon unification in the body. By allowing unsafe user-defined guard goals to be evaluated without suspension constraints on their output bindings, evaluation is made more eager and computation space is more effectively recycled. Computation space used to represent guard goal processes in the process tree can be claimed and released as soon as the guard goal is satisfied or fails. It will not be claimed and then frozen unused pending the further communication of input values to remove from suspension a goal process suspended on a unsafe binding. For deep guard evaluations this frozen space may include space claimed to store ancestor processes suspended upon the suspended user-defined guard process as well as that claimed to store the process performing the suspended unification.

6.6.1. Localisation

The central idea behind localisation [19] is that suspension effects due to the rule of synchronisation arise from what happens during guard evaluation and can be restricted to the locus of that evaluation. In what follows variables will be termed *unsafe* which occur both in the heads of GHC clauses and either in user defined guard goals or argument positions of primitive guard goals liable to bind given values. If all unsafe variables in guard goals are replaced by fresh variables then all transformed calls to guard goals can be allowed to proceed without being subject to a safety suspension condition. Each safety suspension condition can then be achieved by a new primitive which relinks the *unsafe* guard variable with its new replacement. In this way a GHC program can be distilled down to lingua franca clauses interleaved with special primitives to achieve the safe guard suspension mechanism.

The special primitive's job is to allow old variable values to be passed to its new replacement but to ensure that if unifying the new and the old guard variables would bind a calling argument variable before commitment, this special primitive will suspend. As an example take the GHC clause

```
head([A]) :- test(A) | true.
```

and remove the head match in the manner already described.

head(B) :- B => [A], test(A) | true.

A is replaced by a fresh variable $A1$ in the user-defined guard $test/1$ and a special primitive $ward/2$ is added to link the two variables in a way which protects the calling environment from being bound by suspending the call instead. The result is:

head(B) :- B => [A], ward(A, A1), test(A1) | true.

$ward/2$ needs to be able to suspend on several variables because it must handle complex terms. Because complex terms can get progressively bound, it also needs to be able to suspend, awake and pass on values and then maybe suspend again. Other properties of $ward/2$ can be inferred from the fact that the guard goal $test/1$ might need its argument to be bound to a value to succeed. The satisfaction of this guard can only be achieved if the $ward/2$ goal passes bindings to its second argument from its first. However the $ward/2$ goal should not allow a user defined guard goal to export a value from its satisfaction by having $ward/2$ simply unify A with $A1$. Unidirectional unification is more apt.

However, if $ward/2$ is equated with $=>/2$, then the second goal of the guard will succeed in cases where it should not. The calling argument to which B is bound might get bound to $[_I21]$ and the first input matching unidirectional unification succeed sharing A with $_I21$. Since A and $A1$ are still unbound, the second guard safety unidirectional unification can succeed by sharing them. Shortly afterwards the parallel $test/1$ call might bind the variable $A1$ to *incorrect*. The result could be the instantiation of a calling argument variable $_I21$ to *incorrect* before the clause has committed. This behaviour cannot be avoided by insisting that the second goal's unidirectional unification be performed only after the $test/1$ goal has succeeded, because then bindings could never be passed through $ward/2$ into the user defined guard goal.

A better idea is to make the $ward/2$ predicate reluctant unidirectional unification so that the $ward/2$ predicate suspends if, in order to unify unidirectionally its arguments, it has to share variables. Reluctant unidirectional unification is unidirectional unification where bindings are made to the instantiable side whenever non-variable bindings are detected on the non-instantiable side and unidirectional unification is possible. It fails if the two sides can never be unified. Where instantiable side variables need to be associated with complex non-instantiable side terms containing variables, they are bound to a consistent copy of the complex term to avoid variable sharing. The unidirectional unification is reluctant in that variables on the instantiable side only get bound to non-variables or to each other. They never get shared with variables on the non-instantiable side. The predicate suspends instead.

The problem now is that the test is too severe. If the guard goal needs a partially bound or unbound argument in order to succeed, the $ward/2$ predicate must not suspend for ever. For example if the guard goal was defined as follows:

```
test(A) :- true | true.
```

then the result of reluctant unidirectional unification would be for the *ward/2* predicate to suspend with two variable arguments even when the user-defined guard *test/1* has succeeded. The *ward/2* predicate has to suspend on two unbound arguments but only for as long as the guard goal it is warding has not succeeded.

To implement this, a variable binding can be used to signal success of the user-defined guard goal. The signal can be sent to the *ward* predicate by an extra argument, and the predicate defined so that when it gets this signal, it abandons reluctance in one-way unification and tries to relink the original variable and its replacement non-reluctantly. Its unidirectionally ensures that bindings generated by evaluating the guard goal are not passed out to the calling environment. It causes the primitive to suspend should this be attempted. Putting these considerations together gives the following. The original clause

```
head([A]) :- test(A) | true.
```

gets transformed into:

```
head(B) :- B => [A], ward(A, A1, C), satisfy(test(A1), C) | true.
```

The first guard goal implements head matching. The next two implement the original guard goal set about by the GHC suspension test. Up until the user-defined guard goal succeeds, the calling argument variable and its replacement are linked by a reluctant unidirectional unification which does not allow sharing of variables between *ward/3*'s first two arguments. If and when the guard goal succeeds, then *C* is bound by the metacall *satisfy/2*. This signals to *ward/3* which now unidirectionally unifies its first two arguments so as to allow sharing of variables between them. An algorithm for *ward/3*, where the first two arguments are unifiable and in the opposite order, is given in [19].

6.6.2. General Localised Suspension

Transferring this translation scheme into a general scheme requires taking into account multiple occurrences of *unsafe* variables in one or more guard goals. A simple sound way to do this is to place the entire original guard in a *satisfy/2* meta-call if any part of the guard contains an original head argument variable. An aggregate term of all unsafe variables and a consistent copy is also formed by substituting fresh variables for each unsafe variable, as the two terms related by *ward/3*.

The general rule for implementing guard safety suspension in GHC applies to guard goals containing user-defined predicates and primitives liable to bind their arguments.

- 1) All variables both in a head argument and in a user-defined guard goal or in a primitive goal argument which the primitive goal may bind, are formed into the *at risk* set. Distinct fresh substitute variables for this set are created. If the *at risk* set is empty, nothing further is done.
- 2) Every member of the *at risk* set in a guard goal is uniformly replaced by its substitute.
- 3) The whole GHC guard is put into the first argument of a *satisfy/2* meta-call and a fresh variable to be called the *control* variable is put into its second argument.
- 4) A structure with an arbitrary functor and arguments consisting of all members of the *at risk* set of variables is formed. A copy of this structure is made but with substitute variables replacing *at risk* variables. A *ward/3* predicate goal is added to the guard with the structure, the structure copy and the *control* variable of *satisfy/2* as its first three arguments.

Head matches can now be decomposed into the guard in the fashion described earlier. Although *ward/3* can be used to compile GHC to the lingua franca, it cannot be used with *satisfy/2* to compile GHC to flat GHC. While the meta-call *satisfy/2* does not violate anti-substitutability unlike the Parlog meta-call *call/2* [21] 4.7.2, the primitive *ward/3* does.

With unbound variables the goal

```
ward((C,D), (A,A), yes).
```

suspends whereas its anti-substitution

```
ward((C,D), (A,B), yes), A = B.
```

may succeed. Hence *ward/3* cannot be made into a GHC primitive. The same applies to $\Rightarrow/2$.

7. Example Translations

An example shows the result of applying the transformation from GHC. The following GHC clause

```
process([A|B], C, [A|D]) :- check([A|B], E), test([A|D], E) |
                           process(B, C, D).
```

is translated to the lingua franca clause

```
process(J, C, K) :- (J, K) => ([A|B], [A|D]),
                    ward((A, B, D), (F, G, H), I),
                    satisfy((check([F|G], E), test([F|H], E)), I) |
                    process(B, C, D).
```

The first guard goal implements the head match, the second implements the safe guard suspension test for all three relevant variables and the third satisfies the original guard. Only two extra primitive goals and a meta-call are required to implement the localised run-time suspension test and the head match. A different example shows the transformation of a Parlog clause

```
mode test(?, ?, ^).
test([A|B], [A|C], pair(D, E)) :- check(A) & test(C) |
    ( ( analyse(A), synthesise(B, F) ) &
      ( transform(B, F), unfold(C, E) ) ) &
    test(B, C, D).
```

to the following lingua franca clause:

```
test(G, H, I) :- (G, H) => ([A|B], [A|C]),
    wait(check(A), [], J),
    wait(test(C), [J], K) |
    I = pair(D, E),
    wait(analyse(A), [], L),
    wait(synthesise(B, F), [], M),
    wait(transform(B, F), [L, M], N),
    wait(unfold(C, E), [L, M], O),
    wait(test(B, C, D), [L, M, N, O], P).
```

8. Translating back from the Lingua Franca

The reverse translation of translated GHC is simple. The general method presupposes all occurrences of the predicates *ward/3*, *=>/2* and *satisfy/2* are solely a product of the original correct translation.

- 1) The *satisfy/2* predicate is replaced by its first argument.
- 2) The first two arguments of *ward/3* are unified and the goal is removed.
- 3) The two arguments of *=>/2* are unified and the goal is removed.

If the guard is left empty, the space is filled by the primitive *true*. The simplicity of this reverse translation method is what it should be. All that is being done by translating GHC to the lingua franca is to decompose unifications in such a way that appropriate suspension effects and unification directionality can be associated with them. Clearly re-unifying them should restore the status quo.

Reverse translation of the lingua franca into Parlog is performed in three steps. First of all head matches are restored by unifying the two arguments of *=>/2* and removing the *=>/2* goal, then sequential conjunctions are put back, and finally output head arguments are restored. A general algorithm for sequential conjunction replacement is given by

executing the query

```
| ?- q(Goal, New_goals, _, _, _).
```

over the pure Prolog program given in figure 3 with *Goals* bound to a lingua franca conjunction of goals. *New_goals* will be unified with the original Parlog conjunction.

FIGURE 3

Output head arguments are restored by unifying the arguments of all $=/2$ predicates in the body created by the original translation and removing them. They can be uniquely identified by giving them a reserved name during the original translation. Mode declarations cannot always be correctly inferred from the initial translation to the lingua franca so they have to be preserved independently.

9. Lingua Franca Semantics

A meta-interpreter for the full lingua franca invoked by

```
| ?- call(Goals).
```

is given by figure 4.

FIGURE 4

call/1's first clause handles conjunctions. Its second clause evaluates primitives recognised by the suspending primitive *primitive/1*. Its third clause handles committed choice resolution. Frozen clauses retrieved in the second list argument of *clauses/2* define the first argument relation with ";" operators joining adjacent clauses. These clauses have their selectability tested by *reduce/3*. The first clause of *reduce/3* examines whether the leading clause can reduce the goal. The second clause searches the next clause in parallel. The third clause examines the last clause in a group of clauses to be searched in parallel. The sequential search operator after it ensures that the fourth clause is only considered after the first three have been considered and rejected. It continues clause search after all previous *reduce/3*

clauses have been shown to be unable to reduce the goal. *test/3* ensures that the head match is performed before the guard is called, and *match/3* uses the primitive *melt/2* to obtain a fresh melted copy of the frozen clause before input matching the goal with it.

Instead of imposing an inadequately selective guard safety criterion of validity on programs like Parlog or requiring a mandatory run-time safety test in evaluating every clause guard like GHC, the lingua franca supplies a precise means for supporting a run-time safety test anywhere safe guard evaluation must be guaranteed. This test can be used to augment compile time safety analysis [6] p.121-132. Thus unsafe guards in *reduce/3* above can be made safe by replacing *test/3* by

```
test(Goal, C, B) :- match(Goal, C, G, Body) | B = Body, ward(G, G1, X), satisfy(G1, X).
```

The *ward/3* goal acts as a valve preventing the execution of *G1* by the meta-call *satisfy/2* from exporting bindings into *G* and hence through *Goal* to the calling environment.

The lingua franca simplifies CLP language semantics by not making the safety of each CLP clause's guard into a constraint that must be satisfied to execute the CLP language validly. Guaranteeing guard safety is enabled by providing means to let a precise test of safety be applied anywhere it is needed, but the language semantics make it optional to apply it. This test can be realised in a clause using the methods of translation described earlier or it can be exercised generally over a program with unsafe guards by using the safe guard meta-interpreter described above.

The lingua franca exceeds both GHC and Parlog in expressive power by being able to translate each GHC or Parlog clause into a lingua franca clause with the same operational semantics, while GHC and Parlog are unable to do this for each lingua franca clause. Furthermore by being able to translate both CLP languages into the lingua franca and back again a lingua franca implementation can be presented transparently as a GHC, a Parlog and a lingua franca implementation.

The programming style of the lingua franca is a hybrid of GHC's and Parlog's. The lingua franca follows GHC in eschewing provision of an and-sequential operator and in being modeless. This steers programmers away from lapsing into a sequential Prolog-like style and from relying on misleading assurances given by mode declarations. The lingua franca follows Parlog in providing a rich set of control constructs and from not disallowing the use of time of call primitives. Thus it provides a CLP vehicle in its own right rather like Parlog for programming systems with.

By uncoupling evaluation of user defined guards from the suspension mechanism and using *ward/3* to localise suspension effects to the unsafe guard, the lingua franca makes execution of such guard goals more eager and more effectively recycles its computation space than GHC. Computation space used to represent user defined guard goal processes can be claimed and released as soon as the guard goal is satisfied or fails. It will not be claimed and then

frozen unused pending the further communication of input values to remove from suspension a goal process suspended on a unsafe binding.

Because the lingua franca does not require that head matching be concurrent with guard evaluation, it incurs less runtime overheads in this respect than GHC or Parlog. It eliminates the need to provide safe storage for variables shared between head arguments and user defined guard goals, which are accessed during guard goal execution before relevant variable bindings are supplied by head matching. It also expedites use of indexing for clause selection by avoiding early evaluation of guards of clauses, whose heads don't yet match the goal, to detect whether the guard fails.

10. Kernel Parlog

When viewed as an intermediate language in the compilation of Parlog, the lingua franca plays a role similar to Kernel Parlog [6] p.64-69. However, there are several important differences. Unlike Kernel Parlog the lingua franca has been made expressive enough to be used as a programming vehicle of first instance like Parlog and GHC. Thus it preserves full head matching functionality rather than translating it away in the fashion of Kernel Parlog. Furthermore Kernel Parlog does not support primitives like *ward/3* which would enable it to translate away the GHC run-time suspension test. Kernel Parlog execution can only obey the run-time suspension test if its implementation independently supports it.

ward/3's complex action cannot be performed by Kernel Parlog's simple input matching primitives, because that would require ensuring the atomic execution of several of them together. However, Kernel Parlog lacks the means to do this. Part of a method has been described for translating away GHC's run-time suspension test into a complex series of simple matches performed in parallel in the context of translating full GHC into the CLP language $CP[\downarrow, |]$ [14]. However, the translation exploits the multiple environment property of this CLP language where atomic operations of the required kind are possible. Kernel Parlog cannot follow $CP[\downarrow, |]$ in doing this. Nor is *ward/3* suitable for treating as a Kernel Parlog primitive. The design philosophy underlying the choice of Kernel Parlog primitives is aimed at ensuring that processes in the intended And/Or tree implementation model [6] pp.163-164 never require to suspend on more than variable at once. However, *ward/3* has to be able to do this.

11. Conclusion

A strategy for absorbing the differences between two prominent CLP languages by assimilating them into a common language variant has been proposed. It meets different demands of CLP languages by supporting each in a common medium. The strategy has detailed how to translate both GHC and Parlog into a lingua franca and back again so that an implementation of the lingua franca can be transparently presented to a user as either executing GHC or Parlog

directly. By supporting both languages the lingua franca can provide a user with the semantically desirable properties of GHC, with the control features of Parlog or with its own simple programming style on a single implementation according to requirements.

Acknowledgements

I would like to thank Steve Gregory and the paper's reviewers for their many detailed and helpful comments, as well as colleagues at Heriot-Watt university and members of Imperial College's Parlog group.

References

- [1] J.A. Crammond, *Implementation of Committed Choice Languages on Shared Memory Multiprocessors*, PhD thesis, Heriot-Watt Univ., Edinburgh, Scotland, May 1988.
- [2] I.T. Foster, "Logic Operating Systems: Design Issues," in *Logic Programming, Proc. of 4th Int. Conf.*, MIT Press, pp. 910-926, May 1987.
- [3] I.T. Foster and S. Taylor, "Flat Parlog: a Basis for Comparison," Technical Report, Comp. Science Dept., Weizmann Inst. of Science, Israel, Mar. 1987.
- [4] I.T. Foster, "Parlog as a Systems Programming Language," PhD thesis, Dept. of Computing, Imperial College, London Univ., Mar. 1988.
- [5] H. Fujita, A. Okumura, and K. Furukawa, "Partial Evaluation of GHC programs based upon the UR-Set with constraints," in *Logic Programming, Proc. of 5th Int. Conf. and Symp.*, MIT Press, pp. 924-940, Aug. 1988.
- [6] S. Gregory, *Parallel Logic Programming in Parlog - the Language and its Implementation*, Int. Series in Logic Programming, Addison-Wesley, London, Feb. 1987.
- [7] S. Gregory, I.T. Foster, A. Burt, and G.A. Ringwood, *An Abstract Machine for the Implementation of Parlog on Uniprocessors*, New Gen. Computing, vol. 6, pp. 389-420, 1989.
- [8] N. Ichiyoshi, T. Miyazaki, and K. Taki, "A Distributed Implementation of Flat GHC on the Multi-PSI," in *Logic Programming, Proc. of 4th Int. Conf.*, MIT Press, pp. 257-276, May 1987.
- [9] T. Kanamori and M. Maeji, "A preliminary note on the semantics of Guarded Horn Clauses", TR-434, Inst. for New Gen. Comp. Technology, Tokyo, Dec. 1988.
- [10] Y. Kimura and T. Chikayama, "An abstract KL1 machine and its Instruction Set", TR-246, Inst. for New Gen. Comp. Technology, Tokyo, 1987.

- [11] M. Kishi, E. Kuno, K. Rokusawa, and N. Ito, "The Dataflow-based Parallel Inference Machine to support two basic languages in KL1", TR-114, Inst. for New Gen. Comp. Technology, Tokyo, 1985.
- [12] J. Levy, "A GHC Abstract Machine and Instruction Set," in *Lecture Notes in Comp. Science 225, 3rd Int. Conf. on Logic Programming*, Springer-Verlag, pp. 157-171, July 1986.
- [13] J. Levy and E. Shapiro, "Translation of Safe GHC and Safe Concurrent Prolog to FCP," Technical Report CS87-08, Dept. of Comp. Science, Weizmann Inst. of Science, Israel, June 1987.
- [14] V. Saraswat, "GHC: Operational semantics, Problems and Relationship with CP(\downarrow , |)," in Proc. of the Symp. on Logic Programming, pp. 347-358, San Francisco, Sep. 1987.
- [15] M. Sato, H. Shimizu, A. Matsumoto, K. Rokusawa, and A. Goto, "KL1 Execution Model for PIM Cluster with Shared Memory, in *Logic Programming, Proc. of 4th Int. Conf.*, MIT Press, pp. 338-355, 1987.
- [16] E. Shapiro, "A Subset of Concurrent Prolog and its Interpreter", TR-003, Inst. for New Gen. Comp. Technology, Tokyo, Feb. 1983.
- [17] E. Shapiro, "The Family of Concurrent Logic Programming Languages," in *ACM Computing Surveys*, vol. 21(3), pp. 413-510, Sep. 1989.
- [18] L. Sterling and R.D. Beer, "Meta-Interpreters for Expert System Construction," in *Journal of Logic Programming*, vol. 6, pp. 163-178, 1989.
- [19] H. Taylor, "Localising the GHC suspension test," in *Logic Programming, Proc. of 5th Int. Conf. and Symp.*, MIT Press, pp. 1257-1271, Aug. 1988.
- [20] K. Ueda, "On the Operational Semantics of Guarded Horn Clauses," TM-136, Inst. for New Gen. Comp. Technology, Tokyo, 1985.
- [21] K. Ueda, "Guarded Horn Clauses," Doctor of Eng. Thesis, Tokyo Univ., Japan, Mar. 1986.

Algorithm record all distinct unbound variables in left hand argument
apply Unify Procedure to left and right hand arguments
IF Unify Procedure returns *failure*
THEN restore all variables from trail AND return *failure*
ELSE check recorded variables IF any bound
THEN restore all bound variables AND return *suspend*
ELSE return *success*

Unify dereference right hand argument R and left hand argument L
IF R is a variable
THEN bind R to L AND trail R
ELSE EITHER L is variable bind L to R AND trail L
OR L is constant IF R is not same term
THEN return *failure*
OR L is complex term IF R does not have same functor and arity
THEN return *failure*
ELSE FOR each argument R_i of R
apply Unify Procedure to R_i and L_i
IF Unify Procedure returns *failure*
THEN return *failure*

Fig. 1. One Way Unification Algorithm

```
p((A , B), C1, In, [], parallel) :-  
    p(A, A1, In, _, parallel),  
    p(B, B1, In, _, parallel),  
    join(A1, B1, C1).  
  
p((A , B), C1, In, Results, sequential) :-  
    p(A, A1, In, Results1, sequential),  
    p(B, B1, In, Results2, sequential),  
    append(Results1, Results2, Results),  
    join(A1, B1, C1).  
  
p((A & B), C1, In, [], parallel) :-  
    p(A, A1, In, Results1, sequential),  
    append(Results1, In, In1),  
    p(B, B1, In1, _, sequential),  
    join(A1, B1, C1).  
  
p((A & B), C1, In, Results, sequential) :-  
    p(A, A1, In, Results1, sequential),  
    append(Results1, In, In1),  
    p(B, B1, In1, Results2, sequential),  
    append(In1, Results2, Results),  
    join(A1, B1, C1).  
  
p(A, A, [], [], parallel) :-    literal(A).  
p(A, wait(A, [], Out), [], [Out], sequential) :-  
    literal(A).  
p(A, wait(A, In, Out), In, [Out], _) :-  
    literal(A), In \\= [].  
  
join((A,B),C,(A,B,C)).  
join(A,B,(A,B)) :- literal(A).  
  
literal(A) :- A \\= (_,_), A \\= (_ & _),
```

Fig. 2 Sequential Conjunction Elimination

```
q((A , B), C, Common, All, Results) :-
    split((A,B), A1, B1),
    q(A1, A2, Common1, All1, Results1),
    q(B1, B2, Common2, All2, Results2),
    and(A2, B2, C, Results1, All2, Common2),
    append(Results1, Results2, Results),
    append(All1, All2, All),
    intersection(Common1, Common2, Common).

q(wait(A,In,Out), A, In, In, [Out]).
q(A, A, [], [], []) :-      A \\= (_,_), A \\= wait(_,_,_).

and(A, B, C, Results, All, _) :-      intersection(All, Results, []),
                                       conjoin(A, B, C).

and(A, B, A&B, Results, _, Common) :-subset(Results, Common).

subset([], _).
subset([A|B], C) :- member(A, C), subset(B, C).

intersection([], _, []).
intersection([A|B], C, [A|D]) :- member(A, C),
                                  intersection(B, C, D).

intersection([A|B], C, D) :-  \\+ member(A, C),
                              intersection(B, C, D).

member(A, [B|_]) :- A == B.
member(A, [B|C]) :- A \\== B, member(A, C).

conjoin((A,B), C, (A,D)) :- conjoin(B, C, D).
conjoin(A, B, (A,B))      :- A \\= (_,_).

split((A,B), A, B).
split((A,B), (A,A1), B1) :- split(B, A1, B1).
```

Fig. 3 Sequential Conjunction Replacement

```
call((A, B)) :- true          | call(A), call(B).
call(A)      :- primitive(A) | satisfy(A, _);
call(A)      :- true          | clauses(A, Cls), reduce(A, Cls, B), call(B).

reduce(Goal, [C|Cls], B1)    :- test(Goal, C, B)          | B = B1.
reduce(Goal, [C|Cls], B1)    :- reduce(Goal, Cls, B)      | B = B1.
reduce(Goal, [(C;_)|Cls], B1) :- test(Goal, C, B)          | B = B1;
reduce(Goal, [(_;C)|Cls], B1) :- reduce(Goal, [C|Cls], B) | B = B1.

test(Goal, Clause, B) :- match(Goal, Clause, Guard, Body) | B = Body, satisfy(Guard, _).

match(Goal, C, G1, B1) :- melt(C, (H :- G | B)) | Goal => H, G = G1, B = B1.
```

Fig. 4 Lingua Franca Meta-Interpreter