PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

Guarded Horn Clause Languages: Are They Deductive and Logical?

Carl Hewitt*
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Gul Agha†
Department of Computer Science
Yale University
New Haven, CT 06520
agha@cs.yale.edu

September 16, 1988

Abstract

Concurrent languages which use a procedural interpretation of guarded Horn clause resolution are an active area of research. We argue that proof theory does not provide a good model for the semantics of such languages. Although guarded Horn clause languages (*GHCL's*) use directed *forward chaining*, they dynamically add *arrival order assumptions* apart from those made in the program. Because of these additional assumptions, programs do *not* specify a logical deductive system. We also argue for an execution model of *GHCL's* which assumes a *guarantee of delivery* of communications. However, a logical reading of *GHCL* programs is not expressive enough to infer the consequences of a such a guarantee. It is our conjecture that the semantics of *actors* provides an adequate model for *GHCL*'s.

*The first author's work has been supported by DARPA under ONR contract number N00014-85-K-0124

†The second author's work has been supported in part by DARPA under ONR contract number N00014-86-K-0310.

1 Introduction

A family of concurrent logic languages using a notation based on Horn clause resolution have been defined. These languages include *Parlog* [CG86], *Concurrent Prolog* [Sha87], and *Guarded Horn Clauses* [Ued86]. A Horn clause consists of an atomic formula called the *head*, and a set of atomic formulae, called the *body*, with a "$\Leftarrow$" between the head and the body, and "&"s between the atomic formulae in the body, e.g.,

$$H \Leftarrow B_1 \& \ldots \& B_n$$

where an atomic formula has the usual meaning of predicate symbol applied to some functions. A Horn clause is taken to be universally quantified over all the variables that occur in the terms of its atomic formulae,

$$(\forall X, Y, Z)\, H \Leftarrow B_1 \& \ldots \& B_n$$

where X, Y, Z are the variables occurring in H and the B_i. A special case is where the

body is empty, for example,

$$fact(0, 1)$$

says that the factorial of 0 is 1, where $fact(X, Y)$ denotes the relation of Y being the factorial of X.

A program in a Horn clause language is a set of Horn clauses together with a *goal* which is either a single atomic formula or a conjunction (i.e., "and") of atomic formulae. It is called a goal because the system will try to find values for its variables which make it provable from the clauses in the program. The goal is taken to be (implicitly) existentially quantified. For example,

```
?- fact(3,Y).
```

asks what is the value of `Y` such that `Y` is the factorial of 3.

In order to prove a goal, a Horn clause programming system has to prove each of its atomic subgoals. In this way, a goal may create an AND node in a proof tree, since it calls for proving a conjunction. To prove an atomic subgoal, the system can try any Horn clause in the program whose head unifies with the subgoal.[1] These possibilities form an OR node in a proof tree, since a proof of any one will suffice.

Each possibility yields further information about the values of the variables, and also yields some further subgoals, from the body of the Horn clause used. Whenever one of the possibilities fails, the system tries to prove the same subgoal using a different clause. An OR node fails if each of its possibilities fails, and an AND node fails if any one of its subgoals fails.

Strictly speaking, a *guarded* Horn clause,

```
H :- G1,...,Gm | B1,...,Bn
```

is not really a Horn clause since it involves the *commit* operator "`|`". Here, as before, `H` is the *head* and `B1,...,Bn` is the *body*; in addition, `G1,...,Gm` is the *guard*. The procedural interpretation is that if all atoms in the guard succeed, then a system is *committed* to the clause, in the sense that no other clause will be tried for proving the goal in question; thus that goal will succeed if and only if all the atoms in the body succeed. For example, in *GHC* the recursive part of a factorial program could be written as follows:

```
fact(N,F) :- N>0 | -(N,1,N1),
                   fact(N1,F1), *(N,F1,F).
```

In addition these languages have *flat* versions which disallow recursion in a guard. In other words, only system predicates such as `N>0`, as opposed to user defined predicates, are allowed in guards. Thus, in Flat GHC (*FGHC*) there is no unification of variables in a goal with non-variables during head or guard evaluation; this means that variable occurrences in heads and guards have what is called "input mode." Flatness further simplifies evaluation and allows efficient implementation.

We will use *FGHC* notation for our examples but the discussion applies equally well to other guarded Horn clause languages (*GHCL's*).

2 Alternative Interpretations

We now consider some alternative interpretations of *FGHC*, as well as some alternative approaches to base-level implementation languages. Just as we did above, the developers of *FGHC* have traditionally given a *backward chaining* interpretation in which the head of a clause `H` is considered as a goal to be satisfied, the guards `G1,G2,...,Gm` as precondi-

[1] Two atomic formulae are *unified* by a substitution of values for their variables which renders them equal.

tions that must be satisfied before subgoals can be processed, and `B1,B2,...,Bn` as subgoals that must be proved to establish `H`. In backward chaining, the system hypothesizes a conclusion and uses the rules provided to find the hypothesis supporting facts [Win84].

However, it may be more natural to interpret *FGHC* in terms of constraint-driven or directed *forward chaining*, in which the head `H` of a clause is interpreted as a constraint to be satisfied, the guards `G1,G2,...,Gm` as preconditions that must be satisfied before other constraints can be processed, and `B1,B2,...,Bn` are interpreted as further constraints propagated from `H`.[2] In forward chaining, the system uses known facts to deduce new ones. In *GHCL's*, facts are represented by clauses.

The forward-chaining interpretation fits *GHCL's* better because *GHCL's* do *not* actually search for alternative ways to satisfy a goal. Instead, they just elaborate one possible solution to the constraints imposed by `H`. In *GHCL's*, once a guard is satisfied, the system is committed to that clause, and even if the clause fails to yield a value that contributes to an overall solution, no other clause will be tried. By contrast PROLOG *backtracks*, i.e., it undoes some variable bindings and tries to find other values for them that can satisfy the entire set of clauses.

The forward chaining interpretation of *GHCL's* can also be seen as a *message passing* interpretation, whereby given a clause

```
H :- G1,G2,...,Gm | B1,B2,...,Bn
```

the head `H` matches incoming messages, the guard atoms `G1,...,Gm` must all be satisfied before the clause can commit, and the body atoms `B1,...,Bn` all send messages after the clause commits. Under this interpretation, *GHCL's* are very similar to actor languages.

For each guarded Horn clause there is a corresponding Horn clause obtained by ignoring the commit operator, and this Horn clause has a legitimate declarative reading in logic. Nevertheless, the relationship between the Horn clause logical semantics and computations of *GHCL's*' programs remains to be characterized.

3 Arrival Order Nondeterminism

The execution of *GHCL's* involves an exponentially increasing number of assumptions about the order in which messages arrive. We can illustrate this issue with a simple example, a bank account that responds to `deposit`, `withdraw` and `balance` messages. This account can be represented by a variable, say `Acct1Messages`.

```
account(Acct1Messages, 100, 'Clark')
```

which means that the unification variable `Acct1Messages`, is the message stream for an account with balance 100 and owner Clark. The implementation of a shared account would consist of a a clause for each kind of message that might be processed by a bank account. Thus a withdrawal message might be handled in *FGHC* using two clauses, one for the case where there is enough money in the account to cover the request and the second for the case where there is not. Note that `[Head|Tail]` represents a list with head `Head` and tail `Tail` and `[ ]` denotes the empty list. The clauses may look something like this.

[2] We use the term constraint-driven in a very restricted sense. *GHCL's* are not constraint-oriented languages with the same generality as [Ste80]. The term *directed* is appropriate since the "goal" clause constrains which new facts are to be used in deriving other facts.

```
account([withdraw(Amount, Response)|
        MoreMessages],Balance,Owner):-
  /*Withdraw Amount into the account*/
 >=(Balance, Amount) |
  /*balance is greater than request*/
  plus(Balance, NewBalance, Amount),
   /*Bind NewBalance */
  Response = withdrawalReceipt(Amount,
                    Owner, NewBalance),
   /*Bind Response to a receipt*/
  account(MoreMessages, NewBalance,
             Owner).
   /*ask account to process MoreMessages
         with balance NewBalance*/
```

```
account([withdraw(Amount,Response) |
        MoreMessages],Balance,Owner):-
  /*Withdraw Amount from the account*/
 <(Balance, Amount) |
  /*balance is less than request*/
   Response = overdraftNotice(Amount,
                        Owner, Balance),
    /*Bind Response to an overdraft notice*/
   account(MoreMessages, Balance, Owner).
    /*ask account to process MoreMessages
    keeping the same balance and owner*/
```

If two users named Ueda and Shapiro need to share common access to Clark's account, then a stream merge must be explicitly declared. The following expressions declare the variables `UedaAcct1Messages` and `ShapiroAcct1Messages` such that when these two message streams are merged, the result is `Acct1Messages`:

```
merge(UedaAcct1Messages,
      ShapiroAcct1Messages,Acct1Messages)

UedaMessages = [UedaAcct1Messages |
                    MoreUedaMessages]

ShapiroMessages = [ShapiroAcct1Messages |
                     MoreShapiroMessages]
```

where the implementation of `merge` uses a standard technique for stream merges described in [Sha87]. Now Ueda and Shapiro can concurrently communicate with the account using their respective message streams. If Ueda attempts to withdraw 70 using

```
UedaAcct1Messages=[withdraw(70,
                        UedaResponse)]
```

while concurrently Shapiro attempts to withdraw 80 using

```
ShapiroAcct1Messages=[withdraw(80,
                        ShapiroResponse)])
```

then one of them is going to get an overdraft notice.

Let us try to state the possibilities in a logical framework. Let T_0 be the theory which is derived from the Horn clauses that are obtained by ignoring the commit operators in the above expressions. In order to logically derive from T_0 the values of `UedaResponse` and `ShapiroResponse`, some additional assumptions must be made. Making these assumptions, we can obtain new theories T_1 and T_2 by taking the union of T_0 and, respectively, the first assumption or the second assumption below:

$$\{Acct1Messages = [withdraw(70, UedaResponse), withdraw(80, ShapiroResponse)]\}$$

$$\{Acct1Messages = [withdraw(80, ShapiroResponse), withdraw(70, UedaResponse)]\}$$

The assumptions made to derive T_1 and T_2 are called *arrival order assumptions*. The requirement for such assumptions is an inherent characteristic of concurrent systems, where it is often the case that subsequent behavior can be critically affected by the order of arrival of communications.

Now in T_1, we can deduce

```
UedaResponse=withdrawalReceipt(70,Clark,30)

ShapiroResponse=overdraftNotice(80,Clark,30)
```

which states that Ueda gets his money while Shapiro gets an overdraft notice, whereas in T_2, we can deduce

```
ShapiroResponse=withdrawalReceipt(80,Clark,20)

UedaResponse=overdraftNotice(70,Clark,20)
```

which states that Ueda instead gets an overdraft notice while Shapiro gets his money.

The above discussion can be formalized in a metatheory which records the assumptions that need to be made in order to account for possible *FGHC* computations. We define a relation $\prec_c$ to represent the notion of an *immediate computational extension* of a theory.

$$(T \prec_c T_0) \Leftrightarrow \{(T \equiv T_0) \vee (T \equiv T_1)\}$$

More generally a notion *computational extension*, $(\prec_c^*)$ can be defined simply as the transitive closure of $\prec_c$ relation. In other words, a computational extension is a theory which includes assumptions that cannot be inferred from a logical reading of a given set of axioms but have instead been added by the computational system.

Because arrival order assumptions represent interleavings of messages (or, alternately the order in which the constraints are propagated), the number of these assumptions may grow exponentially with time. The need for these assumptions is not dependent on whether one is using a backward chaining or a forward chaining interpretation of *FGHC*. Furthermore, the *computationally extends* relation does not provide a basis for choosing between the two possibilities and therefore does not serve as an adequate logical semantics for *FGHC*.

Notice that the two theories have contradictory assumptions. Because a logical deductive system cannot allow two contradictory statements to be inferred from the same set of facts, it is our conjecture that there is no logical deductive system such that the computations of a *GHCL* program correspond exactly to the proveable theorems.

The computational extension of a theory is equivalent to augmenting the original theory with a set of axioms specifying the order in which constraints are to be processed, or in the message-passing interpretation, the order in which messages arrive. The computational system provides this extension.

In the context of *GHCL's*, an *event* is the binding of the head and the guards of a clause (i.e., a commitment). The *activation order* is the link between events caused by constraint propagation (or alternately, goal reduction). The *arrival order* is the order of messages on a stream; in case of shared streams, such as the one in Clark's account in the example above, this order is determined by the merge. The order of events in any concurrent system is a partial order obtained by the transitive closure of the arrival and activation orders.

Early work in actor semantics showed that the laws of parallel processing [HB77] imply realizability of events in a concurrent system in a global time [Cli81]. Global time is a retroactive construct and dependent on the frame of reference; it is constrained by the fact that it may not violate the transitive closure of arrival and activation orders. This limitation also applies to computational extensions. In actor semantics, a history consistent with the initial configuration is obtained by the transitive closure of the *possible transition* relation [Agh86].

4 Fair Merges

The merge described above need not be *fair*. If Ueda keeps sending messages, Shapiro's message may never be received. The assumption that communications sent will eventually be received is similar to the notion that a function when called will actually be invoked. The

guarantee of delivery of communications in actor systems means that such merges are fair. The term *complete merge* is more appropriate because there are a number of other definitions of *fairness.*

The guarantee of delivery simplifies reasoning about programs. For example, it can allow us to reason about convergence in the computation of a function despite the possible presence of other diverging invocations of the function. In practical terms, the guarantee implies that a continuously functioning operating system can be brought down gracefully [Agh86].

One consequence of the guarantee of delivery is *unbounded nondeterminism* in the results of the execution of a program. For example, a program can be written which will produce one of an infinite number of integers and which is nevertheless guaranteed to stop. Unbounded nondeterminism cannot be modeled by choice-point nondeterminism with a finite number of choices at any given time. This limitation is a simple consequence of Konig's lemma.

For example, using (nondeterministic) turing machines, it is impossible to specify a program which may have one of an infinite number of possible results and which is nevertheless guaranteed to halt. In a model of concurrency which assumes the guarantee of delivery (or fair merges), it is possible to write such programs.

The usual interpretation of *GHCL's* is based on choice-point nondeterminism between clauses that match a given set of constraints. Using this interpretation a fair merge cannot be defined. Shapiro [Sha87] provides two possible implementations of fair merge in Concurrent Prolog. However, these implementations assume that the underlying Concurrent Prolog machine is "weakly stable," a property which in turn cannot be defined in Concurrent Prolog.

Unbounded nondeterminism has been handled in actor semantics by defining a potentially infinite transition relation, called the *subsequent transition* [Agh86]. The subsequent transition relates a given configuration to the first configuration along any path (defined by the possibility relation) in which all communications pending in the given configuration have been processed. Such a relation can also provide an operational model of *GHCL's* with fair merge.

5 Comparison with Actors

We briefly compare programming in *FGHC* with programming in a core actor language. While the actor metaphor differs considerably, both the code and its execution have a rather similar functionality. The difference is that actors have the characteristics of object-based languages, whereas object-based languages have to be simulated on top of a language such as *FGHC.*

We provide the code for the shared account discussed above in an actor base-level implementation language named ACORE [Man86], which is a realization of the minimal actor language ACT [Agh86]. Each actor has a number of message handlers (indicated by => below), one of these handlers will be applicable to the incoming communication. Note that there is OR-concurrency between the message handlers and AND-concurrency between different commands within a handler. There is also AND-concurrency in the selection of a *pending communication*[3] which is to be processed "next," where the notion of "next" is relative to a frame of reference.

History sensitive behavior in actors is captured by the concept of *replacement behavior.* The replacement process is fully pipelined to

[3] A pending communication is a communication which has been sent but not processed.

provide maximal concurrency. In ACORE, if the replacement behavior can be denoted by a change in the parameters of the same behavior definition, a "*ready*" command is used. The command provides new values for the parameters. As soon as these values have been determined, the actor is ready to accept the next message.

The code for a communication handler[4] to process withdrawals in an account is as follows:

```
(DefBehavior account (balance owner)
 (=> (:withdraw amount)
  (If (>= balance amount)
   ;Withdraw amount requested
   (Then
    (Let ((newBalance (- balance amount))
     ;let newBalance be balance less withdrawal
     (Ready (balance newBalance))
     ;Account is ready for the next message
     (Return (withdrawalReceipt
              amount owner newBalance))))
    (Else
     (Ready)
     ;Account is ready for the next message
     (Complain (overdraftNotice
              amount owner balance)))))))
```

The interactions corresponding to those discussed above for *FGHC* are as follows:

```
(DefName Acct1 (Create account 1000 'Clark))
```

declares a new identifier named `Acct1` and binds it to a new account with balance 100 and owner Clark. Again suppose that two users named Ueda and Shapiro need to share access to Clark's account. The following commands give them the ability to communicate with `Acct1`:

```
(Send Ueda Acct1)

(Send Shapiro Acct1)
```

[4]Also called a "method" in object-oriented languages.

Now if Ueda attempts to withdraw 70 from `Acct1` using the command `(withdraw Acct1 70)`, while concurrently Shapiro attempts to withdraw 80, then the operation of the account will be serialized so that one of them will get a withdrawal receipt and the other an overdraft complaint.

6 Conclusions

Code and interactions in *FGHC* are very similar in structure and results to those in actor base-level implementation languages, as the above example shows in detail. This is rather surprising and confirms the fact that a *GHCL* such as *FGHC* has the requisite structure and functionality to serve as a base-level implementation interface between concurrent hardware and software, much in the same way as actor languages can.

The closeness of *FGHC* to actor core languages raises the issue of how their semantics are related. A denotational semantics for actor languages based on system configurations has been defined. Actor Theory provides a meaning for the scripts of actor programming languages, obtained recursively by analyzing the script as a system of communicating actors [The83][Agh86]. On the other hand, to our knowledge no formal semantics has yet been proposed for *FGHC*. It is true that some types of reasoning about *FGHC* can be carried out using a declarative reading of the programs. Such reasoning is in fact similar in structure to the concept of *serializer induction* in actors which permits the inference of static properties of a program.

Another interesting issue is of the relative efficiency of implementing the two languages. Because *FGHC* is based on the use of unification variables to implement local changes, it may well be that actor based system may have greater efficiency than *FGHC*. Furthermore, it

appears that *FGHC* can be implemented using actors with the same efficiency as any other implementation method.

The base-level language of the classic von Neumann architecture provides instructions to perform a sequence of fetch-compute-store cycles. This implies that von Neumann architectures and their programming languages such as COBOL and FORTRAN are both inherently sequential; and in fact, as we all know, they fit together very well. Unfortunately, the enormously long sequences of fetch-compute-store cycles imply a tremendous traffic between the processor and memory, and so the link between them becomes a chief limitation on the speed of execution. One of the objectives of new base-level languages, or to use Shapiro's terminology, high-level machine languages, is to break this bottleneck.

A number of base-level languages have been proposed besides *GHCL's* and actor languages. Some of these languages, such as vector and array processing languages—which provide the model for machines like the Cray—are only slight extensions of the basic von Neumann model. Others, such as the SIMD languages, are quite special purpose, though elegant and powerful for those applications that can fully use them. Actor languages and guarded Horn clause languages, on the other hand, are general purpose base-level languages for implementing concurrent systems. Research on the relationship between these two kinds of languages should continue to be fruitful.

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[CG86] K. J. Clark and S. Gregory. Parlog: parallel programming in logic. *ACM TOPLAS*, 8(1):1–49, 1986.

[Cli81] W. D. Clinger. *Foundations of Actor Semantics.* AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[HB77] C. Hewitt and H. Baker. Laws for communicating parallel processes. In *1977 IFIP Congress Proceedings*, pages 987–992, IFIP, August 1977.

[Man86] Carl R. Manning. Acore: an actor core language reference manual. Internal Memo, September 1986. MIT Message Passing Semantics Group Design Note: 7.

[Sha87] E. Shapiro. A subset of concurrent prolog and its interpreter. In *Concurrent Prolog: Collected Papers*, pages 27–83, M.I.T. Press, 1987.

[Ste80] G. L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* A.I. Tech Report 595, MIT Artificial Intelligence Laboratory, August 1980.

[The83] D. Theriault. *Issues in the Design and Implementation of Act2.* Technical Report 728, MIT Artificial Intelligence Laboratory, June 1983.

[Ued86] K. Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, 1986.

[Win84] P. Winston. *Artificial Intelligence.* Addison-Wesley Publishing Co., Reading, MA, 1984.