MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1106 October, 1990

Experience with Acore: Implementing GHC with Actors

Jeff Palmucci
Carl A. Waldspurger
David Duis
Paul Krause

Abstract: This paper presents a concurrent interpreter for the programming language Guarded Horn Clauses, abbreviated GHC. GHC is a general-purpose concurrent logic programming language. It has a clean, simple semantics based upon unification and choice nondeterminism.

Unlike typical implementations of GHC in logic programming languages, the interpreter is implemented in the Actor language Acore. The primary motivation for this work was to probe the strengths and weaknesses of Acore as a platform for developing sophisticated programs. We chose to implement a concurrent interpreter for GHC because this large, complex application provided a rich testbed for exploring Actor programming methodology.

The interpreter is a pedagogical investigation of the mapping of GHC constructs onto the Actor model. Because we opted for simplicity over efficiency, the interpreter is inefficient in both time and space.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-0460, in part by the Systems Development Foundation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

Contents

1 Introduction **3**
- 1.1 Goals . . . 3
  - 1.1.1 Actor Programming Methodology . . . 3
  - 1.1.2 A Pedagogical Investigation of GHC . . . 3
- 1.2 Roadmap . . . 4

2 A Brief Overview of GHC **5**
- 2.1 Procedures, Programs, and Horn Clauses . . . 5
- 2.2 Terms and Values . . . 5
- 2.3 Parallelism . . . 6

3 Relevant Acore Concepts **10**
- 3.1 Actors: Behaviors and Acquaintances . . . 10
- 3.2 Message Passing . . . 10
- 3.3 Actor Locking . . . 10
- 3.4 Sponsors, Ticks, and Stifling . . . 11
- 3.5 Forwarding Actors . . . 11
- 3.6 Diagramming Actor Computations . . . 11

4 Mapping GHC into Acore **12**
- 4.1 Unification and the Rules of Suspension . . . 12
  - 4.1.1 Unification Algorithm . . . 12
  - 4.1.2 A Problem with Unification . . . 12
  - 4.1.3 Implementation of the Rules of Suspension . . . 14
  - 4.1.4 Unification and Compound Terms . . . 15
- 4.2 Control Structure . . . 16
- 4.3 Example . . . 20

5 Future Work **24**
- 5.1 Elimination of Suspended Variables . . . 24
- 5.2 Speculative Concurrency . . . 25
- 5.3 Decentralization . . . 25

6 Conclusion **27**

7 Acknowledgments **28**

A Actor Behaviors Used To Implement GHC **31**
- A.1 Templates and Activations . . . 31
- A.2 Common Handlers . . . 31
- A.3 Constants . . . 32
- A.4 Predicates and Functions . . . 32
- A.5 Procedures . . . 33
- A.6 Clauses . . . 34
- A.7 Variables . . . 35

B Difficulties with Acore 36
B.1 Exception Handling 36
B.2 Resource Encapsulation with Sponsors 37
B.3 Locking and Serialization 38
B.4 Sharing Mechanisms 39
B.5 Data Abstraction 39
B.6 The Acore Environment 40

1 Introduction

This paper presents a concurrent implementation of the programming language Guarded Horn Clauses, abbreviated GHC. GHC is a general-purpose concurrent logic programming language [Ued86]. It has a clean, simple semantics based upon unification and choice nondeterminism.

This interpreter is, to the best of our knowledge, the only implementation of GHC which is *object-oriented.* Unlike typical implementations of GHC written in logic programming languages, this interpreter is written in the Actor language Acore [Man87]. Acore is a concurrent, object-oriented language designed for open systems.

Further, previous GHC implementations have relied upon the *closed-world assumption*, which states that all information necessary for a computation is known at the time of its invocation. Anything which is not known and cannot be derived from existing information is assumed to be false. In contrast, our implementation does not make this assumption. It is an *open system* in the sense that program clauses can be added dynamically as a computation unfolds [Hew85]. This allows new information to be introduced and used as it becomes available. Thus a program which relies on information which is not immediately available does not fail, but instead *suspends*, waiting for the arrival of this information.

Finally, this interpreter is one of the few complete implementations of full GHC; most previous work has focused on the *flat* subset of GHC (FGHC), which allows no user-defined predicates in guard clauses. Full GHC is considerably more difficult to implement due to the complex and subtle interactions of GHC's rules of suspension.

1.1 Goals

1.1.1 Actor Programming Methodology

Our primary motivation was to probe the strengths and weaknesses of the Actor language Acore as a platform for developing sophisticated programs. We chose to implement a concurrent interpreter for GHC because this large, complex application provided a rich testbed for exploring Actor programming methodology.

Our work explores the challenges of programming in a model which significantly differs from conventional sequential and parallel models. The Actor model is characterized by asynchrony and nondeterminism, factors which alter the way a programmer must reason about computations. The interpreter exploits many actor concepts and techniques which are useful in such an environment.

As of this writing, the GHC interpreter is one of the largest, most sophisticated programs written in Acore. Developing the interpreter provided us with extensive experience using asynchronous message-passing, behavior replacement, forwarding actors, speculative concurrency, and resource encapsulation with sponsors.

1.1.2 A Pedagogical Investigation of GHC

Implementing GHC in an object-oriented language provides those familiar with object-oriented programming languages with an explicit representation of GHC constructs and

control flow. In logic-based interpreters, a great deal is left implicit in the declarative style of the implementation language. Newcomers to logic programming may find that this presents a barrier to understanding the mechanisms of GHC.

This interpreter was also intended to be more expressive than existing GHC interpreters in two ways. First, the interpreter implements full GHC, instead of its more limited flat subset, FGHC. Second, by implementing GHC in a language designed to support open systems, we allow programmers to avoid the limitations of the closed-world assumption.

The interpreter is a pedagogical investigation of the mapping of GHC constructs onto the Actor model. Our interpreter is designed to expose maximal parallelism by adopting a very fine-grained model of concurrency. Because we opted for simplicity over efficiency, the interpreter is inefficient in both time and space. A discussion of possible optimizations is given in the Future Work section.

1.2 Roadmap

The next two sections present quick overviews of our source and destination languages, GHC and Acore. Section 4 details the implementation of the interpreter, focusing first on the unification structure and then on the control structure. Section 5 contains ideas for future optimizations of the interpreter. Appendix A gives specifications for the behaviors of the actors used in our implementation. Finally, Appendix B outlines problems we encountered in using Acore, and suggests several improvements.

2 A Brief Overview of GHC

This section presents a terse description of GHC. For more details, consult [Ued87] and [Ued86]. The reader already familiar with GHC may wish to skip to the next section.

2.1 Procedures, Programs, and Horn Clauses

A GHC program is a set of Guarded Horn Clauses, which are either *goal clauses* or *program clauses.* In GHC a goal clause is written

$$:- G_1, \ldots, G_n.$$

and program clauses have the form

$$H :- G_1, \ldots, G_m \mid B_1, \ldots, B_n. \quad (m, n \geq 0).$$

A goal clause is a conjunction of goals (i.e., $G_1 \wedge \cdots \wedge G_n$, where $n \geq 1$), that is used to start a computation by invoking program clauses. Each goal is a term of the form $p(X_1, \ldots, X_k)$, where $k \geq 0$.

The program clauses in GHC are sets of subgoals which satisfy a particular goal. In the program clause above, H is the *head*, the G_i are the *guard* goals, $\mid$ is the *commit operator*, and the B_j are the *body* goals. Logically, the program clause means that the conjunction of the goals $G_1, \ldots, G_m, B_1, \ldots, B_n$ implies the goal H. The trivial clause *true* is often used to express an empty clause.

A GHC computation is invoked by a request to resolve a goal clause. The system attempts to satisfy the goal clause via *resolution* — matching program clauses are used to *reduce* the goals into a number of subgoals, with the eventual aim of deriving the empty clause, *true*. In essence, an original "caller" goal is reduced into the body goals of a matching program clause. This process is repeated recursively until the computation "bottoms out" at primitive goals that either succeed or fail.

The extra-logical commit operator is used to control subgoal resolution in a GHC program. In a procedural interpretation, commit resembles an **if-then** structure. For example,

$$fact(X,Y) :- X = 0 \mid Y = 1.$$

can be interpreted as: **if** X unifies with 0 **then** unify Y with 1. Unlike a procedural '**if**', there is no '**else**' case, i.e., if the goal $X = 0$ is not satisfied, then the guard will not commit and the body will not export any bindings. If a procedure has multiple clauses they will execute concurrently. The first clause whose guards are satisfied commits, aborting the execution of the other active clauses. This non-backtracking interpretation distinguishes GHC from theorem provers such as Prolog.

2.2 Terms and Values

A goal is a formula in the Horn clause logic. A formula may be a variable symbol such as p or q (known as an *atomic formula*), or may be constructed by applying a predicate

symbol to a set of terms, such as $p(X_1, \ldots, X_k)$. Each term may itself be an atomic formula or consist of a predicate or function symbol applied to a set of terms[1]; GHC is a compositional logic. Each term is assigned a value according to the semantics of GHC and a set of constraints expressed by program clauses.

An unbound variable does not have a value initially assigned to it. Its value is determined by its *unification history*, i.e., the serialization of all unifications on that variable. For example, the serialization $A = B$, $B = 1$ gives A the value 1, but $C = D$ ensures only that C and D will have the same value, if any.

Compound terms in argument positions are not solved as goals by application of program clauses. Instead, they act as data structures, each having itself as a value. For example,

$$\begin{array}{rcl} & :- & cdr(cons(1, cons(2,3)), Result). \\ cdr(cons(A,B),C) & :- & true \mid C = B. \end{array}$$

unifies *Result* with the data structure $cons(2,3)$ even though *cons* may not yet (or ever) have any defining program clauses.

The value of a function application, $f(X_1, \ldots, X_n)$, is determined by calling the function f on the value of the arguments $X_1, \ldots, X_n$. A function call with no arguments is a constant and yields itself as a value. The only non-constant functions in GHC are system defined. System functions, such as the arithmetic operators $+$, $-$, $*$, and $/$, are used to implement primitives that return values.

GHC also contains some system defined predicates [Ued86]. These goals take a special infix syntax, as opposed to user-defined goals which are typically written $p(E_1, \ldots, E_n)$. Although Ueda is able to derive the rest of these predicates from the primitive unify predicate (=) by using an infinite number of program clauses, in practice they must be provided by the language implementation. Much of the expressive power of GHC stems from the subtle interactions of these system predicates.

Primitive	*Meaning*
=	Unifies LHS with RHS
:=	Unifies LHS with value of RHS
=:=	Unifies value of LHS with value of RHS
<>	Succeeds iff value of LHS does not unify with value of RHS.
=<	Succeeds iff value of LHS unifies with value of RHS or any of its successors.
>=	Succeeds iff value of LHS unifies with value of RHS or any of its predecessors.
<	Succeeds iff value of LHS unifies with any successor to value of RHS.
>	Succeeds iff value of LHS unifies with any predecessor of value of RHS.

2.3 Parallelism

GHC was designed as a simple, general-purpose, inherently parallel programming language. Here *inherently parallel* means a programming language which is by nature parallel—instead of providing primitives which allow for *explicit* parallelism on top of an inherently sequential language, rules or primitives are provided to control *implicit* parallelism only

[1] By convention, variables are capitalized, and constants (functions and data structures) are lowercase.

when necessary. Examples of languages which utilize explicit parallelism are MultiLisp [Hal85], which employs *futures* to achieve concurrency, and Argus [Lis83], which provides constructs such as *guardians* and *coenter* statements for concurrent actions and processes. Examples of implicitly parallel languages include Acore [Man87], Id [Nik87], and Flat Concurrent Prolog [Sha87].

The resolution of non-guarded Horn clauses is implicitly parallel in two different respects. First, there is OR parallelism between the clauses of a procedure. At least one of the clauses that match the goal must satisfy its guard for a successful goal resolution. The semantics of the procedure do not explicitly suggest which clause to fire. Secondly, AND parallelism exists between the goals in a clause. *All* of the clause's goals must succeed for the clause to succeed, but again, no order is implied.

Guarded Horn Clauses place further restrictions on OR-parallelism. In GHC, it is only possible for one clause in a procedure to commit. This should be contrasted with Prolog, whose backtracking control structure guarantees that *all* clauses will eventually fire.[2]

This non-backtracking restriction makes GHC easier to implement efficiently, but incomplete as a theorem prover. Since only one OR-parallel path is taken, it is possible to bind a logic variable so that it will appear the same to everyone that has access to it. Concurrent logic programming languages which allow multiple paths usually require an environment mechanism in which the value of a variable is chosen by the branch of computation that is taken. GHC has no need for such a mechanism.

Since a logic variable can only be assigned a value once, the process of choosing the OR-parallel branch to follow cannot be allowed to affect this value. Therefore, observable bindings in GHC program can only be made after a clause commits, and the OR-parallel branch is chosen. GHC avoids exporting bindings through its two *rules of suspension*, [Ued86]:

(a) The guard of a clause can never export any binding to (or, make any binding which is observable from) the caller of that clause, and

(b) The body of a clause cannot export any binding to (or, make any binding which is observable from) the guard of that clause before that clause is selected for commitment.

The rules of suspension also have an important effect on the AND-parallelism of GHC. Any uncommitted goal that attempts to export a binding must suspend until the binding is made through a branch of computation which has already committed. Conceptually, AND-parallel branches of computation will suspend until enough variables are instantiated by other branches to choose a single correct branch of computation.

As an example of the effect of these rules, consider the following example, which appears in [Ued86]:

$$\begin{array}{rll} & :- & p(X), q(X). \\ p(Y) & :- & Y = ok \mid true. \\ q(Z) & :- & true \mid Z = ok. \end{array}$$

[2]However, in Prolog implementations which support the *cut* operator, backtracking can be explicitly circumvented.

The top level goal $p(X), q(X)$ will succeed, but along the way we see an excellent example of the interplay of the rules of suspension.

1. Both subgoals execute in parallel. However, $p(X)$ must suspend: the binding $Y = ok$ in the guard would export a binding for X, which would violate the first rule of suspension.

2. The guard of $q(X)$, on the other hand, is the trivial *true*, and thus $q(X)$ commits and can export the binding $Z = ok$ from its body.

3. This binding in turn binds X to ok via the intervening variable Z. This does not violate the second rule of suspension, since $q(X)$ has committed, and therefore its body may export bindings.

4. At this point $p(X)$ can unsuspend since its guard goal $Y = ok$ no longer exports a new binding through X. Y has been bound to ok above, and the unification is reduced to a simple equality check.

5. The trivial body goal *true* succeeds, and the goal $p(X)$ succeeds.

6. Since each goal in the top-level clause has been satisfied, the top-level clause is satisfied and computation halts.[3]

Care must be taken in defining the guard of a GHC clause. One view of unification is that it is the process of substituting another object for a logic variable. The inverse operation, which Ueda defines as *anti-substitution* [Ued86] should not affect the semantics of the clause.

For example, given some goal $p(ok)$, we can make an equivalent conjunction through anti-substitution: replacing the term ok by the fresh variable T and conjoining the goal $ok = T$ to the goal $p(T)$. The clauses $p(ok)$ and $p(T), ok = T$ are equivalent.

Because of the anti-substitutability property, the head must be considered to be a part of the guard, and variables appearing in the head are subject to the same restrictions as variables appearing in the guard. For example, the following clauses are equivalent:[4]

$$\begin{array}{llll} p(a,b) & :- & true & \mid true. \\ p(A,b) & :- & A = a & \mid true. \\ p(a,B) & :- & B = b & \mid true. \\ p(A,B) & :- & A = a, B = b & \mid true. \end{array}$$

as are:

$$\begin{array}{llll} fact(0,Y) & :- & true & \mid Y = 1. \\ fact(X,Y) & :- & X = 0 & \mid Y = 1. \end{array}$$

[3] Additional examples involving the rules of suspension can be found in [Ued86].

[4] These examples are relevant to our implementation because we syntactically desugar predicates out of clause heads. This ensures that heads contain only distinct variables, and facilitates a more convenient implementation of the first rule of suspension.

However, since the head of a GHC clause is part of the guard,

$$fact(0,1) :- true \mid true. \tag{1}$$

$$fact(X,Y) :- X = 0, Y = 1 \mid true. \tag{2}$$

are equivalent to each other, but *not* to either of the preceding two clauses, because the guard goal $Y = 1$ cannot export a binding from the guard. This does not mean that (1) and (2) are not legal GHC clauses, merely that neither will immediately unify with

$$:- fact(0, Answer). \tag{3}$$

3 Relevant Acore Concepts

In this section, we will discuss our approach to implementing GHC in Acore. We introduce necessary Acore concepts, describe the difference between *templates* and *activations*, and then outline the relevant *acquaintances* and *behaviors* of the actors we use.

The following sections present an extremely terse introduction to the Actor model and the Acore programming language. An introduction to Acore programming is presented in [Man90]. For additional information, the reader is urged to consult [Agh86] and [Man87]. A detailed description of the actor behaviors used in this system can be found in Appendix A.

3.1 Actors: Behaviors and Acquaintances

Acore is an object-oriented programming language, and as such, many of its constructs correspond to those in other languages. Objects are called *actors*. An actor is defined by specifying a *behavior* and a set of *acquaintances*. A behavior is a set of *handlers*, which correspond to methods. Every actor has the ability to *replace* its current behavior with a new behavior.[5] Acore acquaintances correspond to instance variables (such as in Smalltalk) or slots (such as in CLOS). An actor's acquaintances are those other actors to whom the actor may send messages. Acore also provides *complaints*, which correspond to exceptions in other languages.

3.2 Message Passing

Unlike message passing in many object-oriented languages based on call-return semantics, Acore messages are asynchronous and delivered in a nondeterministic order. Consider an actor A_1 which sends two messages M_1 and M_2 to another actor A_2. M_2 may be sent without waiting for a response to M_1, and in fact A_2 may even receive M_2 *before* M_1.

3.3 Actor Locking

Message handlers atomically lock all of an actor's acquaintances when an incoming message is accepted for processing. No other handlers may begin execution while the actor is locked. Each message handler defined for an actor's behavior is specified to be either **serialized** or **unserialized**. An **unserialized** handler is not permitted to modify an actor's state, and thus immediately unlocks the actor. A **serialized** handler unlocks the actor only after a **ready** or **replace** expression specifying the next state of the actor (i.e. the new values for its behavior and acquaintances) has been executed. Additional expressions may follow the execution of the **ready** or **replace**, but they are evaluated using the old values of the actor's acquaintances. This allows actors to exploit concurrency by *pipelining* the processing of messages.

[5] Smalltalk-80 provides a similar method defined on class **Object** called **become:** [Gol83].

3.4 Sponsors, Ticks, and Stifling

Sponsors are the Acore encapsulation of resource allocation [Man87]. Every message has as one of its acquaintances its *sponsor*, from whom it must periodically request "ticks." Ticks are the basic unit of processing in the Acore world: it costs one tick to send a message. When a transaction runs out of ticks, it must request more from its sponsor. Sponsors are arranged hierarchically with the end-user at the top: when the top-level sponsor runs out of ticks, it must request more from the end-user. If a transaction is denied more ticks by its sponsor, its execution is terminated.

3.5 Forwarding Actors

A commonly used Acore behavior is that of a *forwarding actor.* Forwarding actors forward all messages they receive to their forwardee. Forwarding actors are used heavily in our implementation of unification (see Section 4.1).

3.6 Diagramming Actor Computations

Throughout this paper, we present various examples in the form of actor event diagrams [Agh86]. The format of these diagrams is fairly simple. Each actor depicted in a diagram is represented by a vertical line, its *lifeline.* Time, in relation to an actor, continues downward along its lifeline. Messages between actors are represented as arrows between lifelines. The content of the message appears at the origin of the arrow in the form *:keyword arguments.*

Only the messages and actors that are directly involved in an example are shown in the diagram. Our lifelines are simplified and serve only as actor abstractions, informally presenting an intuitive depiction of actor interactions.

4 Mapping GHC into Acore

In this section we present our implementation of GHC, focusing on the manner in which GHC constructs can be represented using the Actor model.

4.1 Unification and the Rules of Suspension

4.1.1 Unification Algorithm

A central issue in any GHC implementation is its unification algorithm, and the Actor model permits an elegant implementation of unification. Since all GHC data types in our implementation are actors, we perform the unification of two actors by changing one actor into a forwarding actor to the other. To an observer, the two actors are effectively the same actor, since an observer cannot distinguish a forwarding actor from its forwardee. Further, the forwardee cannot tell whether it has received a message directly from the sender or via a forwarder.

Unfortunately, this implementation of unification is directional. In the next two sections, therefore, we will refer to the actor handling the unification as the "source", and the other as the "target."

A simple example will clarify the consequences of our directional unification scheme. We may legally unify an unbound variable *Foo* to a constant *bar*, as diagrammed in Figure 1a, but the converse is illegal: unifying *bar* to *Foo* has the effect of changing it from a constant to an uninstantiated variable. Our unification algorithm must differentiate between the two and perform the unification only **from** *Foo* **to** *bar*, no matter which of the two initially received the **:unify** message. This reversal is diagrammed in Figure 1b.

If both objects have been bound to constants, then the unification succeeds only if the two objects are equivalent. Figure 1c diagrams a failing unification. For compound terms, we simply check that their names are identical and then recursively unify their arguments.

One obvious problem with this unification algorithm is the possibility of extremely long forwarding chains. To eliminate this problem in most cases, we never forward to another forwarder, but always to the target's forwardee.

4.1.2 A Problem with Unification

A complication occurs in our unification algorithm when actor locking is taken into account.[6] Consider the following GHC program:

$$:- \quad A = B, B = A.$$

Figure 2 diagrams the message sends that result from a naive interpretation of our directional unification algorithm. The message **:unify** A will be sent to the unbound variable actor B, and **:unify** B will be sent to the unbound variable A. The first of these goals to execute will work perfectly. One of the unbound variables will become a forwarder

[6] See Appendix B.3 for a more detailed discussion of Actor locking.

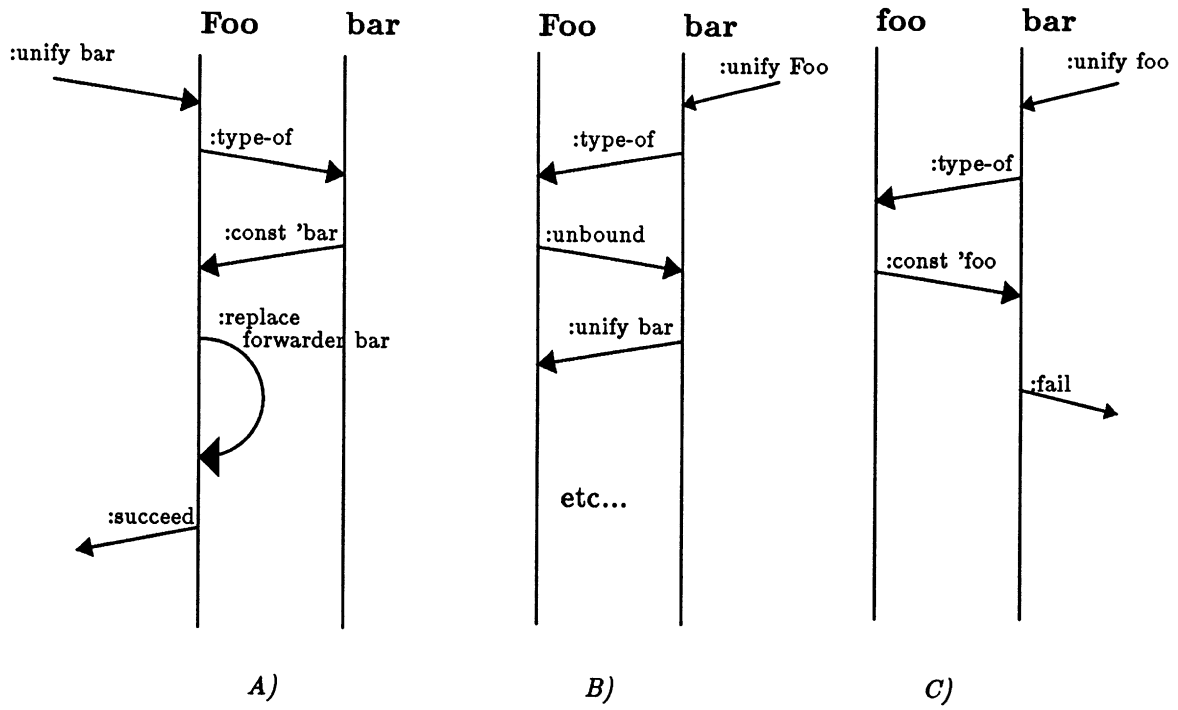

Figure 1: Examples of unification operations. We maintain the GHC convention that capitalization represents a variable. Notice that the direction of the unification must be switched in *B*.

to the other. However, notice that the second goal will do the same thing. We will thus be left with a forwarding cycle. Any messages sent to either of these "variables" will become trapped in a loop.

To remedy this problem, a variable actor must not be allowed to unify with itself. A simple equality check is insufficient, however, due to the semantics of actor locking. Since an unbound variable's behavior may be replaced by a forwarding behavior during unification, the unification handler must lock the actor to prevent the processing of other messages. Even a simple check must be expressed as a message-send to the actor. Thus, if a variable actor is actually unifying to itself, this message will be buffered until the actor is unlocked. Since the unlocking can only occur after a successful check, the unification will result in deadlock.

In order to alleviate this problem, we exploit the concept of actor identifications. Each actor must have a unique id in the system. All actor ids are completely ordered. Since logic variables are represented by actors, the creation of a variable will automatically yield a unique actor identification number. When unifying two unbound variables, we always forward in the direction of ascending id ordering. This eliminates the possibility of forwarding

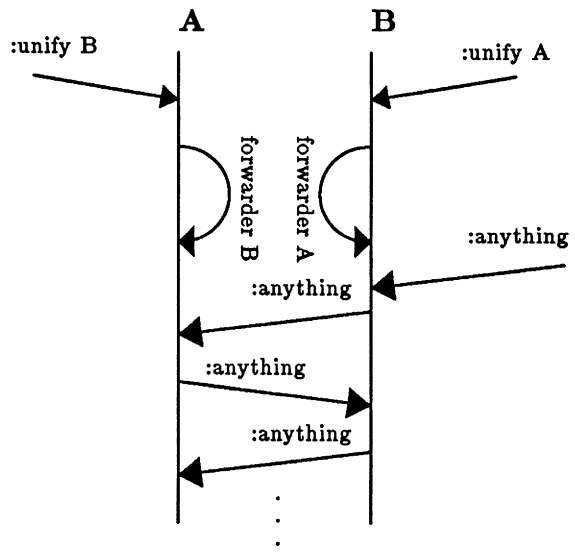

Figure 2: An example of a forwarding cycle.

cycles.[7]

4.1.3 Implementation of the Rules of Suspension

The rules of suspension specify that certain variables may not export bindings, and that therefore certain unifications must *suspend* until the bindings in question are legal. To implement this we differentiate between *normal* and *suspended* variables. The protocol for suspended variables is different than that for normal variables. There are two methods for unifying suspended variables, the normal **:unify** and a special **:passive-unify**. A *passive-unify* is a directional unification. That is, it cannot export a binding from the source to the target, but it can import such unifications from target to source.

We use suspended variables in the head of a clause to prevent the export of illegal bindings, as required by the rules of suspension.[8] Likewise, when we run the bodies of clauses concurrently, we must make all variables shared by guard and body suspended, since the second rule of suspension states that the body must not export bindings to the guard.

When a suspended variable is *passive-unified* to a constant, we proceed as if the variable were not suspended, since the unification cannot export a binding which is visible to the outside world, as in Figure 3a.[9]

[7]This may not be a feasible solution for future actor systems that may not have unique actor ids.

[8]Static analysis techniques could avoid suspending any variables which can be determined to be nonexporting. See Section 5 for further discussion.

[9]A new name for the suspended variable was GENSYMed when its enclosing clause or goal activated, and therefore the variable is not visible to the outside world. See Figure 10.

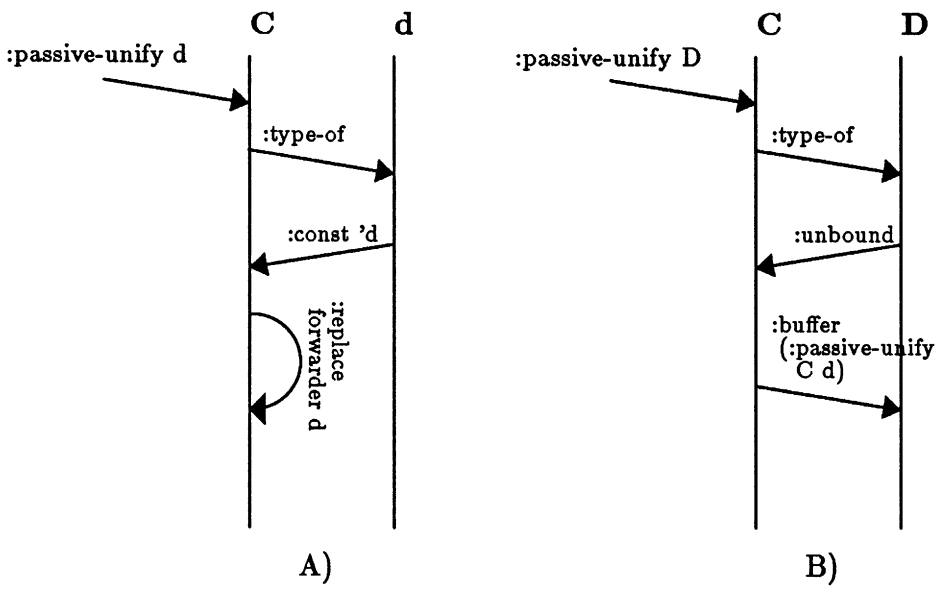

Figure 3: Some examples of passive-unification. C is a suspended variable, D is unbound, and b is a constant. In A the target is a constant, so we can proceed as in normal unification. In B, however, the passive-unification message is buffered on the unbound variable D. This message will be resent when D becomes bound. The passive-unification will then continue as in A.

When the target of the unification is unbound, however, we cannot perform a normal unification. This would introduce the possibility of instantiating the target variable through the suspended variable. To solve this problem, the target variable buffers the **:passive-unify** message, as shown in Figure 3b. When that target variable is instantiated, it re-sends the **:passive-unify** messages it has in its buffer. Since the target variable is now a constant, we may perform a normal unification.

4.1.4 Unification and Compound Terms

The unification and passive-unification of compound terms is a straightforward extension of the methods above. Unification of a compound term to a variable is reversed: the **:unify** or **:passive-unify** message is sent to the variable instead of the term. Unifica-

tion between two compound terms fails if their names and arities are not the same. If they are the same, we recursively unify corresponding arguments, failing as soon as any sub-unification fails, succeeding only after all succeed.

Passive-unifying a suspended variable to a compound term is slightly more complex. Since a **:unify** sent to the term will be rerouted to the variable as outlined above, and since directly unifying the suspended variable to the compound term involves the same problems as unifying to an uninstantiated variable, we must adopt a new approach.

When passive-unified to a compound term, a suspended variable replaces itself with a new term having the same name and arity as the target. The arguments of the new term, however, are all suspended variables. These arguments are then passive-unified to their counterparts in the target term.[10]

When a clause commits, bindings made in the body may be exported. At this time all suspended variables should become normal variables, and their bindings should become permanent. To accomplish this, we send an **:unsuspend** message to each suspended variable in the body.[11] When suspended variables receive this message, they become normal variables and their suspended bindings become visible to the world.

4.2 Control Structure

The following simple factorial program will serve to demonstrate our control structure:

$$fact(0,Y) :- true \mid Y = 1. \tag{4}$$

$$fact(X,Y) :- X > 0 \mid fact(X-1,Z), Y := X * Z. \tag{5}$$

$$:- fact(A,B), A = 5. \tag{6}$$

The structure of a running GHC program takes on the form of an **AND-OR** tree. The root branching in the tree is always an **AND**-branching corresponding to the **AND** of top-level goals, such as those in (6). Each of these goals will succeed if any of its procedure's clauses succeeds. Thus each goal defines an **OR**-branching of degree equal to the number of clauses in its procedure.[12] Each clause introduces an **AND**-branching, which succeeds only if all of its goals succeed. Thus successive levels of goals and clauses combine to form an **AND-OR** tree.

To control the computational explosion of the tree, GHC prunes all but one clause at each **OR** branching in the tree. This occurs when a clause *commits*: when all of its guard goals have succeeded. Conceptually, a clause which commits must be the correct path to the success of the goal. All competing clauses may be discarded.

Our implementation of the control structure is quite straightforward. We use two behaviors to control the two types of branching in GHC. For each procedure used when resolving a goal, we create a procedure actor. A procedure actor is responsible for the resolution of

[10] This method is easily understood but inefficient. See Section 5 on Future Work for further discussion.

[11] We send the **:unsuspend** message only to the body variables since the semantics of GHC do not permit guard clauses to export bindings.

[12] Primitive goals form leaves.

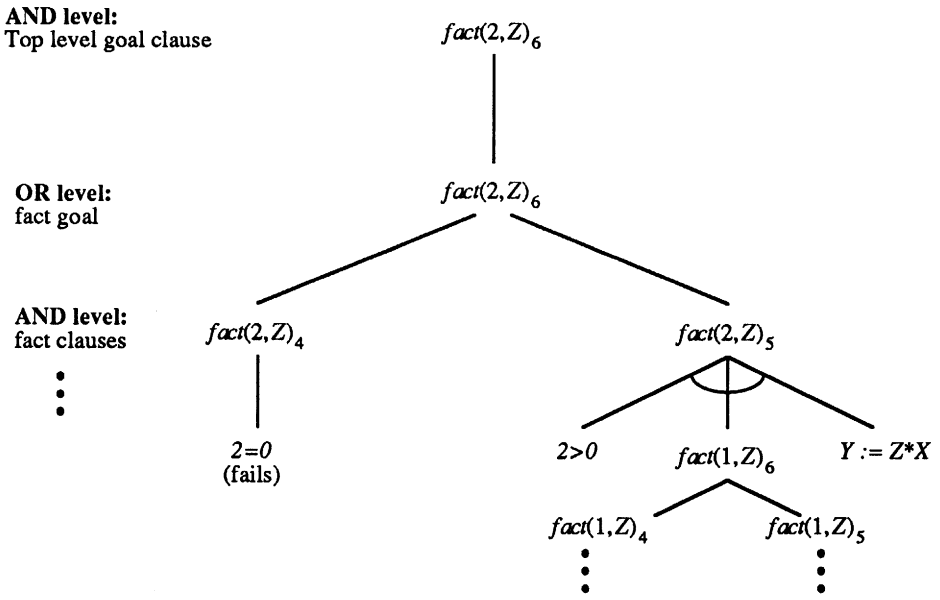

Figure 4: *AND-OR* branchings in the tree for the factorial 5 example. Subscripts correspond to equation labels from the example above.

1. For each clause:
   (a) Create a new sponsor for the clause to run under.
   (b) Create a new clause actor to handle this clause resolution.
   (c) Start execution of the clause by the method outlined in figure 7.
2. Upon receiving a **:request-commit** message from one of these clauses:
   (a) Stop computation in the other clauses by stifling their sponsors.
   (b) Send a **:committed** message to the calling clause.
3. Propagate any **:fail** or **:succeed** messages to the calling clause.

Figure 5: Pseudocode for a procedure actor during goal resolution.

the clauses which define the procedure. Pseudo-code describing procedure actor behavior is detailed in figure 5, and an example run appears in Figure 6.

For each clause in a procedure, we create a clause actor. A clause actor attempts to compute the **AND** of its guard and body predicates. Each of these predicates is in turn either a user-defined procedure or one of the primitives listed in section 2.2. If the predicate is a primitive, the tree has bottomed out in a leaf node, from which a **:fail** or **:done** message will be returned. Pseudocode describing clause actor behavior may be found in Figure 7, and a sample run appears in 8.

At the top level is a goal-handler actor, a specialized clause actor which prints the

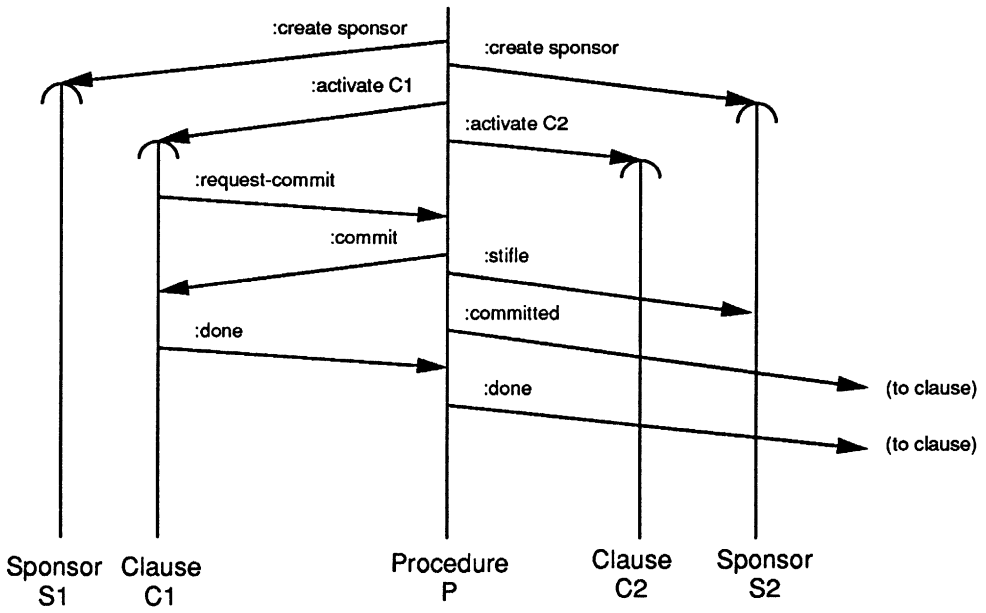

Figure 6: An example of a procedure with two clauses. In this case, the clause represented by C_1 and S_1 has requested a commit first. Therefore, operation on clause C_2 was terminated by stifling the sponsor S_2.

1. For each subgoal:
   (a) Create a new procedure actor to handle this goal resolution.
   (b) **:passive-unify** the head of the new procedure to the corresponding arguments in the goal.
   (c) Start resolution of the procedure by the method outlined in figure 5.
2. Upon receiving **:committed** messages from all the guard procedures, signal **:request-commit** to the calling procedure.
3. Upon receiving a **:done** message from every subprocedure, signal **:done** to the calling procedure.
4. Upon receiving a **:fail** message from any of the subprocedures: If the **:request-commit** signal has already been sent, signal **:fail** to the calling procedure, else stifle the sponsor associated with this clause.

Figure 7: Pseudocode for a clause actor during goal resolution.

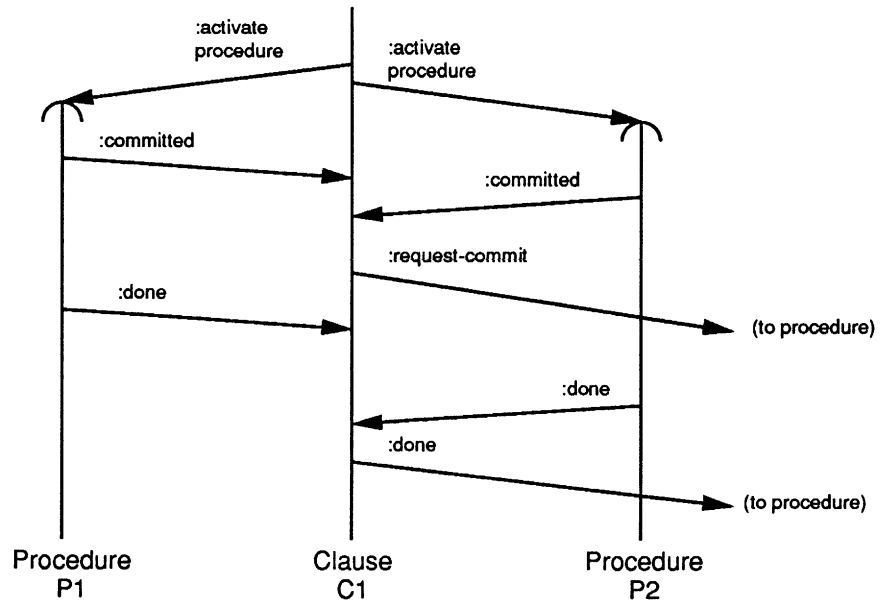

Figure 8: An example of an executing clause activation. Notice that the clause received **:committed**s from both its component procedures before requesting a commit. The same is true for **:done** messages.

success or failure of the **AND** of its instantiated goals to a terminal output stream.

Notice that during the recursive resolution of a procedure or clause actor, several actors of the other type must be created. In order to facilitate the creation of these actors we introduce clause and procedure template actors.

Procedure templates are created at parse time from syntactic information about each of the clauses making up that procedure. This information is in the form of clause templates. The clause templates, in turn, keep track of which goals are necessary to the success of the goal. The pseudocode for procedure and clause actor creation is shown in Figures 9 and 10 respectively.

We employ sponsors to handle the computational pruning mentioned above. Each time a new clause is used, we create a new sponsor to manage its computational resources. Because this sponsor creation occurs during resolution, the sponsor's parent will be the procedure's sponsor, and thus our sponsors are arranged hierarchically. Moving up the tree reveals that a hierarchy of sponsors are actually created during the resolution of parent clauses. Looking again at Figure 4, we see that nodes $fact(2,Z)_4$ and $fact(2,F)_5$ both create sponsors. The parent sponsor of $fact(1,Z)_4$ and $fact(1,Z)_5$ is the one created by $fact(2,Z)_5$. This is crucial because if we wish to stifle $fact(2,Z)_5$ we must be sure all sub-processing will also be stifled. This is guaranteed by sponsor semantics.

1. Create a procedure actor (with no subclauses).
2. For each clause template:
   (a) Create a new clause actor by the method outlined in figure 10.
   (b) Add the new clause actor to the new procedure.

Figure 9: Pseudocode for the creation of a new procedure activation from a procedure template.

1. Create a clause actor (with no subgoals), along with the local and suspended variables needed for the execution.
2. For each goal:
   (a) Lookup the correct procedure template for the resolution of this goal.
   (b) Use it to create a new procedure actor by the method in figure 9.
   (c) Add the new procedure actor to the new clause.

Figure 10: Pseudocode for the creation of a new clause activation from a clause template.

4.3 Example

Now that the different aspects of our interpreter have been presented, we will work through a detailed example designed to illustrate how they interact to interpret a GHC program.

In working through these examples, Figures 11 and 12 should be helpful. Figure 11 contains an actor event diagram which shows the execution of the unification structure for this example. Figure 12 is another event diagram showing the control structure. In order to keep the diagrams readable, they diverge slightly from the actual execution of the interpreter. The *true* goals and the guard variable Z_{guard} have been omitted.[13] The body variable Z_{body} is abbreviated by Z_b in the figures. In addition, only the message sends relevant to GHC have been mapped.

We will revisit Ueda's example which we first saw in Section 2.3, stepping through the pseudocode for our clause and procedure actor behaviors as we simulate the resolution of the top level goal. Since this is a relatively simple example, we will assume a worst-case arrival ordering on unification messages in order to display how our implementation avoids violating Ueda's rules of suspension.

$$
\begin{array}{lll}
 & :- & p(X), q(X). \\
p(Y) & :- & Y = ok \mid true. \\
q(Z) & :- & true \mid Z = ok.
\end{array}
$$

[13] Note that these can be eliminated by static evaluation.

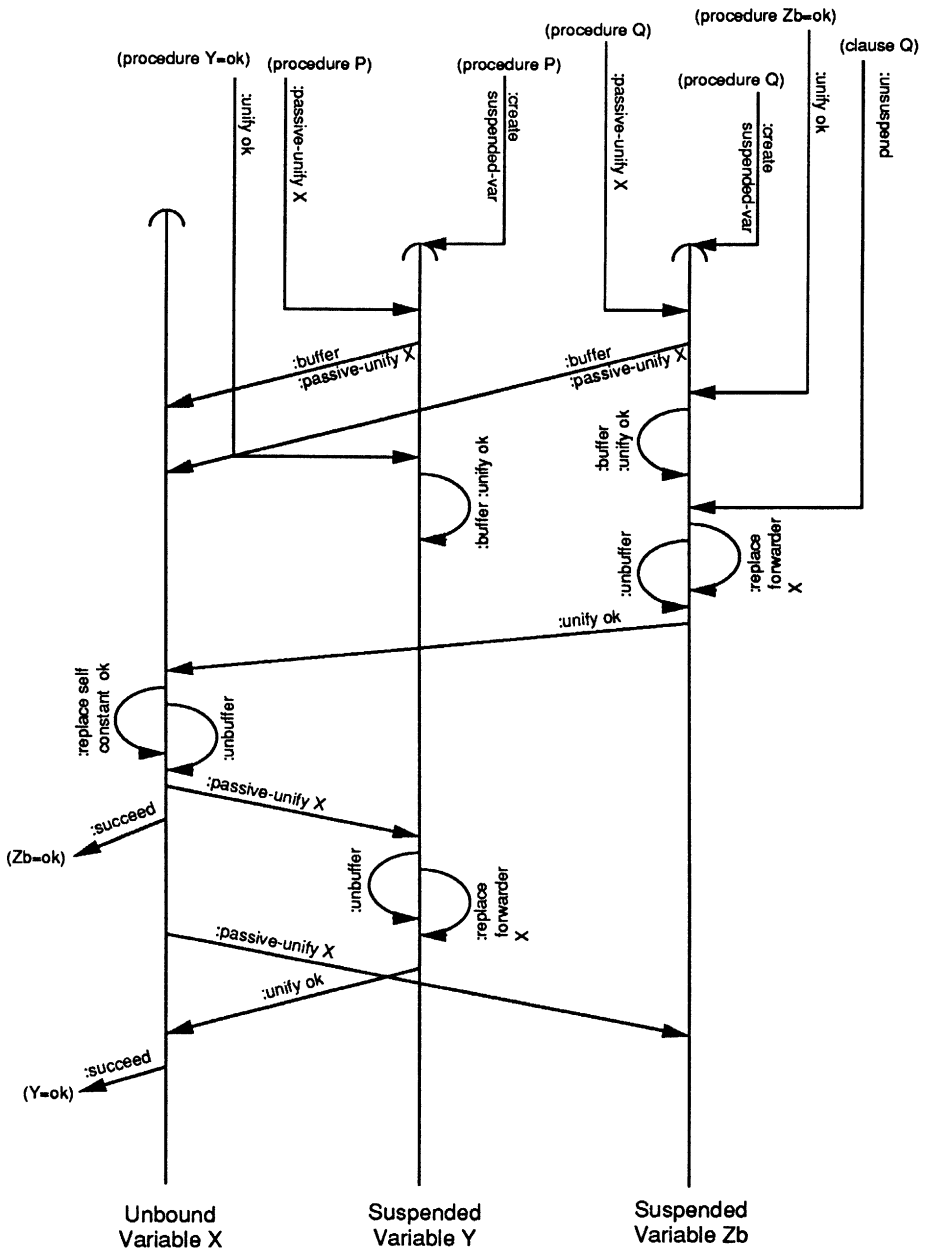

Figure 11: The actor event diagram for the unification structure. Actors whose lifelines appear only in Figure 12 will be referred to by a label of the form (*actor name*).

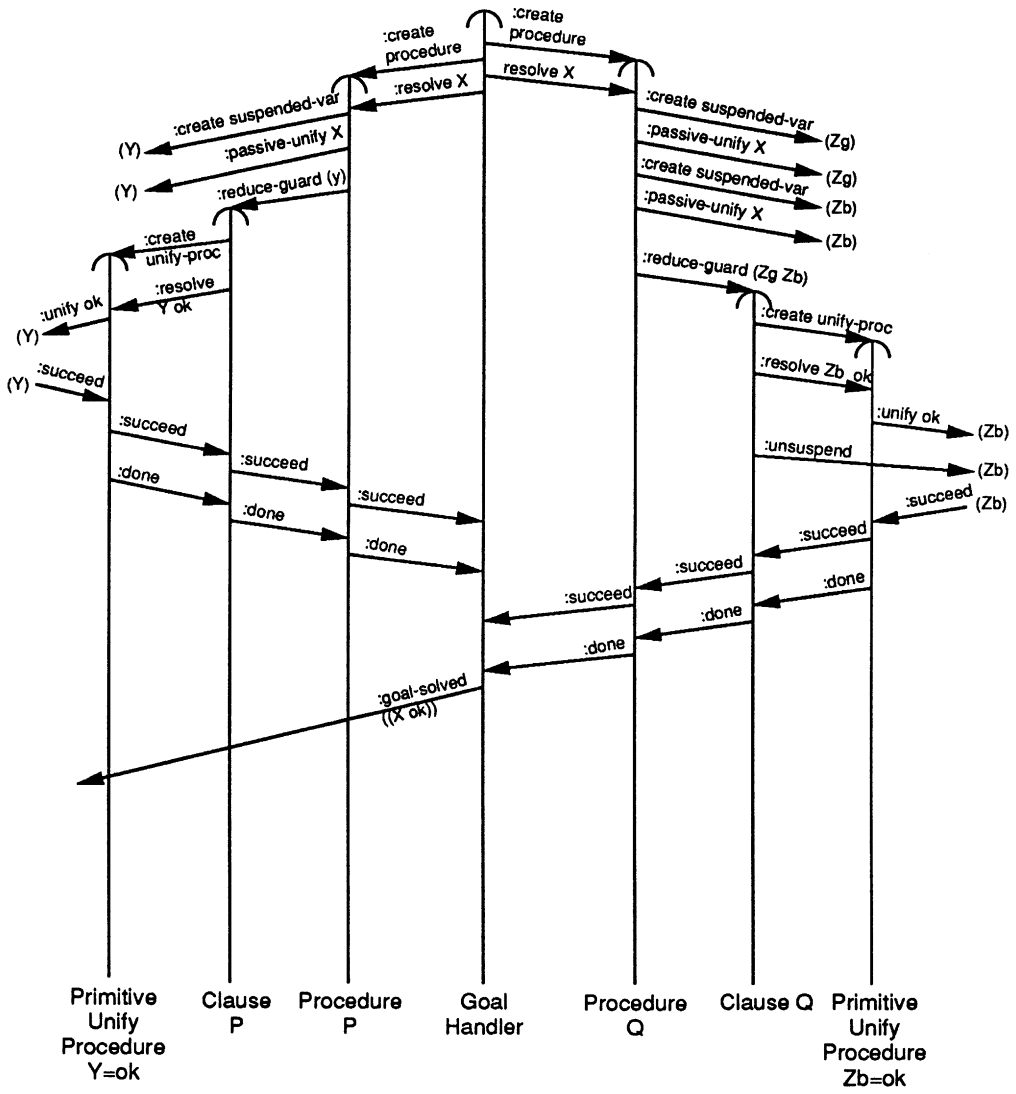

Figure 12: The actor event diagram for the control structure. Actors whose lifelines appear only in Figure 11 will be referred to by a label of the form (*actor name*).

We begin by creating procedure actors for the goals $p(X)$ and $q(X)$, per Figure 9. These goals are executed concurrently, but let us look at $p(X)$ first. We activate the clause for $p(Y)$ and create a suspended variable Y. We *:passive-unify* it to X. Since X is uninstantiated, it buffers the *:passive-unify* message. Now the guard goal, $Y = ok$ is executed, and the message *(:unify ok)* is sent to Y. Since Y is a suspended variable, it buffers the message, waiting to be instantiated. Thus the guard goal $Y = ok$, which would export a binding for X, is effectively suspended. This illustrates our implementation of the first rule of suspension.

We now turn to the resolution of $q(X)$, assuming for our worst-case purposes that it has not yet begun. First we activate the clause for $q(Z)$ and create a suspended variable Z_{guard}, and *:passive-unify* it to X as above. The guard and the body execute in parallel, and thus we must be careful that unification of Z in the body does not export a binding to the Z in the guard. To accomplish this, we create a new suspended variable Z_{body} and :passive-unify it to X as above. The *:passive-unify* for both Z_{guard} and Z_{body} are buffered by X.

At this time we execute the guard and body concurrently. The guard succeeds on the trivial *true* and sends a *:request-commit* to the clause actor. The clause commits, and Z_{body} is unsuspended, causing it to unify with X. At this point, the body goal unifies Z_{body} to *ok*. Since Z_{body} is unsuspended, this results in true unification (otherwise it would be buffered until the unsuspension occurred), causing both Z_{body} and X to unify to *ok*. When X is instantiated, it unbuffers its stored messages, causing Z_{guard} to unify with *ok*.

The instantiation of X also unbuffers the message waiting to *:passive-unify* Y to X. Since X is now a constant, Y is bound directly to that constant, and the unification of Y to *ok* succeeds since X is already bound to *ok*. The clause now commits and the body–the trivial *true*–succeeds as well.

Thus the top-level goal succeeds and returns a success message indicating the new binding for X. Our worst-case examination of the resolution of the top-level goal is complete.

5 Future Work

Because of time constraints, we were forced to leave several ideas related to the efficiency of our interpreter unimplemented. This section discusses three of those ideas and describes possible approaches to their implementation.

5.1 Elimination of Suspended Variables

The rules of suspension cannot be implemented without some form of suspended variable. Unfortunately, one of our most costly operations is the *passive-unification* of a suspended variable. An obvious optimization of our interpreter, therefore, would be to eliminate as many suspended variables as possible.

Our current implementation of lists is as nested terms.[14] Further, since our unification algorithm performs a recursive unification of suspended variables to terms,[15] *passive-unifying* a suspended variable to a list can become extremely costly. Lists would clearly be better implemented as some flat structure.

A second way to eliminate suspended variables is to keep track of which predicates export bindings. As an example, notice that

$$X > Y \tag{7}$$

will bind neither X nor Y, while

$$X = Y \tag{8}$$

could bind both X and Y, and

$$X := Y \tag{9}$$

could bind only X.

Now examine the following factorial clause:

$$fact(X,Y) :- X =:= 0 \mid Y = 1. \tag{10}$$

Notice that there is no way X will be bound in this clause. In this case we can optimize our unification: we can avoid creating a new suspended variable to avoid exporting a binding through X. Unfortunately, with an open world assumption, we cannot generalize this technique easily. For example, it would be highly desirable to declare an entire procedure non-exporting, and since we can declare individual clause guards non-exporting, this would seem a logical extension. However, to do this we must have examined all of its clauses, and since clauses may be added dynamically in an open-world model, this is impossible.

Since GHC is not generally implemented using an open world database, it may be reasonable to abandon the open world property in favor of improved efficiency. This would of course depend on the target application.

[14] In fact, our notation for [*a b c*] is $cons(a, cons(b, cons(c, null)))$.

[15] See Section 4.1.4 for more details. A brief example will make the inefficiency painfully obvious: We wish to unify suspended variable S_1 to a simple list $cons(a, cons(b, null))$. This involves replacing S_1 with a new predicate $cons(S_2, S_3)$, *passive-unifying* S_2 to a, and then unifying S_3 with the predicate $cons(b, null)$. To unify S_3 we must repeat the process just outlined. In general, we must recurse at every level of nesting.

5.2 Speculative Concurrency

GHC is an inherently parallel programming language, and as such, allows a fully concurrent implementation to evaluate the bodies of clauses even before they have been selected for commitment. However, only one clause body will ultimately be selected for commitment from among the clauses of a procedure. A satisfactory trade-off between the resources wasted on clause bodies never selected for commitment and productive concurrency must be made.

Deciding when to evaluate the bodies of clauses before they have been selected for commitment is a complex question. It is certainly desirable to get a head-start on the evaluation of the clause body to which we will ultimately commit. However, we must weigh the resources wasted on "garbage" computation against the benefits of greater concurrency along the critical path. While there is no clear universal answer at this time, specific situations may be more easily analyzed.

The branching factor in the computational tree is directly related to the number of clauses in a procedure. Therefore the proportion of wasted resources in a procedure with few clauses could be low. On the other hand, procedures with many clauses may waste a great deal of resources, since many branches could be explored before committing to a single branch.[16]

We must consider, however, the computational resources at hand. In some cases, it may be to our advantage to run the bodies of procedures with many clauses concurrently simply because we have many idle processors. A more sophisticated strategy may dynamically choose to allow speculative concurrency when a sufficient number of processors are available. This approach would be similar to the "throttling" of loop unfoldings used to control parallelism in dataflow architectures [Arv88]. Another strategy may adapt resource usage to specific procedures based on a strategy similar to trace scheduling [Ell85]. Under such a strategy, the amount of speculative concurrency invoked would be adapted dynamically: for example, a procedure with a base case clause and several recursive case clauses may commit to the recursive case clauses far more frequently than the base case. A clever dynamic scheduler could choose to race the recursive case clauses concurrently while avoiding premature execution of the base case.

5.3 Decentralization

Our control structure seems to be very decentralized. Clauses only communicate with procedures one level above and one below in the and-or tree, and their inter-procedure communication is also localized. The unification structure, however, needs some improvement.

We currently implement unification through forwarding. If several goals use the same variable during a computation, messages to this variable are sent to the same object. This is obviously a significant problem with respect to centralization. It is probably necessary,

[16] Further, the Actor mail system abstraction does not guarantee that the message to commit to a particular clause will be received soon after it is sent. Thus many useless branches of the computation tree might be explored to an arbitrary depth before one is selected for commitment. This is true even in the case where the branch sponsor is stifled, since the arrival of the stifle message may be delayed for an indeterminate length of time. See B.2 for more on this subject.

however, as long as the object is uninstantiated in order to keep the unification structure manageable. However, two instantiations of the same value are functionally equivalent. Therefore, after an unbound variable is instantiated as a term we could replicate its value rather than forward to it. In addition to the greater decentralization resulting from this scheme, we would also save one message send for each operation performed on the object by skipping the forwarder "middle man" and sending directly to the replicated object.

6 Conclusion

We have developed a working interpreter for full GHC using the Actor language Acore. Our interpreter is more expressive than previous implementations, avoiding the closed-world assumption and the restrictions disallowing user-defined predicates in guard clauses imposed by FGHC.

We also believe our implementation provides a unique view of GHC. Its object-oriented design provides a mapping of GHC concepts onto an object-oriented model, thereby allowing the examination of GHC constructs in an object-oriented framework.

In the course of our work, we have discovered several areas in which Acore could be improved. We have suggested specific improvements in Acore's exception handling, sponsorship constructs, locking mechanisms, and data abstraction facilities.

Nevertheless, Acore provided a powerful set of mechanisms well-suited to the rapid prototyping of a straightforward implementation of GHC. In particular, Acore's implicit concurrency and its notions of sponsorship and forwarding enabled us to create a simple mapping of GHC constructs onto actor behaviors. The implicit concurrency of the actor model made a highly concurrent implementation possible. Actor notions of message forwarding and behavior replacement were important in the development of our unification scheme. In addition, Acore's sponsorship mechanism for managing computational resources allowed us to perform effective pruning of active computations using a distributed control structure.

7 Acknowledgments

We would like to thank Carl Manning, Peter Huang, and Joseph "Psi" Mankoski of the Message Passing Semantics Group for their support and assistance with the Acore environment. Our thanks also to Jacob Levy, who provided invaluable guidance and patient explanations. Ken Kahn provided numerous thoughtful comments on an earlier draft of this paper. And, of course, our heartfelt thanks to Carl Hewitt, for luring us into this endeavor.

References

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

[Arv88] Arvind and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture", MIT LCS Computation Structures Group Memo #271, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Mass., June 20, 1988.

[Bri87] J. P. Briot and Akinori Yonezawa. "Inheritance and Synchronization in Concurrent OOP", *ECOOP '87 Proceedings*, Paris, France, June 15–17, 1987.

[Gol83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company, 1983.

[Hal85] R. H. Halstead. "MultiLisp: A Language For Concurrent Symbolic Computation", *ACM Transactions on Programming Languages and Systems, vol. 7, no. 4*, October 1985.

[Hew85] C. E. Hewitt. "The Challenge of Open Systems", *Byte*, vol. 10. pages 223–242, April, 1985.

[Kah87] K. Kahn. "Vulcan: Logical Concurrent Objects", in B. Shriver and P. Wegner (eds.), *Research Directions in Object Oriented Programming* and in *Concurrent Prolog: Collected Papers*, E. Shapiro (ed.), MIT Press, Cambridge, Mass., 1987.

[Lam80] B. Lampson and D. Redell. "Experiences with Processes and Monitors in Mesa", *Communications of the ACM, vol. 23, no. 2*, February 1980.

[Lev88] Jacob Levy. *Personal communications*, 1988.

[Lis83] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems, vol. 5, no. 3*, July 1983.

[Lis86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*, MIT Press, Cambridge, Mass., 1986.

[Man87] C. Manning. *Acore: An Actor Core Language, Reference Manual*, MIT AI Laboratory MPSG Apiary Design Note #7, MIT Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, Mass., August 10, 1987.

[Man90] C.R. Manning. "Introduction to Programming Actors in Acore" in *Towards Open Information Systems Science*, Hewitt, Agha, Manning, and Inman (eds.), MIT Press, Cambridge, Mass., 1990 (forthcoming).

[Ell85] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*, Ph.D. thesis, Yale University, February 1985.

[Nik87] R. S. Nikhil. "Id Nouveau Reference Manual", Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Mass., April 1987.

[Rei87] T. Reinhardt. *The Common Library: An Acore Development Facility, Reference Manual*, MIT AI Laboratory MPSG Apiary Design Note #8, MIT Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, Mass., March 31, 1987.

[Sha87] E. Y. Shapiro. "A Subset of Concurrent Prolog and its Interpreter" in *Concurrent Prolog: Collected Papers*, E. Shapiro (ed.), MIT Press, Cambridge, Mass., 1987.

[Ued86] K. Ueda. *Guarded Horn Clauses*, Ph.D. Thesis, University of Tokyo, March 1986.

[Ued87] K. Ueda. "Guarded Horn Clauses" in *Concurrent Prolog: Collected Papers*, E. Shapiro (ed.), MIT Press, Cambridge, Mass., 1987.

[Yon87] A. Yonezawa., et al. "Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1" in *Object Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro (ed.), MIT Press, 1987, pp 55-89

A Actor Behaviors Used To Implement GHC

This section provides a "reference-manual" to the types of actors we used in our implementation of GHC.

A.1 Templates and Activations

One facet of a truly concurrent implementation of GHC is that at any time there may be multiple instantiations of individual GHC objects, e.g., a recursive call to *Fact* will activate several distinct copies of the clauses defined for procedure *Fact*. For this reason we make a distinction between *templates*, which are objects created at parse-time, and *activations*, which are instances of those *templates* created at run-time. When a *template* receives an **:activate** message, it returns a new object with the same structure as itself but with newly `GENSYM`ed variables. This prevents interactions between multiple instantiations of a single *template*.

Another reason for making this distinction is that in an open system, clauses may be added to procedures at any time.

A.2 Common Handlers

Variable, constant, predicate, and function activations accept the following messages:

- (**:unify** *GHC-actor*) Returns 'succeed if unification is successful, or 'fail otherwise.
- (**:unify-to** *GHC-actor*) Used by the unbound variable handler to check if the unification will result in a forwarding loop, as explained in Section 4.1.2. The receiver of this message will know[17] that an unbound variable is trying to unify to it. It must then take steps to accomplish this unification. This is done through **:safe-unify** below.
- (**:safe-unify** *GHC-actor*) Unifies self and GHC-actor. Assumes that the resulting forwarding link cannot create a forwarding cycle.[18]
- (**:passive-unify** *GHC-actor*) Suspended variables perform special "passive" unification as explained in Section 4.1.4. All GHC actors except suspended variables may ignore **:passive-unify** messages because:

  (1) The initial generation of **:passive-unify** messages occurs during clause head-resolution, and

  (2) The only other way **:passive-unify** can be sent is by an **:unbuffer** operation on a variable. In this case, we are guaranteed that this variable is the same as the *waiting-on* acquaintance of the generating suspended variable. Therefore, if the

[17] However, this is not known with certainty. See Section 4.1.2 on why this is not absolutely known.

[18] An aside on the length of forwarding chains: It is interesting to note that the GHC actor received as an argument was guaranteed not to be a forwarder at the time the message was sent. This is because the message was sent by the actor itself, not a forwarder that leads to it. Therefore, assuming some measure of fairness (and some fairness is actually guaranteed by the rules of suspension), the forwarding chains that are created will not be extremely long.

receiver of a **:passive-unify** message is not a suspended variable, we know that it used to be one and that it has since been unified with the target of the buffered **:passive-unify** message.

All activation-type actors accept the following messages:

- **(:print-value)** Returns a symbolic representation of **self.**
- **(:buffer** *message*) Variables buffer *message*, pending later unbuffering. Predicates, functions, and constants don't actually buffer *message*, but immediately cause it to be sent.
- **(:buffer-until-unsuspended** *message*) Suspended variables buffer *message*, pending later unbuffering. Variables, predicates, functions, and constants don't actually buffer *message*, but immediately cause it to be sent.

A.3 Constants

Constant actors have a single acquaintance, *value*. They accept only the **:value** selector message.

A.4 Predicates and Functions

Predicate actors exist in two forms: *templates* and *activations.*

Predicate *templates* have no interesting handlers. They have the following acquaintances:

- *procedure-template* If this is a user-defined predicate, this is the name of the procedure template which defines the predicate. If this is a built-in predicate, this is the actual primitive actor which defines the predicate.
- *argument-templates* A list of templates for the arguments to the predicate.

Predicate *activations* have the following acquaintances and handlers:

- *procedure-template* As explained above.
- *argument-activations* The activations of the argument templates.
- **(:activate-suspended** *target*) This message creates a new predicate activation whose arguments are all suspended variables. This is used only when a suspended variable is passive-unifying to a predicate. See Section 4.1.3.
- **(:unsuspend)** This message unsuspends any suspended variables the predicate may have as arguments (possibly as a result of an **:activate-suspended** message.)

Function actors also may be *templates* or *activations.* However, unlike predicates, they may not be user-defined. In all other respects they conform to the above specification for predicate actors.

A.5 Procedures

Procedures exist in four forms: *template*, *executing*, *committed*, and *terminated*. The last three forms trace the progression of a procedure from its initial activation as an *executing-procedure*, to a *committed-procedure* as it commits to one of its clauses, and finally to a *terminated-procedure* when all of its subcomputations are complete.

Template-procedures have the following acquaintances and handlers:

- *clause-templates* The templates for the clauses in the procedure.
- *activations* A list of current activations of this template so that executing procedures may be informed of newly added clauses. This is necessary due to the open nature of our implementation.
- (**:resolve** *term*) Creates an executing procedure and begins reducing the guards for each clause. See Figure 4.
- (**:add-clause** *clause*) Adds a new clause template for *clause* to the set of clauses. Informs any activations of the procedure of the new clause.
- (**:remove-activation** *activation*) Removes the terminated activation *activation* from the list of activations kept by the procedure.

Executing-procedures are those procedures currently resolving themselves against a given predicate. They have the following acquaintances and handlers:

- *procedure-template* A handle on the parent template.
- *predicate* The predicate that we are resolving against.
- *sponsors* A list of sponsors which are funding clause reductions. Used when we commit to a particular clause and wish to stifle all the others.
- (**:add-clause** *clause*) Adds a new clause. Invoked by parent template when new clauses are added to the template definition.
- (**:request-commit**) Invoked by a subclause which has satisfied its guard. Stifles all other clauses, and then replaces **self** with a *committed-procedure*.

Committed-procedures are those procedures which have committed to one of their subclauses and are now monitoring the state of the computation. They accept the following messages:

- (**:fail**) Invoked by a failing clause. Propagated back up the **AND-OR** tree.
- (**:done**) Invoked by a clause whose sub-computations are all complete. After propagating this message back up the **AND-OR** tree, replaces **self** with a *terminated-procedure*

Terminated-procedures are those procedures which were previously committed and whose subcomputations have all terminated. Terminated procedures ignore all messages they receive. These procedures exist only as a defensive programming measure, for two reasons:

- Stifled processes don't die immediately, and therefore clauses may still send messages even after they are stifled.
- The mail system may arbitrarily delay messages, and therefore a message may arrive long after it was sent.

A.6 Clauses

Clauses, like procedures, exist in *template*, *executing*, *committed*, and *terminated* forms. The semantics of each form are the same as those of the corresponding procedure forms.

Clause *templates* have the following acquaintances and handlers:

- *variable-templates* A list of variable templates for the variables in this clause. They are `GENSYM`ed on clause activation.
- *guard-templates* A list of the guard goals.
- *body-templates* A list of the body goals.
- **(:reduce-guard** *predicate*) Creates an activation of this clause. See Figure 10.

Executing-clauses have the following acquaintances and handlers:

- *variables* A list of variable activations.
- *commit-count* The number of guard goals yet to commit before the clause may commit. Initially equal to the number of guard goals.
- *done-count* The number of guard and body goals yet to signal **:done** before the clause may consider its computation complete and send a **:done** message to its procedure. Initially equal to the total number of guard and body goals.
- **(:committed)** Sent by a guard goal which has committed. Decrements *commit-count*. If *commit-count* reaches zero, the clause sends a **:commit-request** to its procedure, and if it is acknowledged, replaces **self** with a *committed-clause*.
- **(:done)** Sent by a subgoal (guard or body) all of whose computations are complete. Decrements the done-count. If *done-count* reaches zero, the clause propagates the **:done** message up the **AND-OR** tree and replaces **self** with a *terminated-clause*.
- **(:fail)** Sent by a subgoal which has failed. If sent by a guard-goal, the clause stifles itself.

Committed-clauses have the following acquaintances and handlers:

- *done-count* As above.
- **(:done)** Decrements *done-count*. If it reaches zero, propagates the **:done** message up the **AND-OR** tree and replaces **self** with a *terminated-clause*.
- **(:fail)** Propagates the **:fail** message up the **AND-OR** tree.

Terminated-clauses have no interesting acquaintances or handlers. They ignore all messages and serve the same function as *terminated-procedures*, above.

A.7 Variables

There are two types of variables: *unbound-variables* and *suspended-variables*. A suspended-var is tentatively bound to its *waiting-on* acquaintance, buffering all messages except **:passive-unify** and **:unsuspend**. It will unsuspend when either of the following occurs:

- (1) Our clause commits and explicitly send an **:unsuspend** message, or
- (2) *waiting-on* is either partially or fully instantiated, causing us to retry the unification.

Unbound-variables are variables which have not yet been the target of any unification. Their acquaintances and handlers are:

- *name-symbol* The symbolic name of the variable, in a form suitable for terminal output.
- *messages* A list of buffered messages.

Suspended-variables are those variables which, if unified, might export bindings in violation of the rules of suspension. They have the following acquaintances and handlers:

- *name-symbol* As above.
- *waiting-on* The actor that the *suspended-variable* is passively bound to, and therefore upon whose instantiation the *suspended-variable* is waiting.
- *messages* As above.

B Difficulties with Acore

The design, development, and debugging of the GHC interpreter took place using the the Actor language Acore as implemented at the MIT Artificial Intelligence Laboratory. The GHC interpreter is one of the largest, most complicated programs written in Acore to date. As a result of our extensive experience with Acore and its environment, we have reached several conclusions regarding the present implementation and possible improvements. Since Acore is an evolving, experimental language, our suggestions may be of immediate practical use.

B.1 Exception Handling

Acore signals exceptions by sending *complaints*, as described in [Man87, p.204]:

> ... the handling of the complaint is specified by the closest lexically enclosing *let-except* which handles the complaint generated. It is an error for a complaint to be generated in a context with no exception handler to handle it, so most contexts should provide at least a default *otherwise* exception handler. For example, the default request handlers with expression bodies provide default exception handling; the expression context it provides is one place where a default (forward the complaint to the customer) makes sense.

Note, (1) that complaints are dynamically propagated up the chain of customers which begins at the customer for the current request, and (2) that "it is an error" to fail to provide a handler. However, **is-request** handlers and procedures defined with **defprocedure** do *not* propagate complaints up the customer chain. Unfortunately, it is extremely inconvenient to manually wrap a **let-except** form around every usage of procedures and is-requests. Even worse, many of the Acore CommonLib [Rei87] routines define or use these forms. For example, Acore **apply** is defined using a **defequate-procedure**, and complaints generated during **apply** result in disaster. We actually experienced this problem in the course of developing our interpreter. Our "fix" is more of a kludge than a solution: we avoid misbehaving CommonLib routines and define a macro **complaint-request** as a syntactic sugar for wrapping **let-except**s around every **is-request** form. The proper solution to this, aside from modifying the Acore language, is to rewrite parts of the CommonLib to avoid forms which are dangerous with respect to complaint-handling. Another option is to change the definition of **defprocedure** and **is-request** to propagate complaints up the customer chain. Finally, all lexical blocks could be required, as contexts, to propagate complaints (and assume the behavior of **with-exceptions**, below).

There is another problem with sponsors which is more serious. The lifetime of a **let-except** handler ends when values are returned for the new let-bindings. However, it is quite possible that a complaint could be generated *after* the binding values are returned and the **let-except** handler has disappeared! In this case there will be no handler for the complaint, and disaster ensues. This is a significant problem since an Acore expression may "return" before it is fully executed—the use of **future**s in **let-except** bindings is a good example of such an expression.

We propose one possible solution to all of the above problems concerning exception-handling using Acore complaints. It seems clear that an exception-handler should persist at least as long as the expressions it contains. Starting with this assumption, and noting also that raising and catching exceptions may be useful outside the context of a **let-except** form, we suggest the introduction of a new form, **with-exceptions**, with the following syntax:

(with-exceptions
    (expression-body)
    (except-when {*((:keyword (valA valB ...)) commands ...)*}*))

The semantics of this form are that *all* unhandled complaints generated during the execution of *expression-body* (including any generated by any computation which it may spawn) will be examined and possibly handled by the the *commands* specified in the appropriate **except-when** arm. If the *:keyword* in some **except-when** arm matches the keyword name of the generated complaint, then that complaint's arguments (if any) are bound to *(valA valB ...)* and *commands ...* are executed. Note that it is possible for multiple complaints to be raised and handled during the execution of *expression-body*. Nested **with-exception** handlers form nested enclosing scopes through which complaints are propagated. It would be fairly straightforward to implement **with-exception** handlers using the following scheme:

Associate with every request message an *exception-handler* actor which is passed as an argument in the same way that customer, sponsor, and reply-keyword are currently passed in requests. This exception-handler actor should contain the code for the **except-when** handlers and a pointer to its enclosing exception-handler. Exception-handler actors should persist until the computations which they enclose have completely terminated, at which point they may be garbage-collected.

In fact, it may prove useful to encapsulate all of the information associated with a message send into a single actor which obeys an abstract protocol. For example, an actor with acquaintances *(sponsor customer reply-keyword exception-handler)* could be used in place of separate actors in **:request** messages. Such an actor would respond to selectors such as **:sponsor** and **:reply-keyword**, etc. The advantage of this approach is that the information associated with a message send could more easily be changed by Acore language developers and sophisticated users. As long as the actor used could respond to the abstract protocol for simple message sends, additional messages could be recognized which allow for more complex behavior. For example, our GHC interpreter could have defined the useful exception-handling mechanism described above using such a scheme.

B.2 Resource Encapsulation with Sponsors

We have encountered several problems with sponsors as implemented in the current release of Acore and the CommonLib. Our implementation relies heavily on the ability to stifle sponsors in order to abort computations. Unfortunately, complaints are used as the mechanism through which computations are aborted. Since there are currently several problems with Acore complaint handling (as detailed above), sponsors do *not* work correctly

in the current release. Even after the language implementers helped fix **simple-sponsor** in the CommonLib, incorrect results were still observed. Since correctly functioning sponsors are crucial to our GHC implementation, this impeded our progress significantly. Our short-term solution to this problem was to define our own types of sponsors called **hierarchical-sponsor**s which may achieve a lower concurrency than **simple-sponsor**s, but are at least correct. Our implementation can trivially be adapted to use **simple-sponsor**s once the CommonLib implementation has been fully debugged.

Other more abstract, implementation-independent issues should also be addressed. For example, it should be noted that a computation is not immediately terminated as soon as its sponsor receives a **:stifle** message. This is because the computation can proceed until its current supply of ticks (granted from a previous request) runs out. Only then will it request more ticks from its sponsor and receive a **:sponsorship-denied** complaint. This is inefficient, and could become a serious efficiency problem if a large tick quantum is granted when more ticks are requested. A simple solution would be to define this quantum to consist of only a single tick, but this is unacceptable since the overhead of the sponsorship scheme would dominate the computation and cripple the progress of "real" work. Perhaps a better solution would be for each sponsor to keep a list of its subsponsors (currently only a link to one's parent sponsor is maintained), which it could *recursively* stifle upon reception of a **:stifle** message.

In any case, however, there is always the possibility that residual "junk" messages may be floating around in the mail system even after a **:stifle** message has been sent. This is due to the nondeterministic arrival ordering in the Actor model of computation. Since messages may be delayed an arbitrary length of time before being delivered, a computation may generate an arbitrary number of messages between the time that a **:stifle** message is sent and received. This forces Acore programmers to adopt a defensive programming style which must *explicitly* handle unwanted messages from dying actors. Unfortunately, this seems unavoidable since it is a consequence of the underlying Actor model and not a modifiable part of the Acore language definition.

B.3 Locking and Serialization

Another concrete problem which we encountered in our implementation is the granularity of actor locking in Acore. Acore handlers can be either **serialized** or **unserialized**. An **unserialized** handler cannot change its actor's state and therefore does not need to lock its actor. This allows the actor to immediately begin concurrent processing of another incoming message. On the other hand, a **serialized** handler locks its actor, buffering any incoming messages until the handler completes its state change via the execution of a **ready** or **:replace**, which update its actor's acquaintances and behavior script.

We feel that this binary partitioning of handlers into **serialized** and **unserialized** is too coarse. Some specific problems with this approach are:

- It precludes orthogonal state-changing operations from running concurrently. This can be viewed as both a semantic burden for the programmer and an efficiency issue. For example, suppose an actor has two acquaintances, A and B. During the execution of a serialized handler **set-A** which changes A, it would be perfectly correct

to concurrently execute an unserialized handler **get-B** which is a selector defined to examine the value of B. The Acore compiler could find most if not all of these cases by performing a compile-time analysis of the acquaintances actually changed by a handler. Of course, there should also be some manual override mechanism whereby a programmer could explicitly specify that certain handlers should not be run concurrently even if it is possible to do so—perhaps one really *shouldn't* look at B while changing A in a particular application. For example the *Interact* metacircular interpreter for Acore was forced to resort to using the low-level **system-request** primitive which explicitly ignores locking. We consider this to be a rather serious problem which should be addressed by the Acore language implementors.

- An Acore programmer can't write a recursive serialized handler, or even a serialized handler which invokes other handlers defined in the same behavior. Therefore, a **serialized** handler which wants to achieve this functionality must do everything inline itself or rely upon external functions defined with **deffunction**. This is *very* annoying.

A good discussion of problems related to synchronization and serialization in concurrent object-oriented languages is presented in [Bri87]. A variety of attempts to overcome these problems include the *waiting-mode* mechanism of ABCL/1 [Yon87], and the explicit manipulation of an actor's message queue in Vulcan [Kah87]. Another approach to locking which is common in Algol-like languages such as Mesa is to use *monitors* [Lam80] to achieve concurrency and avoid inconsistency.

A final locking problem occurs during unhandled complaints: If a **serialized** handler receives an unhandled complaint, it correctly propagates that complaint up the customer chain, but it *fails to unlock* the actor locked by the serialized handler.

B.4 Sharing Mechanisms

A sharing mechanism such as inheritance is an important feature found in almost all modern object-oriented languages. Such a mechanism is conspicuously absent in Acore. While it is true that there is nothing to prevent an Acore programmer from explicitly coding inheritance by delegation, there is nothing to facilitate such an endeavor. Although we recognize that supporting inheritance is not simple, we feel that the Acore language definition should explicitly support some sharing mechanism and provide convenient syntactic forms for it. A good overview of inheritance schemes for concurrent object-oriented languages can be found in [Bri87].

B.5 Data Abstraction

The actor model provides a very clean, clear encapsulation of abstract data types and the operations which these support. Nevertheless, it would be nice to allow Acore programmers to declare the scope of each handler as *opaque* (private, invisible, internal) or *visible* (public, exported, external). An *opaque* handler could only accept requests from **self** (and perhaps

its subclasses if inheritance is implemented), while a *visible* handler could accept requests from any actor. A similar form of abstraction is available in CLU [Lis86] and Argus [Lis83].

B.6 The Acore Environment

We have several brief comments regarding the current Acore environment at the MIT Artificial Intelligence Laboratory.

- The entire system is very slow. This is a consequence of multiple layers of interpretation (Acore to PRACT to Lisp) on serial machines (Symbolics 3600s).
- The CommonLib Acore Library [Rei87] is not very robust. In particular, we experienced problems with the definitions of **simple-sponsor** and **hash-table**. The problem with sponsors involved the correct propagation of complaints, and a failure to properly handle the case in which a sponsor is stifled in the middle of a request for more sponsor ticks. The problem with hash tables involved differences between Lisp and Acore **cons** cells/actors and their hashed values.
- Many useful primitive functions are not provided by the CommonLib, forcing Acore programmers to escape into the underlying Lisp environment via `#L` or `#I` forms. A good example of a non-existent Acore function that is extremely useful is an Acore analogue of the **si:equal-hash** (or **sxhash**) function provided by Lisp.
- The current documentation is incomplete and often incorrect. Examples of this include the fact that the actor equality primitive is **eq?** and not the documented `==` as well as the fact that the documented **select** macro is broken.
- A common problem while developing in Acore is not knowing which actors are "hung." Debugging would be much easier if Traveler provided a way to display actors that don't respond after a certain timeout.

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 4/19/95

Report # AIM-1166

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 41 (47-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP - (1) UN#'ED TITLE PAGE	
(2-41) PAGES #'ED 1-40)	
(42-44) SCANCONTROL, DOD (2)	
(45-47) TRGTS (3)	

Scanning Agent Signoff:

Date Received: 4/19/95 Date Scanned: 4/20/95 Date Returned: 4/27/95

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AIM 1106	2. GOVT ACCESSION NO. AD-A231409	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Experience with Acore: Implementing GHC with Actors		5. TYPE OF REPORT & PERIOD COVERED memorandum 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jeff Palmucci, Carl A. Waldspurger, David Duis, Paul Krause		8. CONTRACT OR GRANT NUMBER(s) N00014-88-K-0460 N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE October 1990 13. NUMBER OF PAGES 40
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

parallel programming	message-passing
object-oriented	actors
Concurrent logic programming	Guarded Horn Clauses (GHC)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Abstract: This paper presents a concurrent interpreter for the programming language Guarded Horn Clauses, abbreviated GHC. GHC is a general-purpose concurrent logic programming language. It has a clean, simple semantics based upon unification and choice nondeterminism.

(continued on back)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 continued:

Unlike typical implementations of GHC in logic programming languages, the interpreter is implemented in the Actor language Acore. The primary motivation for this work was to probe the strengths and weaknesses of Acore as a platform for developing sophisticated programs. We chose to implement a concurrent interpreter for GHC because this large, complex application provided a rich testbed for exploring Actor programming methodology.

The interpreter is a pedagogical investigation of the mapping of GHC constructs onto the Actor model. Because we opted for simplicity over efficiency, the interpreter is inefficient in both time and space.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 4/20/1995

M.I.T. Libraries
Document Services