Overlay GHC: an Extension of Guarded Horn Clauses for Overlay Programming

Kenji Saito∗
Keio University / National Institute of Information and Communications Technology, Japan

Abstract

Today’s high-speed network allows sophisticated applications of overlay networks. Meanwhile, usage of multicore processors has been spreading. The level of concurrency we need to handle has been rising rapidly, which necessitates a language that can express massive concurrency in a natural way, which can work with both tightly and loosely-coupled multiprocessor environments.

This paper describes still work-in-progress design of "Overlay GHC", an overlay network programming language based on concurrent logic language GHC (Guarded Horn Clauses)[10], as a candidate for such a language.

1 Introduction

1.1 Problem Statement

Today’s high-speed network has allowed sophisticated applications of the Internet such as P2P (peer-to-peer) systems. Designs of P2P systems are characterized by their usage of overlay networks such that participants can potentially take symmetrical roles. This implies higher level of concurrency, as those participants are autonomous agents.

Meanwhile, all major CPU manufacturers are moving to multi-core architectures, as uni-core processors have stopped performance scaling. Usage of multi-core processors has been spreading widely.

The level of concurrency we need to handle has been rising rapidly, which necessitates programming languages that can express massive concurrency in a natural way, which can work with both tightly and loosely-coupled multiprocessor environments.

1.2 Contributions of This Work

This paper describes still work-in-progress design of Overlay GHC, an overlay network programming language based on concurrent logic programming language GHC (Guarded Horn Clauses)[10], as a candidate for such a language. This language allows us to rapidly program an overlay network with a bird’s-eye view, and to code a whole logical network in one program.

2 Background

2.1 GHC: Guarded Horn Clauses

Preliminaries The basic elements of GHC programs are terms. A term is formed from function symbols and variables. Function symbols are symbols beginning with small letters, and variables are those beginning with capital letters or ‘.’. A variable alone is a term. Otherwise, a term is of the following form:

\( f(\gamma_1, \gamma_2, \ldots, \gamma_n) \) \( (n \geq 0) \)

where \( f \) is a function symbol, and \( \gamma \)'s are terms. For example, \( X, a, \text{cons}(a, X), \text{cons}(a, \text{cons}(b, X)) \) are terms.

An atom (atomic formula) is of the following form:

\( p(\gamma_1, \gamma_2, \ldots, \gamma_n) \) \( (n \geq 0) \)

where \( p \) is a predicate symbol, and \( \gamma \)'s are terms. Predicate symbols are symbols beginning with small letters. For example, \( \text{is list}(\text{cons}(a, X)) \) is an atom.

Syntax of GHC A GHC program is a set of guarded Horn clauses of the following form:

\( H \leftarrow G_1, \ldots, G_m | B_1, \ldots, B_n. \) \( (m, n \geq 0) \)

where \( H \), \( G \)'s, and \( B \)'s are atoms. \( H \) is called the head, and \( G \)'s and \( B \)'s are called guard goals and body goals, respectively. A goal may be a unification goal:

\( \gamma_1 = \gamma_2 \)

where \( \gamma_1 \) and \( \gamma_2 \) are terms. ‘\|’ is the commit operator. \( H \) and \( G \)'s together are called the guard, and \( B \)'s as a group are often called the body. If there are no goals in the guard or body, it is denoted by \text{true}.

A program is invoked by a goal clause:

\( \leftarrow B_1, \ldots, B_n. \) \( (n \geq 0) \)

where \( B \)'s are goals.
Declarative Semantics of GHC  A guarded Horn clause can be read as follows: “If every goal in its guard and body is true, its head is true.”

The result of every successful execution of a GHC program conforms the above semantics (soundness). There may be some clauses not applied in an execution, so that the result might not be the only solution (incompleteness). Incompleteness reflects the fact that there is indeterminacy in concurrent programs.

Operational Semantics of GHC Intuitively, each guarded Horn clause is considered as a rewrite rule of a goal, where its guard specifies the conditions to be satisfied for the rule to be applied, and its body specifies the actual goals to replace with. If more than one clause can be applied, one of them is selected nondeterministically. This irreversible act of applying a clause is called commitment.

Assignment to variables is called binding. Bindings are produced after commitments, and any attempts to bind the bound variables with incompatible terms fail. A binding to a non-variable term is called instantiation.

Two terms are unified when they become lexically identical by binding the variables in each with the corresponding terms of the other.

An informal operational semantics of GHC follows:

1. Goal execution: Every goal in the goal clause is executed concurrently by the following steps:
   
   (a) Head unification: Variables appearing in the head of a clause is analogous with formal parameters of procedural programming languages. Clauses whose heads are unifiable with the calling goal become the candidates for commitment; variables appearing in their heads are bound with the corresponding terms in the calling goal.
   
   (b) Suspension rule: Guard goals of the candidates are executed concurrently, with a restriction imposed by the suspension rule: any attempts to instantiate the calling goal are suspended.
   
   (c) Commitment: The execution of the calling goal commits to a clause whose guard succeeds; the body of the committed clause replaces the calling goal. Unification goals in the body may instantiate the calling goal.

2. Success: A unification goal succeeds if its arguments are unified. A non-unification goal succeeds if it is eventually replaced by unification goals that succeed, or by an empty body. A program succeeds if every goal in its goal clause succeeds.

3. Failure: A unification goal fails if its arguments are not unifiable. A non-unification goal fails if its execution has no candidates for commitment, or the guard of every candidate fails. A program fails if any goal in its goal clause fails.

Process Interpretation of GHC Programs A GHC program defines a concurrent program in the following way:

1. Recursively defined predicates define processes.

2. Conjunction of processes define a network of processes.

3. Arguments of goals define local states of processes.

4. Shared variables among goals define communication channels. These variables are often used to represent streams (sequences of data).

Figure 1 is a stack program written in GHC, based on an example from [7]. \texttt{io:printstream(Os)} is a process pro-

\begin{verbatim}
:- stack(CommandStream, Data, OutputStream).
stack([pop(Cs), [X][List], Os] :- true | stack(Cs, [X][List], Os).
stack([push(X)Cs], List, Os) :- true | stack(Cs, [X][List], Os).
stack([listCs], List, Os) :- true | Os = [List|Os], stack(Cs, List, Os1).
stack([], List, Os) :- true | Os = [].
:- stack([push(1), push(2), list, pop, list, pop, list], []),
    io:printstream(Os).
\end{verbatim}

* \texttt{:-} is the coding expression for ‘\texttt{Ã—}’.
* \texttt{[a\mid b]}, \texttt{[a, b]} and \texttt{[]} are syntactic sugars for ‘cons(a, b)’, ‘cons(a, cons(b, nil))’ and ‘nil’, respectively.

Figure 1. Example: stack program in GHC

vided in our Overlay GHC implementation that prints the elements of stream \texttt{Os}. The program produces the sequence ‘[2, 1]’, ‘2’, ‘[1]’, ‘1’, and ‘[]’.

2.2 KL1: Kernel Language One

KL1[11] is an extension of GHC, designed as the system description language for parallel inference machines. While GHC is concerned with concurrency only, KL1 is also concerned with parallelism, or how to execute concurrent programs, although the semantics of GHC is not broken. By removing pragmas (directives to the language system) from a KL1 program, the corresponding GHC program is obtained.
3 Design

3.1 Overlay GHC

Unlike our previous work[8], we take the same approach as KL1; how to execute GHC programs in the distributed real-time settings is described by pragmas and some built-in predicates/functions. GHC itself remains unchanged.

Introduction of Goal Placement Pragmas A remote goal-placement pragma @node_id(ID) has been introduced, where ID is expressed in the form of a URL. Usage of ‘xmpp://’ to specify a Jabber ID or ‘pgp://’ to specify a PGP public key user ID is expected for example (language systems are assumed to have locating mechanisms that map those identifiers onto actual communication targets).

Figure 2 is a symmetric group chat program that uses this pragma. A new user joins the chat by invitation basis.

In the program, terminal() is the main process representing a user. keyboard() is responsible for taking inputs, including ones from other users, and forward them to the next user in the circular overlay network. checkInput() is subordinate to keyboard(), and interprets commands. This is where new terminal() is spawned on a remote processor when the user input is of the form ‘join(Name, ID)’. display() is responsible for generating the console output stream. ‘\=" is a not-unifiable operator. ‘otherwise’ succeeds when the guards of all other candidate clauses fail.

Programming practices in Overlay GHC are characterized around this technique of remote goal placement, by which many interesting concepts can be realized.

Detection and Treatment of Failures In a distributed computation, a program may fail even if the code is logically sound, because of processor or communication failures. But the program in Figure 2 does not assume occurrences of failures. We have been investigating how failures can be detected and treated in natural ways in Overlay GHC. We feel that programmers need to be able to handle at least two models of failures: fail-stop and arbitrary failures.

In the fail-stop failure model, a failed processor stops computing without a side effect. In Overlay GHC, this will cause a perpetual suspension, without breaking the semantics of GHC as there is no concept of time in Horn clauses.

We have been investigating to introduce timeout(Time, Unit) as a built-in predicate to be used in guards, to detect the state of perpetual suspension caused by fail-stop, and recover from such a state. Figure 3 is an example of a program using timeout().

In the arbitrary failure model, a failed processor may bind variables with arbitrary terms. Redundancy needs to be programmed to detect such failures.

\begin{verbatim}
checkAlive(Stop, Os) :- timeout(30, sec) | Os = [ping(Pong)|Os1], checkPong(Pong, Stop), checkAlive(Stop, Os1).
checkAlive(stop, Os) :- true | process-failure-of-the-peer.
checkPong(ping(Pong, Stop) :- true | timeout(10, sec) | Stop = stop.
\end{verbatim}

* Sends out ping(Pong) every 30 seconds, and detects a failure if Pong is not instantiated for more than 10 seconds.

Figure 3. Ping program in Overlay GHC

Dynamic Many-to-One Communication A binding to a variable is one-time only, which makes it impossible for multiple writers to simultaneously write into a stream.

This difficulty has been compensated in concurrent logic languages by usage of merge processes that takes more than one streams and outputs a merged single stream, which can easily be implemented in those languages. However, in overlay programming, it is often the case that more dynamic many-to-one communication needs to be supported.

We have been investigating to introduce queue data type to solve this problem. Figure 4 shows an example.

\begin{verbatim}
createStack(Q, Os) :- true | Q := createQueue(Cs), stack(Cs, [], Os).
stacker(Q, . . .) :- true | Q \langle= push(X), . . ., stacker(Q, . . .).
:- createStack(Q, Os), . . ., stacker(Q, . . .)@node_id(ID1), stacker(Q, . . .)@node_id(ID2).
\end{verbatim}

* \langle= is the enqueue operator.

Figure 4. Multiple writers to a stream

A queue is created over a variable that represents a stream. Multiple writers can put data into the queue, which are merged as a single stream output headed by the variable.

Security Considerations Obviously, remote goal placement must only be allowed if the remote processor agrees to execute the goal. We are in the process of determining what the best way is for realizing this. (Our current implementation has a list of Jabber IDs from which the user is willing to accept goals.)

3.2 Remote Goal Placement Applications

Mobility and Ad Hoc Communication Mobility realizes a sustaining self over movements, as near-by entities always
provide the same services to maintain such a self. Ad hoc communication is a related concept, where the network construction is improvised on the spot. These concepts can be realized by portable devices placing part of their functionalities on processors in the vicinity.

**Distributed Computers** Components of a computing machine can be distributed over remote processors, in order to distribute loads, or to route I/O for particular purposes. Overlay GHC with its remote goal placement is suitable for expressing such distribution of functionalities.

As an example, we have an experience in implementing a software version of a single-core 4bit CPU in Overlay GHC using multiple workstations for educational purposes.

**Global Operating System Shell** A global OS is a system responsible for the management and coordination of activities for sharing the resources on the Earth. It will be realized by a distributed computer consisting of autonomous agents connected by LPLC (local production, local consumption) basis for the best autonomy and resilience.

Overlay GHC has been intended to be a shell language for such an OS. It works as a glue language to form global OS services and applications by combining existing functionalities in the distributed computer.

---

**Figure 2. Simple group chat program in Overlay GHC**

4 Implementation

The implementation of an Overlay GHC interpreter is ongoing as a plug-in for wija, an XMPP-based instant messaging system and a global OS shell prototype we have been developing in Java. The latest version is available from the following URL:

- http://www.media-art-online.org/ghc/

Our implementation of Overlay GHC has **package programming interface** that enables other plug-ins for wija to provide a set of Overlay GHC primitives and processes, which helps programmers to combine functionalities over the global OS shell prototype.

5 Related Work

5.1 Distributed KL1

Distributed KL1 is a predecessor to Overlay GHC as an extension of GHC for distributed programming. It has been implemented as DKLIC[5] language system that is an extension to KLIC, a KL1 to C compiler. DKLIC has a notion similar to remote goal placement, but it is called by the term **remote predicate call**, as predicate migration does not seem to have been implemented.
Problems such as multiple writers to a stream or processor failures do not seem to have been addressed either.

5.2 P2/OverLog

P2[4] is a system using OverLog language to express overlay networks in a compact and reusable form.

OverLog and Overlay GHC share the similar goals (rapid and declarative implementations of overlay networks) and similar approaches (logic programming). They are different in that OverLog is a query language based on DataLog, a subset of Prolog, while Overlay GHC is a descendant of concurrent logic programming languages. Further comparison studies are planned between the two environments.

5.3 Erlang

Erlang[2] is a functional programming language designed for expressing massive concurrency.

Erlang and Overlay GHC are related in that concurrent computation in Erlang is based on actor model[1], and the model is regarded as a special interpretation of concurrent logic programming[3]. Further comparison studies are also planned between those languages.

5.4 StreamIt

StreamIt[9] is a language for high performance streaming applications. It has two goals: to provide high-level stream abstractions, and to serve as a common machine language for grid-based architectures (instead of C as a common machine language for von Neumann architectures).

Because stream programming is one aspect of concurrent logic programming, many language features of StreamIt are also expressible in Overlay GHC. At this moment, Overlay GHC is not too concerned about performance of the generated overlay networks, but perhaps it is worth investigating to compile an Overlay GHC program into StreamIt code.

6 Future Work

We need to accumulate more experiences with programming in Overlay GHC to better our design. As an example, Kademlia[6], a distributed hash table (DHT), is being implemented in Overlay GHC. Our efforts for implementing existing overlay networks in the language will be followed by other DHT's and various distributed algorithms.

At the same time, we are planning to develop a scripting language tentatively called OG that corresponds one-to-one with Overlay GHC. At a different level than StreamIt, Overlay GHC is a machine language for non-von Neumann architectures (readers are reminded that KL1 is the system description language for parallel inference machines). Our limited experiences in training programmers to use Overlay GHC suggest that a higher-level language may be necessary to ease the burden of those programmers who are not familiar with logic and/or concurrent programming.

7 Conclusions

This paper introduced Overlay GHC, an overlay network programming language based on concurrent logic programming language GHC. The language allows us to rapidly program an overlay network with a bird’s-eye view, and to code a whole logical network in one program.

As an example of overlay programming with the language, Kademlia is being implemented, to be followed by other DHT's and various distributed algorithms. Development of a corresponding scripting language is also planned.

References