The repeated demise of logic programming and why it will be reincarnated

Carl Hewitt

MIT EECS (emeritus)
at alum.mit.edu try carlhewitt

Abstract

Logic Programming is the proposal to implement systems using mathematical logic. Perhaps the first published proposal to use mathematical logic for programming was John McCarthy's Advice Taker paper.

Planner was the first language to feature "procedural plans" that were called by "pattern-directed invocation" using "goals" and "assertions". A subset called Micro Planner was implemented by Gerry Sussman, Eugene Charniak and Terry Winograd and was used in Winograd's natural-language understanding program SHRDLU, Eugene Charniak's story understanding work, and some other projects. This generated a great deal of excitement in the field of AI. It also generated controversy because it proposed an alternative to the logic approach that had been one of the mainstay paradigms for AI. The question arose as what the difference was between the procedural and logical approaches. It took several years to answer this question. The upshot is that the procedural approach has a different mathematical semantics (based on the denotational semantics of the Actor model) from the semantics of mathematical logic.

There were some surprising results from this research including that mathematical logic is incapable of implementing general concurrent computation even though it can implement sequential computation and some kinds of parallel computation including the lambda calculus. Also along the way a large number of logic programming experiments were carried out although none met with great success. Also classical logic blows up in the face of inconsistent information that is becoming more ubiquitous with the growth of the Internet.

Now we are in the midst of a huge paradigm shift to massive concurrency with the advent of Web Services and many-core computer architectures. This paradigm shift enables and requires a new generation of systems incorporating ideas from mathematical logic in their implementation. The result will be that logic programming will be reincarnated. But something is often transformed when reincarnated!

Actors

Actors are the universal primitives of concurrent digital computation. In response to a message that it receives, an Actor can make local decisions, create more Actors, send more messages, and designate how to respond to the next message received. A Serializer is an Actor that is continually open to the arrival of messages. Messages sent to a Serialize always arrive although delivery can take an unbounded amount of time. (The Actor model can be augmented with metrics.)

Unbounded nondeterminism is the property that the amount of delay in servicing a request can become unbounded as a result of arbitration of contention for shared resources *while still guaranteeing that the request will eventually be serviced.*

Arguments for unbounded nondeterminism include the following:

- There is no bound that can be placed on how long it takes a computational circuit called an *Arbiter* to settle.
  - Arbiters are used in computers to deal with the circumstance that computer clocks operate asynchronously with input from outside, "e.g..", keyboard input, disk access, network input, "etc."
  - So it could take an unbounded time for a message sent to a computer to be received and in the meantime the computer could traverse an unbounded number of states.
- Electronic mail enables unbounded nondetermism since mail can be stored on servers indefinitely (perhaps the server is down temporarily) before being delivered.
- Communication links to servers on the Internet can be out of service indefinitely.

Actor Programming Language Semantics

Actors can be used to define the semantics of concurrent programming languages.

Actor script semantics

Actor script semantics are defined by the behavior of Actors that serve as the script elements of an Actor programming language.

For example consider the following concurrent programming language in which each <expression> is one of the following kinds:

<identifier>

When Communication[Eval[environment] customer] is received, send environment Communication[lookup[<identifier>]customer]

send <recipient> <communication>

When Communication[Eval[environment] customer] is received, send <recipient> Communication[Eval[environment] customer'] where customer' is a new Actor such that when customer' receives the communication r, then send <communication> Communication[Eval[environment] customer''] where customer'' is a new actor such that when customer'' receives the communication c, then send r c

<recipient>.<message>

When Communication[Eval[environment] customer] is received, send <recipient> Communication[Eval[environment] customer'] such that when customer' receives the communication r, then send <message> Communication[Eval[environment] customer''] such that when customer'' receives the communication m, then send r Communication[m, customer]

receiver <pattern>$_i$ <expression>$_i$

When Communication[Eval[environment] customer] is received, send customer a new actor r such that

when r receives a communication com, then try environment.bind[<pattern>$_i$ com] and

1. if a new environment' is created send <expression>$_i$ Communication[Eval[environment']]
2. otherwise try <pattern>$_{i+1}$

behavior <pattern>$_i$ <expression>$_i$

When Communication[Eval[environment] customer] is received, send customer a new actor r such that

when r receives Communication[message customer'], then try environment.bind[<pattern>$_i$ message] and

1. if a new environment' is created send <expression>$_i$ Communication[Eval[environment'] customer']
2. otherwise try <pattern>$_{i+1}$

{<expression1>, <expression2>}

When Communication[Eval[environment] customer] is received, send <expression1> Communication[Eval[environment]] and concurrently send <expression2> Communication[Eval[environment] customer]

let <identifier> = <expression1> *in* <expression2>

When message[Eval[environment] customer] is received, then create a new environment' by environment.bind[<identifier> <expression1>.Eval[environment]] and send <expression2> Communication[Eval[environment'] customer]

serializer <expression>

When Communication[Eval[environment] customer] is received, then send customer a new actor s such that communications sent to s are processed in FIFO order with a behavior Actor that is initially <expression>.Eval[environment] and:

When communication com is received by s, then send the behavior Actor Communication[com customer'] where customer' is a new actor such that when it receives an Actor then it is used as the behavior Actor for the next communication received by s.

Example Actor script

An example Actor script for a simple storage cell that can contain any Actor address is as follows:

```
Cell ≡
  receiver
    Communication[Create[initial] customer]
      send customer
           serializer readWrite(initial)}
```

The above script which creates a storage cell makes use of the behavior readWrite which is defined as follows:

```
readWrite(contents) ≡
  behavior
    Communication[read[] customer]
      {send customer contents,
       readWrite(contents)}
    Communication[write[x] customer]
      {send customer x,
       readWrite(x)}
```

For example the following expression creates a cell x with initial contents 5 and then concurrently writes to it with the values 7 and 9.

```
let x = Cell.Create[5]
  in {x.write[7],
      x.write[9],
      x.read[]}
```

The value of the above expression is either 7 or 9.

A Limitation of Logic Programming

In his 1988 paper on the early history of Prolog, Bob Kowalski published the thesis that "computation could be subsumed by deduction" and quoted with approval "Computation is controlled deduction." which he attributed to Pat Hayes. Contrary to Kowalski and Hayes, Hewitt's thesis was that logical deduction was incapable of carrying out concurrent computation in

open systems because of indeterminacy in the arrival order of messages.

Indeterminacy in Concurrent Computation

Hewitt [1985], Hewitt and Agha [1991], and other published work argued that mathematical models of concurrency did not determine particular concurrent computations as follows: The Actor model makes use of arbitration for determining which message is next in the arrival ordering]of an Actor that is sent multiple messages concurrently. For example *Arbiters* can be used in the implementation of the arrival ordering of an Actor which is subject to physical indeterminacy in the arrival order.

In concrete terms for Actor systems, typically we cannot observe the details by which the arrival order of messages for an Actor is determined. Attempting to do so affects the results and can even push the indeterminacy elsewhere. Instead of observing the internals of arbitration processes of Actor computations, we await outcomes. Physical indeterminacy in arbiters produces indeterminacy in Actors. The reason that we await outcomes is that we have no alternative because of indeterminacy.

According to Chris Fuchs [2004], quantum physics is a theory whose terms refer predominately to our interface with the world. It is a theory not about observables, not about *beables*, but about '*dingables*' We tap a bell with our gentle touch and listen for its beautiful ring.

The semantics of indeterminacy raises important issues for autonomy and interdependence in information. In particular it is important to distinguish between *indeterminacy* in which factors outside the control of an information system are making decisions and *choice* in which the information system has some control.

It is not sufficient to say that indeterminacy in Actor systems is due to unknown/unmodeled properties of the network infrastructure. The whole point of the appeal to quantum indeterminacy is to show that aspects of Actor systems can be *unknowable and the participants can be entangled.*

Actor Model and Mathematical Logic

What does the mathematical theory of Actors have to say about logic programming? A closed system is defined to be one which does not communicate with the outside. Actor model theory provides the means to characterize all the possible computations of a closed Actor system. So mathematical logic can characterize (as opposed to implement) all the possible computations of a closed Actor system. However, this is impossible for an open Actor system S in which the addresses of outside Actors are passed into S in the middle of computations so that S can communicate with these outside Actors. These outside Actors can then in turn communicate with Actors internal to S using addresses supplied to them by S.

Prolog-like concurrent message-passing programming languages

Keith Clark, Herve Gallaire, Steve Gregory, Vijay Saraswat, Udi Shapiro, Kazunori Ueda, etc. developed a family of Prolog-like concurrent message passing programming languages using unification of shared variables and data structure streams for messages. Claims were made that these languages were based on mathematical logic. This kind of programming language was used as the basis of the Japanese Fifth Generation Project (ICOT).

However, the Prolog-like concurrent programming languages (like the Actor model) were based on message passing and consequently were subject to the same indeterminacy. This was the basis of the argument in Carl Hewitt and Gul Agha [1991] that the Prolog-like concurrent programming languages were neither deductive nor logical.

Updating the Scientific Community Metaphor

The Scientific Community Metaphor paper was published in 1981 by Bill Kornfeld and Carl Hewitt as an approach to understanding |scientific communities by extending pattern directed invocation programming languages that invoke high level procedural plans on the basis of messages, *e.g.*, assertions and goals. Their work built on the philosophy, history and sociology of science with its analysis that scientific research depends critically on monotonicity, concurrency, commutativity, and pluralism to propose, modify, support, and oppose scientific methods, practices, and theories.

A programming language named Ether was developed that invokes procedural plans to process goals and assertions concurrently by dynamically creating new rules during program execution. Ether also addressed issues of conflict and contradiction with multiple sources of knowledge and multiple viewpoints.

At this point the metaphor needs an update. Some ideas are presented in the sections below.

Monotonicity, Concurrency, Commutatvity, Pluralism, Skepticism, and Provenance

Scientific communities have characteristics of monotonicity, concurrency, commutativity, pluralism,

skepticism, and provenance whose semantics can be studied using the Actor model.

Monotonicity. Once something is published it cannot be taken back. Results are published so they are available to a community. Published work is collected and indexed. Retractions can be published in case of error. Publications are sometimes lost or difficult to retrieve. Sometimes it is easier to rederive a result than to look it up.

Concurrency. The activities overlap in time. The participants interact with each other by message passing. Resources are limited by including processing, communications, and storage

Commutativity. Publications can be read regardless of whether they initiate new research or become relevant to ongoing research. Initiating work on a new scientific raises the question whether the answer has already been published. While working on a project, attention needs to be paid to ongoing developments that can affect the project. The order in which information is received can influence how it is processed.

Pluralism. Publications include heterogeneous, overlapping and possibly conflicting information. There is no central arbiter of truth in scientific communities. Scientific fads sometimes sweep up almost everyone in a field. Sponsors can try to control scientific activities.

Skepticism. Skepticism is an important commitment of the Scientific Community Metaphor. Great effort is expended to try to undermine current information and replace it with better information.

Provenance. The provenance of information is of intense interest in the Scientific Community Metaphor.

Viewpoints

Ether used viewpoints to modularize information in publications. However a great deal of information is shared across viewpoints. So Ether made use of inheritance so that information in a viewpoint could be readily used in other viewpoints. Sometimes this inheritance is not exact as when the laws of physics in Newtonian mechanics are derived from those of Special Relativity. In such cases Ether used translation instead of inheritance. Imre Lakatos studied very sophisticated kinds of translations of mathematical (e.g., the Euler formula for polyhedra) and scientific theories.

Logical Viewpoints (Theories, Context). Viewpoints were used to implement natural deduction (Fitch [1952]) in Ether. In order to prove classically a goal of the form `(P implies Q)` in a viewpoint `V`, it is sufficient to create a new viewpoint `V'` that inherits from `V`, assert `P` in `V'`, and then prove `Q` in `V'`. An idea like this was originally introduced into programming language proving by Rulifson, Derksen, and Waldinger [1973] except since the Scientific Community Metaphor is concurrent rather than being sequential it does not rely on being in a single viewpoint that can be sequentially pushed and popped to move to other viewpoints.

Negotiation Viewpoints. More challenging semantics are processes for negotiation as studied in the sociology and philosophy of science by Michel Callon, Paul Feyerabend, Elihu M. Gerson, Bruno Latour, John Law, Karl Popper, Susan Leigh Star, Anselm Strauss, Lucy Suchman, etc.

Observational Viewpoints. According to Relational Quantum Physics [Laudisa and Rovelli 2005], the way distinct physical systems affect each other when they interact (and not of the way physical systems "are") exhausts all that can be said about the physical world. The physical world is thus seen as a net of interacting components, where there is no meaning to the state of an isolated system. A physical system (or, more precisely, its contingent state) is reduced to the net of relations it entertains with the surrounding systems, and the physical structure of the world is identified as this net of relationships. In other words, "Quantum physics is the theoretical formalization of the experimental discovery that the descriptions that different observers give of the same events are not universal."

The concept that quantum mechanics forces us to give up is: the description of a system independent from the observer providing such a description; that is the concept of the absolute state of a system. *I.e., there is no observer independent data at all*. According to Zurek [1982], "Properties of quantum systems have no absolute meaning. Rather they must be always characterized with respect to other physical systems."

Does this mean that there is no relation whatsoever between views of different observers? Certainly not. According to Rovelli [1996] "*It is possible to compare different views, but the process of comparison is always a physical interaction (and all physical interactions are quantum mechanical in nature).*"

Further limitations of Classical Mathematical Logic

In addition to the inability to implement concurrent computation, the semantics of mathematical logic suffers from difficulties including unruly combinatorics and the semantic failure of classical proof theory and model theory in the face of inconsistency as discussed below.

Unruly combinatorics

General mathematical theorem proving has been intractable in practice although verification has been more successful.

Semantic failure of classical proof theory and model theory in the face of inconsistency

Claim: *All very large knowledge bases about human information system interactions are inconsistent.*

In the face of inconsistency, classical mathematical logic fails, *e.g.,*

- Proof theory fails because everything can be proved: If Ψ is inconsistent, then $\forall\Omega\ \Psi \vdash \Omega$
- Model theory fails because there are no models: If Ψ is inconsistent, then $\neg\exists M \vdash_M \Psi$

Irrelevance

Classical mathematical logic allows many irrelevant derivations, e.g., $\vdash ((P \wedge \neg P) \Rightarrow Q)$, $\vdash (P \Rightarrow Q \vee Q \Rightarrow R)$, and $\vdash (P \Rightarrow (P \vee Q))$

Dichotomy

In classical mathematical logic $\vdash (P \vee \neg P)$ and $\vdash (\neg \neg P \Rightarrow P)$. The idea was that truth values could be assigned to ground propositions and a ground proposition is either true or false. But inconsistency means unsatisfiability and therefore there is no assignment of truth values to ground propositions that works in the face of inconsistency.

Why does it matter that classical mathematical logic fails for very large knowledge bases of human information system commitments?

One of my colleagues made the argument that it is not fault of classical logic that very large knowledge bases of human information system commitments are inconsistent. It just means that the knowledge base has bugs just like our programs. A problem with the argument is that the semantics of classical logic collapses in the face of inconsistency whereas the Actor denotational semantics does not collapse in the face of programs with bugs. So we need to replace classical logic with a reasoning system that does not collapse in the face of inconsistency.

Direct Logic

A foolish consistency is the hobgoblin of little minds-- Emerson

We need a revised mathematical logic to address the limitation of classical mathematical in dealing with inconsistency and irrelevancy.

The goals of Direct Logic are to clarify provenance in case of inconsistency and to make derivations more relevant. Consequently Direct Logic does not embrace full indirect proof $(\Psi \vdash \Phi), (\Psi \vdash \neg\Phi) \vdash \neg\Psi$ because it would immediately blow up in case of an inconsistency. Similarly Direct Logic does not embrace disjunctive expansion $\Psi \vdash \Psi \vee \Phi$ because it (together with disjunction elimination $\Psi \vee \Phi, \neg\Psi \vdash \Phi$) also cause a blow up via the following derivation [Lewis and Langford 1959]]: $\Psi, \neg\Psi \vdash \Psi \vee \Phi, \neg\Psi \vdash \Phi$

However, Direct Logic does allow the direct form of indirect proof $(\Psi \vdash \neg\Psi) \vdash \neg\Psi$.

Direct Logic uses ideas from Relevance Logic and Intuitionistic Logic. (Note that Direct Logic is distinct from the Direct Predicate Calculus [Ketonen and Weyhrauch 1984].)

Sequences of formulas

Direct Logic makes use of unordered sequences of formulas separated by commas intuitively meaning *and* (*i.e.* conjunction).

Basic Rules for $\vdash$

The basic rules for $\vdash$ are as follows:

$$\Psi \vdash \Psi$$
$$(\Psi \vdash \Theta) \vdash (\Psi, \Phi \vdash \Theta)$$
$$(\Psi \vdash \Phi, \Theta) \vdash (\Psi \vdash \Phi)$$
$$(\Psi \vdash \Phi), (\Omega \vdash \Theta) \vdash (\Psi, \Omega \vdash \Phi, \Theta)$$
$$(\Psi \vdash \Phi), (\Phi \vdash \Theta) \vdash (\Psi \vdash \Theta)$$

Direct Indirect Proof

Direct Logic only allows for the direct form of indirect proof which is the following rule:

$$(\Psi \vdash \neg\Psi) \vdash \neg\Psi$$

Rule for $\wedge$

The basic rule for $\wedge$ (conjunction) is

$$\Psi, \Phi \cong \Psi \wedge \Phi$$

The above rule justifies the following rules:

$$((\Psi \wedge \Phi) \vdash \Theta) \cong (\Psi, \Phi \vdash \Theta)$$
$$(\Phi \vdash (\Psi \wedge \Theta)) \cong (\Phi \vdash \Psi, \Theta)$$

Rules for $\vee$

The most basic rule for $\vee$ is

$$\Psi \vee \Phi \cong (\neg\Psi \vdash \Phi) \wedge (\neg\Phi \vdash \Psi)$$

Note that the provenance of $\Rightarrow$ is derivational as opposed to truth functional.

For the convenience of users, the negation elimination rule for $\vee$ (disjunction) is:

$$\neg(\Phi \vee \Psi) \cong \neg\Psi \wedge \neg\Phi$$

And the negation elimination rule for $\wedge$ (conjunction) is:

$$\neg(\Phi \wedge \Psi) \cong \neg\Psi \vee \neg\Phi$$

The last two equivalences above lead to the following equivalence:

$$\neg\neg\neg\Psi \cong \neg\Psi$$

Rule by Cases for ∨
Direct Logic has the *Rule by Cases* is a follows**:**

$$\Psi\vee\Phi,\ (\Psi\vdash\Theta),\ (\Phi\vdash\Theta)\vdash\Theta$$

Rules for ∧ and ∨
For convenience of users, Direct Logic has the following equivalences for ∧ and ∨:

$(\Psi\wedge\Psi)\cong\Psi$
$(\Psi\wedge\Phi)\cong(\Phi\wedge\Psi)$
$(\Psi\wedge(\Phi\wedge\Theta))\cong((\Psi\wedge\Phi)\wedge\Theta)$

$(\Psi\vee\Psi)\cong\Psi$
$(\Psi\vee\Phi)\cong(\Phi\vee\Psi)$
$(\Psi\vee(\Phi\vee\Theta))\cong((\Psi\vee\Phi)\vee\Theta)$

$(\Psi\wedge(\Phi\vee\Theta))\cong((\Psi\wedge\Phi)\vee(\Psi\wedge\Theta))$
$(\Psi\vee(\Phi\wedge\Theta))\cong((\Psi\vee\Phi)\wedge(\Psi\vee\Theta))$

Rule for ⇒
The most basic rule for ⇒ is

$$\Psi\Rightarrow\Phi\ \cong\ (\Psi\vdash\Phi)\ \wedge\ (\neg\Phi\vdash\neg\Psi)$$

Note that the provenance of ⇒ is derivational as opposed to truth functional.

Integers and XML
A theory of integers and XML is needed. So Direct Logic allows quantification over the integers and XML expressions where there is a standard encoding of logical formulas into XML. If $m\in\omega$ then

$$\wedge_{n\geq m}\Psi[n]\ \cong\Psi[m]\ \wedge\ \wedge_{n\geq m+1}\Psi[n]$$

$$\vee_{n\geq m}\Psi[n]\ \cong\Psi[m]\ \vee\ \vee_{n\geq m+1}\Psi[n]$$

The fundamental rules for quantification are:

$$\forall n\in\omega\ \Psi[n]\ \cong\ \wedge_{n\geq 0}\Psi[n]$$

$$\exists n\in\omega\ \Psi[n]\ \cong\ \vee_{n\geq 0}\Psi[n]$$

For the convenience of users, the following rules are provided:

$0\in\omega$
$\forall n\in\omega\ n+1\in\omega$
$\forall n\in\omega\ 0\leq n$
$\forall n,m\in\omega\ n=m\ \Leftrightarrow\ n+1=m+1$
$\forall n,m\in\omega\ n\leq m\ \Leftrightarrow\ (n=m\ \vee\ n+1\leq m)$
$\forall n,m\in\omega\ n\leq m\ \vee\ n=m\ \vee\ m\leq n$
$\exists n\in\omega\ \Psi[n]\ \Rightarrow\ (\exists m\in\omega\ \Psi[m]\ \wedge\ \forall i\in\omega\ \Psi[i]\Rightarrow m\leq i)$

Induction
Direct Logic has the rule of induction:

$$\Psi[0],\ (\forall n\in\omega\ \Psi[n]\vdash\Psi[n+1])\vdash\forall n\in\omega\ \Psi[n]$$

Reflection
Direct Logic has reflection of provability:

$$\forall\alpha\in\mathrm{XML}\ \lceil\vdash\alpha\rceil\vdash\lceil\alpha\rceil$$
$$\forall\alpha\in\mathrm{XML}\ \lceil\neg\vdash\alpha\rceil\vdash(\neg\vdash\lceil\alpha\rceil)$$

Rules for ∧ and ∨
In addition, Direct Logic has the following structural equivalences for ∧ and ∨

$\wedge_{i\in\omega}\Psi_i\ \wedge\ \vee_{i\in\omega}\Phi_i\cong\wedge_{i,j\in\omega}(\Psi_i\vee\Phi_i)$
$\vee_{i\in\omega}\Psi_i\ \vee\ \wedge_{i\in\omega}\Phi_i\cong\vee_{i,j\in\omega}(\Psi_i\wedge\Phi_i)$

Negation Elimination
For convenience, Direct Logic has the following structural equivalences for negation elimination

$\neg\vee_{i\in\omega}\Psi_i\ \cong\ \wedge_{i\in\omega}\neg\Psi_i$
$\neg\wedge_{i\in\omega}\Psi_i\ \cong\ \vee_{i\in\omega}\neg\Psi_i$
$\neg\forall x\in\mathrm{XML}\ \Psi[x]\cong\exists x\in\mathrm{XML}\ \neg\Psi[x]$
$\neg\exists x\in\mathrm{XML}\ \Psi[x]\cong\forall x\in\mathrm{XML}\ \neg\Psi[x]$

Incompleteness
Theorem (after Carnap): Let Ψ be a predicate on formulas

$$\mathrm{Fix}(\Psi)\dashv\vdash\Psi(\mathrm{Fix}(\Psi))$$
$$\text{where } \mathrm{Fix}(\Psi)\equiv\Omega(\Omega)$$
$$\text{where } \Omega=\lambda(P)\ \Psi(P(P))$$

Theorem (after Gödel): In Direct Logic $\mathbf{Paradox}$ ***but*** $\neg\vdash\mathbf{Paradox}$ where $\mathbf{Paradox}\equiv\mathrm{Fix}(\lambda(x)\lceil\neg\vdash x\rceil)$

Caveat
The rules of Direct Logic have been stated as broadly as possible to make the logic more useful. *However, it has not yet been verified that that it does not blow up*. If it does, then its rules will have to be adapted.

Using Provenance to Undercut a Derivation

In order to illustrate how provenance can be used to undercut a derivation consider the following (inconsistent) sentences:

(Says[person,s] ∧ Reliable[person])⇒ s

Says[Curveball, WMD]
Says[Tenet, Reliable[Curveball]]
Reliable[Tenet]]
Says[CIAstaff,¬Reliable[Curveball]]
Reliable[CIAstaff]

The derivation of WMD from Says[Curveball, WMD] and Reliable[Curveball] is undercut by the derivation of ¬Reliable[Curveball] from

Says[CIAstaff,¬Reliable[Curveball]] and Reliable[CIAstaff].

Future Work

Developments in hardware and software technology for the Internet since the original paper was published are tending to increase the importance of the Scientific Community Metaphor.

Legal concerns (e.g., HIPAA, Sarbanes-Oxley, "The Books and Records Rules" in SEC Rule 17a-3/4 and "Design Criteria Standard for Electronic Records Management Software Applications" in DOD 5015.2 in the US) are leading organizations to store information monotonically forever. It has just now become less costly in many cases to store information on magnetic disk than on tape. With increasing storage capacity, sites can monotonically record what they read from the Internet as well as monotonically recording their own operations.

Search engines currently provide rudimentary access to all this information. Future systems will provide interactive question answering broadly conceived that will make all this information much more useful.

Massive concurrency (i.e., Web services and many-core computer architectures) lies in the future posing enormous challenges and opportunities.

The future of Logic Programming lies in the further development of Direct Logic and the Scientific Community Metaphor in the context of massive concurrency.

Acknowledgments

Sol Feferman, David Israel, Ben Kuipers, Pat Langley, Vladimir Lifschitz, John McCarthy, Fanya Montalvo, Ray Perrault, Mark Stickel, Richard Waldinger, and others provided valuable feedback at seminars at Stanford, SRI, and UT Austin in which I presented earlier versions of the material in this paper. Subsequently Richard Waldinger and the AAMAS-06 and KR-06 reviewers made valuable suggestions for improvement of material in this paper.

Appendix

Perhaps the first published proposal to use mathematical logic for programming was John McCarthy [1958]. Planner is a programming language designed by Carl Hewitt at MIT, and first published in 1969. Hewitt championed the "procedural embedding of knowledge" in the form of high level procedural plans in contrast to the logical approach pioneered by John McCarthy who advocated expressing knowledge declaratively in mathematical logic for Artificial Intelligence. This raised a fundamental question: "What is the difference between the procedural and logical approaches?" The upshot is that the procedural approach has a different mathematical semantics (based on the denotational semantics of the Actor model) from the semantics of mathematical logic.

Planner was the first language to feature "procedural plans" that were called by "pattern-directed invocation" using "goals" and "assertions". A subset called Micro Planner was implemented by Gerry Sussman, Eugene Charniak and Terry Winograd and was used in Winograd's natural-language understanding program SHRDLU, Eugene Charniak's story understanding work, and some other projects. This generated a great deal of excitement in the field of AI.

However, computer memories were very small by current standards because they were expensive, being made of iron ferrite cores at that time. So Planner adopted the then common expedient of using backtracking control structures to economize on the use of computer memory. In this way, the computer only had to store one possibility at a time in exploring alternatives.

Bruce Anderson at Edinburgh University implemented a subset of Micro Planner and Julian Davies at Edinburgh essentially the whole of Planner. At SRI, Jeff Rulifson, Jan Derksen, and Richard Waldinger developed QA4 (later evolved into QLisp) which built on the constructs in Planner and introduced a context mechanism to provide modularity for expressions in the data base as well as enriched data structures (sets and bags) along with associative-commutative unification for these data structures.

Peter Landin had introduced a powerful control structure using his J (for Jump) operator that could perform a nonlocal goto into the middle of a procedure invocation. In fact the J operator could jump back into the middle of a procedure invocation even after it had already returned! Drew McDermott and Gerry Sussman called Landin's concept "Hairy Control Structure" and used it in the form of a nonlocal goto for the Conniver programming language. Scott Fahlman used Conniver in his planning system for robot construction tasks.

These difficulties with control structure inspired the development of the Actor model of computation. Hewitt reported: "*... we have found that we can do without the paraphernalia of "hairy control structure" (such as possibility lists, non-local gotos, and assignments of values to the internal variables of other procedures in CONNIVER.)... The conventions of ordinary message-passing seem to provide a better structured, more intuitive foundation for constructing the communication systems needed for expert problem-solving modules to cooperate effectively.*"

Bob Kowalski, who had been one of the principal members of the logic paradigm community, then adapted in collaboration with Alain Colmerauer some theorem proving ideas into a form similar to a subset of Micro Planner called Prolog. Indeed, Prolog can be viewed as largely a reinvention of a subset of Micro Planner, *e.g.*, Micro Planner (unlike Prolog) had the capability to use pattern-directed invocation of procedural plans from assertions as well as goals.

References

Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems* Doctoral Dissertation. 1986.

Bruce Anderson. "Documentation for LIB PICO-PLANNER" School of Artificial Intelligence, Edinburgh University. 1972.

William Athas and Nanette Boden "Cantor: An Actor Programming System for Scientific Computing" in Proceedings of the NSF Workshop on Object-Based Concurrent Programming. 1988. Special Issue of SIGPLAN Notices.

Fisher Black. "A deductive question answering system" Harvard University. Thesis. 1964.

H. Blair and V. S. Subrahmanian. "Paraconsistent Logic Programming". Theoretical Computer Science, 68(2) 1989

Daniel Bobrow, "A Model for Control Structures for Artificial Intelligence Programming Languages" IJCAI 1973.

Will Clinger. *Foundations of Actor Semantics* MIT Mathematics Doctoral Dissertation. June 1981.

F. S. Correa da Silva, J. M. Abe, and M. Rillo. "Modeling Paraconsistent Knowledge in Distributed Systems". Technical Report RT-MAC-9414, Instituto de Matematica e Estatistica, Universidade de Sao Paulo, 1994.

Julian Davies. "Popler 1.5 Reference Manual" University of Edinburgh, TPU Report No. 1, May 1973.

Scott Fahlman. *A Planning System for Robot Construction Tasks* MIT AI TR-283. June 1973.

Adam Farquhar, Anglela Dappert, Richard Fikes, and Wanda Pratt. "Integrating Information Sources Using Context" Logic Knowledge Systems Laboratory. KSL-95-12. January, 1995.

Frederic Fitch. *Symbolic Logic: an Introduction*. Ronald Press, New York, 1952.

Michael Gelfond and Vladimir Lifschitz. "Logic programs with classical negation" 7th International Conference on Logic Programming. MIT Press 1990.

C. Cordell Green: "Application of Theorem Proving to Problem Solving" IJCAI 1969:

Irene Greif. *Semantics of Communicating Parallel Processes* MIT EECS Doctoral Dissertation. August 1975

Ramanathan Guha. *Contexts: A Formalization and Some Applications* PhD thesis, Stanford University, 1991.

Carl Hewitt. "Planner: A Language for Proving Theorems in Robots" IJCAI 1969.

Carl Hewitt. "Procedural Embedding of Knowledge In Planner" IJCAI 1971.

Carl Hewitt. "Description and Theoretical Analysis (Using Schemata) of Planner, A Language for Proving Theorems and Manipulating Models in a Robot" AI Memo No. 251, MIT Project MAC, April 1972.

Carl Hewitt, Peter Bishop and Richard Steiger. "A Universal Modular Actor Formalism for Artificial Intelligence" IJCAI 1973.

Carl Hewitt and Henry Baker Laws for Communicating Parallel Processes IFIP. August 1977.

Carl Hewitt. "Viewing Control Structures as Patterns of Passing Messages" Journal of Artificial Intelligence. June 1977.

Carl Hewitt. "The Challenge of Open Systems" Byte Magazine. April 1985.

Carl Hewitt and Gul Agha. "Guarded Horn clause languages: are they deductive and Logical?" International Conference on Fifth Generation Computer Systems. Ohmsha 1988.

Carl Hewitt. "New Foundations for Concurrency: Denotational Actor Semantics" Submitted for publication. 2005.

Carl Hewitt. "What is Commitment? Physical, Organizational, and Social" Submitted for publication. 2005.

J. Ketonen and R. Weyhrauch. "A decidable fragment of Predicate Calculus" Theoretical Computer Science. 1984.

Bill Kornfeld and Carl Hewitt. "The Scientific Community Metaphor" IEEE Transactions on Systems, Man, and Cybernetics. January 1981.

Bill Kornfeld. *Parallelism in Problem Solving* MIT EECS Doctoral Dissertation. August 1981.

Robert Kowalski. "The limitation of logic" Proceedings of the 1986 ACM fourteenth annual conference on Computer science.

Robert Kowalski "Predicate Logic as Programming Language" Memo 70, Department of AI, Edinburgh University. 1973.

Robert Kowalski. "The Early Years of Logic Programming" CACM. January 1988.

Clarence Lewis and H. L. Langford. *Symbolic Logic* 2nd Ed. Dover. 1959.

Federico Laudisa and Carlo Rovelli. "Relational Quantum Mechanics", *The Stanford Encyclopedia of Philosophy (Fall 2005 Edition)*.

Henry Lieberman. "A Preview of Act 1" MIT AI memo 625. June 1981.

Edwin Mares. "Relevance Logic" The Stanford Encyclopedia of Philosophy. Summer 1998.

John McCarthy. "Programs with common sense" Symposium on Mechanization of Thought Processes. National Physical Laboratory. Teddington, England. 1958.

John McCarthy. "Generality in Artificial Intelligence" CACM. December 1987.

John McCarthy. "A logical AI Approach to Context" Technical note, Stanford Computer Science Department, 1996.

Drew McDermott and Gerry Sussman. "The Conniver Reference Manual" MIT AI Memo 259. May 1972.

Robin Milner "Elements of interaction: Turing award lecture", CACM. January 1993.

Joan Moschovakis. "Intuitionistic Logic" The Stanford Encyclopedia of Philosophy. Spring 2004.

Graham Priest and Koji Tanaka. "Paraconsistent Logic" The Stanford Encyclopedia of Philosophy. Winter 2004.

Carlo Rovelli "Relational quantum mechanics" International Journal of Theoretical Physics 35 1637-1678. 1996.

Jeff Rulifson, Jan Derksen, and Richard Waldinger. "QA4, A Procedural Calculus for Intuitive Reasoning" SRI AI Center Technical Note 73, November 1973.

Davide Sangiorgi and David Walker. "The Pi-Calculus : A Theory of Mobile Processes" Cambridge University Press. 2001.

Gerry Sussman, Terry Winograd and Eugene Charniak. "Micro-Planner Reference Manual (Update)" AI Memo 203A, MIT AI Lab, December 1971

Ehud Shapiro. "The family of concurrent logic programming languages" ACM Computing Surveys. September 1989.

Terry Winograd. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*. MIT AI TR-235. January 1971.

Wojciech Zurek. Physics Review Letters. D26 1862. 1982.