Development of Logic Programming:
What went wrong, What was done about it, and What it might mean for the future

Carl Hewitt
at alum.mit.edu try carlhewitt

Abstract

Logic Programming can be broadly defined as "*using logic to deduce computational steps from existing propositions*" (although this is somewhat controversial). The focus of this paper is on the development of this idea. Consequently, it does not treat any other associated topics related to Logic Programming such as constraints, abduction, *etc.*

The idea has a long development that went through many twists in which important questions turned out to have surprising answers including the following:

- Is computation reducible to logic?
- Are the laws of thought consistent?

This paper describes what went wrong at various points, what was done about it, and what it might mean for the future of Logic Programming.

Church's *Foundation of Logic*

Arguably, Church's *Foundation of Logic* was the first Logic Programming language [Church 1932, 1933].[1] It attempted to avoid the known logical paradoxes by using partial functions and disallowing proof by contradiction.

The system was very powerful and flexible. Unfortunately, it was so powerful that it was inconsistent [Kleene and Rosser 1935] and consequently the logic was removed leaving only the functional lambda calculus [Church 1941].

What went wrong:
A logical system that was developed by Church to be a new foundation for logic turned out to have inconsistencies that could not be removed.

What was done about it:

- Logic was removed from the system leaving the functional lambda calculus, which has been very successful
- Much later a successor system Direct Logic [Hewitt 2008a] was developed that overcame these problems of Church's *Foundation of Logic*.

Advice Taker

McCarthy [1958] proposed the Logicist Programme for Artificial Intelligence which included the Advice Taker with the following main features:

1. *There is a method of representing expressions in the computer. These expressions are defined recursively as follows: A class of entities called terms is defined and a term is an expression. A sequence of expressions is an expression. These expressions are represented in the machine by list structures [Newell and Simon 1957].*
2. *Certain of these expressions may be regarded as declarative sentences in a certain logical system which will be analogous to a universal Post canonical system. The particular system chosen will depend on programming considerations but will probably have a single rule of inference which will combine substitution for variables with modus ponens. The purpose of the combination is to avoid choking the machine with special cases of general propositions already deduced.*
3. *There is an immediate deduction routine which when given a set of premises will deduce a set of immediate conclusions. Initially, the immediate deduction routine will simply write down all one-step consequences of the premises. Later, this may be elaborated so that the routine will produce some other conclusions which may be of interest. However, this routine will not use semantic heuristics; i.e., heuristics which depend on the subject matter under discussion.*
4. *The intelligence, if any, of the advice taker will not be embodied in the immediate deduction routine. This intelligence will be embodied in the procedures which choose the lists of premises to which the immediate deduction routine is to be applied.*
5. *The program is intended to operate cyclically as follows. The immediate deduction routine is applied to a list of premises and a list of individuals. Some of the conclusions have the form of imperative sentences. These are obeyed. Included in the set of imperatives which may be obeyed is the routine which deduces and obeys.*

What went wrong:

- The imperative sentences deduced by the Advice Taker could have impasses in the following forms:
  - *lapses* in which no imperative sentences were deduced
  - *conflicts* in which inconsistent sentences were deduced.
- The immediate deduction routine of the Advice Taker was extremely inefficient

[1] Of course this was back when computers were humans!

What was done about it:

- McCarthy, *et. al.,* developed Lisp (one of the world's most influential programming languages) in order to implement ideas in the Advice Taker and other AI systems.
- McCarthy changed the focus of his research to solving epistemological problems of Artificial Intelligence
- The Soar architecture was developed to deal with impasses [Laird, Newell, and Rosenbloom 1987].

Uniform Proof Procedures based on Resolution

John Alan Robinson [1965] developed a deduction method called resolution which was proposed as a uniform proof procedure for proving theorems that converted everything to clausal form and then used a method analogous to modus ponens to attempt to obtain a proof by contradiction by adding the clausal form of the negation of the theorem to be proved.

The first use of Resolution was in computer programs to prove mathematical theorems and in the synthesis of simple sequential programs from logical specifications [Wos 1965; Green 1969; Waldinger and Lee 1969; Anderson and 1970; 1971, etc.]. In the resolution uniform proof procedure theorem proving paradigm, the use of procedural knowledge was considered to be "cheating" [Green 1969].

What went wrong:

- Converting all information to clausal form is problematic because it hides the underlying structure of the information.
- Using resolution as the only rule of inference is problematical because it hides the underlying structure of proofs
- It proved to be impossible to develop efficient enough uniform proof procedures for practical domains.[2]
- Using proof by contradiction is problematical because the axiomatizations of all practical domains of knowledge are inconsistent in practice. And proof by contradiction is not a sound rule of inference for inconsistent systems.

What was done about it:

- The *Procedural Embedding of Knowledge* paradigm [Hewitt 1971] was developed as an alternative to *Resolution Uniform Proof Procedure* paradigm.
- Strongly paraconsistent logic (such as Direct Logic [Hewitt 2008a]) was developed to isolate inconsistencies during reasoning. (See section below on the future of Logic Programming.)

[2] In other words, taking a first order axiomatization of a large practical domain, converting it to clausal form, and then using a uniform resolution proof procedure was found to be so wildly inefficient that answers to questions of interest could not be found even though they were logically entailed.

Planner

The two major paradigms for constructing semantic software systems were procedural and logical. The procedural paradigm was epitomized by Lisp [McCarthy et. al. 1962] which featured recursive procedures that operated on list structures. The logical paradigm was epitomized by uniform resolution theorem provers [Robinson 1965].

Planner [Hewitt 1969] was a kind of hybrid between the procedural and logical paradigms. It featured a procedural interpretation of logical sentences in that an implication of the form (P *implies* Q) can be procedurally interpreted in the following ways [Hewitt 1969]:

- *Forward chaining*
  - When assert P, assert Q
  - When assert not Q, assert not P
- *Backward chaining*
  - When goal Q, goal P
  - When goal not P, goal not Q

Planner was the first programming language based on the pattern-directed invocation of procedural plans from assertions and goals. The development of Planner was inspired by the work of Karl Popper [1935, 1963], Frederic Fitch [1952], George Polya [1954], Allen Newell and Herbert Simon [1956], John McCarthy [1958, et. al. 1962], and Marvin Minsky [1968]. It was a rejection of the resolution uniform proof procedure paradigm.

A subset called Micro-Planner was implemented by Gerry Sussman, Eugene Charniak and Terry Winograd. Micro-Planner was used in Winograd's natural-language understanding program SHRDLU [Winograd 1971], Eugene Charniak's story understanding work, work on legal reasoning [McCarty 1977], and some other projects. This generated a great deal of excitement in the field of AI. Being a hybrid language, Micro Planner had two different syntaxes. So it lacked a certain degree of elegance. In fact, after Hewitt's lecture at IJCAI'71, Allen Newell rose from the audience to remark on the lack of elegance in the language!

Computers were expensive. They had only a single slow processor and their memories were very small by comparison with today. So Planner adopted the then common expedient of using backtracking [Golomb and Baumert 1965]. In this way it was possible to economize on the use of time and storage by working on and storing only one possibility at a time in exploring alternatives.

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 [Landin 1965]. In fact the J operator enabled a program to 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 [McDermott and Sussman 1972].

Pat Hayes [1974] remarked:

> *Their* [Sussman and McDermott] *solution, to give the user access to the implementation primitives of Planner, is however, something of a retrograde step (what are Conniver's semantics?), although pragmatically useful and important in the short term. A better solution is to give the user access to a meaningful set of primitive control abilities in an explicit representational scheme concerned with deductive control.*

However, there was the germ of a good idea in Conniver; namely, using co-routines to computationally shift focus to another branch of investigation while keeping alive the one that has been left. Scott Fahlman used this capability of Conniver to good effect in his planning system for robot construction tasks [Fahlman 1973] to introduce a set of higher-level control/communications operations for its domain. However, the ability to jump back into the middle of procedure invocations didn't seem to be what was needed to solve the difficulties in communication that were a root cause of the control structure difficulties. Conniver was also useful in that it provoked further research into control structures for Planner-like languages.

In 1972 Alan Kay visited MIT and gave an inspiring lecture that explained some of his ideas for Smalltalk-72, building on the message-passing of Planner and Simula [Dahl and Nygaard 1967] as well as the Logo work of Seymour Papert with the "little person" model of computation used for teaching children to program (cf. [Whalley 2006]). However, the message passing of Smalltalk-72 was quite complex [Ingalls 1983]. Also, as presented by Kay, Smalltalk-72 (like Simula before it) was based on co-routines rather than true concurrency.

The Actor model [Hewitt, Bishop, and Steiger 1973] was a new model of computation that differed from previous models of computation in that it was inspired by the laws of physics. It took some time to develop programming methodologies for the Actor model. 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.*

What went wrong:

1. Although pragmatically useful at the time it was developed, backtracking proved to be too rigid and uncontrollable.
2. Planner had a single global data base which was not modular or scalable.
3. Although pragmatically useful for interfacing with the underlying Lisp system, the syntax of Planner was not a pretty sight.

What was done about it:

1. Concurrency based on message passing was developed as an alternative to backtracking.
2. *QA-4* [Rulifson, Derksen, and Waldinger 1973] developed a hierarchical context system to modularize the data base. Contexts were later generalized into viewpoints in the *Scientific Community Metaphor*. The idea is fundamental to the strongly paraconsistent theories of Direct Logic [Hewitt 2008a]. (See section below on the future of Logic Programming.)
3. Prolog [Kowalski 1974, Colmerauer and Roussel 1996] was basically a subset of Planner that restricted programs to clausal form using backward chaining and consequently had a simpler more uniform syntax. (But Prolog did not include the forward chaining of Planner.)

Japanese Fifth Generation Project (ICOT)

Beginning in the 1970's, Japan became dominant in the DRAM market (and consequently most of the integrated circuit industry). This was accomplished with the help of the Japanese VLSI project that was funded and coordinated in good part by the Japanese government Ministry of International Trade and Industry (MITI) [Sigurdson 1986]. MITI hoped to enlarge this victory by taking over the computer industry with a new Fifth Generation Computing System Project (named ICOT). However, Japan had come under criticism for "copying" the US. One of the MITI goals for ICOT was to show that Japan could innovate new computer technology and not just copy the Americans.

According to Kowalski [2004],

> *The announcement of the FGCS* [Fifth Generation Computing Systems] *Project in 1981 triggered reactions all over the world.*
>
> *...*
>
> *Logic Programming was virtually unknown in mainstream Computing at the time, and most of its research activity was in Europe. So it came as a big shock - nowhere more so than in North America - when it eventually became obvious that logic programming was to play a central, unifying role in the FGCS Project.*

ICOT, partly influenced by Logic Programming enthusiasts, tried to go all the way with Logic

Programming. Kowalski later recalled "*Having advocated LP* [Logic Programming] *as a unifying foundation for computing, I was delighted with the LP focus of the FGCS project.*" [Fuchi, Kowalski, Ueda, Kahn, Chikayama, and Tick 1993] By making Logic Programming (which was mainly being developed outside the US) the foundation, MITI hoped that the Japanese computer industry could leapfrog the US.

This meant that ICOT had to deal with concurrency and consequently developed concurrent programming languages based on clauses that were loosely related to logic [Shapiro 1989]. However, it proved difficult to implement clause invocation in these languages as efficiently as procedure invocation in object-oriented programming languages. Simula-67 originated a hierarchical class structure for objects so that message handling procedures (methods) and object instance variables could be inherited by subclasses. Ole-Johan Dahl [1967] invented a powerful compiler technology using dispatch tables that enabled message handling procedures in subclasses of objects to be efficiently invoked. The combination of efficient inheritance-based procedure invocation together with class libraries and browsers (pioneered in Smalltalk) was better than the slower pattern-directed clause invocation of the FGCS programming languages. Consequently, the ICOT programming languages never took off and instead concurrent object-oriented message-passing languages like Java and C# became the mainstream.

The greater efficiency of object-oriented programming languages was especially ironic because:

> *Even if it made sense for MITI to boldly seek a vast step forward in computer technology, the particular approach MITI ultimately authorized was widely criticized in corporate Japan. If the central focus of the Fifth Generation was Artificial Intelligence, the emphasis on great speed in making inferential steps seemed unnecessary.* [Saxonhouse 2001]

"*The* [ICOT] *project aimed to leapfrog over IBM, and to a new era of advanced knowledge processing applications"* [Sergot 2004] But the MITI strategy backfired because the software wasn't good enough (as explained above) and so the Japanese companies refused to productize the ICOT hardware.

What went wrong:
The way that it used Logic Programming was a principle contributing cause to the failure of ICOT because Logic Programming proved not to be competitive with object-oriented programming.

What was done about it:
- Japanese companies refused to productize the ICOT architecture.
- ICOT languished and then suffered a lingering death.

Computation is not Subsumed by Deduction

Kowalski developed the thesis that "*computation could be subsumed by deduction"* [Kowalski 1988a] which he states was first proposed by Hayes [1973] in the form "*Computation = controlled deduction."* [Kowalski 1979]. This thesis was also implicit in one interpretation of Cordell Green's earlier work. [Green 1969]

But computation in general cannot be subsumed by deduction and computation in general is not controlled deduction. Hewitt and Agha [1991] argued that mathematical models of concurrency did not determine particular concurrent computations:

> *The Actor Model makes use of arbitration for determining which message is next in the arrival order of an Actor that is sent multiple messages concurrently. For example Arbiters can be used in the implementation of the arrival order of messages sent to an Actor which are subject to indeterminacy in their arrival order. Since arrival orders are in general indeterminate, they cannot be deduced from prior information by mathematical logic alone. Therefore mathematical logic cannot implement concurrent computation in open systems.*

Instead of deducing the outcomes of arbitration, we await outcomes. Indeterminacy in arbiters produces indeterminacy in concurrent comptation. The reason that we await outcomes is that we have no alternative because of indeterminacy.

According to Hewitt [2007]:

> "What does the mathematical theory of Actors have to say about this? 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 in terms of the Representation Theorem [Hewitt 2006]: The denotation $\mathsf{Denote}_{\mathfrak{Z}}$ of an Actor system $\mathfrak{Z}$ represents all the possible behaviors of $\mathfrak{Z}$ as
>
> $$\mathsf{Denote}_{\mathfrak{Z}} = \bigsqcup_{i\in\omega} \mathsf{Progression}_{\mathfrak{Z}}^{i}(\bot_{\mathfrak{Z}})$$
>
> *where* $\mathsf{Progression}_{\mathfrak{Z}}$ is an approximation function that takes a set of approximate behaviors to their next stage and $\bot_{\mathfrak{Z}}$ is the initial behavior of $\mathfrak{Z}$."

Consequently, Logic Programming can represent but not in general implement concurrent systems. On the other hand, inconsistent theories do not have any classical logical models at all. So the tables have turned!

What went wrong:
The thesis that computation is subsumed by deduction failed because concurrent computation could not be implemented.

What was done about it:
A mathematical foundation for concurrent computation was developed based on domain theory [Scott and Strachey 1971, Clinger 1981, Hewitt 2007].

The Laws of Thought are Inconsistent

Platonic Ideals were to be perfect, unchanging, and eternal. Beginning with the Hellenistic mathematician Euclid [circa 300BC] in Alexandria, theories were intuitively supposed to be both consistent and complete. However, using a kind of reflection, Gödel [1931] (later generalized by Rosser [1936]) proved that mathematical theories are incomplete, i.e., there are propositions which can neither be proved nor disproved. This was accomplished using a kind of reflection by showing that in each sufficiently strong theory $\mathcal{T}$, there is a paradoxical proposition $\mathsf{Paradox}_{\mathcal{T}}$ which is logically equivalent to its own unprovability, *i.e.,* $\neg \vdash_{\mathcal{T}} \mathsf{Paradox}_{\mathcal{T}}$.

Some restrictions are needed around reflection to avoid inconsistencies in mathematics (e.g., Liar Paradox, Russell's Paradox, Curry's Paradox, etc.), Various authors, (*e.g.,* [Feferman 1984a, Restall 2006]) have raised questions about how to do it.

The Tarskian framework of stratifying theories into a hierarchy of metatheories in which the semantics of each theory is formalized in its metatheory [Tarski and Vaught 1957] is currently standard. However, according to Feferman [1984a]:

> "*...natural language abounds with directly or indirectly self-referential yet apparently harmless expressions—all of which are excluded from the Tarskian framework."*

Self-referential propositions about cases, documentation, and code are common in large software systems. These propositions are excluded by the Tarskian framework substantially limiting its application to Software Engineering. To overcome such limitations, Direct Logic[3] was developed as an unstratified strongly paraconsistent reflective inference systems with the following goals [Hewitt 2008]:

- *Provide a foundation for strongly paraconsistent theories in Software Engineering.*
- *Formalize a notion of "direct" inference for strongly paraconsistent theories.*
- *Support all "natural" deductive inference [Fitch 1952; Gentzen 1935] in strongly paraconsistent theories with the exception of general Proof by Contradiction and Disjunction Introduction.*[4]
- *Support mutual reflection among code, documentation, and use cases of large software systems.*
- *Provide increased safety in reasoning about large software systems using strongly paraconsistent theories.*

In the new system, reflection was restricted to propositions which are *Admissible*.[5] In this way the classical paradoxes of reflection were blocked, *i.e.,* the Liar, Russell, Curry, Kleene-Rosser, *etc.*

To demonstrate the power of Direct Logic, a generalization of the incompleteness theorem was proved paraconsistently without using the assumption of consistency on which Godel/Rosser had relied for their proofs. Then there was a surprising new development: since it turns out that the Gödelian paradoxical proposition $\mathsf{Paradox}_{\mathcal{T}}$ is self-provable (*i.e.* $\vdash_{\mathcal{T}} \mathsf{Paradox}_{\mathcal{T}}$), it follows that every reflective strongly paraconsistent theory in Direct Logic is inconsistent!

According to Hewitt [2008]:

> "This means that the formal concept of TRUTH as developed by Tarski, *et. al.* is out the window. At first, TRUTH may seem like a desirable property for propositions in theories for large software systems. However, because a paraconsistent reflective theory $\mathcal{T}$ is necessarily inconsistent about $\vdash_{\mathcal{T}} \mathsf{Paradox}_{\mathcal{T}}$, it is impossible to consistently assign truth values to propositions of $\mathcal{T}$. In particular it is impossible to consistently assign a truth value to the proposition $\vdash_{\mathcal{T}} \mathsf{Paradox}_{\mathcal{T}}$. If the proposition is assigned the value TRUE, then (by the rules for truth values) it must also be assigned FALSE and vice versa. It is not obvious what (if anything) is wrong or how to fix it."

Of course this is contrary to the traditional view of Tarski. *E.g.*,

> *I believe everybody agrees that one of the reasons which may compel us to reject an empirical theory is the proof of its inconsistency: a theory becomes untenable if we succeeded in deriving from it two contradictory sentences . . . . It seems to me that the real reason of our attitude is...: We know (if only intuitively) that an inconsistent theory must contain false sentences.* [Tarski 1944]

On the other hand, Frege [1915] suggested that, in a logically perfect language, the word 'true' would not appear! According to McGee [2006], he argued that "*when we say that it is true that seawater is salty, we don't add anything to what we say when we say simply that seawater is salty, so the notion of truth, in spite of being the central notion of* [classical] *logic, is a singularly ineffectual notion. It is surprising that we would have occasion to use such an impotent notion, nevermind that we would regard it as valuable and important*."

Why did Gödel and the logicians who followed him not go in this direction? Feferman [2006b] remarked on "*the*

[3] Direct Logic is distinct from the Direct Predicate Calculus [Ketonen and Weyhrauch 1984].

[4] In this respect, Direct Logic differs from Quasi-Classical Logic [Besnard and Hunter 1995] for applications in information systems, which does include Disjunction Introduction.

[5] A proposition Ψ is Admissible for a theory $\mathcal{T}$ if and only if

$$(\neg\Psi) \vdash_{\mathcal{T}} (\vdash_{\mathcal{T}} \neg\Psi)$$

shadow of Hilbert that loomed over Gödel from the beginning to the end of his career." Also Feferman [2006a] conjectured that "*Gödel simply found it galling all through his life that he never received the recognition from Hilbert that he deserved*." Furthermore, Feferman maintained that "*the challenge remained well into his last decade for Gödel to demonstrate decisively, if possible, why it is necessary to go beyond Hilbert's finitism in order to prosecute the constructive consistency program*." Indeed Gödel saw his task as being "*to find a consistency proof for arithmetic based on constructively evident though abstract principles*" [Dowson 1997 pg. 263].

Also Gödel was a committed Platonist, which has an interesting bearing on the issue of the status of reflection. Gödel invented arithmetization to encode abstract mathematical propositions as integers. Direct Logic provides a similar way to easily formalize and paraconsistently prove Gödel's argument. But it is not clear that Direct Logic is fully compatible with Gödel's Platonism.

With an argument just a step away from inconsistency, Gödel (with his abundance of caution [Feferman 1984b, Dawson 1997]) could not conceive going in that direction. In fact, you could argue that he set up his whole hierarchical framework of metatheories and object theories to *avoid* inconsistency. A Platonist of his kind could argue that Direct Logic is a mistaken formalism because, in Direct Logic, all strongly paraconsistent reflective theories are necessarily inconsistent. In this view, the inconsistency simply proves the necessity of the hierarchy of metatheories and object theories.

However, reasoning about large software systems is made more difficult by attempting to develop such a hierarchy for the chock full of inconsistencies theories that use reflection for code, use cases, and documentation. In this context, it is not especially bothersome that theories of Direct Logic are inconsistent about $\vdash_T \text{Paradox}_T$

What went wrong:
When formalizing reasoning using reflection for large software systems, the reasoning process itself produced inconsistencies about certain specialized propositions that make assertions about their own unprovability.
What was done about it:
It was decided to ignore these inconsistencies because: 1. Since Direct Logic is strongly paraconsistent, the inconsistencies do no great harm. 2. The existence of these inconsistencies does not matter for large software systems, which are chock full of other inconsistencies that do matter.

The Future of Logic Programming

Keith Clark, Alain Colmerauer, Pat Hayes, Robert Kowalski, Alan Robinson, Philippe Roussel, *etc.* deserve a lot of credit for promoting the concept of Logic Programming and helping to build the Logic Programming community. And the traditions of this community should not be disrespected. At the same time, the term "logic programming" (like "functional programming") is highly descriptive and should mean something. Over the course of history, the term "functional programming" has grown more precise and technical as the field has matured. Logic Programming should be on a similar trajectory. Accordingly, "Logic Programming" should have a more precise characterization, *e.g.,* "*the logical deduction of computational steps".*

Today we know much more about the strengths and limitations of Logic Programming than in the late 1960's. For example, Logic Programming is not computationally universal and is strictly less general than the Procedural Embedding of Knowledge paradigm [Hewitt 1971]. Logic Programming and Functional Programming will both be very important for concurrent computation. Although neither one by itself (or even both together) can do the whole job, what can be done is extremely well suited to massive concurrency.

The following fundamental principles to extend Logic Programming based on the Scientific Community Metaphor were developed to extend Logic Programming [Kornfeld and Hewitt 1981, Hewitt 2008]:

- *Monotonicity:* Once something is published it cannot be withdrawn. Scientists publish their results so they are available to all. Published work is collected and indexed in libraries. Scientists who change their mind can publish later articles contradicting earlier ones. However, they are not allowed to go into the libraries and "erase" old publications.
- *Concurrency*: Scientists can work concurrently, overlapping in time and interacting with each other.
- *Commutativity*: Publications can be read regardless of whether they initiate new research or become relevant to ongoing research. Scientists who become interested in a scientific question typically make an effort to find out if the answer has already been published. In addition they attempt to keep abreast of further developments as they continue their work.
- *Sponsorship*: Sponsors provide resources for computation, i.e., processing, storage, and communications. Publication and subscription require sponsorship although sometimes costs can be offset by advertising.
- *Pluralism*: Publications include heterogeneous, overlapping and possibly conflicting information. There is no central arbiter of truth in scientific communities.
- *Skepticism*: Great effort is expended to test and validate current information and replace it with better information.
- *Provenance*: The provenance of information is carefully tracked and recorded.

A major research issue is investigating Strongly Paraconsistent Logic Programming using Direct Logic [Hewitt 2008a, 2008b] to deal with theories of practical domains that are chock full of inconsistencies, *e.g.,* domains associated with large software systems.

In this respect, the Deduction Theorem of logic plays a crucial role in relating logical implication to computation. The *Classical Deduction Theorem* can be stated as follows:

$$(\vdash (\Psi \rightarrow \Phi)) \longleftrightarrow (\Psi \vdash \Phi)$$

which states that $\Psi \rightarrow \Phi$ can be proved if and only if Φ can be inferred from Ψ. Thus procedures can search for a proof of the implication $\Psi \rightarrow \Phi$ by simply searching for a proof of Φ from Ψ. ***However, the Classical Deduction Theorem is not valid for the strongly paraconsistent theories of Direct Logic.***

Consequently for Direct Logic, the *Two-Way Deduction Theorem* [Hewitt 2008c] was developed taking the following form:

$$(\vdash_T (\Psi \rightarrow \Phi)) \longleftrightarrow ((\Psi \vdash_T \Phi) \wedge (\neg\Phi \vdash_T \neg\Psi))$$

which states that $\Psi \rightarrow \Phi$ can be proved in a strongly paraconsistent theory T if and only if Φ can be inferred from Ψ *and* $\neg\Psi$ can be inferred from $\neg\Phi$**.** In this way, the Two-Way Deduction Theorem enables procedures to search for strongly paraconsistent proofs of implications in Direct Logic.

What went wrong:

1. Pure Logic Programming proved to be too restrictive to handle the information processing for open systems.
2. The Classical Deduction Theorem (a mainstay principle of Logic Programming) was found not to be valid for the strongly paraconsistent theories of Direct Logic.

What was done about it:

1. Less restrictive principles were developed based on the Scientific Community Metaphor that generalized principles of Logic Programming.
2. A replacement for the classical Deduction Theorem (the Two-way Deduction Theorem) was developed which facilitates Logic Programming for strongly paraconsistent theories in Direct Logic.

Acknowledgements

Elihu M. Gerson and Dan Shapiro made extensive suggestions for improving this paper. The discussion of Conniver was improved by comments of Scott Fahlman. Erik Sandewall and Richard Waldinger provided suggestions on how to characterize Logic Programming that led to substantial improvements. Jeremy Forth made helpful comments and suggested including the section on the future of Logic Programming. The referees made many helpful suggestions.

References

Philippe Besnard and Anthony Hunter. *Quasi-classical Logic: Non-trivializable classical reasoning from inconsistent information* Symbolic and Quantitative Approaches to Reasoning and Uncertainty 1995.

Alonzo Church *A Set of postulates for the foundation of logic* Annals of Mathematics. Vol. 33, 1932. Vol. 34, 1933.

Eugene Charniak *Toward a Model of Children's Story Comprehension* MIT AI TR-266. December 1972.

Alain Colmerauer and Philippe Roussel. *The birth of Prolog* History of Programming Languages ACM Press. 1996

Ole-Johan Dahl and Kristen Nygaard. *Class and subclass declarations* IFIP TC2 Conference on Simulation Programming Languages. May 1967.

Julian Davies. *Popler 1.6 Reference Manual* University of Edinburgh, TPU Report No. 1, May 1973.

John Dawson *Logical Dilemmas. The Life and Work of Kurt Gödel* AK Peters. 1997.

John Dawson. *What Hath Gödel Wrought?* Synthese. Jan. 1998.

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

Solomon Feferman (1984b) *Kurt Gödel: Conviction and Caution* Philosophia Naturalis Vol. 21.

Solomon Feferman (1984a) *Toward Useful Type-Free Theories, I* in Recent Essays on Truth and the Liar Paradox. Ed. Robert Martin (1991) Claraendon Press.

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

Kazuhiro Fuchi, Robert Kowalski, Kazunori Ueda, Ken Kahn, Takashi Chikayama, and Evan Tick. *Launching the new era* CACM. 1993.

Kurt Gödel (1931) On formally undecidable propositions of Principia Mathematica Monatshefte für Mathematik und Physik, Vol. 38. (translated in A Source Book in Mathematical Logic, 1879-1931. Harvard Univ. Press. 1967) (translated by Bernard Meltzer. Basic Books. 1962.) http://home.ddc.net/ygg/etext/godel/

Cordell Green. *Application of Theorem Proving to Problem Solving* IJCAI'69.

Pat Hayes *Computation and Deduction* Mathematical Foundations of Computer Science: Proceedings of Symposium and Summer School, Štrbské Pleso, High Tatras, Czechoslovakia, September 3-8, 1973.

Pat Hayes *Some Problems and Non-Problems in Representation Theory* AISB. Sussex. July, 1974.

Carl Hewitt *PLANNER: A Language for Proving Theorems in Robots* IJCAI'69.

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

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

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

Carl Hewitt *The repeated demise of logic programming and why it will be reincarnated* What Went Wrong and Why: Lessons from AI Research and Applications. Technical Report SS-06-08. AAAI Press. March 2006.

Carl Hewitt (2008a) *Large-scale Organizational Computing requires Unstratified Reflection and Strong Paraconsistency* Coordination, Organizations, Institutions, and Norms in Agent Systems III. Jaime Sichman, Pablo Noriega, Julian Padget and Sascha Ossowski (ed.). Springer-Verlag. http://organizational.carlhewitt.info/

Carl Hewitt (2008b). *The downfall of mental agents in the implementation of large software systems* What went wrong. AAAI Magazine. Summer 2008.

Carl Hewitt (2008c) *Common sense for concurrency and strong paraconsistency using unstratified inference and reflection* Submitted for publication. *http://commonsense.carlhewitt.info/*

Daniel Ingalls. *The Evolution of the Smalltalk Virtual Machine* Smalltalk-80: Bits of History, Words of Advice. Addison Wesley. 1983.

Jussi Ketonen and Richard Weyhrauch. *A decidable fragment of Predicate Calculus* Theoretical Computer Science. 1984.

Stephen Kleene and John Barkley Rosser *The inconsistency of certain formal logics* Annals of Mathematics Vol. 36. 1935.

William Kornfeld and Carl Hewitt *The Scientific Community Metaphor* MIT AI Memo 641. January 1981.

Robert Kowalski *Algorithm = logic + control* CACM. July, 1979

Robert Kowalski (1988a) *The Early Years of Logic Programming* CACM. January 1988.

Robert Kowalski *History of the Association of Logic Programming October* 2004.

John Laird, Allen Newell, and Paul Rosenbloom *SOAR: an architecture for general intelligence* Artificial Intelligence. Vol. 33. No 1. 1987.

Peter Landin. *A Generalization of Jumps and Labels* UNIVAC Systems Programming Research Report. August 1965. (Reprinted in *Higher Order and Symbolic Computation.* 1998)

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

John McCarthy, Paul Abrahams, Daniel Edwards, Timothy Hart, and Michael Levin. *Lisp 1.5 Programmer's Manual* MIT Computation Center and Research Laboratory of Electronics. 1962.

L. Thorne McCarty. *Reflections on TAXMAN: An Experiment on Artificial Intelligence and Legal Reasoning* Harvard Law Review. Vol. 90, No. 5, March 1977.

Drew McDermott and Gerry Sussman *The Conniver Reference Manual* MIT AI Memo 259A. January 1974.

Drew McDermott *The Prolog Phenomenon* ACM SIGART Bulletin. Issue 72. July, 1980.

Marvin Minsky (ed.) *Semantic Information Processing* MIT Press. 1968.

Allen Newell and Herbert Simon *The logic theory machine: A complex information processing system* IRE Trans. Information Theory IT-2:61-79. 1956.

George Polya (1957) *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving* Combined Edition Wiley. 1981.

Greg Restall (2006). *Curry's Revenge: the costs of non-classical solutions to the paradoxes of self-reference* (to appear in The Revenge of the Liar ed. J.C. Beall. Oxford University Press. 2007) July 12, 2006. *http://consequently.org/papers/costing.pdf*

John Alan Robinson *A Machine-Oriented Logic Based on the Resolution Principle.* CACM. 1965.

John Barkley Rosser. *Extensions of Some Theorems of Gödel and Church* Journal of Symbolic. Logic. 1(3) 1936.

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

Eric Sandewall. *From Systems to Logic in the Early Development of Nonmonotonic Reasoning* CAISOR. July, 2006.

Gary Saxonhouse *What's All This about Japanese Technology Policy?* Cato Institute. August 17, 2001 *http://www.cato.org/pubs/regulation/reg16n4c.html*

Marek Sergot. *Bob Kowalski: A Portrait* Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski, Part I Springer. 2004.

Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages* Oxford Programming Research Group Technical Monograph. PRG-6. 1971.

Jon Sigurdson *Industry and state partnership: The historical role of the engineering research associations in Japan* 1986 *http://findarticles.com/p/articles/mi_qa3913/is_199812/ai_n8814325/print*

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

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

Alfred Tarski (1944) *The semantic conception of truth and the foundations of semantics* Philosophy and Phenomenological Research 4 (Reprinted in Readings in Philosophical Analysis, Appleton-1944)

Alfred Tarski and Robert Vaught (1957). "Arithmetical extensions of relational systems" *Compositio Mathematica* 13.

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