Inconsistency Robustness for Logic Programs
Carl Hewitt

To cite this version:
Carl Hewitt. Inconsistency Robustness for Logic Programs. Inconsistency Robustness, 2015, 978-1-84890-159. hal-01148496v6

HAL Id: hal-01148496
https://hal.archives-ouvertes.fr/hal-01148496v6
Submitted on 27 Apr 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright
Inconsistency Robustness for Logic Programs

Carl Hewitt

This article is dedicated to Alonzo Church and Stanisław Jaśkowski

Abstract
This article explores the role of Inconsistency Robustness in the history and theory of Logic Programs. Inconsistency Robustness has been a continually recurring issue in Logic Programs from the beginning including Church's system developed in the early 1930s based on partial functions (defined in the lambda calculus) that he thought would allow development of a general logic without the kind of paradoxes that had plagued earlier efforts by Frege, etc.¹

Planner [Hewitt 1969, 1971] was a kind of hybrid between the procedural and logical paradigms in that it featured a procedural embedding of logical sentences in that an implication of the form (p implies q) can be procedurally embedded in the following ways:

- **Forward chaining**
  - When asserted $p$, **Assert** $q$
  - When asserted $\neg q$, **Assert** $\neg p$

- **Backward chaining**
  - When goal $q$, **SetGoal** $p$
  - When goal $\neg p$, **SetGoal** $\neg q$

Developments by different research groups in the fall of 1972 gave rise to a controversy over Logic Programs that persists to this day in the form of following alternatives:

1. **Procedural Interpretation:** Logic Program using procedural interpretation of logic-clause syntax for a program¹ [Kowalski 2014]

2. **Procedural Embedding:** Logic Program in which each computational step² is logically inferred³

¹ a Logic Program is written as $\Psi = (\Phi_1 \land ... \land \Phi_n)$, which is logically equivalent to the disjunctive clause $\Psi \lor \neg \Phi_1 \lor ... \lor \neg \Phi_n$, where $\Psi$ and each of the $\Phi_i$ is either $P[t_1, ..., t_m]$ or $\neg P[t_1, ..., t_m]$ for some atomic predicate $P$ and terms $t_j$.

² e.g., using the Actor Model of computation

³ e.g., using Direct Logic
This article argues for the second alternative based on the following considerations:

- Programs in logic-clause syntax are a special case of the second alternative because each computational step of a program in logic-clause syntax is logically inferred as backward chaining or forward chaining.
- Reducing propositions to logic-clause syntax can obscure their natural structure.
- Procedural interpretation of logic-clause syntax can obscure the natural structure of derivations.

Logic-clause syntax is far too limited to be of use for general Logic Programs.

Kowalski advocates a bold thesis: “Looking back on our early discoveries, I value most the discovery that computation could be subsumed by deduction.”

However, mathematical logic cannot always infer computational steps because computational systems make use of arbitration for determining which message is processed next. Since reception orders are in general indeterminate, they cannot be inferred from prior information by mathematical logic alone. Therefore mathematical logic alone cannot in general implement computation. Logic Programs (like Functional Programs) are useful idioms even though they are not universal. For example Logic Programs can provide useful principles and methods for systems which are quasi-commutative and quasi-monotonic even though the systems themselves cannot be implemented using Logic Programs.

A fundamental principle of Inconsistency Robustness is to make contradictions explicit so that arguments for and against propositions can be formalized. In the Inconsistency Robustness paradigm, deriving contradictions is a progressive development and contradictions are not “game stoppers” that they would be using classical logic (in which reasoning about inconsistent information can make erroneous inferences). Contradictions can be helpful instead of being something to be “swept under the rug” by denying their existence or fruitlessly attempting complete elimination in systems of practice that are pervasively inconsistent.

A contradiction is manifest when both a proposition and its negation are asserted even if by different parties, e.g., New York Times said “Snowden is a whistleblower.”, but NSA said “Snowden is not a whistleblower.”

---

1 e.g., Natural Deduction using Direct Logic
This paper explores the role of Inconsistency Robustness in the development and theory of Logic Programs, which is an interesting test case involving pervasive inconsistency in an area in which, traditionally, inconsistency was not supposed to occur.

### Inconsistency Robustness

In this article, boxes like this one are used below to call out instances of inconsistency robustness.

---

### Uniform Proof Procedures based on Resolution

*By this it appears how necessary it is for nay man that aspires to true knowledge to examine the definitions of former authors; and either to correct them, where they are negligently set down, or to make them himself. For the errors of definitions multiply themselves, according as the reckoning proceeds, and lead men into absurdities, which at last they see, but cannot avoid, without reckoning anew from the beginning; in which lies the foundation of their errors...*

[Hobbes *Leviathan*, Chapter 4]³

An important limitation of classical logic for inconsistent theories is that it supports the principle that from an inconsistency anything and everything can be inferred, e.g. "The moon is made of green cheese."

For convenience, I have given the above principle the name IGOR⁶ for Inconsistency in Garbage Out Redux. IGOR can be formalized as follows in which a contradiction about a proposition Φ infers any proposition Ψ, i.e., Φ, ¬Φ ⊢ Ψ.

Of course, IGOR *cannot* be part of inconsistency-robust logic because it allows every proposition to be inferred from a contradiction.

---

³Using the symbol ⊢ to mean "infers in classical mathematical logic." The symbol was first published in [Frege 1879].
The IGOR principle of classical logic may not seem very intuitive! So why is it included in classical logic?

- **Classical Proof by Contradiction:** \((\Psi \models \Phi, \neg \Phi) \Rightarrow (\models \neg \Psi)\), which can be justified in classical logic on the grounds that if \(\Psi\) infers a contradiction in a consistent theory then \(\Psi\) must be false. In an inconsistent theory, Classical Proof by Contradiction leads to explosion by the following proof in classical logic by a which a contradiction about \(P\) infers any proposition \(\Theta\):

  \[ P, \neg P \models \neg \Theta. \quad \therefore P, \neg P \models \neg \Theta \]

- **Classical Contrapositive for Inference:** \((\Psi \models \Phi) \Rightarrow (\neg \Phi \models \neg \Psi)\), which can be justified in classical logic on the grounds that if \(\Psi \models \Phi\), then if \(\Phi\) is false then \(\Psi\) must be false. In an inconsistent theory, Classical Contrapositive for Inference leads to explosion by the following proof in classical logic by a which a contradiction about \(P\) infers any proposition \(\Theta\) by the following proof:

  Since \(\models P, \neg \Theta \models P\) by monotonicity. Therefore \(\models \neg P \models \neg \Theta\) by Classical Contrapositive for Inference. Consequently \(P, \neg P \models \Theta\).

- **Classical Extraneous \(\lor\) Introduction:** \(\Psi \models (\Psi \lor \Phi)\), which in classical logic says that if \(\Psi\) is true then \(\Psi \lor \Phi\) is true regardless of whether \(\Phi\) is true.\(^7\) In an inconsistent theory, Extraneous \(\lor\) introduction leads to explosion via the following proof in classical logic in which a contraction about \(P\) infers any proposition \(\Theta\):

  \[ P, \neg P \models (P \lor \Theta), \neg P \models \Theta \]

- **Classical Excluded Middle:** \(\lor (\Psi \lor \neg \Psi)\), which in classical logic says that \(\Psi \lor \neg \Psi\) is true regardless of whether \(\Psi\) is true. Excluded Middle is the principle of Classical Logic that for every proposition \(X\) the following holds: \(\text{Excluded Middle}[X] \equiv \neg (X \land \neg X)\). However, Excluded Middle is not suitable for inconsistency-robust logic because it is equivalent\(^8\) to saying that there are no inconsistencies, \(i.e.,\) for every proposition \(X\),

\[ \text{Noncontradiction}[X] \equiv \neg (X \land \neg X) \]

Using propositional equivalences, note that

\[ \text{Excluded Middle}[\Phi \lor \Psi] \equiv (\Psi \lor \neg \Psi \lor \Phi) \land (\Phi \lor \neg \Phi \lor \Psi) \]

Consequently, \(\text{Excluded Middle}[\Phi \lor \Psi] \Rightarrow (\Psi \lor \neg \Psi \lor \Phi)\), which means that the principle of Excluded Middle implies \(\Psi \lor \neg \Psi \lor \Phi\) for all propositions \(\Psi\) and \(\Phi\). Thus the principle of Excluded Middle is not inconsistency robust because it implies every proposition \(\Phi\) can be proved\(^9\) given any contradiction \(\Psi\). [Kao 2011]

\(^7\) using propositional equivalences
Classical Logic is unsafe for use with potentially inconsistent information.\textsuperscript{10}

[Robinson 1965] developed a deduction method called resolution which was proposed as a uniform proof procedure using first-order logic for proving theorems which

\begin{quote}
Converted everything to clausal form and then used a method analogous to modus ponens to attempt to obtain a contradiction by adding the negation of the proposition to be proved.
\end{quote}

Resolution uniform proof procedures were used to generate some simple proofs [Wos, et. al. 1965; Green 1969; Waldinger and Lee 1969; Anderson and Bledsoe 1970; \textit{etc.}]. In the resolution uniform proof procedure theorem proving paradigm, the use of procedural knowledge was considered to be "cheating."\textsuperscript{11}

Kowalski (also see [Decker 2003, 2005, 2010]) advocates that inference for inconsistent information systems\textsuperscript{12} be performed using resolution theorem proving.\textsuperscript{13} Unfortunately, in the presence of inconsistent information, resolution theorem provers can prove propositions using invalid arguments.
For example, consider the following illustration:\(^1\)
A resolution theorem prover proved \(\Phi \lor \Psi\) and as a consequence it was added to the information system. Sometime afterward, another resolution theorem prover proved \(\Phi\) making use of the previously proved \(\Phi \lor \Psi\). Unfortunately, \(\Phi\) was proved by resolution theorem proving using an invalid argument because\(^{14}\)

- The first resolution theorem prover proved \(\Psi \lor \Phi\) because\(^{ii}\) \(\therefore \Psi\) where \(\Psi\) holding has no bearing\(^{vi}\) on whether \(\Phi\) holds.\(^v\)
- Using an invalid argument,\(^{vi}\) the second resolution theorem proved \(\Phi\) using the previously proved \(\Psi \lor \Phi\) because\(^{vii}\) \(\therefore \neg \Psi\).

The above illustration illustrates how resolution theorem proving can prove propositions\(^ix\) using invalid arguments.

For a large information base \(\Omega\) of theories of practice (e.g., a climate models, theories of the human brain, etc.), Kowalski advocates use of the resolution set of support strategy\(^{15}\) as follows: In proving a proposition \(\Phi\), the set of support strategy chooses resolutions involving resolving \(\neg \Phi\) with \(\Omega\) and resolving the resulting clauses with each other and with \(\neg \Phi\).\(^{16}\)

---

\(^1\) The illustration below is presented in general terms for simplicity of exposition because presenting a detailed example involves significant complexity with lots of irrelevant details. An important point is that resolution theorem provers rely on overlap of vocabulary in deciding on which propositions to resolve. Consequently, they can become entangled in irrelevancy when deriving a contradiction when two propositions (having no bearing on whether the other holds) share vocabulary.

\(^{ii}\) even though the resolution theorem prover could not prove \(\Psi\) in the time that it took to prove \(\Psi \lor \Phi\).

\(^{iii}\) unknown to the resolution theorem prover although \(\Psi\) and \(\Phi\) have some common vocabulary, which can lead a resolution theorem prover astray.

\(^{iv}\) Unfortunately, the following rule of classical logic is not inconsistency robust: \(\Psi \lor \Phi\). However, a resolution theorem prove must honor all of the rules of classical logic.

\(^{v}\) because the resolution theorem provers unknowingly used \(\therefore \Psi, \neg \Psi\) as the basis for the proof of \(\Phi\) although \(\Psi\) holding has no bearing on \(\Phi\) holding even though they have some common vocabulary.

\(^{vi}\) unknown to the resolution theorem prover.

\(^{vii}\) even though the resolution theorem prover could not prove \(\neg \Psi\) in the time that it took to prove \(\Phi\).

\(^{ix}\) for pervasively inconsistent information systems of practice, e.g., theories for climate change and the human brain.
Any proposition \( X \) can be proved\(^1\) by interacting with a resolution theorem prover\(^2\) from an information base containing propositions \( \theta \) and \( \neg \theta \)\(^iii\) as follows:

1) \( \theta \vdash \text{SetOfSupport}(\theta \lor X) \) because \( \theta, \neg(\theta \lor X) \vdash \text{ResolutionTheoremProver} \text{False} \)

2) \( \theta \lor X, \neg \theta \vdash \text{SetOfSupport} X \) because

\[ \theta \lor X, \neg \theta, \neg X \vdash \text{ResolutionTheoremProver} \text{False} \]

3) A resolution theorem prover using \textit{set of support} working on an information base containing both \( \theta \) and \( \neg \theta \) can be used to first prove \( \theta \lor X \) using 1) and then can be used to prove \( X \) using 2).

Direct Logic was developed to address limitations\(^17\) of classical logic as a foundation for Computer Science:

- Classical Direct Logic is intended to be thought to be consistent by an overwhelming consensus of working professional mathematicians. See [Hewitt 2013] for discussion of Classical Direct Logic.
- Inconsistency Robust Direct Logic is for pervasively inconsistent theories of practice, e.g., theories for climate modeling and for modeling the human brain. Classical Direct Logic can be freely used in theories of Inconsistency Robust Direct Logic. See [Hewitt 2010] for discussion of Inconsistency Robust Direct Logic.

Inconsistency Robust Direct Logic is the state of the art for possibly inconsistent information.\(^v\) However, enforcing the constraint that a resolution theorem prover use only the rules of Inconsistency Robust Direct Logic is inefficient as well as extraordinarily complicated and awkward. Consequently, resolution theorem provers are unsuitable for inconsistency robust reasoning.

---

\(^1\) using an invalid argument because it used inconsistent assumptions
\(^2\) using the set of support strategy
\(^iii\) a resolution theorem prover using set of support might not realize that \( \theta \) and \( \neg \theta \) are inconsistent because if they are not in the set of support, they will not be resolved and it might not be immediately apparent from their clausal forms that they are inconsistent.
\(^iv\) set of support is underlined
\(^v\) set of support is underlined
\(^v\) Many other paraconsistent logics have been proposed that are partially inconsistency robust.
In the late 1960’s, the two major paradigms for constructing semantics 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 including property lists that were updated imperatively. The logical paradigm was epitomized by uniform Resolution Theorem Provers [Robinson 1965].

Procedural Embedding redux
In the late 1960’s, the two major paradigms for constructing semantics 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 including property lists that were updated imperatively. The logical paradigm was epitomized by uniform Resolution Theorem Provers [Robinson 1965].

Planer

Inconsistency Robustness: Uniform Proof Procedures
Uniform proof procedures using resolution was intended be a general theorem proving paradigm. But it suffered immense inefficiency in practice. Changing an axiomatization to improve performance was considered to be “cheating.”

Inconsistency Robustness: Resolution
Using resolution as the only rule of inference is problematical because it can obscure the natural structure of propositions and the natural structure of derivations using Natural Deduction. Also, Resolution Theorem Proving can prove propositions using invalid arguments for systems of practice because of pervasive inconsistencies.

Inconsistency Robustness: Resolution
Using resolution as the only rule of inference is problematical because it can obscure the natural structure of propositions and the natural structure of derivations using Natural Deduction. Also, Resolution Theorem Proving can prove propositions using invalid arguments for systems of practice because of pervasive inconsistencies.

Inconsistency Robustness: Uniform Proof Procedures
Uniform proof procedures using resolution was intended be a general theorem proving paradigm. But it suffered immense inefficiency in practice. Changing an axiomatization to improve performance was considered to be “cheating.”

Planner [Hewitt 1969, 1971] was a kind of hybrid between the procedural and logical paradigms in that it featured a procedural embedding of logical sentences in that an implication of the form \( p \implies q \) can be procedurally embedding in the following ways:¹

Forward chaining
- **When asserted** \( p \), **Assert** \( q \)
- **When asserted** \( \neg q \), **Assert** \( \neg p \)

Backward chaining
- **When goal** \( q \), **SetGoal** \( p \)
- **When goal** \( \neg p \), **SetGoal** \( \neg q \)

¹ In modern notation [see appendix of this paper]:

Forward chaining
- **When** \( \vdash \psi \rightarrow \vdash \phi \)
- **When** \( \vdash \neg \phi \rightarrow \vdash \psi \)

Backward chaining
- **When** \( \models \phi \rightarrow \models \psi \)
- **When** \( \models \neg \psi \rightarrow \models \neg \phi \)
Planner was the first programming language based on using explicit assertions and goals processed using pattern-directed invocation. The development of Planner was inspired by the work of Jaśkowski [1934], Newell and Simon [1956], McCarthy [1958], McCarthy *et. al.* [1962], Minsky [1958], Polya [1954], and Popper [1935, 1963].

**Planner represented a rejection of the resolution uniform proof procedure paradigm in favor of Procedural Embedding making use of new program language constructs** for computation including using explicit assertions and goals that invoked programs.

A subset called Micro-Planner was implemented by Gerry Sussman, Eugene Charniak and Terry Winograd as an extension to Lisp primarily for pragmatic reasons since it saved memory space and processing time (both of which were scarce) by comparison with more general problem solving techniques, e.g., [Polya 1957]:

- Lisp was well suited to the implementation of a Micro-Planner interpreter.
- The full functionality of Lisp libraries were immediately available for use by Micro-Planner programs.
- The Lisp compiler could be used to compile Lisp programs used by Micro-Planner applications to make them smaller and run faster. (It was unnecessary to first implement a Micro-Planner compiler.)

Computers were expensive. They had only a single slow processor and their memories were very small by comparison with today. So Planner adopted some efficiency expedients including the following:

- Backtracking [Golomb and Baumert 1965] was adopted to economize on the use of time and storage by working on and storing only one possibility at a time in exploring alternatives. In several ways, backtracking proved unwieldy helping to fuel the great control structure debate. Hewitt investigated some preliminary alternatives in his thesis.

- A unique name assumption was adopted by assuming that different names referred to different objects, which saved space and time. For example names like Peking and Beijing were assumed to refer to different objects.

- A closed world assumption could be implemented by conditionally testing whether an attempt to prove a goal exhaustively failed. Later this capability was given the misleading name “negation as failure” because for a goal G it was possible to say: “if attempting to achieve G exhaustively fails then assert (Not G).*

---

1 Prolog later also adopted these same efficiency expedients.

2 satirized as “*the less that can proved, the more that can be assumed!*”

---

9
• Being a hybrid language, Micro Planner had two different syntaxes, variable binding mechanisms, etc. 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! However, variants of this syntax have persisted to the present day.

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 Artificial Intelligence.

**Inconsistency Robustness: Procedural Embedding**

Planner was designed as a program language for Procedural Embedding. However, efficiency expedients were made in its implementation that unfortunately resulted in inflexible problem solving strategies as well as awkward, limited reasoning capabilities.

Although Winograd made an impressive demo, the successors of Planner and SHRDLU were incapable of practically realizing Procedural Embedding because of limited hardware performance and lack of effective software frameworks and tooling. Because of decades of subsequent progress, it has become feasible to developed practical, principled systems for Procedural Embedding.
Logic Programs in ActorScript are a further development of Planner. For example, suppose there is a ground-complete predicate \texttt{Link[anode, anotherNode, aCost]} that is true exactly when there is a path from \texttt{anode} to \texttt{anotherNode} with \texttt{aCost}.

\begin{verbatim}
When ⊨ Path[anode, anode, aCost] →
  \texttt{aCost=0}  // when a goal is set for a cost between \texttt{anode} and itself
\end{verbatim}

The following goal-driven Logic Program works forward from \texttt{start} to find the cost to \texttt{finish}:

\begin{verbatim}
When ⊨ Path[start, finish, aCost] →
  \texttt{aCost=Minimum \{nextCost + remainingCost}
    | \texttt{Link[start, next\#start, nextCost],}
    \texttt{Path[next, finish, remainingCost]}}\texttt{▮}
  // a cost from \texttt{start} to \texttt{finish} is the minimum of the set of the sum of the
  // cost for the next node after \texttt{start} and
  // the cost from that node to \texttt{finish}
\end{verbatim}

The following goal-driven Logic Program works backward from \texttt{finish} to find the cost from \texttt{start}:

\begin{verbatim}
When ⊨ Path[start, finish, aCost] →
  \texttt{aCost=Minimum \{remainingCost + previousCost}
    | \texttt{Link[previous\#finish, finish, previousCost],}
    \texttt{Path[start, previous, remainingCost]}}\texttt{▮}
  // the cost from \texttt{start} to \texttt{finish} is the minimum of the set of the sum of the
  // cost for the previous node before \texttt{finish} and
  // the cost from \texttt{start} to that Node
\end{verbatim}

Note that the above Logic Programs work together concurrently providing information to each other.
**Control Structure Controversies**

Peter Landin introduced a powerful co-routine 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 program language [McDermott and Sussman 1972]. Hewitt and others were skeptical about hairy control structure. 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?)

The difficulties using backtracking in Planner and Conniver were useful in that they provoked further research into control structures for procedural embedding.

**Inconsistency Robustness: Control Structures**

There was there germ of a good idea (previously emphasized in Polya [1957] and “progressive deepening” [de Groot 1965]) 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 and communications operations for its domain. However, the ability to jump back into the middle of procedure invocations seemed awkward and confusing. Hairy Control Structure didn’t seem to be what was needed as the foundation to solve the difficulties in communication that were a root cause of the control structure difficulties.

**Control structures are patterns of passing messages**

In November 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, 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]).

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
grounded by the laws of physics so that it could be completely general in terms of control structure.\textsuperscript{21} It took some time to develop practical program languages for the Actor model.

Work on Planner was temporarily suspended in favor of intensive investigation of the Actor model.\textsuperscript{1}

\begin{quote}
\textbf{Inconsistency Robustness: Message Passing}

Planner aimed to extend what could be programmed using logical methods but did not take a stand about the theoretical limits of these methods. However, once the Actor model was invented in late 1972, it became clear that logical inference alone would not suffice for computation because the order of Actor message reception could not always be logically inferred. [Hewitt 1976] reported

\begin{quote}
... 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. (emphasis in original)
\end{quote}
\end{quote}

\textbf{Edinburgh Logic for Computable Functions}

Like Planner,\textsuperscript{22} Edinburgh Logic for Computable Functions [Milner 1972; Gordon, Milner, and Wadsworth 1979] was capable of both forward chaining as well as backward chaining. This was accomplished by a purely functional program operating on a special data type called “Theorem” to produce new theorems by forward and backward chaining. Sub-goaling strategies (called tactics) were represented as higher-order functions taking strategies as arguments and returning them as results with goal failure implemented using exceptions.

\begin{quote}
\textbf{Inconsistency Robustness: Logic for Computable Functions}

Edinburgh Logic for Computable Functions was a notable advance in that its logical soundness was guaranteed by the type system. However, its problem solving generality is limited since it was not concurrent because it was purely functional.
\end{quote}

\textsuperscript{1} Work on Logic Programs later resumed in the Scientific Community Model [see section below].
Procedural Embedding versus Procedural Interpretation of Logic-clause Syntax

At Edinburgh, Pat Hayes and Bob Kowalski collaborated on resolution theorem proving. Then Hayes visited Stanford where Bruce Baumgart published his *Micro-Planner Alternate Reference Manual* in April 1972. Hayes says that from the time that he learned about Micro-Planner it seemed obvious to him that it was based on controlled deduction.

When he returned to Edinburgh, he talked about his insight with anyone who would listen and gave internal seminars at two of the major departments at Edinburgh concerned with logic. In the third department, Hayes point seemed irrelevant because they were busy getting their hands on the latest “magic machinery” for controlling reasoning using Popler [Davies 1973], a derivative of Planner. Hayes wrote a joint paper with Bruce Anderson on “The Logicians Folly” against the resolution uniform proof procedure paradigm [Anderson and Hayes 1972].

Gerry Sussman and Seymour Papert visited Edinburgh spreading the news about Micro-Planner and SHRDLU casting doubt on the resolution uniform proof procedure approach that had been the mainstay of the Edinburgh Logicists. According to Maarten van Emden [2006]

*The run-up to the workshop* [Machine Intelligence 6 organized by Donald Michie in 1970] was enlivened by telegrams from Seymour Papert at MIT announcing on alternating days that he was (was not) coming to deliver his paper entitled "The Irrelevance of Resolution", a situation that caused Michie to mutter something about the relevance of irresolution. The upshot was that a student named Gerry Sussman appeared at the appointed time. It looked as if this was going to be his first talk outside MIT. His nervousness was compounded by the fact that he had been instructed to go into the very bastion of resolution theorem proving and tell the assembled experts how totally misguided they were in trying to get anything relevant to AI with their chosen approach.

I had only the vaguest idea what all this was about. For me theorem proving was one of the things that some people (including Kowalski) did, and I was there for the programming. If Bob and I had anything in common, it was search. Accordingly I skipped the historic Sussman lecture and arrived late for the talk scheduled to come after Sussman's. Instead, I found an unknown gentleman lecturing from a seat in the audience in, what I thought a very English voice. It turned out that a taxi from the airport had delivered Seymour Papert after all, just in time for the end of Sussman's lecture, which was now being re-done properly by the man himself.

The effect on the resolution people in Edinburgh of this frontal assault was traumatic. For nobody more so than for Bob Kowalski. Of course there was no shortage of counter objections, and the ad hoc creations of
MIT were not a pretty sight. But the occasion hit hard because there was a sense that these barbarians had a point.

The above developments generated tension among the Logicists at Edinburgh. These tensions were exacerbated when the UK Science Research Council commissioned Sir James Lighthill to write a report on the AI research situation. The resulting report [Lighthill 1973; McCarthy 1973] was highly critical although SHRDLU [Winograd 1971] was favorably mentioned. “Resolution theorem-proving was demoted from a hot topic to a relic of the misguided past. Bob [Kowalski] doggedly stuck to his faith in the potential of resolution theorem proving. He carefully studied Planner.” [Bruynooghe, Pereira, Siemann, and van Emden 2004]

van Emden [2006] recalled:

Kowalski's apparent research program narrowed to showing that the failings so far of resolution inference were not inherent in the basic mechanism. He took great pains to carefully study PLANNER and CONNIVER. And painful it was. One of the features of the MIT work was that it assumed the audience consisted of LISP programmers. For anybody outside this circle (Kowalski most definitely was not a LISP programmer), the flavour is repellent.

According to [Kowalski 2014]

Pat Hayes and I had been working in Edinburgh on a book [Hayes and Kowalski, 1971] about resolution theorem-proving, when he returned from a second visit to Stanford (after the first visit, during which he and John McCarthy wrote the famous situation calculus paper [McCarthy and Hayes, 1968]). He was greatly impressed by Planner, and wanted to rewrite the book to take Planner into account. I was not enthusiastic, and we spent many hours discussing and arguing about the relationship between Planner and resolution theorem proving. Eventually, we abandoned the book, unable to agree.

In the fall of 1972 at MIT, there was universal dissatisfaction with the adequacy of micro-Planner. Consequently, o further the development of Procedural Embedding, the Actor Model was invented, which provided a rigorous basis for defining both:

- **Functional Program**: Actors do not change
- **Logic Program**: Each computational step is logically inferred.

Hayes reported that he was astonished when Kowalski wrote back from Marseilles saying that he and Colmerauer had a revolutionary idea that Horn clauses could be interpreted as backward-chaining programs. Feeling that his ideas were being unfairly appropriated by Kowalski, Hayes complained to the head of their unit Bernard Meltzer and still feeling unsatisfied wrote a summary and exegesis of his ideas in a paper for the proceedings of a summer
school in Czechoslovakia with the idea of recording the priority of his ideas [Hayes 1973].

However, Kowalski felt that his work with Colmerauer bore little resemblance to anything that had been discussed previously in Edinburgh by Hayes claiming that Hayes’ ideas (and the paper that he published) were based on using equations for computation (in the spirit of the work in Aberdeen [Foster and Elcock 1969].

Kowalski [2008] recalled:

In the meanwhile, critics of the formal approach, based mainly at MIT, began to advocate procedural representations of knowledge, as superior to declarative, logic-based representations. This led to the development of the knowledge representation and problem-solving languages Planner and micro-Planner. Winograd’s PhD thesis (1971), using micro-Planner to implement a natural language dialogue for a simple blocks world, was a major milestone of this approach. Research in automated theorem-proving, mainly based on resolution, went into sharp decline.

The battlefield between the logic-based and procedural approaches moved briefly to Edinburgh during the summer of 1970 at one of the Machine Intelligence Workshops organized by Donald Michie (van Emde, 2006). At the workshop, Papert and Sussman from MIT gave talks vigorously attacking the use logic in AI, but did not present a paper for the proceedings. This created turmoil among researchers in Edinburgh working in resolution theorem-proving. However, I was not convinced that the procedural approach was so different from the SL resolution system I had been developing with Donald Kuehner (1971).

During the next couple of years, I tried to reimplement Winograd’s system in resolution logic and collaborated on this with Alain Colmerauer in Marseille. This led to the procedural interpretation of Horn clauses (Kowalski 1973/1974) and to Colmerauer’s development of the programming language Prolog.

In the fall of 1972, Prolog (an abbreviation for “PRogrammation en LOGique” (French for programming in logic)), was developed as a subset of micro-Planner that restricted programs to Horn-clause syntax using backward chaining and consequently had a simpler more uniform syntax than Planner. However, the restriction to Horn-clause syntax tremendously restricted the expressive power of Prolog.

1 Horn-clause syntax for a logic program is of the form $\Psi \leftarrow (\Phi_1 \land \ldots \land \Phi_n)$ which is logically equivalent to the disjunctive clause $\Psi \lor \neg \Phi_1 \lor \ldots \lor \neg \Phi_n$ where $\Psi$ and each of the $\Phi_i$s is $P[t_1, \ldots, t_n]$ for some atomic predicate $P$ and terms $t_i$. 

16
Like Planner, Prolog provided the following:

- An indexed data base of propositions (limited by Prolog to positive predicates with ground arguments) and pattern-direct backward-chaining procedures (limited by Prolog to Horn-clause syntax).
- The Unique Name Assumption, by means of which different names are assumed to refer to distinct entities, e.g., Peking and Beijing are assumed to be different in order to save space and time.
- The Closed World Assumption (available and used in practice in micro-Planner to save space and time although it was not strictly required by micro-Planner).

Prolog was fundamentally different in intent from Planner as follows:

- Planner was designed for **Procedural Embedding using explicit goals and assertions processed by pattern-directed procedures**. Correctness can be checked when an assertion is made that a proposition holds in a theory using a rule of inference. Theories do not limit the problem solving methods that can be used.¹
- Prolog was designed a **backward-chaining interpretation of Horn-clause syntax**. Consequently, a goal established by a Prolog backward-chaining program was intended to be a logical consequence of the propositional interpretation of the Prolog program syntax without any additional checking. A theory expressed in Prolog limits the problem solving method that can be used to backward chaining of Horn-clause syntax programs whose propositional content is the Horn clauses²⁵ of the theory.

A number of Logic Program features of Micro-Planner were omitted from Prolog including the following:²⁶

- Using explicit assertions processed by pattern-directed procedures (i.e., “forward chaining”)
- Logical negation, e.g., (not (human Socrates))
- Explicit goals and subgoals that are distinct from assertions²⁷

**In summary, Prolog was basically a subset** of Planner that restricted programs to a backward chaining interpretation of Horn clauses.

¹ *e.g.*, the full range of techniques in [Polya 1957] should be available.

² excepting that Prolog used unification instead of the pattern matching used in Planner. However, in practice unification is often partially turned off in Prolog programs (at the potential cost of incorrect results) because unification can be expensive.
According to [Kowalski 2014]:

“... it was widely believed that logic alone is inadequate for problem solving, and that some way of controlling the theorem-prover is needed for efficiency. Planner combined logic and control in a procedural representation that made it difficult to identify the logical component. Logic programs with SLD resolution also combine logic and control, but make it possible to read the same program both logically and procedurally. I later expressed this as Algorithm=Logic+Control (A=L+C) [Kowalski, 1979a], influenced by Pat Hayes' [1973] Computation=Controlled Deduction.

The most direct implication of the equation is that, given a fixed logical representation L, different algorithms can be obtained by applying different control strategies, i.e. A_1=L+C_1 and A_2=L+C_2. Pat Hayes [1973], in particular, argued that logic and control should be expressed in separate languages, with the logic component L providing a pure, declarative specification of the problem, and the control component C supplying the problem solving strategies needed for an efficient algorithm A. Moreover, he argued against the idea, expressed by A_1=L+C and A_2=L+C, of using a fixed control strategy C, as in Prolog, and formulating the logic L; the problem to obtain a desired algorithm A_i.”

[Hayes 1974] complained that procedural interpretation was “aliens semantics” for logical propositions that was analogous to “... throwing out the baby and keeping the bathwater.” In other words, Hayes was arguing for Procedural Embedding as in Planner, whereas Kowalski was arguing for fixed control structure as in resolution uniform proof procedures.

Furthermore, there are concerns about the adequacy of logic-clause syntax to express Logic Programs.

---

1 Logic-clause syntax for a logic program is of the form $\psi \leftarrow \Phi_1 \land \ldots \land \Phi_n$ [logically equivalent to the disjunctive clause $\psi \lor \neg \Phi_1 \lor \ldots \lor \neg \Phi_n$] where $\psi$ and each of the $\Phi_i$ is of the form $P[t_1, \ldots, t_\ell]$ or the form $\neg P[t_1, \ldots, t_\ell]$ for some atomic predicate $P$ and terms $t_i$. (Logic-clause syntax is a slight generalization of Horn-clause syntax in that it allows predicates to be negated.)
Scientific Community Model

Research on the Scientific Community Model [Kornfeld and Hewitt 1981, Kornfeld 1981] involved the development of a program language named Ether that invoked 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.

The Scientific Community Model builds on the philosophy, history and sociology of science. It was originally developed building on work in the philosophy of science by Karl Popper and Imre Lakatos. In particular, it initially made use of Lakatos' work on proofs and refutations. Subsequently development has been influenced by the work of Geof Bowker, Michel Callon, Paul Feyerabend, Elihu M. Gerson, Bruno Latour, John Law, Karl Popper, Susan Leigh Star, Anselm Strauss, and Lucy Suchman.

In particular Latour's Science in Action had great influence. In the book, Janus figures make paradoxical statements about scientific development. An important challenge for the Scientific Community Model is to reconcile these paradoxical statements.

Scientific research depends critically on monotonicity, concurrency, commutativity, and pluralism to propose, modify, support, and oppose scientific methods, practices, and theories. Scientific Communities systems

---

**Inconsistency Robustness: Expressiveness of Logic Programs**

Procedural Embedding (starting with Planner) allows the distinction between propositions and programs:

- Propositional information does not have to be reformulated using logic-clause syntax (which can obscure the natural structure of propositions).
- Logic Programs can be more expressive and powerful when not restricted to logic-clause syntax. For example, Logic Programs in ActorScript use Natural Deduction.
- Procedural Embedding can implement problem solving methods that cannot be implemented using Logic Programs (see this article on “Is Computation Subsumed by Deduction?”)
- Assertions have provenance that justify their inference in a theory.
have characteristics of monotonicity, concurrency, commutativity, pluralism, skepticism and provenance:

- **monotonicity**: Once something is published it cannot be undone. 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.
- **concurrency**: Scientists can work concurrently, overlapping in time and interacting with each other.
- **quasi-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.
- **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.
There are a number of controversies involved in the history of Logic Programs in which different researchers took contradictory positions that are addressed in following sections of this article including, “Is computation subsumed by deduction?” and “Did Prolog-style clause programs contribute to the failure of the Japanese Fifth Generation Project (ICOT)” and “What is a Logic Program?”

Is Computation Subsumed by Deduction?

“...a single formalism suffices for both logic and computation, and logic subsumes computation.”

[Kowalski 2014]
The challenge to the generality of Logic Programs as a foundation for computation was officially thrown in *The Challenge of Open Systems* [Hewitt 1985] to which [Kowalski 1988b] replied in *Logic-Based Open Systems*. This was followed up with [Hewitt and Agha 1988] in the context of the Japanese Fifth Generation Project (see section below). All of this was in opposition to Kowalski's thesis: “Looking back on our early discoveries, I value most the discovery that computation could be subsumed by deduction.”

In concrete terms, we cannot observe the internals of the mechanism by which the reception order of messages is determined. Attempting to do so affects the results and can even push the indeterminacy elsewhere. Instead of observing the internals of arbitration processes, we await outcomes. The reason that we await outcomes is that we have no alternative because of indeterminacy. Because of indeterminacy in the physical basis of computation, no kind of deductive mathematical logic can always infer which message will be received next and the resulting computational steps. Consequently, Logic Programs can make inferences about computation but not in general implement computation. Nevertheless, Logic Programs (like Functional Programs) can be a useful idiom.

---

**Inconsistency Robustness: Universality of Deduction**

Contrary to Kowalski, computation in general cannot be subsumed by deduction. Mathematical models of computation do not determine particular computations as follows: Arbiters can be used in the implementation of the reception order of messages, which are subject to indeterminacy. Since reception orders are in general indeterminate, they cannot be inferred from prior information by mathematical logic alone. Therefore mathematical logic alone cannot implement computation in open systems.
Actor systems can perform computations that are impossible by mathematical deduction as illustrated by the following nondeterministic Actor system \( i \) that can compute an integer of unbounded size implemented as follows using ActorScript™:

\[
\text{Unbounded,}_i: \text{Integer} \equiv \\
\text{let } a\text{Counter }\leftarrow \text{Counter}[,]. \quad \text{// let aCounter be a new Counter} \\
\Box a\text{Counter,}_i\text{.go}[,]. \quad \text{// send aCounter a go message and concurrently} \\
\Box a\text{Counter,}_i\text{.stop}[,] \\
\quad \text{// return the value of sending aCounter a stop message}
\]

\text{Actor Counter[,]} \\
\begin{align*}
\text{count} & := 0 \quad \text{// the variable count is initially 0} \\
\text{continue} & := \text{True}. \quad \text{// the variable continue is initially True} \\
\text{stop}[,]: \text{Integer} & \rightarrow \\
\text{count} & \quad \text{// return count} \\
\text{afterward continue} & := \text{false} \\
\text{// continue is false for the next message received} \\
\text{go}[,]: \text{Void} & \rightarrow \\
\text{continue} & \quad \text{// if continue is True,} \\
\text{Hole,}_i\text{.go}[,] & \quad \text{// send go[,] to this counter after} \\
\text{afterward Error! Bookmark not defined. count} & := \text{count} + 1 \\
\text{// incrementing count} \\
\text{False}: \text{Void} & \quad \text{// if continue is False, return Void}
\end{align*}

By the semantics of the Actor model of computation [Clinger 1981; Hewitt 2006], Unbounded,\(_i\) will result in computing an integer of unbounded size.

The procedure Unbounded above can be axiomatized as follows:

\[
\forall [n: \text{Integer}] \rightarrow \\
\exists [a\text{Request}: \text{Request}, \quad a\text{Integer}: \text{Integer}] \rightarrow \\
\text{Unbounded sent}_{a\text{Request}[,]} \Rightarrow \\
\text{Sent}_{\text{Response}_{a\text{Request}[,}}} \text{Returned}[a\text{Integer}] \land a\text{Integer} > n
\]

However, the above axiom does not compute any actual output! Instead the above axiom simply asserts the existence of unbounded outputs for Unbounded,\(_i\).

\( ^1 \) using the ActorScript programming language [Hewitt 2010a]
**Theorem.** The nondeterministic function defined by Unbounded (above) cannot be implemented by a nondeterministic Logic Program.  

**Proof.**

The task of a nondeterministic Logic Program $P$ is to start with an initial set of axioms and prove $\text{Output} = n$ for some numeral $n$. Now the set of proofs of $P$ starting from initial axioms will form a tree. The branching points will correspond to the nondeterministic choice points in the program and the choices as to which rules of inference to apply. Since there are always only finitely many alternatives at each choice point, the branching factor of the tree is always finite. Now König's lemma says that if every branch of a finitary tree is finite, then so is the tree itself. In the present case this means that if every proof of $P$ proves $\text{Output} = n$ for some numeral $n$, then there are only finitely many proofs. So if $P$ nondeterministically proves $\text{Output} = n$ for every numeral $n$, it must contain a nonterminating computation in which it does not prove $\text{Output} = n$ for some numeral $n$.

The following arguments support unbounded nondeterminism in the Actor model [Hewitt 1985, 2006]:

- 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 nondeterminism since mail can be stored on servers indefinitely before being delivered.

- Communication links to servers on the Internet can be out of service indefinitely.

---

1 the lambda calculus is a special case of Logic Programs
The following Logic Programs procedurally embed information about Unbounded:

When ⊢ aRequest anActor sent aMessage →
  // When asserted that anActor is sent aRequest with aMessage
  ⊢ aRequest anActor received aMessage  // assert that anActor received aRequest
When ⊢ aResponse Sent aResult →
  ⊢ aResponse Received aResult  // assert that aResult is received for aResponse
When ⊢ aRequest Unbounded received [] →
  // When asserted that Unbounded is sent []
  ⊢ Unbounded1[ aRequest ] Counter sent []
  ⊢ Response Unbounded1[ aRequest ] Received Returned[ aCounter : Counter ] →
    // Set a goal that aCounter is returned for the request Unbounded1[ aRequest ].
  ⊢ Unbounded2[ aRequest ] aCounter sent go [],
  ⊢ Unbounded3[ aRequest ] aCounter sent stop [],
  ⊢ Response Unbounded2[ aRequest ] Received Returned[ Void ] →
  ⊢ Response Unbounded3[ aRequest ] Received Returned[ anInteger : Integer ] →
    ⊢ Response aRequest Returned[ anInteger ]
When ⊢ aRequest Counter received [] →
  ⊢ Response aRequest Returned[ aCounter ],
  ⊢ Response aRequest currentCount anAccount = 0,
  ⊢ Response aRequest continue aCounter = True
When ⊢ aRequest aCounter: Counter received Stop [] →
  ⊢ Response aRequest Sent Returned[ currentCount aRequest ],
When ⊢ aRequest currentCount Counter = n →
  ⊢ Response aRequest currentCount aCounter = n,
  ⊢ Response aRequest continue aCounter = False
When ⊢ aRequest aCounter: Counter received go [] →
  ⊢ aRequest ¬ continue aCounter →  // Set a goal that continue aCounter = False
    ⊢ Response aRequest Sent Returned[ Void ],
  ⊢ Response aRequest Unchanged aRequest currentCount aCounter,
  ⊢ Response aRequest Unchanged aRequest continue aCounter
  ⊢ aRequest continue aCounter →  // Set a goal that continue aCounter = True
    ⊢ Counter1[ aRequest ] aCounter sent go [],
    // Assert that the cheese has been left at Counter1[ aRequest ]
    ⊢ aRequest currentCount aCounter = anInteger
    ⊢ Counter1[ aRequest ] currentCount aCounter = anInteger + 1,
  ⊢ Response Counter1[ aRequest ] Received Returned[ anInteger : Integer ] →
  ⊢ Response aRequest Sent Returned[ anInteger ]
Note that the above logic programs do not make use of global time.¹

Inconsistency Robustness: Modeling Change

Direct Logic can be used to model change in a way that is physically realizable as opposed to systems that make use of physically unrealizable global time:

1. **Global State Spaces**: a computation can be represented as a global state that determines all information about the computation. It can be nondeterministic as to which will be the next global state, e.g., in simulations where the global state can transition nondeterministically to the next state as a global clock advances in time, e.g., Simula [Dahl and Nygaard 1967].¹

2. **Actors**: a computation can be represented as a configuration. Information about a configuration can be indeterminate. For example, there can be arbiters that are meta-stable and messages in transit that will be delivered at some indefinite (unbounded) time.

The Japanese 5th Generation Project (ICOT)

Beginning in the 1970’s, Japan took the DRAM market (and consequently most of the integrated circuit industry) away from the previous US dominance. 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].

**Project Inception**

MITI hoped to repeat this victory by taking over the computer industry. 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.

**Trying to go all the way with Prolog-style clause programs**

ICOT tried to go all the way with Prolog-style clause programs. Kowalski later recalled “Having advocated LP [Logic Programs] as a unifying foundation for computing, I was delighted with the LP [Logic Program] focus of the FGCS [Fifth Generation Computer Systems] project.” [Fuchi, ¹ which is physically unrealizable
Kowalski, Ueda, Kahn, Chikayama, and Tick 1993) By making Prolog-style clause programs (mainly being developed outside the US) the foundation, MITI hoped that the Japanese computer industry could leapfrog the US. “The [ICOT] project aimed to leapfrog over IBM, and to a new era of advanced knowledge processing applications.” [Sergot 2004]

Unfortunately, ICOT misjudged the importance of direct message passing, e.g., in the Actor Model [Hewitt, Bishop, and Steiger 1973] which had been developed in reaction to the limitations of Planner.

ICOT had to deal with the concurrency and consequently developed concurrent program languages based on Prolog-style clauses [Shapiro 1989] similar to the above Logic Program. However, it proved difficult to implement clause-procedure invocation in these languages as efficiently as procedures in object-oriented program 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 compiler technology originally developed for Simula far out-performed the ICOT compilers for Prolog-style clause languages developed by ICOT.

The clausal syntax used by ICOT was awkward because it only allowed relational syntax for procedure calls and consequently was not compositional requiring the use of multiple ports for communication.
Using clausal syntax, ICOT encountered difficulties dealing with concurrency, e.g., readers-writers concurrency. Concurrency control for readers and writers in a shared resource is a classic problem. The fundamental constraint is that readers are allowed to operate concurrently but a writer is not allowed to operate concurrently with other writers and readers.

The interface for the readers/writer guardian is the same as the interface for the shared resource:

\[\text{Interface ReadersWriter with read[Query]} \rightarrow \text{QueryAnswer},\]
\[\quad \text{write[Update]} \rightarrow \text{Void}\]

Cheese diagram for ReadersWriter implementations:

Note:
1. At most one activity is allowed to execute in the cheese.\(^1\)
2. The cheese has holes.\(^ii\)
3. The value of a variable\(^\text{iii}\) cannot change in the cheese.\(^iv\)

\(^{\text{i}}\) Cheese is yellow in the diagram  
\(^{\text{ii}}\) A hole is grey in the diagram  
\(^{\text{iii}}\) A variable is orange in the diagram  
\(^{\text{iv}}\) Of course, other external Actors can change.
Inconsistency Robustness:
Clausal Concurrent Programming Languages
The combination of efficient inheritance-based virtual procedure
invocation (pioneered in Simula) together with class libraries and browsers
(pioneered in Smalltalk) provided better tools than the slower pattern-
directed invocation of the FGCS Prolog-style clause programs.
Consequently, the ICOT program languages never took off and instead
object-oriented program languages like Java and JavaScript became the
mainstream.

Downfall
The technical managers at ICOT were aware of some of the pitfalls that had
tripped up previous Artificial Intelligence (AI) researchers. So they
deliberately avoided calling ICOT an AI Project. Instead they had the vision
of an integrated hardware/software system [Uchida and Fuchi 1992].
However, the Prolog-style clause-program paradigm turned not to be a
suitable foundation because of poor modularity and lack of efficiency by
comparison with direct message passing [Hewitt and Agha 1988]. Another
problem was that multi-processors found it difficult to compete in the
marketplace because at the time single processors were rapidly increasing in
speed and connections between multiple processors suffered long latencies.
The reliance on Prolog-style clause procedures was a principle
contributing cause to the failure of ICOT to achieve commercial
success.

Inconsistency Robustness: ICOT
MITI's Fifth Generation strategy backfired because Japanese companies
refused to productize ICOT hardware.
However, the architects of ICOT did get some things right:
- The project largely avoided the Mental Agent paradigm [Hewitt
  2009]
- The project correctly placed tremendous emphasis on research in
  concurrency and parallelism as an emerging computing paradigm.
What is a Logic Program?

“It would be like saying Prolog and SLD-Resolution is the only way to do Logic Programming. To some extent, the LP [“Logic Programming”] community’s insistence on clinging to this “exclusive method” has contributed to the relative disinterest in LP following its development in the 1980’s and 1990’s.” [Aït-Kaci 2009] (Emphasis added.)

Developments by different research groups in the fall of 1972 gave rise to a controversy over Logic Programs that persists to this day in the form of following alternatives:

1. Logic Programs using *procedural interpretation of logic-clause syntax for procedures* [Kowalski 2014]¹
2. Logic Programs in which *each computational step* ii is logically inferred iii

This article argues for the second conception based on the following considerations:

- Logic-clause syntax is inadequate to express Logic Programs, e.g., logic-clause syntax lacks the ability to construct sets using \{x | \models \Phi(x)\}, where \Phi is a ground-complete predicate.
- Programs in logic-clause syntax are a special case of the second alternative because each computational step of a program in logic-clause syntax is logically inferred by forward chaining and backward chaining.
- Reducing propositions to logic-clause syntax can obscure their natural structure.
- Procedural interpretation of logic-clause syntax for procedures can obscure the natural structure of proofs.iv

Logic Programs need to deal with both of the following:

- *Mathematical Theories* that are thought to be consistent by the overwhelming consensus of professional working mathematicians, *e.g.* Hilbert spaces, homology theory, etc.

---

¹ Logic-clause syntax for a logic program is \( \Psi \equiv (\Phi_1 \land \ldots \land \Phi_n) \) where \( \Psi \) and each of the \( \Phi_i \) is either \( P[t_1, \ldots, t_m] \) or \( \neg P[t_1, \ldots, t_n] \) for some atomic predicate \( P \) and terms \( t_i \).
² in the Actor Model of computation
³ *e.g.* using Direct Logic
⁴ *e.g.* ActorScript Logic Programs using Inconsistency Robust Natural Deduction in Direct Logic
• Theories of Practice that are pervasively inconsistent, e.g., theories of climate change and the human brain. Mathematical theories are often freely used within theories of practice.

Going beyond the limitations of logic-clause syntax, there is a core subset of Logic Program constructs that are applicable to both Classical Direct Logic and Inconsistency Robust Direct Logic.¹

Inconsistency Robustness: Characterization of Logic Programs

Developments by different research groups in the fall of 1972 gave rise to a controversy over Logic Programs that persists to this day in the form of following alternatives:

1. Each computational step (e.g. as defined in the Actor Model) of a Logic Program is logically inferred (e.g. in Direct Logic) with explicit assertions and goals processed using pattern-directed invocation.

2. A Logic Program is expressed using logic-clause syntax that is interpreted as both a backward-chaining program and a forward-chaining program.

¹ See appendix of this article.
Logic Programs versus Kowalski’s “Logic Programming”

Kowalski has for a long time advocated “Logic Programming” originally characterized as follows:37

The driving force behind logic programming is the idea that a single formalism suffices for both logic and computation, and that logic subsumes computation. ...

Logic programming aims ... to unify different areas of computing by exploiting the greater generality of logic [over computational models']. It does so by building upon and extending38 one of the simplest, yet most powerful logics imaginable, namely the logic of Horn clauses.

Kowalski’s Logic Programming is a scientific research programme in the sense of [Lakatos 1980]. According to [Kowalski 2006]:

Admittedly, I have been messianic in my advocacy of Logic, and I make no apologies for it. Pushing Logic as hard as I could has been my way of trying to discover its limits.

Kowalski’s version of Logic Programing has the following limitations:

- based on the following mistaken assumptions:
  - “that logic subsumes computation”39
  - that Horn clauses are the basis for a powerful logic40
- lacks inconsistency-robust inference because it is based on classical logic making use of resolution theorem proving41
- lacks the precision of a well-defined scientific concept.42
- lacks mathematical rigor because [Kowalski 2014] includes less well-defined capabilities such as “abduction”

---

1 e.g., Turing Machines, relational model of data base queries, the Actor Model, etc.

2 using logic-clause syntax for a Logic Program, e.g., $\Psi \subseteq (\Phi_1 \land ... \land \Phi_n)$ where $\Psi$ and each of the $\Phi_i$ is either $P[t_1, ..., t_m]$ or $\neg P[t_1, ..., t_n]$ for some atomic predicate $P$ and terms $t_i$.

3 Contrary to Kowalski, there are computations that cannot be performed using logical inference. See discussion earlier in this article.

4 Contrary to Kowalski, the logic-clause generalization of Horn-clause syntax is inadequate for expressing Logic Programs as explained earlier in this article. Also, Horn clauses do not form the basis for the most powerful logic because they are based on first-order logic, which has unintended models of axioms. For example, first-order logic cannot even characterize the integers up to isomorphism. See discussion in article 1-2 in this volume.

5 As explained earlier in this article, resolution theorem provers can make invalid inferences using inconsistent axioms.

6 which could be fixed by defining the concept as “programming using Logic Programs” for a well-defined definition of a Logic Program.
### Overview

<table>
<thead>
<tr>
<th>Contradiction</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kowalski advocates using resolution theorem proving to make inferences about inconsistent information systems. Unfortunately, using resolution theorem provers allows inferring every proposition from an inconsistent information system.</td>
<td>Using Direct Logic, inconsistent-robust inference allows inference about inconsistent information systems without enabling inference of every sentence.</td>
</tr>
<tr>
<td>Planner was intended to be a general purpose programming language for the procedural embedding of knowledge. Partly as a result of limitations of contemporary computers, pragmatic decisions were made in the implementation of Planner that limited its generality.</td>
<td>Programming languages (like ActorScript) have been developed that incorporate general Logic Programs.</td>
</tr>
<tr>
<td>Kowalski, et. al. advocated that Horn-clause syntax for programs be used as the foundation of Logic Programs. But Horn-clause syntax for programs (and slightly more general logic-clause syntax) lack the generality and modularity needed for Logic Programs.</td>
<td>Programming languages (like ActorScript) are not restricted to logic-clause syntax for Logic Programs.</td>
</tr>
<tr>
<td>The Japanese Fifth Generation Project (ICOT) attempted to create a new computer architecture based on logic-clause programs.</td>
<td>ICOT failed to gain commercial traction although it was an important research project.</td>
</tr>
<tr>
<td>Kowalski claims that mathematical logical deduction subsumes computation. However, there are computations that cannot be implemented using deduction (i.e. Logic Programs) and there are important applications that cannot be implemented using only Logic Programs.</td>
<td>Actor programming languages (like ActorScript) use direct message passing to Actors and are not restricted to just Logic Programs.</td>
</tr>
<tr>
<td>Previously, change was modeled in Logic Programs using a global time that is physically meaningless.</td>
<td>The Actor model enables change in computation to be modeled by causal partial orders of message-passing events.</td>
</tr>
</tbody>
</table>

The above examples are intended to be case studies in Inconsistency Robustness in which information is formalized, contradictions are derived using Inconsistency Robust reasoning, and arguments are formalized for and
against contradictory propositions. A challenge for the future is to automate
the reasoning involved in these case studies.

**Conclusion**

*Max Planck, surveying his own career in his Scientific Autobiography*
*Planck 1949*, sadly remarked that ‘a new scientific truth does not
triumph by convincing its opponents and making them see the light, but
rather because its opponents eventually die, and a new generation grows
up that is familiar with it.’
*Kuhn 1962*

A fundamental principle of Inconsistency Robustness is to make
contradictions explicit so that arguments for and against propositions can
be formalized. This paper has explored the role of Inconsistency Robustness in
the history and theory of Logic Programs by making contradictions explicit
and using them to explicate arguments. The development of Logic Programs
has been shown to be a productive area for applying principles of
Inconsistency Robustness.

The examples presented in this paper are intended to be case studies in
Inconsistency Robustness in which information is formalized, contradictions
are derived using Inconsistency Robust reasoning, and arguments are
formalized for and against contradictory propositions with developments that
included:

- Some arguments were dropped.
- Some arguments triumped.
- Arguments were combined and improved.
- New arguments were developed

A challenge for the future is to automate the reasoning involved in these case
studies.38

**Acknowledgements**

Scott Fahlman made suggestions and comments that materially improved an
erlier version of this article. Richard Waldinger made an important
suggestion for improving the definition of Logic Programs and otherwise
greatly helped to improve the paper. Jeremy Forth made helpful suggestions.
Bob Kowalski and Pat Hayes made important contributions and suggestions
that greatly improved an earlier version article. I especially thank Bob for
sharing his view of his recent research. Maarten van Emde corrected some
typos. Dale Schumacher provided useful suggestions in the section on ICOT.
In 2013, this article was substantially revised to address issues of Inconsistency Robustness. Bob Kowalski provided useful comments on the abstract, which contributed to better delineation similarities and differences in our views. Alan Bundy pointed out many crucial places where the presentation needed improvement. Alan Karp pointed out a major bug in the Logic Program in the end notes. Ken Kahn provided helpful comments on clausal concurrent programming languages. Extensive discussions with Eric Kao helped clarify the discussion of the work of Kowalski. Harold Boley provided helpful suggestions on the presentation.

Of course, any remaining errors are entirely my own.

**Bibliography**


Hassan Aït-Kaci. *Children's Magic Won't Deliver the Semantic Web* CACM. March 2009


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


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


Pat Hayes. *Some Problems and Non-Problems in Representation Theory* AISB’74.
Carl Hewitt *PLANNER: A Language for Proving Theorems in Robots* IJCAI’69.
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’73.
Carl Hewitt (2006b) *What is Commitment? Physical, Organizational, and Social* COIN@AAMAS’06.


Alfred Horn. *On sentences which are true of direct unions of algebras* Journal of Symbolic Logic. March 1951.


Ken Kahn and Vijay Saraswat. *Actors as a Special Case of Concurrent Constraint Programming* ECOOP/OOPSLA. October 1990.


William Kornfeld (1981a) *The Use of Parallelism to Implement a Heuristic Search* IJCAT’81.


Robert Kowalski *Predicate Logic as Programming Language* IFIP’74.


Robert Kowalski *Algorithm = Logic + Control* CACM. July 1979.


Max Planck Scientific Autobiography and Other Papers 1949.


Jon Sigurdson *Industry and state partnership: The historical role of the engineering research associations in Japan* 1986.


Appendix. Inconsistency Robust Logic Programs

Notation of Direct Logic

The aims of logic should be the creation of “a unified conceptual apparatus which would supply a common basis for the whole of human knowledge.”

[Tarski 1940]

In Direct Logic, unrestricted recursion is allowed in programs by using recursive definitions.

There are uncountably many Actors.\textsuperscript{39} For example, CreateReal,\texttt{[ ]} can output any real number\textsuperscript{i} between 0 and 1 where\textsuperscript{ii}

\begin{equation*}
\text{CreateReal,\texttt{[ ]}} \equiv [(0 \text{ either } 1), \textbf{\texttt{Postpone}} \text{CreateReal,\texttt{[ ]}}]
\end{equation*}

where
\begin{itemize}
\item \text{CreateReal,\texttt{[ ]}} is the result of sending the Actor CreateReal the message \texttt{[ ]}
\item \texttt{(0 either 1)} is the nondeterministic choice of 0 or 1,
\item \texttt{[first, rest]} is the list that begins with \texttt{first} and whose remainder is \texttt{rest}, and
\item \textbf{Postpone expression} delays execution of \texttt{expression} until the value is needed.
\end{itemize}

Each \text{CreateReal,\texttt{[ ]}} is one of uncountably many Actors.

There are uncountably many propositions (because there is a different proposition for every real number). Consequently, there are propositions that are not the abstraction of any element of a denumerable set of sentences. For example, \( p \equiv [x: \mathbb{R}] \rightarrow ([y: \mathbb{R}] \rightarrow (y=x)) \) defines a different predicate \( p[x] \) for each real number \( x \), which holds for only one real number, namely \( x \).\textsuperscript{iii}

\textsuperscript{i} using binary representation.

\textsuperscript{ii} Typically, a result returned by the non-deterministic procedure \texttt{Real} is not computable in the sense there is no computable deterministic procedure that can compute its digits.

\textsuperscript{iii} For example \( (p[3])[y] \) holds if and only if \( y=3 \).
It is important to distinguish between sentences, and propositions. Sentences\(^1\) (which without free variables) can be abstracted into propositions that can be asserted. Furthermore, terms\(^2\) can be abstracted into Actors (i.e. objects in mathematics).

Abstraction and parsing are becoming increasingly important in software engineering. e.g.,

- The execution of code can be dynamically checked against its documentation. Also Web Services can be dynamically searched for and invoked on the basis of their documentation.
- Use cases can be inferred by specialization of documentation and from code by automatic test generators and by model checking.
- Code can be generated by inference from documentation and by generalization from use cases.

Abstraction and parsing are needed for large software systems so that that documentation, use cases, and code can mutually speak about what has been said and their relationships.

For example:

**Proposition**

\[\forall [n:N] \rightarrow \exists [m:N] \rightarrow m > n\]

i.e., for every \(N\) there is a larger \(N\)

**Sentence**

\[(\forall [n:N] \rightarrow (\exists [m:N] \rightarrow (m > n)))\]

i.e., the sentence that for every \(N\) there is a larger \(N\)

\(^1\) which are grammar tree structures
\(^2\) which are grammar tree structures
In Direct Logic, a sentence is a grammar tree (analogous to the ones used by linguists). Such a grammar tree has terminals that can be constants. And there are uncountably many constants, e.g., the real numbers:

Of course, because the digits of 3.14159... are computable, there is an expression\textsuperscript{1} such that \([\text{expression}\textsubscript{1}] = 3.14159...\) that can be used to create the sentence \(\langle \text{expression}\textsubscript{1} < (\text{expression}\textsubscript{1} + 1) \rangle\).

However the sentence \(\langle \text{expression}\textsubscript{1} < (\text{expression}\textsubscript{1} + 1) \rangle\) is not the same as \(\langle 3.14159... < (3.14159... + 1) \rangle\) because it does not have the same vocabulary and it is a much larger sentence that has many terminals whereas \(\langle 3.14159... < (3.14159... + 1) \rangle\) has just 3 terminals:

```
< +
3.14159... 3.14159... 1
```

Consequently, sentences cannot be enumerated.

**Note:** Type theory of Classical Direct Logic is much stronger than constructive type theory with constructive logic\textsuperscript{40} because Classical Direct Logic has all of the power of Classical Mathematics.
Notation of Direct Logic

- **Type i.e., a Type** is a *discrimination* of the following:
  - \(\tau \text{ Type<} \tau \text{>}\)
  - Boolean: Type<Boolean>\(^{41}\) and \(\mathbb{N}^{42}: \text{Type<}\mathbb{N}\text{>}\)
  - If order: \(\mathbb{N}_n\), then
    - Proposition<order>\(\text{Type<} \text{Proposition<order>}\text{>}\) \(^{ii}\) and
    - Sentence<order>\(\text{Type<} \text{Sentence<order>}\text{>}\) \(^{iii}\)
  - \(\tau_1 \cap \tau_2: \text{Type<} \tau_1 \cap \tau_2 \text{>}\) \(^{43}\), \([\tau_1, \tau_2]: \text{Type<} [\tau_1, \tau_2] \text{>}\) \(^{44}\),
  - \([\tau_1] \rightarrow \tau_2: \text{Type<} [\tau_1] \rightarrow \tau_2 \text{>}\) \(^{iv}\), and \(\sigma_2^{\text{v}}: \text{Type<} \tau_1 \cap \tau_2 \text{>}\) \(^{v}\)
  - \(\text{Set<}\text{Type<Set<} \tau \text{>}\text{>}\)

- **Propositions, i.e., a Proposition** is a *discrimination* of the following where order: \(\mathbb{N}_n\):
  - If \(\Phi: \text{Proposition<} \text{order}\text{>}, \) then \(\neg \Phi: \text{Proposition<} \text{order}\text{>}\).
  - If \(\Phi, \Psi: \text{Proposition<} \text{order}\text{>}, \) then
    - \(\Phi \land \Psi, \Phi \lor \Psi, \Phi \Rightarrow \Psi, \Phi \Leftrightarrow \Psi: \text{Proposition<} \text{order}\text{>}\).
  - If \(p: \text{Boolean and } \Phi, \Psi: \text{Proposition<} \text{order}\text{>, then}
    - \((p \land \Phi \, \text{True} \land p \land \Phi = \text{False} \land p \land \Phi) : \text{Proposition<} \text{order}\text{>}\) \(^{45}\).
  - If \(x_1, x_2: \tau, \) then \(x_1 = x_2, x_1 \subseteq x_2, x_1 \subset x_2: \text{Proposition<} \text{order}\text{>}\).
  - If \(x: \tau_1 \) and \(p: \text{Proposition<} \text{order}\text{>}, \) then
    - \(p[x]: \text{Proposition<} \text{order} + 1 \text{>}\) \(^{vi}\).
  - If \(p: \text{Proof, T: Theory, and } \Phi_1 \text{ to } \Phi_n: \text{Proposition<} \text{order}\text{>, then}
    - \((\Phi_1, ..., \Phi_n: \text{Proof, T: Theory, and } \Phi_1 \text{ to } \Phi_n: \text{Proposition<} \text{order}\text{>})\) \(^{46}\).
  - If \(s: \text{Sentence<} \text{order}\text{> with no free variables, then}
    - \(s: \text{Proposition<} \text{order}\text{>}\).

\(^{1}\) The type of \(\tau\) means that "self-referential" types cannot be constructed in Direct Logic.

\(^{ii}\) If \(x: \text{Proposition \Rightarrow } \exists[y: \text{order<} \mathbb{N}_n \text{> \rightarrow x: \text{Proposition<} \text{order}\text{>}}\)

\(^{iii}\) If order1 \(\leq\) order2, then \(x: \text{Proposition<} \text{order1} \Rightarrow x: \text{Proposition<} \text{order2}\text{>\)

\(^{iv}\) Type of computable procedures from \(\tau_1\) into \(\tau_2\).

\(^{v}\) If \(f: [\tau_1] \rightarrow \tau_2\) and \(x: \tau_1\) then \(f(x): \tau_2\)

\(^{vi}\) The type of \(p[x]\) means that "self-referential" propositions cannot be constructed in Direct Logic.
Grammar trees (i.e. expressions and sentences) are defined as follows:

- **Expressions**: i.e., an **Expression** in a discrimination of the following:
  - **Constant**: Type Constant.
  - (Boolean): Constant Type Boolean.
  - (N): Constant Type N.
  - (Theory): Constant Type Theory and (Proof): Constant Type Proof.
  - If order: N, then (Proposition: Order): Constant Proposition Order.
  - If order: 1, then (Sentence: Order): Constant Sentence Order.
  - If x: Constant, then x: Expression.
  - Expression: Type Expression.
  - If x: Variable, then x Expression.
  - If for i in 1 to n, f_i: Variable in d_i and y, x_i: Variable in d_i, f_i: Expression in x_i, and y: Expression, then: let f_i[x_i]: τ -> d_i, f_i[x_i] y: τ: Expression.
  - If for i in 1 to n, x_i: Variable in d_i and y, d_i: Expression in x_i, and y: Expression, then let x_i: τ: d_i -> d_i, x_i: d_i: y: τ: Expression.
  - If e_1: Expression and e_2: Expression, then (e_1: Operation: e_2) Expression in [τ_1, τ_2]-> Expression.
  - If e_1: Expression in [τ_1] and x: Expression in τ, then: let x: τ: e_1: Expression.
  - If e: Expression and x: Expression, then: let e[x]: Expression.
  - If e: Expression in [τ_1] and x: Expression in τ, then: let e[x]: Expression.
  - If e: Expression with no free variables, then: e: τ.
• **Sentences**, i.e., a **Sentence** is a discrimination of the following where order: \( \mathbb{N}^+ \):
  - If \( x: \text{Variable} \rightarrow \text{Sentence} < \text{order} \), then \( \{x\}: \text{Sentence} < \text{order} + 1 \).\(^1\)
  - If \( s_1: \text{Sentence} < \text{order} \), then \( \{\neg s_1\}: \text{Sentence} < \text{order} \).
  - If \( s_1, s_2: \text{Sentence} < \text{order} \), then
    \( (s_1 \land s_2), (s_1 \lor s_2), (s_1 \Rightarrow s_2), (s_1 \Leftrightarrow s_2) : \text{Sentence} < \text{order} \).
  - If \( t: \text{Expression} < \text{Boolean} \), \( s_1, s_2: \text{Sentence} < \text{order} \), then
    \( \{t \leftrightarrow \text{True} \Rightarrow s_1 \lor \text{False} \Rightarrow s_2\}: \text{Sentence} < \text{order} \).\(^53\)
  - If \( t_1: \text{Expression} < \tau \) and \( t_2: \text{Expression} < \tau \), then
    \( (t_1 = t_2), (t_1 \subseteq t_2), (t_1 \supseteq t_2) : \text{Sentence} < 1 \).
  - If \( t_1: \text{Expression} < \tau \) and \( t_2: \text{Expression} < \tau \), then
    \( (t_1 \in t_2), (t_1 \notin t_2) : \text{Sentence} < 1 \).
  - If \( x: \text{Variable} < \tau \) in \( s \) and \( s: \text{Sentence} < \text{order} \), then
    \( \{\forall x: \tau \rightarrow s\}, \{\exists x: \tau \rightarrow s\} : \text{Sentence} < \text{order} \).
  - If \( x: \text{Expression} < \tau \) and \( p: \text{Sentence} < \text{order} \), then
    \( \{p(\{x\})\} : \text{Sentence} < \text{order} + 1 \).\(^8\)
  - If \( T: \text{Expression} < \text{Theory} \), \( s_1 \to n: \text{Sentence} < \text{order} \) and \( p: \text{Expression} < \text{Proof} \), then \( \{s_1, \ldots, s_n \vdash s_{k+1}, \ldots, s_n\}: \text{Sentence} < \text{order} \).

---

\(^1\) The type of \( \{x\} \) means that “self-referential” sentences cannot be constructed in Direct Logic.

\(^8\) The type of \( \{p(\{x\})\} \) means that “self-referential” sentences cannot be constructed in Direct Logic.
Foundations with strong parameterized types

Everyone is free to elaborate [their] own foundations. All that is required of [a] Foundation of Mathematics is that its discussion embody absolute rigor, transparency, philosophical coherence, and addresses fundamental methodological issues.
[Nielsen 2014]

Classical Direct Logic develops foundations for mathematics by deriving sets from types and the Peano/Dedekind axioms for the integers to encompass all of standard mathematics including the reals, analysis, geometry, etc.

Sets can be defined as follows:

$$\text{Set}_c = \text{Boolean}^T$$

Sets provide a convenient way to collect together elements with the following axiom.

$$\forall [s:\text{Set}_c] \rightarrow (\forall [e:τ] \rightarrow e \in s \Leftrightarrow s[e]=\text{True})$$

The following strong Peano/Dedekind induction axiom for Proposition $P$ can be used to characterize the natural numbers ($\mathbb{N}$) up to isomorphism with a unique isomorphism:

$$(P[0] \land \forall [i:\mathbb{N}] \rightarrow P[i] \Rightarrow P[i+1]) \Leftrightarrow \forall [i:\mathbb{N}] \rightarrow P[i]$$

---

According to [Scott 1967]: “As long as an idealistic manner of speaking about abstract objects is popular in mathematics, people will speak about collections of objects, and then collections of collections of ... of collections. In other words set theory is inevitable.” [emphasis in original]

According to [Scott 1967]: “there is only one satisfactory way of avoiding the paradoxes: namely, the use of some form of the theory of types... the best way to regard Zermelo's theory is as a simplification and extension of Russell's ...simple theory of types. Now Russell made his types explicit in his notation and Zermelo left them implicit. It is a mistake to leave something so important invisible...”
**Forward Chaining**

Forward chaining is performed using $\vdash$

$\langle \text{"When" } \text{"$\vdash$" PropPattern \rightarrow Exp} \rangle$:
Continuation
When PropPattern holds for Theory, evaluate Exp.

Illustration of forward chaining:

$\vdash$ Human[Socrates] $\triangleright$

When $\vdash$ Human[x] $\rightarrow \vdash$ Mortal[x] $\triangleright$

will result in asserting Mortal[Socrates] for theory $t$.

**Backward Chaining**

Backward chaining is performed using $\models$

$\langle \text{"When" } \text{"$\models$" GoalPattern \rightarrow Exp} \rangle$:
Continuation
When there are goals that matches GoalPatterns for Theory, evaluate Exp.

Illustration of backward chaining:

$\models$ Human[Socrates] $\triangleright$

When $\models$ Mortal[x] $\rightarrow (\models$ Human[x] $\rightarrow \models$ Mortal[x]) $\triangleright$

$\models$ Mortal[Socrates] $\triangleright$

will result in asserting Mortal[Socrates] for theory $t$. 
SubArguments
This section explains how subarguments can be implemented in natural deduction.

When $\vdash_s (\Psi \vdash \Phi) \rightarrow$
Let $t' = \text{Extension}[t]$,
Do $\vdash_{t'} \Psi'$,
$\vdash_{t'} \Phi \rightarrow \vdash_s (\Psi' \vdash \Phi) \mathbf{I}$

Note that the following hold for $t'$ because it is an extension of $t$:

- When $\vdash t \Theta \rightarrow t \Theta \mathbf{I}$
- When $\vdash t \Theta \rightarrow \vdash t \Theta \mathbf{I}$
Index

\(\top\), 45
\(\vdash\), 8, 11, 49
\(\models\), 8, 11, 49
\(\leftrightarrow\), 45
ActorScript, 23
Agha, G., 22
Baumgart, B., 14
Bishop, P., 12
Bledsoe, W., 5
Boley, H., 35
Boolean, 45
Bowker, G., 19
Bruynooghe, M., 15
Bundy, A., 35
Callon, M., 19
Charniak, E., 9
cheese, 28
hole, 28
Church, A., 1
Classical Direct Logic, 7, 31
Clinger, W., 23
Conniver, 12
Davies, J., 14
Decker, H., 5
Dedekind, R., 48
Direct Logic
sentence, 47
Direct Logic
expression, 46
proposition, 45
Edinburgh Logic for
Computable Functions, 13
either, 42
expression
Direct Logic, 46
Expression, 46
Fahlman, S., 34
Feyerabend, P., 19
Gerson, E. M., 19
Gordon, M., 13
Green, C., 5
hairy control structure
J operator, 12
Planner, 12
Hayes, P., 12, 14, 18, 34
IGOR
Inconsistency in Garbage Out
Redux, 3
Inconsistency Robust Direct
Logic, 7, 31
Japanese 5th Generation Project
(ICOT), 26
Jaśkowski, S., 1
Jaśkowski, S., 9
Kahn, K., 35
Kao, E., 4, 35
Karp, A., 35
Kay, A., 12
Kornfeld, W., 19
Kowalski, R., 14, 15, 16, 18, 21,
30, 32, 34, 35
Kuhn, T., 34
Landin, P., 12
Latour, B., 19
Law, J., 19
Lighthill, J., 15
Lisp, 8
Logic
classical, 3
Logic Program, 30, 32
Logic Programming, 32
programming using Logic
Programs, 32
McCarthy, J., 8, 9
McDermott, D.
  hairy control structure, 12
Meltzer, B., 15
Michie, D., 14
Milner, R., 13
Minsky, M., 9
N, 45, 48
Newell, A., 9
Nielsen, 48
Papert, S., 12, 14
patterns of passing messages, 13
Peano, G., 48
Pereira, L., 15
Planck, M., 34
Planner, 1, 8
Polya, G., 9, 17
Popper, K., 9, 19
Postpone, 42
Procedural Embedding, 8
program control structure, 12
Prolog (PROgrammation en LOGique), 16
proposition
   Direct Logic, 45
Proposition, 45
resolution, 5
Robinson, J., 5
sentence
   Direct Logic, 47
Sentence, 47
Set, 45, 48

SHRDLU, 15
Siekmann, J., 15
Simon, H., 9
Simula-67, 12
Smalltalk-72, 12
Star, S., 19
Steiger, R., 12
Strauss, A., 19
Suchman, L., 19
Sussman, G., 9, 14
   hairy control structure, 12
Tarski, A., 42
Type, 45
uniform proof procedure, 5
van Emde, M., 15, 34
van Emde, M., 15
Waldinger, R., 5, 34
When, 8, 11, 49
Winograd, T., 9
Wos, L., 5
Zermelo, E., 48
End Notes

1 Church's system was quickly shown to be inconsistent because it allowed Gödelian “self-referential” propositions, which lead to inconsistency in Mathematics [Hewitt 2011].

2 [Kowalski1988]

3 According to [Kowalski 1979]:
   “an inconsistent system can ... organize useful information...
   Thus [finding a ] contradiction, far from harming an information system, helps to indicate areas in which it can be improved. It facilitates the development of systems by successive approximation – daring conjectures followed by refutation and reconciliation. It favours bold, easily falsified beliefs, which can be weakened if need should arrive, over save, timid beliefs, which are difficult to strengthen later on. Better to make mistakes and to correct them than to make no progress at all.”

4 Raising issues of Inconsistency Robustness, The Obama administration deleted the following statement from its 2008 campaign website:
   “Protect Whistleblowers: Often the best source of information about waste, fraud, and abuse in government is an existing government employee committed to public integrity and willing to speak out. Such acts of courage and patriotism, which can sometimes save lives and often save taxpayer dollars, should be encouraged rather than stifled. We need to empower federal employees as watchdogs of wrongdoing and partners in performance. Barack Obama will strengthen whistleblower laws to protect federal workers who expose waste, fraud, and abuse of authority in government. Obama will ensure that federal agencies expedite the process for reviewing whistleblower claims and whistleblowers have full access to courts and due process.”
   It may be that Obama's Administration's statement on the importance of protecting whistleblowers went from being a promise for his administration to a political liability. There are manifest contradictions in what Obama said then and what he is doing now.


6 In Latin, the principle is called ex falso quodlibet which means that from falsity anything follows.

7 [Nekham 1200, pp. 288-289]; later rediscovered and published in [Lewis and Langford 1932]
8 [Pospesel 2000] has discussed extraneous ∨ introduction on in terms of the following principle: \( \Psi, (\Psi \lor \Phi \vdash \Theta) \vdash \Theta \). However, the above principle immediately derives extraneous ∨ introduction when \( \Theta \) is \( \Psi \lor \Phi \). In Direct Logic, argumentation of the above form would often be reformulated as follows to eliminate the spurious \( \Phi \) middle proposition: \( \Psi, (\Psi \vdash \Theta) \vdash \Theta \).

9 Using ∨-Elimination, i.e., \( \neg \Phi, (\Phi \lor \Psi) \vdash \Psi \).

10 Turing noted that classical logic can be used to make invalid inferences using inconsistent information “without actually going through [an explicit] contradiction.” [Diamond 1976] Furthermore, [Church 1935, Turing 1936] proved that it is computationally undecidable whether a mathematical theory of practice is inconsistent.

11 [Green 1969]

12 [Kowalski 1988]

13 Resolution Theorem Proving is not Inconsistency Robust because it can be used to prove that there are no contradictions as follows:

Using Resolution Theorem Proving, \( \neg(\Psi \land \neg \Psi) \) can be proved because

\[
\Psi, \neg \Psi \vdash \text{False}
\]

It is possible to use Inconsistency-Robust Resolution as follows:

\[
\Psi, \neg \Psi, \neg \Psi \lor \Phi, \Psi \lor \Omega \vdash \Phi \lor \Omega
\]

that requires the additional assumption \( \Psi \lor \neg \Psi \) in order to make the inference.

Of course, it is possible to add the classical resolution rule to a theory \( T \) by adding the following: \( \Psi \lor \Phi, \neg \Phi \lor \Theta \vdash \Psi \lor \Theta \)

14 The argument below originated in [Nekham 1200, pp. 288-289] (later rediscovered and published in [Lewis and Langford 1932]) is an argument that an inconsistency \( \Theta \land \neg \Theta \) can be used to infer every proposition \( \Theta \). The Nekham argument is not valid in Inconsistency Robust Direct Logic because it make use of the rule of Extraneous \( \lor \)–Introduction, i.e., \( \Psi \vdash (\Psi \lor \Phi) \). The Nekham argument can be formalized is as follows: \( \Theta \vdash (\Theta \lor X) \) and therefore \( \Theta, \neg \Theta \vdash X \) because \( \neg \Theta, (\Theta \lor X) \vdash X \).

15 [Wos et. al., 1965]

16 I am grateful to Kowalski for clarifying his position. [Personal communication March 8, 2014].

17 such as the one immediately above

18 (later generalized, e.g., ActorScript [Hewitt 2013])
A ground-complete predicate is one for which all instances in which the predicate holds are explicitly manifest, i.e. instances can be generated using patterns. See [Ross and Sagiv 1992, Eisner and Filardo 2011].

Execution can proceed differently depending on how sets fit into computer storage units.

Sussman and Steele [1975] mistakenly concluded

“we discovered that the 'Actors' and the lambda expressions were identical in implementation.”

The actual situation is that the lambda calculus is capable of expressing some kinds of sequential and parallel control structures but, in general, not the concurrency expressed in the Actor model. On the other hand, the Actor model is capable of expressing everything in the lambda calculus and more.

Sussman and Steele noticed some similarities between Actor customers and continuations introduced by [Reynolds 1972] using a construct called escape that was a further development of hairy control structure. In their program language Scheme, they called their variant of escape by the name “call with current continuation.” Unfortunately, general use of escape is not compatible with usual hardware stack disciple introducing considerable operational inefficiency. Also, using escape can leave customers stranded. Consequently, use of escape is generally avoided these days and exceptions are used instead so that clean up can be performed. [Hewitt 2009]

There was somewhat similar work that Hayes had discussed with the researchers at Aberdeen on ABSYS/ABSET [Foster and Elcock 1969].

According to [Colmerauer and Roussel 1996]:

*While attending an IJCAI convention in September ’71 with Jean Trudel, we met Robert Kowalski again and heard a lecture by Terry Winograd on natural language processing. The fact that he did not use a unified formalism left us puzzled. It was at this time that we learned of the existence of Carl Hewitt’s programming language, Planner [Hewitt, 1969]. The lack of formalization of this language, our ignorance of Lisp and, above all, the fact that we were absolutely devoted to logic meant that this work had little influence on our later research.*

However, according to [Kowalski 2008]:

*During the next couple of years, I tried to reimplement Winograd’s system in resolution logic and collaborated on this with Alain Colmerauer in Marseille. This led to the procedural interpretation of Horn clauses (Kowalski 1973/1974) and to Colmerauer’s development of the programming language Prolog.*

[Horn 1951]
In practice, Prolog implemented a number of non-logical computational primitives for input-output, etc. Like Planner, for the sake of efficiency, it used backtracking. Prolog also had a non-logical computational primitive like the one of Planner to control backtracking by conditionally testing for the exhaustive failure to achieve a goal by backward chaining. However, Prolog was incapable of expressing strong “Negation as Failure” because it lacked both the assertions and true negation of Planner and thus it was impossible in Prolog to say “if attempting to achieve the goal $G$ exhaustively fails then assert (not $G$).” Prolog extended Planner by using unification (but not necessarily soundly because for efficiency reasons it can omit use of the “occurs” check).

Prolog required a top-level goal $\Phi_1 \land \ldots \land \Phi_n$ to be stated as follows: $\text{False} \leftarrow (\Phi_1 \land \ldots \land \Phi_n)$ [logically equivalent to the disjunctive clause $\neg \Phi_1 \lor \ldots \lor \neg \Phi_n$], which requires that in order to find solutions to the goal, the disjunctive clause must be refuted by deriving an contradiction, which is not Inconsistency Robust.

Some of the material below was contributed by the author for publication in Wikipedia on “The Actor Model.”

[Kowalski 1988a] On the other hand since the fall of 1972 with the invention of the Actor Model, Logic Programs can be rigorously defined very general terms (starting with the McCarthy’s Advice Taker proposal [McCarthy 1958]) as “what can be programmed in mathematical logic.” Of course, what can be programmed in mathematical logic is exactly “each computational step (e.g. as defined in the Actor model) can be logically deduced.” Even allowing the full power of Direct Logic, computation is not reducible to Logic Programs [Hewitt 2011].

[Cf. Plotkin [1976]]

When asserted that aResult is sent in aResponse

Assert in Unbounded1[aRequest] that Counter is sent []

e.g. [Kowalski and Sadri 2015]

ICOT used monotonic mutable lists instead of events in its Prolog-style clause programs.

Later adapted for concurrency, e.g., Java, etc.

Use of multiple ports is an awkward programming idiom that introduces many difficulties, e.g., starvation due not properly servicing a port. For a contrary view, see [Kahn and Saraswat 1990].

[Kowalski 2014]
Computerization of argumentation is still in its infancy, cf., [Toulmin 1959, Woods 2000, Bench-Capon 2012]. Also, there is a great deal of ongoing research in formalizing mathematical proofs [Avigad and Harrison 2014].

The Computational Representation Theorem [Clinger 1981; Hewitt 2006] characterizes computation for systems which are closed in the sense that they do not receive communications from outside:

The denotation $\text{Denote}_S$ of a closed system $S$ represents all the possible behaviors of $S$ as

$$\text{Denote}_S = \lim_{i \to \infty} \text{Progression}_S^i$$

where $\text{Progression}_S^i \to \text{Progression}_S^{i+1}$

In this way, $S$ can be mathematically characterized in terms of all its possible behaviors (including those involving unbounded nondeterminism).

The denotations of the Computational Representation Theorem form the basis of procedurally checking programs against all their possible executions.

c.g. [Shulman 2012, nLab 2014]

$\forall [x: \text{Boolean}] \to x=\text{True} \lor x=\text{False}$

The natural numbers are axiomatised as follows where $\text{Successor}$ is the successor function:

- $0: \mathbb{N}$
- $\forall [i: \mathbb{N}] \to \text{Successor}[i] \neq 0$
- $\forall [i, j: \mathbb{N}] \to \text{Successor}[i] = \text{Successor}[j] \Rightarrow i=j$
- Peano/Dedekind induction axiom for Proposition $P$:

  $$(P[0] \land \forall [i: \mathbb{N}] \to P[i] \Rightarrow P[\text{Successor}[i]]) \Rightarrow \forall [i: \mathbb{N}] \to P[i]$$

40 For $i=1,2$

- If $x: \tau_1$, then $((\tau_1 \uplus \tau_2)[x])? = ((\tau_1 \uplus \tau_2)[x]) \downarrow \tau_i$.
- $\forall [z: \tau] \to z: \tau_1 \uplus \tau_2 \Leftrightarrow \exists [x: \tau_i] \to z = ((\tau_1 \uplus \tau_2)[x])$ Type of 2-element list with first element of type $\tau_1$ and with second element of type $\tau_2$

41 $\text{i}f$ $t \text{ then } \Phi \text{ else } \Psi$

$\Phi_1, \ldots$ and $\Phi_k$ infer $\Psi_1, \ldots$, and $\Psi_n$

$\langle \text{True} \rangle, \langle \text{False} \rangle: \text{Constant}<\text{Boolean}>$

$\langle 0 \rangle, \langle 1 \rangle: \text{Constant}<\mathbb{N}>$

42 mutually recursive definitions of functions $f_1 \ldots n$

43 mutually recursive definitions of variables $x_1 \ldots n$

44 $\text{i}f$ $t_1$ $\text{ then } t_2$ $\text{ else } t_3$
\(\text{if present}
\)

\(\text{if } t \text{ then } s_1 \text{ else } s_1\)

\(\text{if present}\)

[Church 1956; Boolos 1975; Corcoran 1973, 1980]. Also, Classical Direct Logic is not a univalent homotopy type theory [Awodey, Pelayo, and Warren 2013].

\(\text{Set}<\tau>\) is the type of a set of type \(\tau\) with the following axioms:

\[\{\}\colon \text{Set}<\tau>\]  // the empty set \(\{\}\) is a set

\[\forall [x \colon \tau] \to \{x\} : \text{Set}<\tau>\]  // a singleton set is a set

\[\forall [s : \text{Set}<\tau>] \to \bigcup s : \text{Set}<\tau>\]  // all elements of the subsets of a set is a set

\[\forall [x : \tau] \to x \notin \{\}\]  // the empty set \(\{\}\) has no elements

\[\forall [s : \text{Set}<\tau>, f : \tau^\tau] \to (\text{Elementwise}[f])[s] : \text{Set}<\tau>\]  // the function image of a set is a set

\[\forall [s : \text{Set}<\tau>, p : \text{Boolean}^\tau] \to s \upharpoonright p : \text{Set}<\tau>\]  // a predicate restriction of a set is a set

\[\forall [s : \text{Set}<\tau>] \to \{\} \subseteq s\]  // \(\{\}\) is a subset of every set

\[\forall [s_1, s_2 : \text{Set}<\tau>] \to s_1 = s_2 \iff (\forall [x : \tau] \to x \in s_1 \iff x \in s_2)\]

\[\forall [x, y : \tau] \to x \in \{y\} \iff x = y\]

\[\forall [s_1, s_2 : \text{Set}<\tau>] \to s_1 \subseteq s_2 \iff (\forall [x : \tau] \to x \in s_1 \Rightarrow x \in s_2)\]

\[\forall [x : \tau, s_1, s_2 : \text{Set}<\tau>] \to x \in s_1 \cup s_2 \iff (x \in s_1 \lor x \in s_2)\]

\[\forall [x : \text{Set}<\tau>, s : \text{Set}<\text{Set}<\tau>>] \to x \in \bigcup s \iff \exists [s_1 : \text{Set}<\tau>] \to x \in s_1 \land s_1 \in s\]  // union of a set is a set

\[\forall [y : \tau, s : \text{Set}<\tau>, f : \tau^\tau] \to y \in (\text{Elementwise}[f])[s] \iff \exists [x \in s] \to f[x] = y\]  // set image of a set is a set

\[\forall [y : \tau, s : \text{Set}<\tau>, p : \text{Boolean}^\tau] \to y \in s \upharpoonright p \iff y \in s \land p[y]\]

[\text{Dedekind 1888, Peano 1889}]

\(\mathbb{N}\) is identified with the type of natural numbers