

Introducing Hyperlog

Selmer Bringsjord

Rensselaer AI & Reasoning (RAIR) Lab
Department of Cognitive Science
Department of Computer Science
Lally School of Management & Technology
Rensselaer Polytechnic Institute (RPI)
Troy, New York 12180 USA

IFLAI2
11/7/2022
ver 1107221039NY



Logistical topics ...

Still many students missing, & some non-checkmarks on those who chimed in.

The screenshot displays the IFLAI2F22_PAPERTOPICS web application interface. The top navigation bar includes icons for Menu, Home, and a red 'K' icon, followed by buttons for Review, Share, Submit, History, Layout, and Chat. Below this, a secondary bar shows 'Source' (selected), 'Source (legacy)', and 'Rich Text' tabs, along with a 'Recompile' button and a download icon.

The left sidebar contains a file explorer with 'main.tex' and 'main72.bib' listed. Below it is a 'File outline' section with a list of document sections: General Orientation, Formatting, Due Dates/S..., The Required Structure ..., The List For You to (Care..., and Sample from Last Offeri....

The main area is split into two panes. The left pane shows the LaTeX source code for 'main.tex', with line numbers 121 through 150 visible. The code includes comments and LaTeX commands for creating a document structure, such as `\item \textbf{Topic Area}:`, `\item \textbf{Specific Claim}:`, and `\item \textbf{Feedback from Selmer}:`. It also includes a `\checkmark` command and a `\begin{itemize}` block. The right pane shows the rendered PDF preview, which has the title 'IFLAI2F22 Paper Topics & Feedback' and the author 'Prof Selmer Bringsjord'. Below the title is a 'Contents' section with a list of items and their page numbers: 1 General Orientation (1), 2 Formatting, Due Dates/Schedule (1), 3 The Required Structure of the Paper (1), 4 The List For You to (Carefully!) Add Yourself To, F22 (2), 5 Sample from Last Offering of IFLAI2 (F21) (6), and References (7).

DCEC in HyperSlate® ...

Inference Schemata

$$\begin{array}{c}
 \frac{\mathbf{K}(a, t_1, \Gamma), \Gamma \vdash \phi, t_1 \leq t_2}{\mathbf{K}(a, t_2, \phi)} [R_K] \quad \frac{\mathbf{B}(a, t_1, \Gamma), \Gamma \vdash \phi, t_1 \leq t_2}{\mathbf{B}(a, t_2, \phi)} [R_B] \\
 \\
 \frac{}{\mathbf{C}(t, \mathbf{P}(a, t, \phi) \rightarrow \mathbf{K}(a, t, \phi))} [R_1] \quad \frac{}{\mathbf{C}(t, \mathbf{K}(a, t, \phi) \rightarrow \mathbf{B}(a, t, \phi))} [R_2] \\
 \\
 \frac{\mathbf{C}(t, \phi) \ t \leq t_1 \dots t \leq t_n}{\mathbf{K}(a_1, t_1, \dots \mathbf{K}(a_n, t_n, \phi) \dots)} [R_3] \quad \frac{\mathbf{K}(a, t, \phi)}{\phi} [R_4] \\
 \\
 \frac{}{\mathbf{C}(t, \mathbf{K}(a, t_1, \phi_1 \rightarrow \phi_2)) \rightarrow \mathbf{K}(a, t_2, \phi_1) \rightarrow \mathbf{K}(a, t_3, \phi_2)} [R_5] \\
 \\
 \frac{}{\mathbf{C}(t, \mathbf{B}(a, t_1, \phi_1 \rightarrow \phi_2)) \rightarrow \mathbf{B}(a, t_2, \phi_1) \rightarrow \mathbf{B}(a, t_3, \phi_2)} [R_6] \\
 \\
 \frac{}{\mathbf{C}(t, \mathbf{C}(t_1, \phi_1 \rightarrow \phi_2)) \rightarrow \mathbf{C}(t_2, \phi_1) \rightarrow \mathbf{C}(t_3, \phi_2)} [R_7] \\
 \\
 \frac{}{\mathbf{C}(t, \forall x. \phi \rightarrow \phi[x \mapsto t])} [R_8] \quad \frac{}{\mathbf{C}(t, \phi_1 \leftrightarrow \phi_2 \rightarrow \neg \phi_2 \rightarrow \neg \phi_1)} [R_9] \\
 \\
 \frac{}{\mathbf{C}(t, [\phi_1 \wedge \dots \wedge \phi_n \rightarrow \phi] \rightarrow [\phi_1 \rightarrow \dots \rightarrow \phi_n \rightarrow \psi])} [R_{10}] \\
 \\
 \frac{\mathbf{S}(s, h, t, \phi)}{\mathbf{B}(h, t, \mathbf{B}(s, t, \phi))} [R_{12}] \quad \frac{\mathbf{I}(a, t, \text{happens}(\text{action}(a^*, \alpha), t'))}{\mathbf{P}(a, t, \text{happens}(\text{action}(a^*, \alpha), t))} [R_{13}] \\
 \\
 \frac{\mathbf{B}(a, t, \phi) \ \mathbf{B}(a, t, \mathbf{O}(a, t, \phi, \chi)) \ \mathbf{O}(a, t, \phi, \chi)}{\mathbf{K}(a, t, \mathbf{I}(a, t, \chi))} [R_{14}]
 \end{array}$$

DCEC (supported fragment)

First-order (Propositional) Schema

- Assume
- Not Elim, Not Intro
- And Elim, And Intro
- Or Elim, Or Intro
- If Elim, If Intro
- Iff Elim, Iff Intro
- Forall Elim, Forall Intro
- Exists Elim, Exists Intro
- Higher Order Forall Elim, Higher Order Forall Intro
- Higher Order Exists Elim, Higher Order Exists Intro
- Eq Elim, Eq Intro
- Pc Oracle, Fol Oracle

Modal Schema

- $R_1, R_2, R_3, R_4,$
- $R_k, R_b,$
- R_{14}

Inference Schemata

Moda

$$\frac{\mathbf{K}(a, t_1, \Gamma), \Gamma \vdash \phi, t_1 \leq t_2}{\mathbf{K}(a, t_2, \phi)} [R_K] \quad \frac{\mathbf{B}(a, t_1, \Gamma), \Gamma \vdash \phi, t_1 \leq t_2}{\mathbf{B}(a, t_2, \phi)} [R_B]$$

$$\frac{}{\mathbf{C}(t, \mathbf{P}(a, t, \phi) \rightarrow \mathbf{K}(a, t, \phi))} [R_1] \quad \frac{}{\mathbf{C}(t, \mathbf{K}(a, t, \phi) \rightarrow \mathbf{B}(a, t, \phi))} [R_2]$$

$$\frac{\mathbf{C}(t, \phi) \ t \leq t_1 \dots t \leq t_n}{\mathbf{K}(a_1, t_1, \dots \mathbf{K}(a_n, t_n, \phi) \dots)} [R_3] \quad \frac{\mathbf{K}(a, t, \phi)}{\phi} [R_4]$$

$$\frac{}{\mathbf{C}(t, \mathbf{K}(a, t_1, \phi_1 \rightarrow \phi_2)) \rightarrow \mathbf{K}(a, t_2, \phi_1) \rightarrow \mathbf{K}(a, t_3, \phi_2)} [R_5]$$

$$\frac{}{\mathbf{C}(t, \mathbf{B}(a, t_1, \phi_1 \rightarrow \phi_2)) \rightarrow \mathbf{B}(a, t_2, \phi_1) \rightarrow \mathbf{B}(a, t_3, \phi_2)} [R_6]$$

$$\frac{}{\mathbf{C}(t, \mathbf{C}(t_1, \phi_1 \rightarrow \phi_2)) \rightarrow \mathbf{C}(t_2, \phi_1) \rightarrow \mathbf{C}(t_3, \phi_2)} [R_7]$$

$$\frac{}{\mathbf{C}(t, \forall x. \phi \rightarrow \phi[x \mapsto t])} [R_8] \quad \frac{}{\mathbf{C}(t, \phi_1 \leftrightarrow \phi_2 \rightarrow \neg \phi_2 \rightarrow \neg \phi_1)} [R_9]$$

$$\frac{}{\mathbf{C}(t, [\phi_1 \wedge \dots \wedge \phi_n \rightarrow \phi] \rightarrow [\phi_1 \rightarrow \dots \rightarrow \phi_n \rightarrow \psi])} [R_{10}]$$

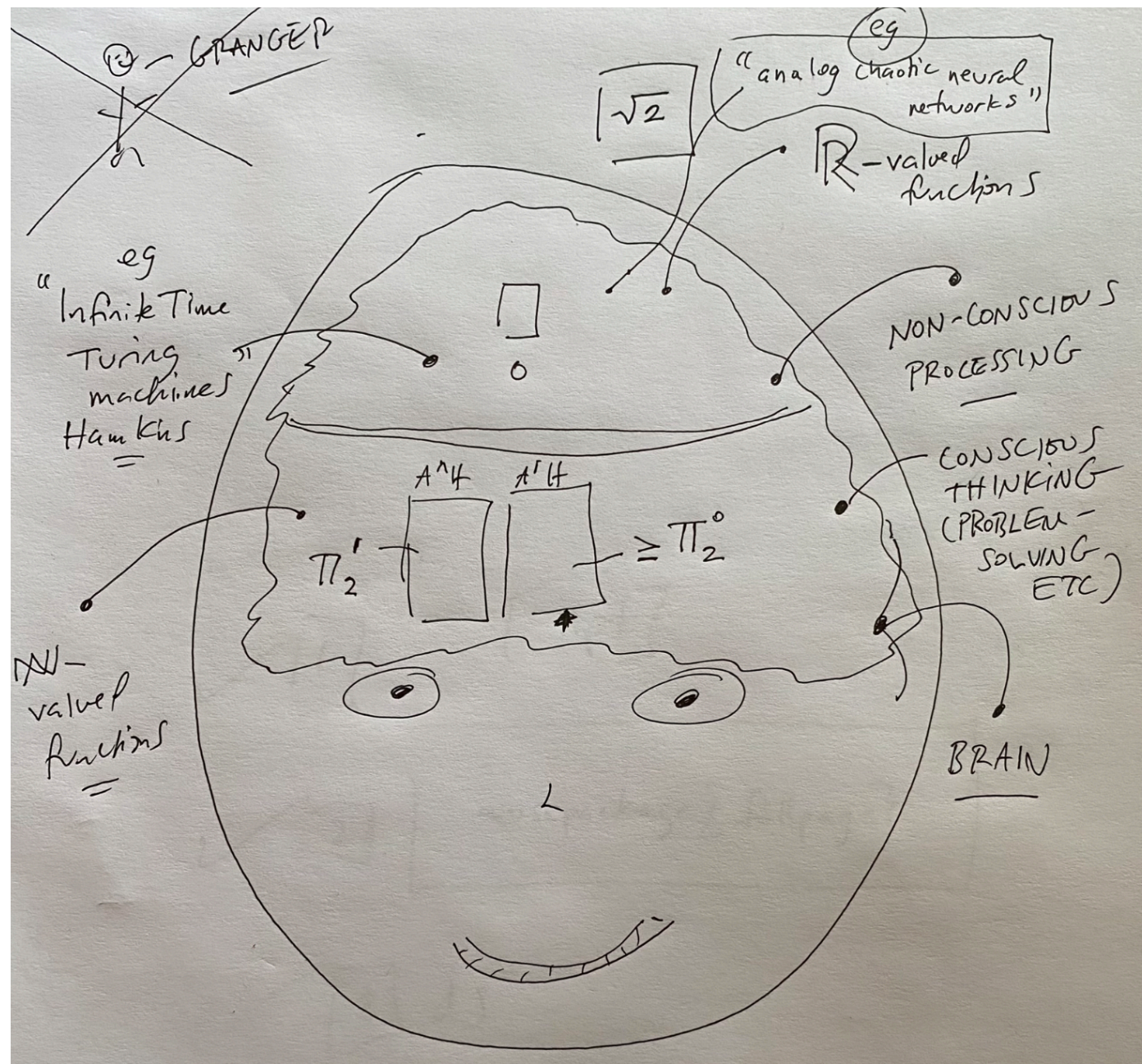
$$\frac{\mathbf{S}(s, h, t, \phi)}{\mathbf{B}(h, t, \mathbf{B}(s, t, \phi))} [R_{12}] \quad \frac{\mathbf{I}(a, t, \text{happens}(\text{action}(a^*, \alpha), t'))}{\mathbf{P}(a, t, \text{happens}(\text{action}(a^*, \alpha), t))} [R_{13}]$$

$$\frac{\mathbf{B}(a, t, \phi) \ \mathbf{B}(a, t, \mathbf{O}(a, t, \phi, \chi)) \ \mathbf{O}(a, t, \phi, \chi)}{\mathbf{K}(a, t, \mathbf{I}(a, t, \chi))} [R_{14}]$$

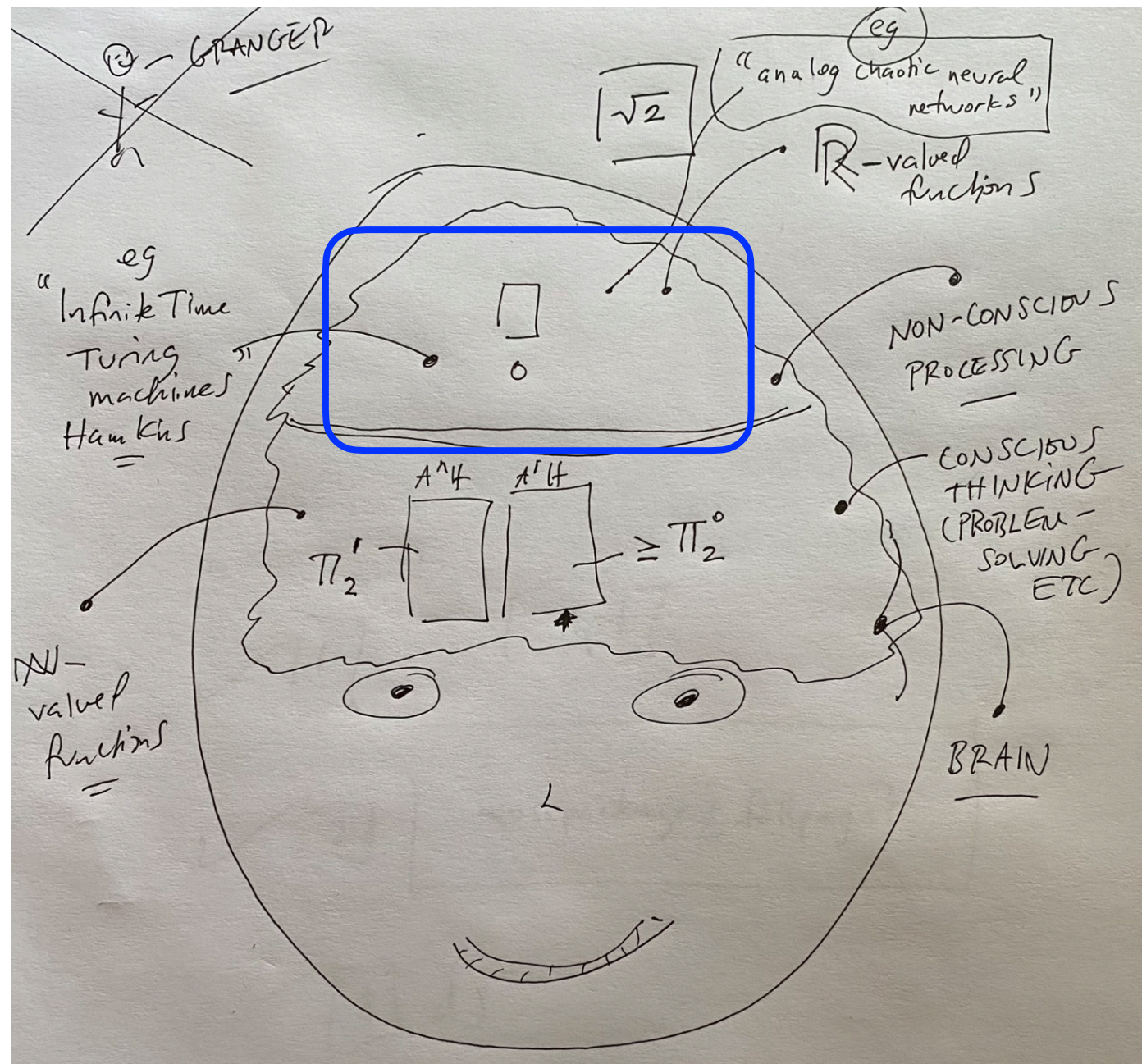
Delivered on promissory
note re building
hierarchies via formal
logic ... questions?

Re computing over \mathbb{R} ?!

What about ... computing over \mathbb{R} ?!




What about ... computing over \mathbb{R} ?!




What about ... computing over \mathbb{R} ?!

Resource I



ELSEVIER

Available online at www.sciencedirect.com



ScienceDirect

Theoretical Computer Science 374 (2007) 277–290

Theoretical
Computer Science

www.elsevier.com/locate/tcs

A new conceptual framework for analog computation

Jerzy Mycka^a, José Félix Costa^{b,*}

^a *Institute of Mathematics, University of Maria Curie-Skłodowska, Lublin, Poland*
^b *Department of Mathematics, I.S.T., Universidade Técnica de Lisboa, Lisboa, Portugal*

Received 3 February 2005; received in revised form 26 December 2006; accepted 15 January 2007

Communicated by F. Cucker

Abstract

In this paper we show how to explore the classical theory of computability using the tools of Analysis: A differential scheme is substituted for the classical recurrence scheme and a limit operator is substituted for the classical minimization. We show that most relevant problems of computability over the non-negative integers can be dealt with over the reals: elementary functions are computable, Turing machines can be simulated, the hierarchy of non-computable functions can be represented (the classical halting problem being solvable at some level). The most typical concepts in Analysis become natural in this framework. The most relevant question is posed: Can we solve open problems of classical computability and computational complexity using, as Popper says, the toolbox of Analysis?

© 2007 Elsevier B.V. All rights reserved.

Keywords: Recursive function theory over the reals; Analog computation; Dynamical systems; Dynamical systems capable of universal computation

What about ... computing over \mathbb{R} ?!

Resource 2

arXiv.org > math > arXiv:math/9808093

Search... All fields Search

Help | Advanced Search

Mathematics > Logic

[Submitted on 21 Aug 1998]

Infinite Time Turing Machines

Joel David Hamkins, Andy Lewis

We extend in a natural way the operation of Turing machines to infinite ordinal time, and investigate the resulting supertask theory of computability and decidability on the reals. The resulting computability theory leads to a notion of computation on the reals and concepts of decidability and semi-decidability for sets of reals as well as individual reals. Every Π^1_1 set, for example, is decidable by such machines, and the semi-decidable sets form a portion of the Δ^1_2 sets. Our oracle concept leads to a notion of relative computability for reals and sets of reals and a rich degree structure, stratified by two natural jump operators.

Comments: 57 pages, 4 figures, to appear in the Journal of Symbolic Logic

Subjects: **Logic (math.LO)**

MSC classes: 03D30; 03D60

Cite as: arXiv:math/9808093 [math.LO]
(or arXiv:math/9808093v1 [math.LO] for this version)

Submission history

From: Joel David Hamkins [view email]
[v1] Fri, 21 Aug 1998 02:52:43 UTC (148 KB)

Download:

- PDF
- PostScript
- Other formats

(license)

Current browse context:
math

< prev | next >
new | recent | 9808





References & Citations

- NASA ADS
- Google Scholar
- Semantic Scholar

5 blog links (what is this?)

Export BibTeX Citation

Bookmark

What about ... computing over \mathbb{R} ?!

arXiv.org > math > arXiv:math/9808093

Search...

All fields

[Help](#) | [Advanced Search](#)

Mathematics > Logic

[Submitted on 21 Aug 1998]

Infinite Time Turing Machines

by [David Hamkins](#), [Andy Lewis](#)

We extend in a natural way the operation of Turing machines to infinite ordinal time, and investigate the resulting supertask theory of computability and decidability on the reals. The resulting computability theory leads to a notion of computation on the reals and concepts of decidability and semi-decidability for sets of reals as well as individual reals. Every Π^1_1 set, for example, is decidable by such machines, and the semi-decidable sets form a portion of the Δ^1_2 sets. Our oracle concept leads to a notion of relative computability for reals and sets of reals and a rich degree structure, stratified by two natural jump operators.

Comments: 57 pages, 4 figures, to appear in the Journal of Symbolic Logic

Subjects: **Logic (math.LO)**

MSC classes: 03D30; 03D60

arXiv: [arXiv:math/9808093](#) [math.LO]

(or [arXiv:math/9808093v1](#) [math.LO] for this version)

Submission history

Author: Joel David Hamkins [[view email](#)]

Submitted: Fri, 21 Aug 1998 02:52:43 UTC (148 KB)

Download

- [PDF file](#)
- [Postscript file](#)
- [Other formats](#) (license)

Current version: **math**

[< prev](#)
[new](#) | [revisions](#)

References

- [NASA](#)
- [Google](#)
- [Semantic Scholar](#)

[5 blog entries](#)

[Export to BibTeX](#)

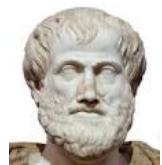
Bookmarks



Introducing HyperLog

.75 ...

Hyperlog: Historico-logico-programming Landscape



First "logic programs" 300 BC



Liebniiz
Dies 1716



Frege
1893



Schöenfinkel
1893

Combinatory Logic



Church

λ -calculus

simple type theory



Logic Theorist
(birth of modern logicist AI)



Simon

1956



Turing

Lisp

Prolog

Fortran

Smalltalk

A
t
h
e
n
a

ML

Scheme

CL

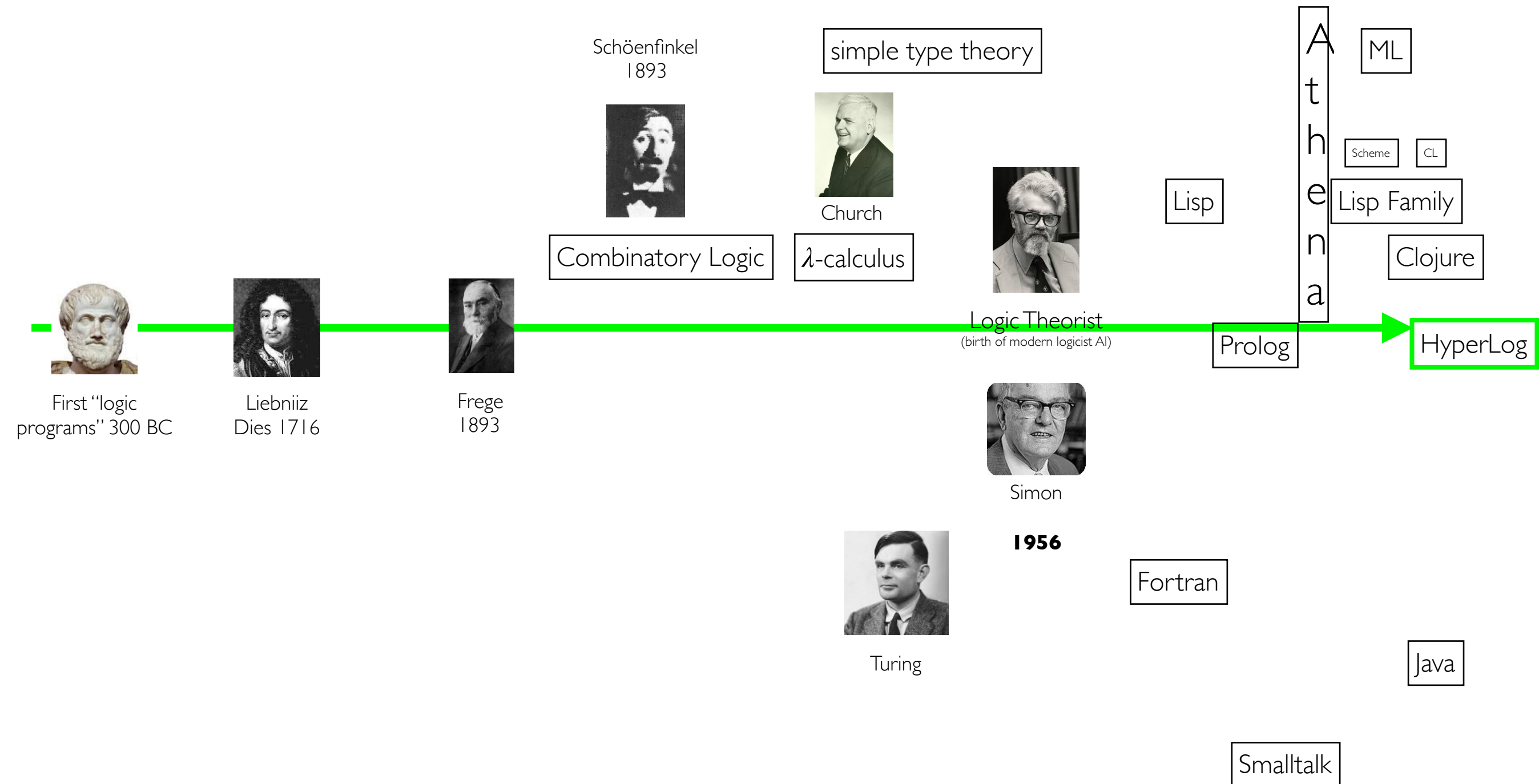
Lisp Family

Clojure

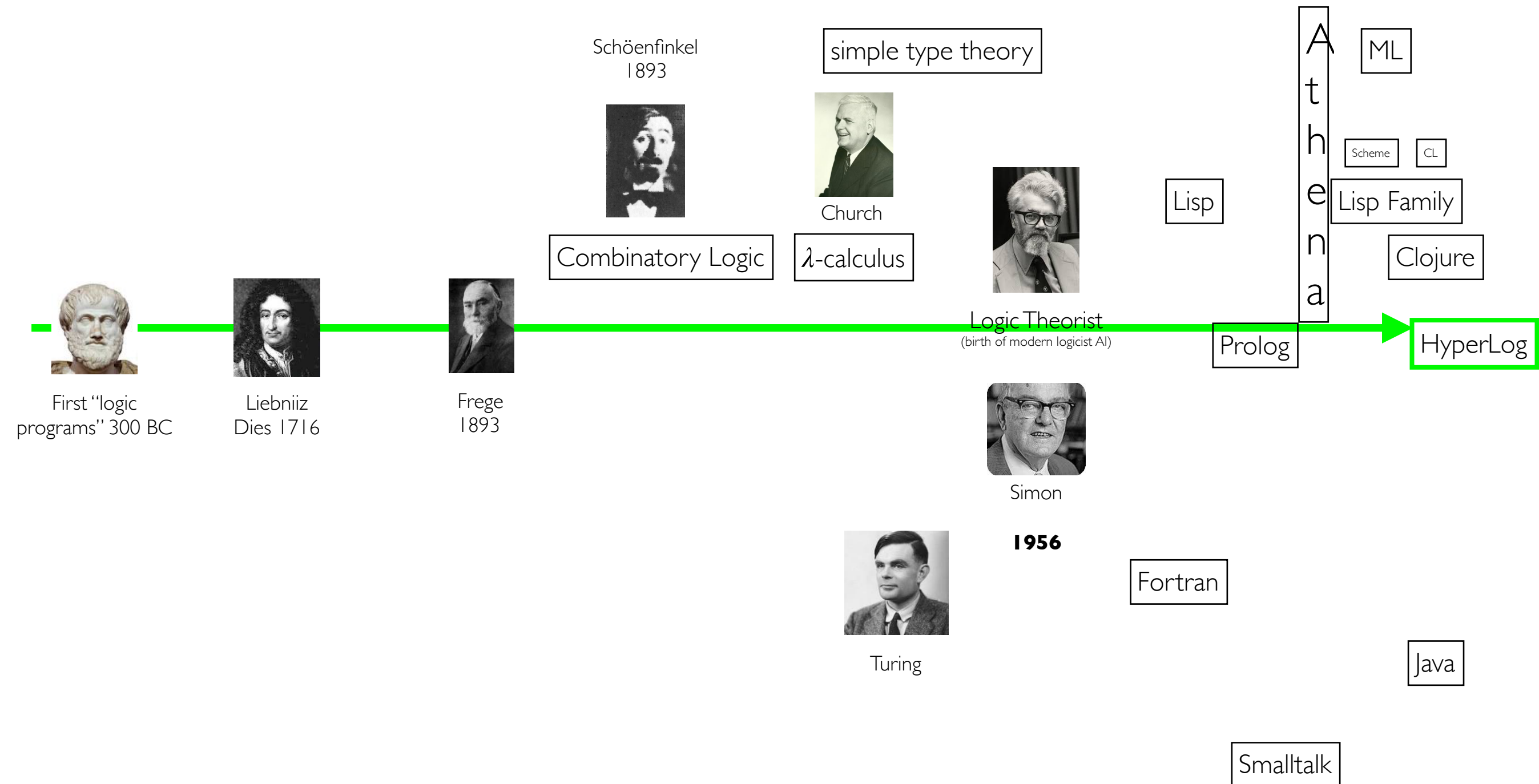
HyperLog

Java

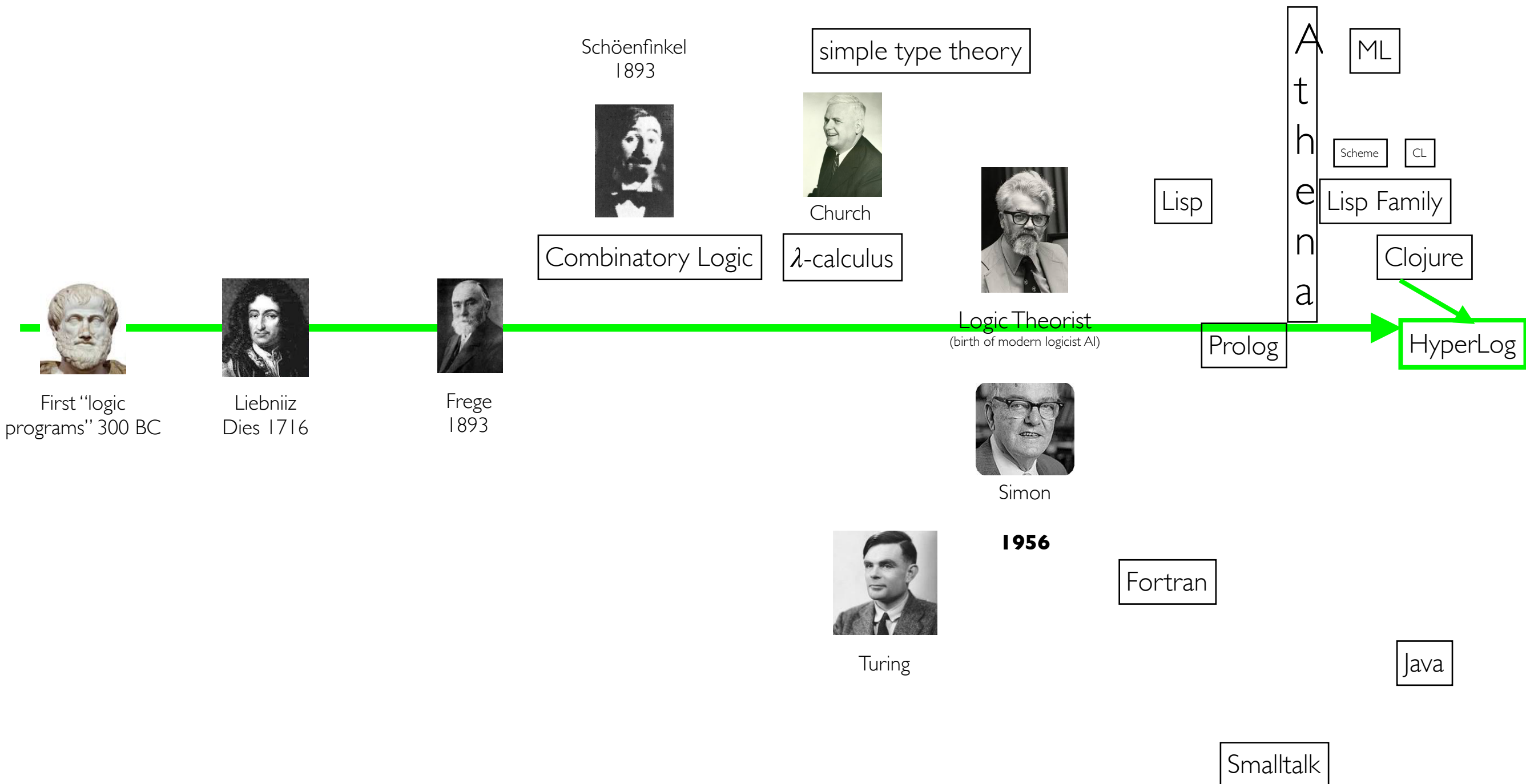
Hyperlog: Historico-logico-programming Landscape



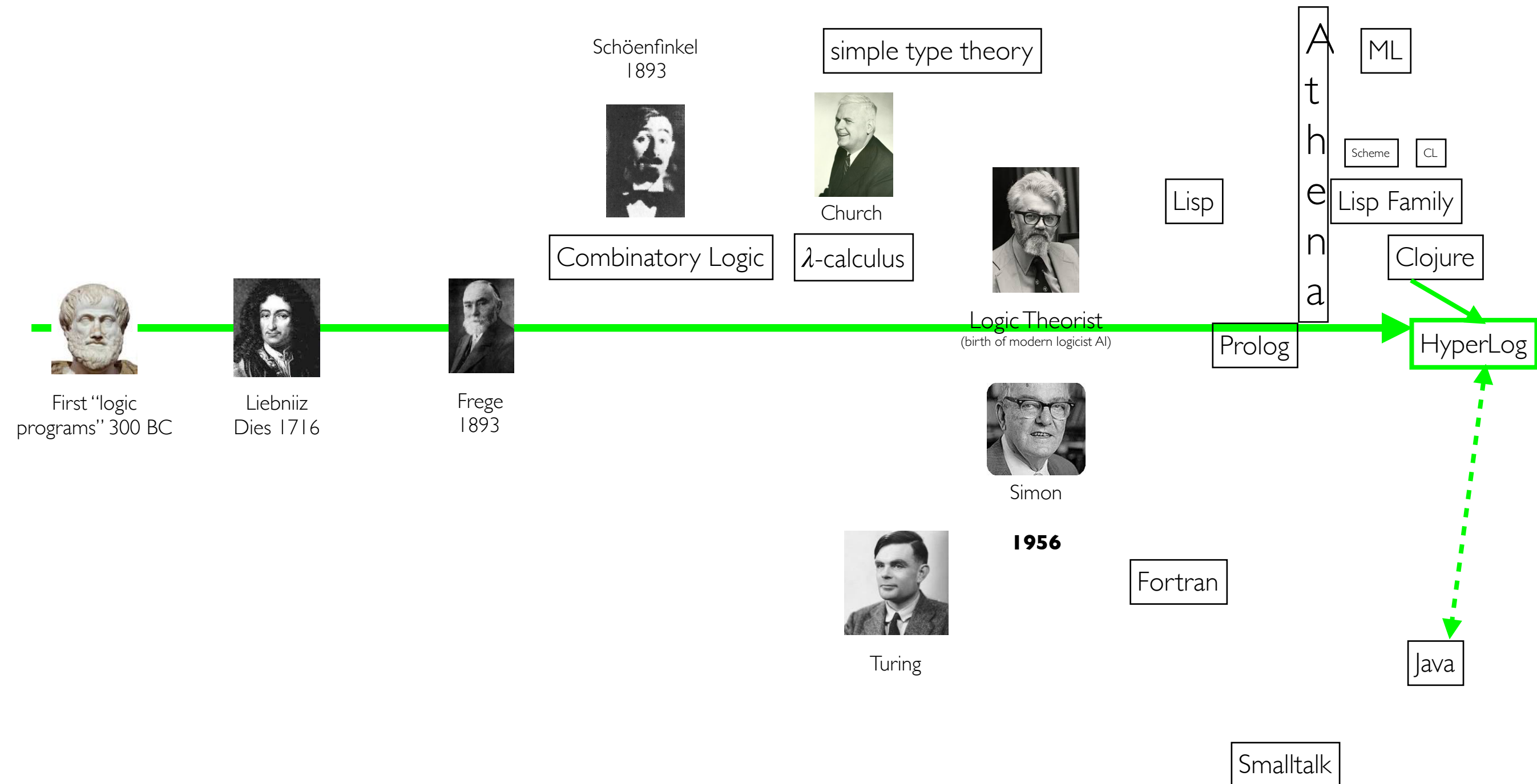
Hyperlog: Historico-logico-programming Landscape



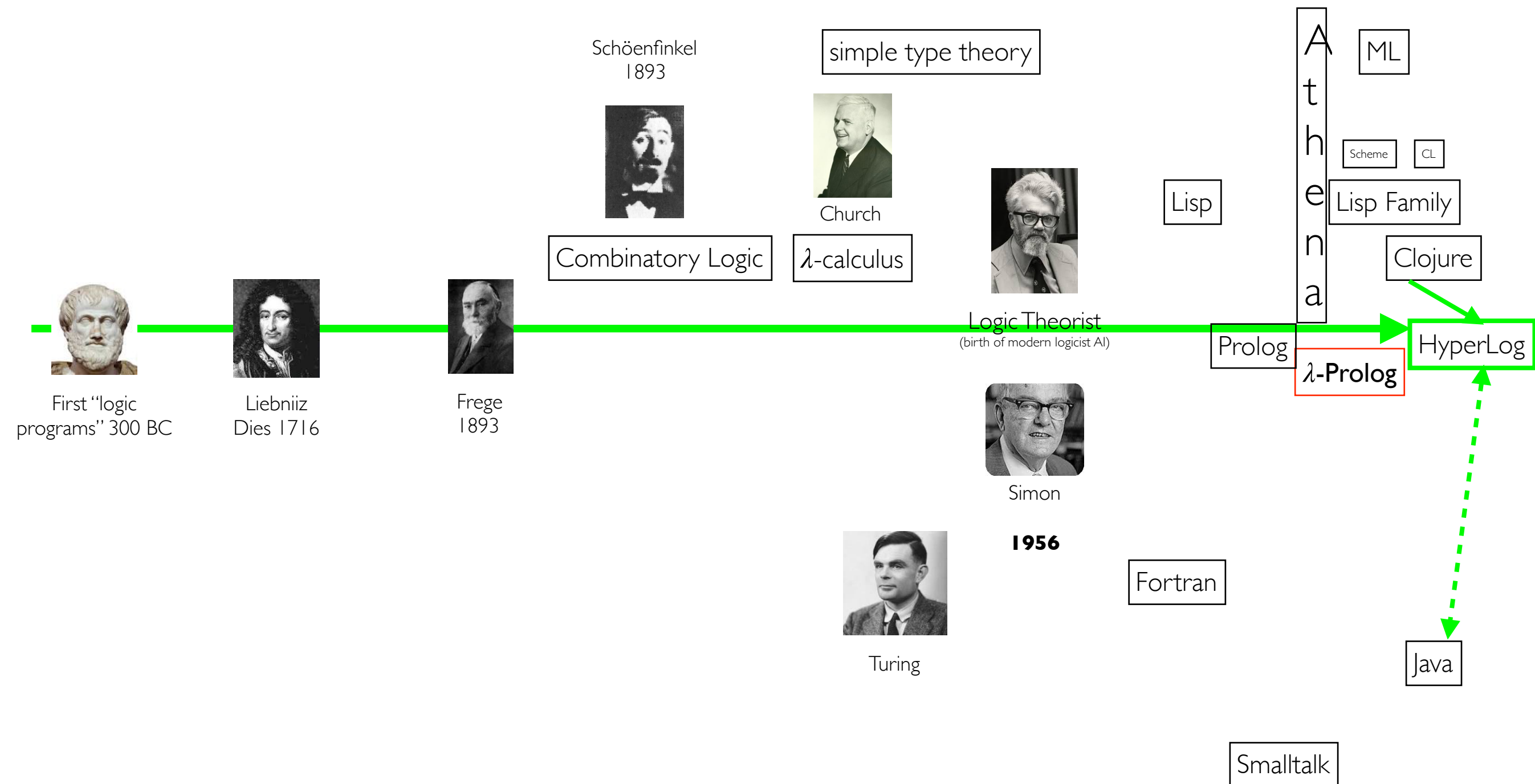
Hyperlog: Historico-logico-programming Landscape



Hyperlog: Historico-logico-programming Landscape



Hyperlog: Historico-logico-programming Landscape



Thinking as Computation

Hector J. Levesque
Dept. of Computer Science
University of Toronto

Constants and variables

A Prolog *constant* must start with a *lower case* letter, and can then be followed by any number of letters, underscores, or digits.

A constant may also be a *quoted-string*: any string of characters (except a single quote) enclosed within single quotes.

So the following are all legal constants:

```
sue opp_sex mamboNumber5 'Who are you?'
```

A Prolog *variable* must start with an *upper case* letter, and can then be followed by any number of letters, underscores, or digits.

So the following are all legal variables:

```
X P1 MyDog The_biggest_number Variable_27b
```

Prolog also has *numeric* terms, which we will return to later.

Atomic sentences

The atomic sentences or *atoms* of Prolog have the following form:

$$\textit{predicate}(\textit{term}_1, \dots, \textit{term}_k)$$

where the predicate is a constant and the terms are either constants or variables.

Note the punctuation:

- immediately after the predicate, there must be a *left parenthesis*;
- between each term, there must be a *comma*;
- immediately after the last term, there must be a *right parenthesis*.

The number of terms k is called the *arity* of the predicate.

If $k = 0$, the parentheses can be left out.

Co

Atomic sentences

A F
follo

The a

So

when
varia

A F
follo

Note

So

- in
- b
- in

Pro

The r
If $k =$

Chapter 3: The

Chapter 3: The P

Conditional sentences

The conditional sentences of Prolog have the following form:

$$head \text{ :- } body_1, \dots, body_n$$

where the head and each element of the body is an atom.

Note the punctuation:

- immediately after the head, there must be a *colon* then *hyphen*, :- .
- between each element of the body, there must be a *comma*.

If $n = 0$, the :- should be omitted.

In other words, an atomic sentence is just a conditional sentence where the body is empty!

Thinkin

Conditional sentences

A Prolog sentence follows the form

So

A Prolog sentence follows the form

So

Prolog

Atomic sentences

The atomic sentence

where x and y are variables

Note that

- in Prolog, x and y are variables
- b and c are constants
- in Prolog, x and y are variables

The sentence

If $k = 1$

Conditional sentences

The conditional sentence

where x and y are variables

Note that

- in Prolog, x and y are variables
- b and c are constants
- in Prolog, x and y are variables

If $n = 1$

In Prolog, the sentence

Prolog programs

Prolog programs are simply *knowledge bases* consisting of atomic and conditional sentences as before, but with a slightly different notation.

Here is the “family” example as a Prolog program:

[family.pl](#)

```
% This is the Prolog version of the family example
child(john,sue).    child(john,sam).
child(jane,sue).    child(jane,sam).
child(sue,george).  child(sue,gina).

male(john).         male(sam).         male(george).
female(sue).        female(jane).      female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

Now let's look at all the pieces in detail ...

Prolog Problems



massachusetts institute of technology — artificial intelligence laboratory

Certified Computation

Konstantine Arkoudas

AI Memo 2001-007

April 30, 2001

© 2001 massachusetts institute of technology, cambridge, ma 02139 usa — www.ai.mit.edu

`unify`, which are by far the two most complicated parts of the system. We only need to trust our five primitive methods. This becomes evident when we ask Athena to produce the relevant certificates. For instance, if we ask Athena to produce the certificate for the method call

```
(!unify (Cons (== s t) Nil))
```

we will obtain the exact same proof that was given in page 17, which only uses the primitive inference rules of our logic.

1.4 Comparison with other approaches

As we mentioned earlier, the idea of using deduction for computational purposes has been around for a long time. There are several methodologies predating DPLs that can be used for certified computation. In this section we will compare DPLs to logic programming languages and to theorem proving systems of the HOL variety.

Comparison with logic programming

The notion of “programming with logic” was a seminal idea, and its introduction and subsequent popularization by Prolog was of great importance in the history of computing. Although logic programming languages can be viewed as platforms for certified computation, they have little in common with DPLs. DPLs are languages for writing proofs and proof strategies. By contrast, in logic programming users do not write proofs; they only write assertions. The inference mechanism that is used for deducing the consequences of those assertions is fixed and sequestered from the user: linear resolution in the case of Prolog, some higher-order extension thereof in the case of higher-order logic programming languages [9, 2], and so on. This rigidity can be unduly constraining. It locks the user into formulating every problem in terms of the same representation (Horn clauses, or higher-order hereditary Harrop clauses [10], etc.) and the same inference method, even when those are not the proper tools to use. For instance, how does one go about proving De Morgan’s laws in Prolog? How does one derive $\neg(\exists x)\neg P(x)$ from the assumption $(\forall x)P(x)$? Moreover, how does one write a schema that does this for any given x and P ? How about higher-order equational rewriting or semantic tableaux? Although in principle more or less everything could be simulated in Prolog, for many purposes such a simulation would be formidably cumbersome.

A related problem is lack of extensibility. Users have no way of extending the underlying inference mechanism so as to allow the system to prove more facts or different types of facts.

The heart of the issue is how much control the user should have over proof construction. In logic programming the proof-search algorithm is fixed, and users are discouraged from tampering with it (e.g., by using impure control operators or clever clause reorderings). Indeed, strong logic programming advocates maintain that the user should have no control at all over proof construction. The user should simply enter a set of assertions, sit back, and let the system deduce the desired consequences. Advocates of weak logic programming allow

relevant certificates. For instance, if we ask Athena to produce the certificate for the method call

```
(!unify (Cons (== s t) Nil))
```

we will obtain the exact same proof that was given in page 17, which only uses the primitive inference rules of our logic.

1.4 Comparison with other approaches

As we mentioned earlier, the idea of using deduction for computational purposes has been around for a long time. There are several methodologies predating DPLs that can be used for certified computation. In this section we will compare DPLs to logic programming languages and to theorem proving systems of the HOL variety.

Comparison with logic programming

The notion of “programming with logic” was a seminal idea, and its introduction and subsequent popularization by Prolog was of great importance in the history of computing. Although logic programming languages can be viewed as platforms for certified computation, they have little in common with DPLs. DPLs are languages for writing proofs and proof strategies. By contrast, in logic programming users do not write proofs; they only write assertions. The inference mechanism that is used for deducing the consequences of those assertions is fixed and sequestered from the user: linear resolution in the case of Prolog, some higher-order extension thereof in the case of higher-order logic programming languages [9, 2], and so on. This rigidity can be unduly constraining. It locks the user into formulating every problem in terms of the same representation (Horn clauses, or higher-order hereditary Harrop clauses [10], etc.) and the same inference method, even when those are not the proper tools to use. For instance, how does one go about proving De Morgan’s laws in Prolog? How does one derive $\neg(\exists x) \neg P(x)$ from the assumption $(\forall x) P(x)$? Moreover, how does one write a schema that does this for any given x and P ? How about higher-order equational rewriting or semantic tableaux? Although in principle more or less everything could be simulated in Prolog, for many purposes such a simulation would be formidably cumbersome.

A related problem is lack of extensibility. Users have no way of extending the underlying inference mechanism so as to allow the system to prove more facts or different types of facts.

The heart of the issue is how much control the user should have over proof construction. In logic programming the proof-search algorithm is fixed, and users are discouraged from tampering with it (e.g., by using impure control operators or clever clause reorderings). Indeed, strong logic programming advocates maintain that the user should have no control at all over proof construction. The user should simply enter a set of assertions, sit back, and let the system deduce the desired consequences. Advocates of weak logic programming allow

relevant certificates. For instance, if we ask Athena to produce the certificate for the method call

```
(!unify (Cons (== s t) Nil))
```

we will obtain the exact same proof that was given in page 17, which only uses the primitive inference rules of our logic.

1.4 Comparison with other approaches

As we mentioned earlier, the idea of using deduction for computational purposes has been around for a long time. There are several methodologies predating DPLs that can be used for certified computation. In this section we will compare DPLs to logic programming languages and to theorem proving systems of the HOL variety.

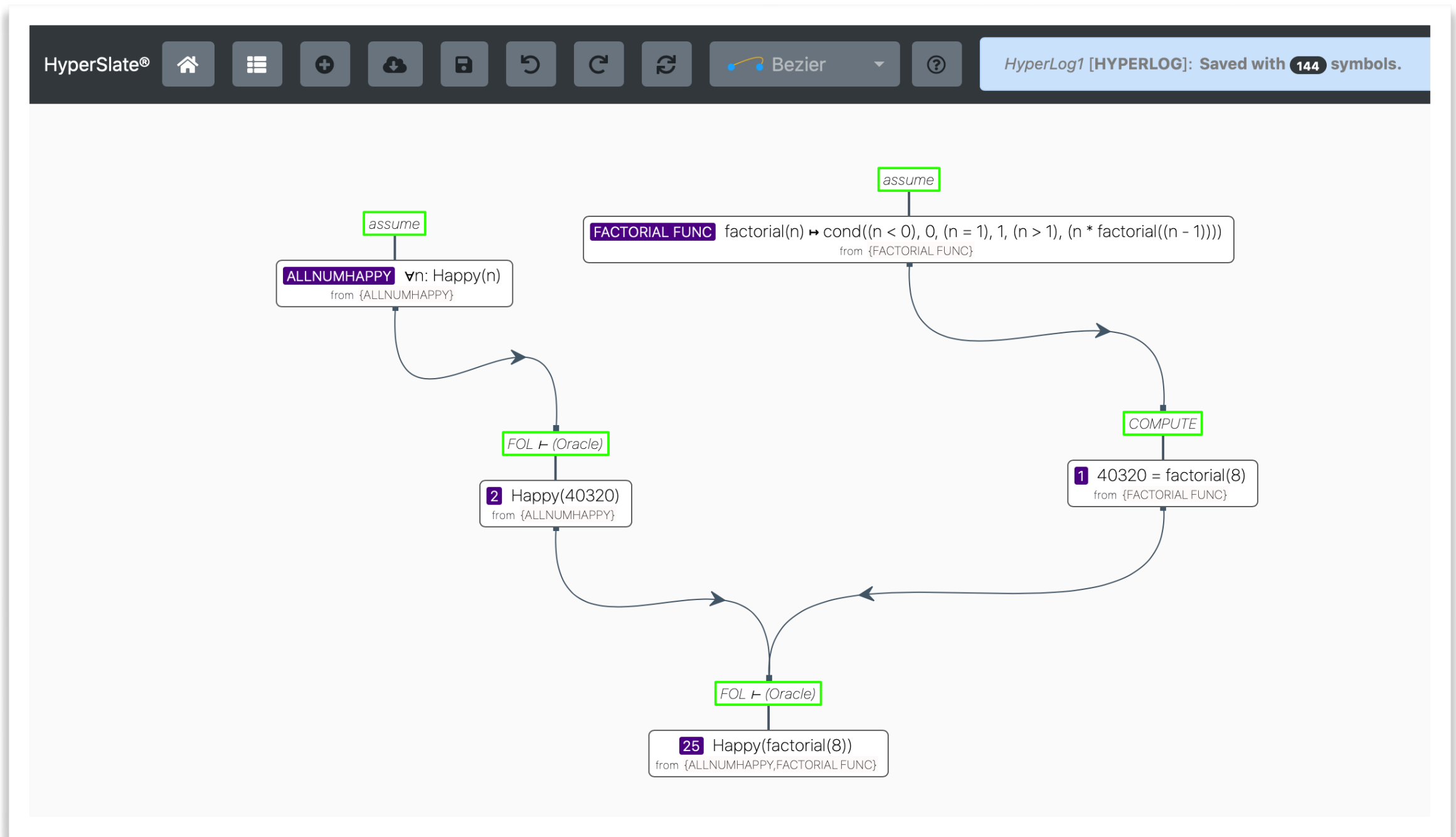
Comparison with logic programming

The notion of “programming with logic” was a seminal idea, and its introduction and subsequent popularization by Prolog was of great importance in the history of computing. Although logic programming languages can be viewed as platforms for certified computation, they have little in common with DPLs. DPLs are languages for writing proofs and proof strategies. By contrast, in logic programming users do not write proofs; they only write assertions. The inference mechanism that is used for deducing the consequences of those assertions is fixed and sequestered from the user: linear resolution in the case of Prolog, some higher-order extension thereof in the case of higher-order logic programming languages [9, 2], and so on. This rigidity can be unduly constraining. It locks the user into formulating every problem in terms of the same representation (Horn clauses, or higher-order hereditary Harrop clauses [10], etc.) and the same inference method, even when those are not the proper tools to use. For instance, how does one go about proving De Morgan’s laws in Prolog? How does one derive $\neg(\exists x) \neg P(x)$ from the assumption $(\forall x) P(x)$? Moreover, how does one write a schema that does this for any given x and P ? How about higher-order equational rewriting or semantic tableaux? Although in principle more or less everything could be simulated in Prolog, for many purposes such a simulation would be formidably cumbersome.

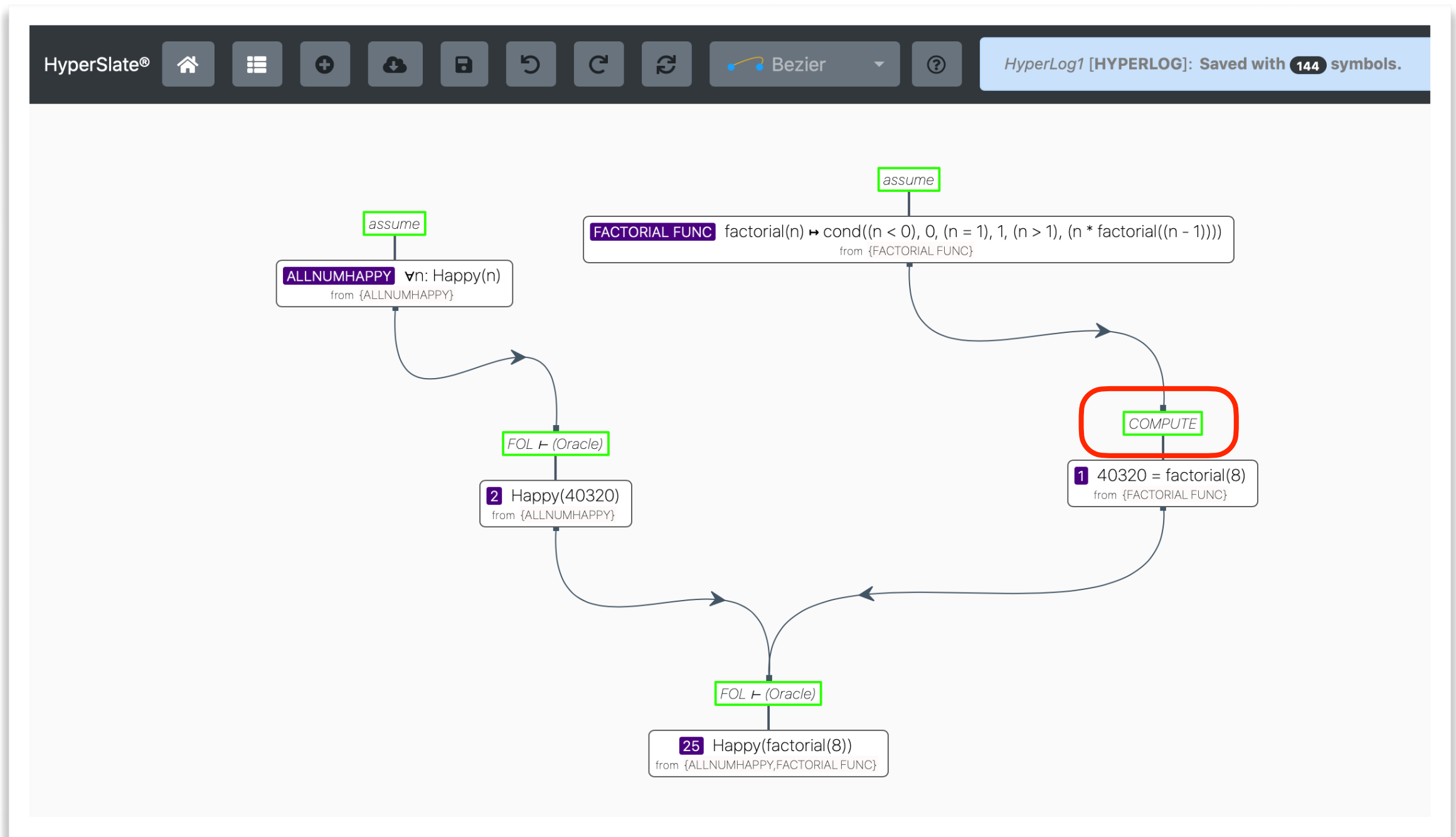
A related problem is lack of extensibility. Users have no way of extending the underlying inference mechanism so as to allow the system to prove more facts or different types of facts.

The heart of the issue is how much control the user should have over proof construction. In logic programming the proof-search algorithm is fixed, and users are discouraged from tampering with it (e.g., by using impure control operators or clever clause reorderings). Indeed, strong logic programming advocates maintain that the user should have no control at all over proof construction. The user should simply enter a set of assertions, sit back, and let the system deduce the desired consequences. Advocates of weak logic programming allow

The Factorial of 8 is Happy!



The Factorial of 8 is Happy!



A New Oracle!

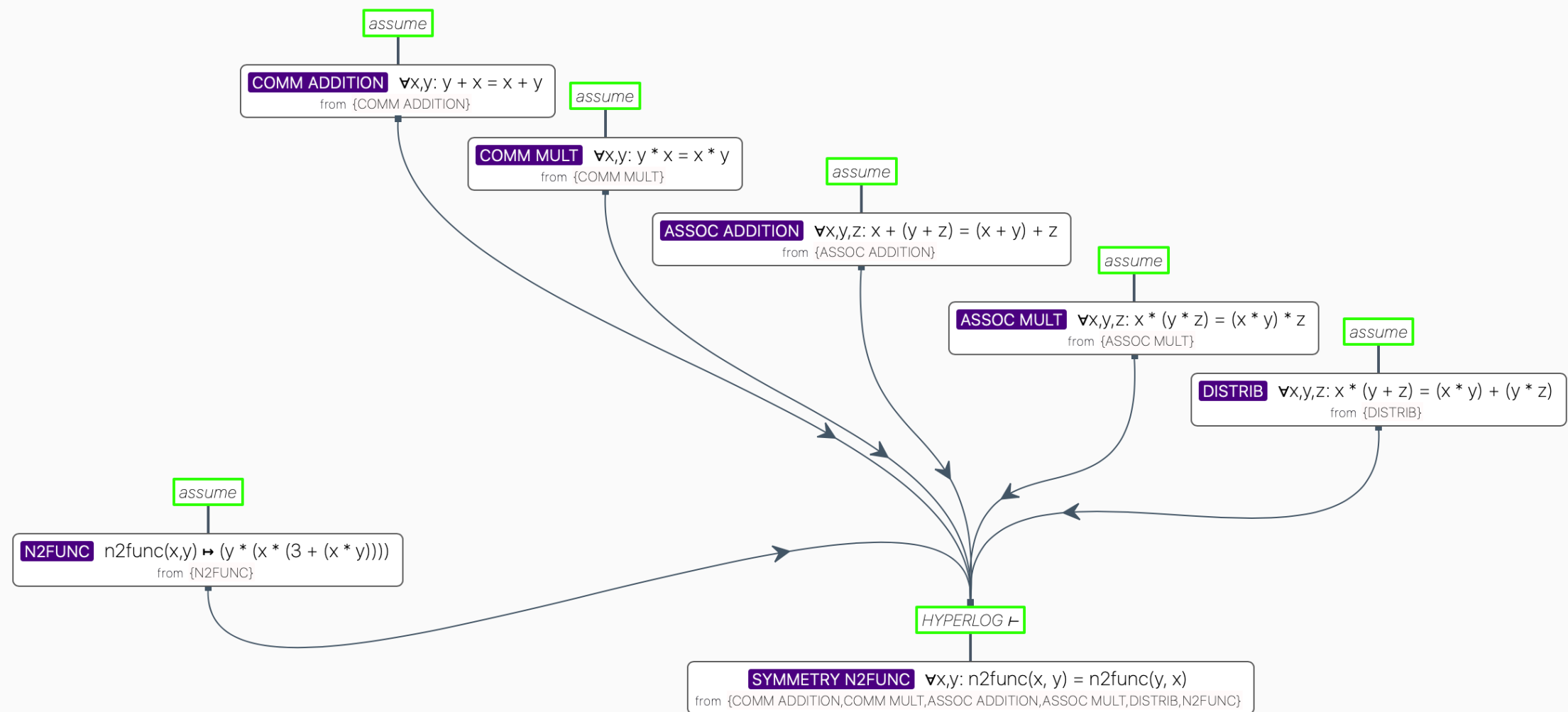
HyperSlate®



Bezier



HLOracularProof [HYPERLOG]: Saved with 90 symbols.



A New Oracle!

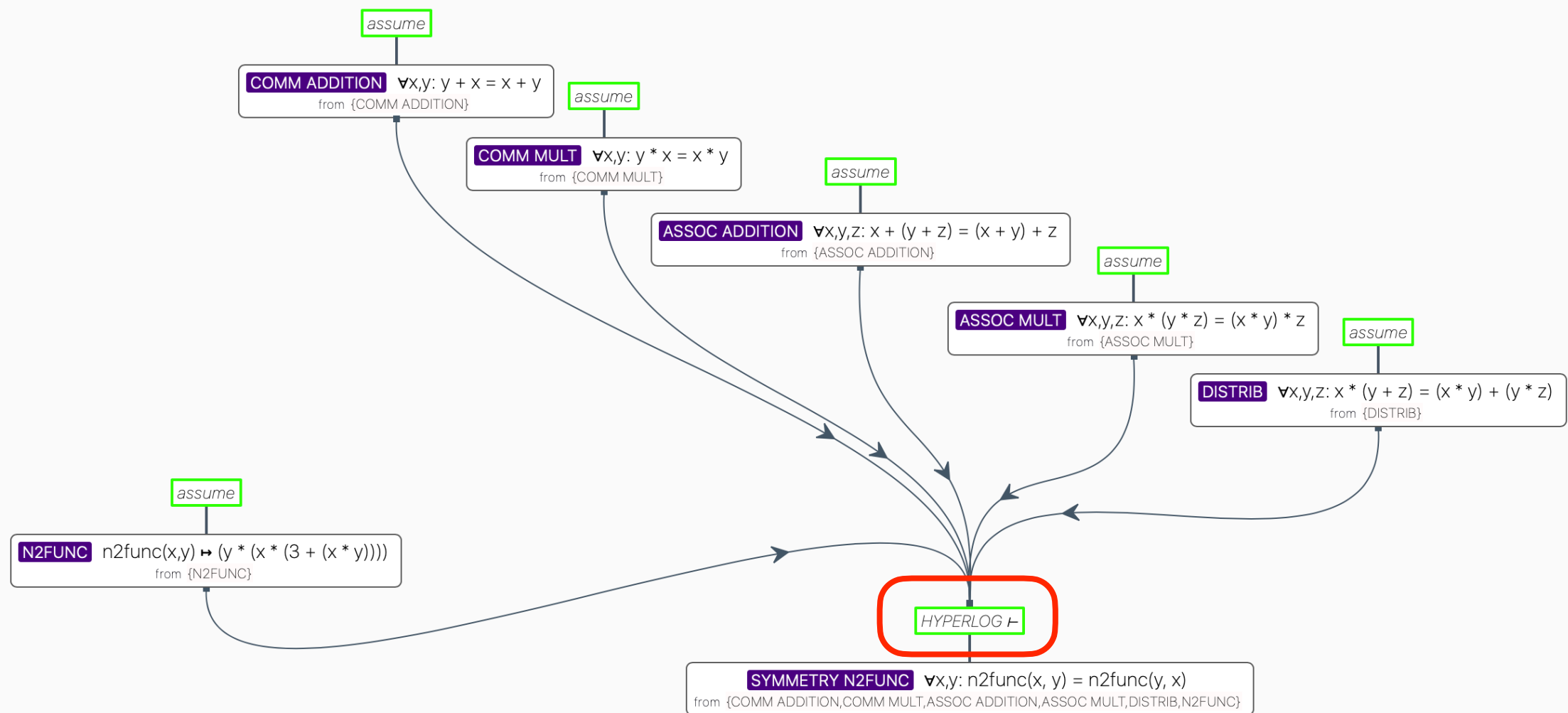
HyperSlate®



Bezier



HLOracularProof [HYPERLOG]: Saved with 90 symbols.



Available in Hyperlog .5

```
"letfn",  
"=", "cond",  
"and", "or", "not", "not=",  
"+", "-", "*", "/", "quot", "rem", "mod",  
"inc", "dec", "max", "min", "+'", "-'", "*'", "inc'", "dec'",  
"==", "<", ">", "<=", ">=", "compare",  
"zero?", "pos?", "neg?", "even?", "odd?",  
"number?", "rational?", "integer?", "ratio?", "decimal?", "float?",  
"double?", "int?", "nat-int?", "neg-int?", "pos-int?",  
"count", "get", "subs", "compare",  
"clojure.string/join", "clojure.string/escape", "clojure.string/split",  
"clojure.string/split-lines", "clojure.string/replace", "clojure.string/replace-first",  
"reverse", "index-of", "last-index-of", "str"
```