An Improved
Proof-Theoretic Compilation
of Logic Programs

Iliano Cervesato
Carnegie Mellon University
Qatar
Overview

• Motivations and background
• Abstract logic programming compilation
• Better compilation
• Modeled compilation
Architecture of a Prolog System

Program

Query

Interpreter

Compiler

Compiled query

Compiled program

WAM

Answer

Interpret

Compile
Proof-Theoretic Interpretation

- **Program/queries**: Logical formulas
- **Answers**: Derivability
- **Semantics**: Uniform proof search

\[ \text{ALPL} \]
**ALPL** [Miller, Nadathur, Pfenning & Scedrov, 91]

- **Computation = proof search**
  - Connectives in $A$: search directives
  - Clauses in $\Gamma$: spec. of how to continue the search when the goal is atomic

- **Uniform proofs**
  - Goal oriented $\Gamma \rightarrow A$
  - Focused $\Gamma \rightarrow B \triangleright a$

In an ALPL, every provable sequent has a uniform proof.
Hereditary Harrop Formulas

\[
A ::= a \mid A \supset B \mid \forall x. A
\]

\[
a ::= p \mid a \, t
\]

\[
\frac{\Gamma, A \rightarrow [c/x]A}{\Gamma \rightarrow [c/x]A}
\]

\[
\frac{\Gamma, A \rightarrow B}{\Gamma \rightarrow A \supset B}
\]

\[
\frac{\Gamma \rightarrow A \supset B}{\Gamma \rightarrow [t/x]A \supset a}
\]

\[
\frac{\Gamma \rightarrow a \supset a}{\Gamma \rightarrow [t/x]A \supset a}
\]

\[
\frac{\Gamma \rightarrow a \supset a}{\Gamma \rightarrow \forall x. A \supset a}
\]

\[
\frac{\Gamma \rightarrow a \supset a}{\Gamma \rightarrow \forall x. A \supset a}
\]

- Term language is left unspecified
  - Must be predicative

Non-determinism
Architecture of a Prolog System

Interpret

Compile

Program

Query

Answer

Interpreter

Compiler

Compiled program

Compiled query

WAM
WAM [Warren, 83]

- Interprets a specialized instruction set for Prolog
- Very fast (~40 times)
- Complex
- Specialized to Prolog
  - Then extended to CLP(R), PROTOS-L, λProlog

- No logical status
  - Where do the instructions come from?
  - What does it do?
Correctness of the WAM (as of 1998)

[Russinoff, 92] [Börger & Rosenzweig, 95]

• Starts from highly operational spec. of Prolog’s semantics
• Complex
• Do not scale to modern logic programming language
Proof-Theoretic Compilation (JICSLP’98)

• Logic-based
  ▪ Transformation between ALPLs
  ▪ Target language is an ALPL

• Logic-independent
  ▪ Applies to any ALPL

• Systematic
  ▪ Easy proofs of correctness

• Abstract and modular
  ▪ Manages gory details

• Used in Twelf and LLF

---

Proof-Theoretic Foundation of Compilation in Logic Programming Languages

Ilario Corradini
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
IlarioCorradini@cs.stanford.edu

Abstract

Commercial implementations of logic programming languages are engineered around compilers based on Warren’s Abstract Machine (WAM) or one of its variants. In spite of their close relationship, the logical machinery behind the proof-theoretic foundation of a logic programming language and its compiled form is still poorly understood. In this paper, we propose a logical foundation of compilation for logic programming languages. We apply this foundation to the logic of a compiler and the corresponding machine for the language of Horn-style Horn formulas and ideas for its future refinement.

1 Introduction

Compiled logic programs can over an order of magnitude fewer steps than their unoptimized counterparts because they do not require the advantages of the declarative nature of high-performance with the efficient requirements of full-scale applications. For these reasons, commercial implementations of logic programming languages are configured with a compiler to translate source programs into intermediate languages and an abstract machine to execute the compiled code efficently. Most systems are based on Warren’s Abstract Machine (WAM) [22], first developed for Prolog. The WAM has been adopted to other logic programming languages, such as C-Prolog [7] and Prolog II [3]. However, the WAM itself is more complex and it has been made for other advanced logic programming languages such as LALR [8] and ALPL [9].

Warren’s work appears as a well-engineered construction but, as in a very promising nature, it lacks any logical structure. This can be done by adding the representation of logical structures and meta-interpreter in the WAM. Indeed, the interpretation of the WAM heavily uses meta-structures to be concrete models for logic programming. Proving and source highly specialized in this language. As a result, the WAM is not the natural prune that it might be together as a

---

Carnegie Mellon Qatar
**Uniform proofs** alternate
- Goal decomposition
- Clause decomposition

**Compilation**
- First, *all* preparation
- Then, *all* search

\[ \mathcal{L} \gg \mathcal{L}' \]

\( \mathcal{L}' \) must be an ALPL with *right rules only*
Determining the Target ALPL

- Keep every right rule of $L$
- Add operators that behave on the right like the connectives of $L$ on the left

\[
\frac{\Gamma \rightarrow A \quad \Gamma \rightarrow B \triangleright a}{\Gamma \rightarrow A \supset B \triangleright a}
\]

\[
\frac{\Gamma \rightarrow \forall x. A \triangleright a}{\Gamma \rightarrow [t/x]A \triangleright a}
\]

\[
\frac{\Gamma \rightarrow a \triangleright a}{\Gamma \rightarrow a = a}
\]

- Logical principle
  - Currying
- Parameterize w.r.t. $\triangleright a$
Compiled HHF

\[
G ::= a \mid (\alpha.C) \supset G \mid \forall x. G \\
\Psi ::= . \mid \alpha.C, \Psi
\]

\[
C ::= \alpha = a \mid C \land G \mid \exists x. C
\]

- Non-determinism is preserved
### Compilation

\[
b \subset a_1 \subset \ldots \subset a_n \quad \Rightarrow \quad \alpha. \ (\alpha = b \land a_1 \land \ldots \land a_n)
\]

- Atoms are not touched

<table>
<thead>
<tr>
<th>Formation</th>
<th>Rules</th>
</tr>
</thead>
</table>
| \[a >> a\]                                    | \[\begin{array}{c}
A >> \alpha \setminus \mathcal{C} \\
B >> G
\end{array}\] \[
A \supset B >> \alpha. \mathcal{C} \supset G
\]
| \[\forall x. A >> \forall x. G\]              |                                            |
| \[a >> \alpha \setminus \alpha = a\]         | \[\begin{array}{c}
B >> \alpha \setminus \mathcal{C} \\
A >> G
\end{array}\] \[
A \supset B >> \alpha \setminus \mathcal{C} \land G
\]
| \[\forall x. A >> \forall x. \exists x. G\]   |                                            |
| \[. >> .\]                                    | \[\begin{array}{c}
\Gamma >> \Psi \\
A >> \alpha \setminus \mathcal{C}
\end{array}\] \[
\Gamma, A >> \Psi, \mathcal{C}
\]
Concrete Example

\[ \forall L. \text{append \;} \text{nil \;} L \; L \]
\[ \forall K. \; \forall L. \; \forall M. \; \forall X. \]
\[ \text{append} \; (X::K) \; L \; (X::M) \]
\[ \subseteq \; \text{append} \; K \; L \; M \]
\[ \exists L. \; \alpha = (\text{append \; nil \; L \; L}) \]
\[ \exists K. \; \exists L. \; \exists M. \; \exists X. \]
\[ \alpha = (\text{append} \; (X::K) \; L \; (X::M)) \]
\[ \land \; \text{append} \; K \; L \; M \]
Meta-Theory

Soundness
1. If $\Gamma \rightarrow A$ and $\Gamma \gg \Psi$ and $A \gg G$, then $\Psi \rightarrow G$
2. If $\Gamma \rightarrow A \triangleright a$ and $\Gamma \gg \Psi$ and $A \gg \alpha \setminus C$, then $\Psi \rightarrow [a/\alpha]C$

Proof: structural induction

Completeness
1. If $\Psi \rightarrow G$ and $\Gamma \gg \Psi$ and $A \gg G$, then $\Gamma \rightarrow A$
2. If $\Psi \rightarrow R$ and $R = [a/\alpha]C$ and $\Gamma \gg \Psi$ and $A \gg \alpha \setminus C$, then $\Gamma \rightarrow A \triangleright a$

Proof: structural induction
What is $\alpha$?

- An ad-hoc second-order mechanism
  - Works very well operationally, but
  - what is its logical status?

- Can we engineer a fully logical compilation scheme?
Idea: Use Term-Level Equality

\[\forall y. \ (p \ t \subset a_1 \subset \ldots \subset a_n)\]

\[\downarrow\]

\[\forall x. \ (p \ x \subset \exists y. \ (x=t \land a_1 \land \ldots \land a_n))\]

- This is currying again, but respecting dependencies
- Compiled head always matches goal for \(p\)
- Compiled clauses have the form \(\forall x. \ (p \ x \subset R)\)
Compiled HHF (2)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Psi, C \rightarrow C \supset a )</td>
<td>( \Psi, C \rightarrow a ) *</td>
</tr>
<tr>
<td>( \Psi, C \rightarrow G )</td>
<td>( \Psi \rightarrow C \supset G )</td>
</tr>
<tr>
<td>( c ) “new” ( \Psi \rightarrow [c/x]G )</td>
<td>( \Psi \rightarrow \forall x. G )</td>
</tr>
<tr>
<td>( \Psi \rightarrow R )</td>
<td>( \Psi \rightarrow \forall x. C \supset a ) *</td>
</tr>
<tr>
<td>( \Psi \rightarrow R \supset a \supset a )</td>
<td>( \Psi \rightarrow t = t )</td>
</tr>
<tr>
<td>( \Psi \rightarrow [t/x]C \supset a )</td>
<td>( \Psi \rightarrow R ) ( \Psi \rightarrow G )</td>
</tr>
<tr>
<td>( \Psi \rightarrow R \land G )</td>
<td>( \Psi \rightarrow [t/x]R )</td>
</tr>
<tr>
<td>( \Psi \rightarrow \exists x. R )</td>
<td></td>
</tr>
</tbody>
</table>

- Non-determinism is still preserved
- Minor infrastructure to produce \( x = t \)
- Remains sound and complete
Macro-Rule

• Builds uniform proofs
  ▪ Necessary sequence of steps to use compiled clause
    \[ \Psi, \forall x.(p x \subset R) \rightarrow [t/x]R \]
    \[ \Psi , \forall x.(p x \subset R) \rightarrow p t \]
  ▪ The backchaining rule
  ▪ View \( \forall x.(p x \subset R) \) as synthetic connective \( \Lambda_p x . R \)
    ▪ That’s our old \( \alpha \)
Concrete Example (2)

∀K. ∀L. ∀M. ∀X.
append (X::K) L (X::M)
⊂ append K L M

∀x1. ∀x2. ∀x3.
append x1 x2 x3
⊂ ∃K. ∃L. ∃M. ∃X.
(x1 = (X::K) ∧
x2 = L ∧
x3 = (X::M) ∧
append K L M)

∀L. append nil L L

∀x1. ∀x2. ∀x3.
append x1 x2 x3
⊂ ∃L. (x1 = nil ∧
x2 = L ∧
x3 = L)
Moded Programs

- Arguments are labeled as input or output
  - Input are ground at call time
  - Output made ground upon return
  - Simple static check

- Moded semantics is based on matching not unification
  - Faster for first-order terms (no occurs-check)
  - Decidable for higher-order term

- Sufficient for CLF
Moded Execution

\[ \forall y. (p \sqsubseteq q_1 \sqsubset \ldots \sqsubset q_n) \]

\[ \forall x. (p \sqsubseteq \exists y. (x=t \land \ldots \land q_n)) \]

- \textbf{\( x=t \)} matches input
- \textbf{\( t=x \)} assigns output
Moded Execution

- $\vdash$: matching operator – for well-moded programs
- $\leftarrow$: assignment operator

Data flow

Control flow

Generic Horn clause

\[
\forall y. (p t t \sqsubseteq q_1 t_1 t_1 \sqsubseteq \ldots \sqsubseteq q_n t_n t_n)
\]

\[
\forall x x. (p x x \sqsubseteq \exists y. (x=:\!\!\!\!\!\!\!\!t \land q_1 t_1 t_1 \land \ldots \land q_n t_n t_n \land x=:\!\!\!\!\!\!\!\!t))
\]
Moded Atomic Goals

- Solving an atomic goal is like a function call
  - Non-deterministic partial function

\[ \exists z. (q t z \land z = :t) \]
Compiled HHF (3)

- Non-determinism is still preserved
- Additional compilation infrastructure needed
- Remains sound and complete

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rule</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Psi, C \rightarrow C \supset a )</td>
<td>( \Psi, C \rightarrow G )</td>
<td>( c ) <em>new</em> ( \Psi \rightarrow [c/x]G )</td>
</tr>
<tr>
<td>( \Psi, C \rightarrow a )</td>
<td>( \Psi \rightarrow C \supset G )</td>
<td>( \Psi \rightarrow \forall x. G )</td>
</tr>
<tr>
<td>( \Psi \rightarrow R )</td>
<td>( \Psi \rightarrow [t/x]C \supset a )</td>
<td>( \Psi \rightarrow \forall x. C \supset a )</td>
</tr>
<tr>
<td>( \Psi \rightarrow R \supset a \supset a )</td>
<td>( \Psi \rightarrow R \bigwedge G )</td>
<td>( \Psi \rightarrow [t/x]R )</td>
</tr>
<tr>
<td>( \Psi \rightarrow t := t )</td>
<td>( \Psi \rightarrow \exists x. R )</td>
<td>( \Psi \rightarrow [t/x]R )</td>
</tr>
<tr>
<td>( \Psi \rightarrow t := t )</td>
<td>( \Psi \rightarrow G )</td>
<td></td>
</tr>
</tbody>
</table>
Consequences of Uniformity

- Two macro-rules
  \[ \Psi, \forall x \in \exists y. (R \land x := s) \rightarrow p_t[u/y]s \]
  \[ \Psi \rightarrow p_t s \]
  \[ \Psi \rightarrow \exists z. (p_t z \land z =: s) \]

- Two synthetic connectives
  - \[ \forall x \in \exists y. (R \land x := s) \text{ as } \Lambda_p x \cdot \exists y. (R ; \text{return } s) \]
  - \[ \exists z. (p_t z \land z =: s) \text{ as } \text{call } p_t =: s \]
Concrete Example (3)

∀K. ∀L. ∀M. ∀X.
append (X::K) L (X::M)
⊂ append K L M

∀x₁. ∀x₂. ∀x₃.
append x₁ x₂ x₃
⊂ ∃K. ∃L. ∃M. ∃X.
(x₁ =: (X::K) ∧
x₂ =: L ∧
∃z. (append K L z ∧ z =: M) ∧
x₃ := (X::M))

∀L. append nil L L
Future Work

• Prove that matching/assignment are sufficient for moded programs
  ▪ Make goal & term selection explicit
  ▪ Declarative mode checking
  ▪ Semi-functional interpretation

• Implement within CLF prototype
  ▪ Backward + forward chaining semantics
  ▪ Linear/affine operators
  ▪ Complex higher-order term language
Thank you!

Questions?