Proof-Theoretic Foundation of Compilation in Logic Programming Languages

Iliano Cervesato
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
iliano@cs.stanford.edu

Abstract

Commercial implementations of logic programming languages are engineered around a compiler based on Warren's Abstract Machine (WAM) or a variant of it. In spite of various correctness proofs, the logical machinery relating the proof-theoretic specification of a logic programming language and its compiled form is still poorly understood. In this paper, we propose a logic-independent definition of compilation for logic programming languages. We apply this methodology to derive the first cut of a compiler and the corresponding abstract machine for the language of hereditary Harrop formulas and then for its linear refinement.

1 Introduction

Compiled logic programs run over an order of magnitude faster than their interpreted source and constitute therefore a key step to combining the advantages of the declarative nature of logic programming with the efficiency requirements of full-scale applications. For this reason, commercial implementations of logic programming languages come equipped with a compiler to translate a source program into an intermediate language, and an abstract machine to execute this compiled code efficiently. Most systems are based on *Warren's Abstract Machine* (WAM) [1, 22], first developed for *Prolog*. The WAM has now been adapted to other logic programming languages such as *CLP*($\mathcal{R}$) [10] and *PROTOS-L* [2]. Extensions to λ*Prolog* [14] are under way [12, 16, 17], but no similar effort has been undertaken for other advanced logic programming languages such as *Lolli* [9] or *Elf* [20].

Warren's work appears as a carefully engineered construction, but, for its very pioneering nature, it lacks any logical status. This contrasts strongly with the deep roots that the interpretation semantics of logic programming has in logic and proof-theory [15]. Indeed, the instruction set of the WAM hardly bears any resemblance to the connectives of the logic underlying *Prolog* and seems highly specialized to this language. As a result, the WAM "resembles an intricate puzzle, whose many pieces fit tightly together in a

Appeared in the *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming — JICSLP'98* (J. Jaffar editor), pp ??–??, MIT Press, Manchester, UK, 16–19 June 1998.

miraculous way" [3], understanding it is complex in spite of the availability of excellent presentations [1], and designing WAM-like systems for other languages is a major engineering project. Several authors have proved the correctness of the WAM with respect to specifications of *Prolog*'s interpretation semantic [3, 21]. However, these results shed very little light on the logical reading of the WAM since they start from highly procedural presentations of the semantic of Horn clauses, as SLD-resolution for example. Adaptations of these proofs to the aforementioned extensions of the WAM to other logic programming languages [2, 4] suffer from the same problem. Therefore, differently from the case of functional programming for example [13, 18], there is no logical account of compilation for logic programs.

In this paper, we give a general definition of compilation that parallels and extends the notion of abstract logic programming language [15]. More specifically, we propose a proof-theoretic characterization of the compilation process based on the duality between left and right rules in a sequent calculus presentation of a logic. In doing so, we do not commit to any particular logic but consider any formalism whose proof theory obeys a minimal set of properties [15]. We use the logic of hereditary Harrop formulas [15] and its linear variant [9] as examples. The high-level rules of an abstract logic programming language leave several implementation choices open, such as the order in which conjunctive goals should be solved and how to instantiate variables. The same will happen in the compiled language. However, similarly to the case of abstract logic programming languages, our abstract compilation scheme can be refined to adopt specific strategies (e.g. left-to-right subgoal selection rule and unification). Modularity is achieved in this way.

The main contributions of this paper are: (1) the individuation of the logic underlying the WAM and a logical justification of the compilation process, (2) the definition of an abstract, logic-independent and modular notion of compilation for logic programming languages, and possibly (3) the first steps toward a logic-based theory of compilation for logic programming languages. We also hope that our approach will help make compilation an integral part of the design of new logic programming languages rather than the collateral engineering task it is now.

This paper is organized as follows. In Section 2, we recall the definition of abstract logic programming language and introduce the logic of hereditary Harrop formulas as an example. We describe our abstract notion of compilation in Section 3, apply it to our case study and prove correctness results. In Section 4, we further exemplify our approach on a logic programming language based on linear logic. Section 5 outlines directions of future work.

2 Abstract Logic Programming Languages

Computation in logic programming is achieved through proof search. Given a *goal* A to be proved in a *program* Γ, we want to be able to interpret the connectives of A as *search directives* and the *clauses* in Γ as specifications

Uniform provability

$$\frac{\Gamma, A, \Gamma' \xrightarrow{u} A \gg a}{\Gamma, A, \Gamma' \xrightarrow{u} a}\;\textbf{u\_Atom} \qquad \frac{}{\Gamma \xrightarrow{u} \mathbf{t}}\;\textbf{u\_True} \qquad \frac{\Gamma \xrightarrow{u} A_1 \quad \Gamma \xrightarrow{u} A_2}{\Gamma \xrightarrow{u} A_1 \wedge A_2}\;\textbf{u\_And}$$

$$\frac{\Gamma, A_1 \xrightarrow{u} A_2}{\Gamma \xrightarrow{u} A_1 \supset A_2}\;\textbf{u\_Imp} \qquad \frac{c \text{ ``}\textit{new}\text{''} \quad \Gamma \xrightarrow{u} [c/x]A}{\Gamma \xrightarrow{u} \forall x.\, A}\;\textbf{u\_Forall}$$

Immediate entailment

$$\frac{}{\Gamma \xrightarrow{u} a \gg a}\;\textbf{i\_Atom} \qquad \text{(No rule for } \mathbf{t}\text{)}$$

$$\frac{\Gamma \xrightarrow{u} A_1 \gg a}{\Gamma \xrightarrow{u} A_1 \wedge A_2 \gg a}\;\textbf{i\_And}_\mathbf{1} \qquad \frac{\Gamma \xrightarrow{u} A_2 \gg a}{\Gamma \xrightarrow{u} A_1 \wedge A_2 \gg a}\;\textbf{i\_And}_\mathbf{2}$$

$$\frac{\Gamma \xrightarrow{u} A_1 \gg a \quad \Gamma \xrightarrow{u} A_2}{\Gamma \xrightarrow{u} A_2 \supset A_1 \gg a}\;\textbf{i\_Imp} \qquad \frac{\Gamma \xrightarrow{u} [t/x]A \gg a}{\Gamma \xrightarrow{u} \forall x.\, A \gg a}\;\textbf{i\_Forall}$$

Figure 1: Uniform Deduction System for $\mathcal{L}_{HH}$.

of how to continue the search when the goal is atomic. These desiderata are given without mentioning the logic the program and the goal belong to.

A proof in any logic is *goal-oriented* if every compound goal is immediately decomposed and the program is accessed only after the goal has been reduced to an atomic formula. A proof is *focused* if every time a program formula is considered, it is processed up to the atoms it defines without need to access any other program formula. A proof having both these properties is *uniform*, and a formalism such that every provable goal has a uniform proof is called an *abstract logic programming language*, abbreviated *ALPL* [15]. An ALPL is conveniently described as a pair $(\mathcal{L}, \longrightarrow)$ consisting of a language $\mathcal{L}$ and an associated notion of (uniform) derivability $\longrightarrow$.

The language of *hereditary Harrop* formulas, abbreviated $\mathcal{L}_{HH}$, is the largest freely generated fragment of intuitionistic logic that passes the above ALPL test [15]. It is the logic underlying the programming language λ*Prolog* [14] and embeds the language of Horn clauses on which *Prolog* is based. Formulas and programs are defined by means of the following grammar:

Formulas: $A ::= a \mid \mathbf{t} \mid A_1 \wedge A_2 \mid A_1 \supset A_2 \mid \forall x.\, A$
Programs: $\Gamma ::= \cdot \mid \Gamma, A$

where a ranges over atomic formulas. We shall treat programs as sequences and omit the leading "$\cdot$" whenever a program is not empty. Moreover, we write t to indicate terms and denote the capture avoiding substitution of a term t for x in a formula A as $[t/x]A$. Notice that we do not specify the underlying language of terms. This is irrelevant to our development: it can be first-order, higher-order as in λ*Prolog*, or be based on any equational theory (which would yield corresponding *constraint logic programming languages*). Propositional $\mathcal{L}_{HH}$ can be handled as well once we omit the quantifiers $\forall x.\, A$. However, we do not admit quantification over atomic formulas.

Syntactically richer definitions of the logic of hereditary Harrop formulas exist [14, 15]. They are obtained by allowing the language of programs to be different from the language of goals. The former differs in minor points from the grammar displayed above. The latter is extended with numerous other logical operators. It is not a coincidence that these additional constructs correspond to the connectives and quantifiers into which program formulas will be compiled in the next section. This syntactic extension does not however alter the expressiveness of this logic.

Figure 1 displays the traditional sequent calculus rules for $\mathcal{L}_{HH}$ written in such a way that every proof is uniform. The specification of the *uniform provability* judgment, written $\Gamma \xrightarrow{u} A$, includes all the right sequent rules for this logic and calls the *immediate entailment* judgment $\Gamma \xrightarrow{u} A \gg a$ when the goal is atomic. Immediate entailment isolates a clause and decomposes it as prescribed by the left sequent rules of $\mathcal{L}_{HH}$ until an axiom applies. The cut-elimination theorem and simple permutations of inference rules suffice to show that any sequent derivation can be transformed into a uniform proof that uses only these rules [15]. Therefore $\mathcal{L}_{HH}$ is an ALPL.

The rules in Figure 1 present three main forms of non-determinism. First, the order in which the two premises of rules **u_And** and **i_Imp** should be proved is left unspecified (*conjunctive non-determinism*). Second, no criterion is given to select the program formula in rule **i_Atom** nor to choose among rules **i_And$_1$** and **i_And$_2$** (*disjunctive non-determinism*). Third, no strategy is specified to construct the term t in rule **i_Forall** (*existential non-determinism*). A concrete implementation must resolve these points.

3 Abstract Compilation

The search for a uniform proof consists of the alternation of two distinct phases: *goal decomposition*, where right sequent rules are applied according to the structure of the goal, and, once the goal is atomic, *clause decomposition* where left sequent rules are systematically used on the selected formula in the program. These two operational modes are clearly distinguished in $\mathcal{L}_{HH}$ by means of the two forms of judgments that describe its semantics (Figure 1). From a logic programming perspective, the connectives appearing in the goal are search directives and therefore goal decomposition is where the bulk of the action takes place. Clause decomposition can instead be viewed as a preparatory phase.

The objective of compilation is to separate these two phases so that all the preparatory steps are performed *before* any search begins [11]. An *abstract logic programming compilation system*, *ALPCS* for short, consists therefore of a *source ALPL* $(\mathcal{L}, \longrightarrow)$, an *intermediate language* $\mathcal{L}^c$ which is itself an ALPL with the characteristic that its notion of derivability $\xrightarrow{c}$ consists of right sequent rules only, and a *compilation function* $\gg$ that maps programs and goals in $\mathcal{L}$ to *compiled programs* and *compiled goals* in $\mathcal{L}^c$, respectively.

The intermediate ALPL $(\mathcal{L}^c, \xrightarrow{c})$ and the compilation function can be determined systematically from the source language and its derivability rules. First, since provable goals in the source language should have a derivation in the intermediate ALPL, the goal language of $\mathcal{L}$ must be a sublanguage of $\mathcal{L}^c$.

We get rid of every left-introduction rule $\mathbf{r}$ of $\longrightarrow$ by devising a dual rule $\mathbf{r}^c$ with an identical structure, but that operates on the right of the sequent arrow. In particular, the number of premises, the way other clauses are propagated to them, and possible interactions of the selected formula with the atomic goal of $\mathbf{r}$ should be preserved in $\mathbf{r}^c$. We should clearly rename the connective or quantifier introduced in $\mathbf{r}$ in order not to alter its search semantics. Although systematic, the process of transforming a left rule into an equivalent right inference figure may appear to step out of logic since we are required to invent new syntactic constructs. However, the source logic $(\mathcal{L}, \longrightarrow)$ is almost invariably a fragment of some larger logic (generally not an ALPL) that exhibits the very same duality principles we just described, but in a native form. Therefore, it will suffice to adopt these operators as the dual constructs of the program connectives of the source language, giving in this way a strong logical reading to the intermediate language. It is easily seen that this process preserves points of non-determinism.

The last question to answer is what to do when the search in $\mathcal{L}^c$ produces an atomic goal a. As in any ALPL, we want to consult the (compiled) program for indications on how to continue the search. Since the appropriateness of a program formula for this purpose should depend on a, we perform a *parametric compilation* of program formulas that keeps the atom they should resolve unspecified. Then, when an atomic formula a is produced, the computation proceeds by picking a (parametric) compiled clause, instantiating it with a and processing it as a goal.

We will now apply the methodology we just informally described to the language of hereditary Harrop formulas, $\mathcal{L}_{HH}$, discussed in Section 2. In order to construct the associated intermediate ALPL $(\mathcal{L}^c_{HH}, \xrightarrow{c})$, we are going to look for connectives in general intuitionistic logic whose right rules are the dual of the immediate entailment rules from Figure 1.

The easiest operator to process in this way is $\mathbf{t}$, which has no immediate entailment rule. The only construct in intuitionistic logic that does not have a right sequent rule is falsehood, $\mathbf{f}$. We adopt it as the dual of $\mathbf{t}$.

Next, we consider conjunction. This connective has two left introduction rules that differ only by the conjunct that is kept when reading them from the conclusion to the premises. The only construct whose right sequent rules behave in this manner is disjunction ($\vee$), as we can clearly see by placing corresponding rules next to each other:

$$\frac{\Gamma \xrightarrow{u} A_i \gg a}{\Gamma \xrightarrow{u} A_1 \wedge A_2 \gg a}\ \mathbf{i\_And}_i \qquad\qquad \frac{\Gamma \longrightarrow A_i}{\Gamma \longrightarrow A_1 \vee A_2}$$

for $i = 1, 2$. Therefore, we use disjunction to compile a conjunction appearing in a program position.

There is one immediate entailment rule for implication, it has two premises which each process one of its two subformulas. The right sequent rule for conjunction is structured in a similar way:

$$\frac{\Gamma \xrightarrow{u} A_1 \gg a \quad \Gamma \xrightarrow{u} A_2}{\Gamma \xrightarrow{u} A_2 \supset A_1 \gg a}\,\textbf{i\_Imp} \qquad \frac{\Gamma \longrightarrow A_1 \quad \Gamma \longrightarrow A_2}{\Gamma \longrightarrow A_1 \wedge A_2}$$

Consequently, we compile occurrences of implication in a program position as conjunctions. Notice that the left premise of rule **i_Imp** prescribes further clause decomposition, while its right premise sets up a goal for search. This distinction does not appear in the right introduction rule for $\wedge$. However, this is exactly what we are after since we want to be able to process every compiled clause as a goal, after instantiation.

The immediate entailment rule for the universal quantifier instantiates the embedded clause with some term t for further decomposition. The existential quantifier $\exists$ performs the same step, but on the right of the sequent arrow:

$$\frac{\Gamma \xrightarrow{u} [t/x]A \gg a}{\Gamma \xrightarrow{u} \forall x.\, A \gg a}\,\textbf{i\_Forall} \qquad \frac{\Gamma \longrightarrow [t/x]A}{\Gamma \longrightarrow \exists x.\, A}$$

We therefore compile each clause occurrence of $\forall$ to an existential quantifier in a goal position.

The only remaining immediate entailment rule is **i_Atom**, which applies once the selected clause has been decomposed to an atomic formula. Then, it checks whether this atom is the same as the (atomic) goal. We are therefore looking for a binary connective operating on atomic formulas and whose only right introduction rule has no premise. No construct in intuitionistic logic behaves in this way. We can however enrich this formalism with a logical operator that checks that two atomic formulas are syntactically equal. We denote syntactic equality has $\doteq$. The rule defining its semantics is displayed below on the right:

$$\frac{}{\Gamma \xrightarrow{u} a \gg a}\,\textbf{i\_Atom} \qquad \frac{}{\Gamma \longrightarrow a \doteq a}$$

Then, we compile program clauses parametrically with respect to the atomic goal they are expected to match. In particular, we associate to an atomic clause a the compiled goal $a \doteq \alpha$, where α is a *parameter*. When a generic clause is used to continue the search for some atomic goal a', we replace its parameter α with a' and use the resulting expression as a goal.

We are now in a position to bundle up this intuition into a formal definition. The intermediate language corresponding to $\mathcal{L}_{HH}$, that we denote $\mathcal{L}^c_{HH}$, is given by the following grammar:

$$\begin{array}{rrcl}
\textit{Goals:} & G & ::= & a \mid \mathbf{t} \mid G_1 \wedge G_2 \mid (\Lambda\alpha.\, C) \supset G \mid \forall x.\, G \\
\textit{C-Goals} & R & ::= & a_1 \doteq a_2 \mid \mathbf{f} \mid R_1 \vee R_2 \mid R \wedge G \mid \exists x.\, R \\
\textit{Clauses:} & C & ::= & a \doteq \alpha \mid \mathbf{f} \mid C_1 \vee C_2 \mid C \wedge G \mid \exists x.\, C \\
\textit{Programs:} & \Psi & ::= & \cdot \mid \Psi, \Lambda\alpha.\, C
\end{array}$$

In order to make the discussion fully precise, we distinguish the *goals* inherited from $\mathcal{L}_{HH}$, that we denote with the letter G, from the goal constructions

Goals

$$\frac{\Psi, \Lambda\alpha.\, C, \Psi' \xrightarrow{cg} [a/\alpha]C}{\Psi, \Lambda\alpha.\, C, \Psi' \xrightarrow{c} a}\ \textbf{c\_Atom} \qquad \frac{}{\Psi \xrightarrow{c} \mathbf{t}}\ \textbf{c\_True} \qquad \frac{\Psi \xrightarrow{c} G_1 \quad \Psi \xrightarrow{c} G_2}{\Psi \xrightarrow{c} G_1 \wedge G_2}\ \textbf{c\_And}$$

$$\frac{\Psi, \Lambda\alpha.\, C \xrightarrow{c} G}{\Psi \xrightarrow{c} (\Lambda\alpha.\, C) \supset G}\ \textbf{c\_Imp} \qquad \frac{c \text{ ``\textit{new}''} \quad \Psi \xrightarrow{c} [c/x]G}{\Psi \xrightarrow{c} \forall x.\, G}\ \textbf{c\_Forall}$$

Compilation goals

$$\frac{}{\Psi \xrightarrow{cg} a \doteq a}\ \textbf{cg\_Eq} \qquad \text{(No rule for } \mathbf{f}\text{)}$$

$$\frac{\Psi \xrightarrow{cg} R_1}{\Psi \xrightarrow{cg} R_1 \vee R_2}\ \textbf{cg\_Or}_\mathbf{1} \qquad \frac{\Psi \xrightarrow{cg} R_2}{\Psi \xrightarrow{cg} R_1 \vee R_2}\ \textbf{cg\_Or}_\mathbf{2}$$

$$\frac{\Psi \xrightarrow{cg} R \quad \Psi \xrightarrow{c} G}{\Psi \xrightarrow{cg} R \wedge G}\ \textbf{cg\_And} \qquad \frac{\Psi \xrightarrow{cg} [t/x]R}{\Psi \xrightarrow{cg} \exists x.\, R}\ \textbf{cg\_Exists}$$

Figure 2: Search Semantics of $\mathcal{L}^c_{HH}$.

resulting from the compilation of a program formula. We call them *compilation goals*, or *c-goals*, and use the letter R for them.

We write $\Lambda\alpha.\, C$ to denote a *compiled clause* with parameter α. We view Λ as an operator that binds every free occurrence of α in C. When using this notation, we shall assume that C does not contain free parameters besides α so that $\Lambda\alpha.\, C$ is a closed expression. Moreover, we require every clause C to mention at most one free parameter. We denote the instantiation of a clause C with parameter α by means of an atomic formula a as $[a/\alpha]C$. A c-goal is then a clause whose (free) parameter has been replaced with an atomic formula (we omit the formal statement and its proof).

The antecedent of an implication goal is a compiled clause since solving the overall goal has the effect of extending the program with this subformula as an extra clause. Observe also that the right conjunct of a c-goal $R \wedge G$ is a goal. This is a consequence of the structure of rule **i_Imp**.

We report the inference rules defining the semantics of $\mathcal{L}^c_{HH}$ in Figure 2. Notice that only right introduction rules are mentioned. In order to achieve simple proofs of soundness and completeness, we distinguish the rules for goals from those applying to c-goals by means of the judgments $\Psi \xrightarrow{c} G$ and $\Psi \xrightarrow{cg} R$, respectively. Notice that they present the same non-deterministic behaviors as the rules for $\mathcal{L}_{HH}$ in Figure 1.

The compilation of the various linguistic entities of $\mathcal{L}_{HH}$ into $\mathcal{L}^c_{HH}$ is presented as a deductive system in Figure 3. Since clauses contain embedded goals and vice-versa, the judgments compiling these expressions, $A \gg \alpha \setminus C$ and $A \gg G$, have a mutually recursive definition. Compiling a program, denoted $\Gamma \gg \Psi$, reduces to compiling its clauses and collecting them. It can be easily shown that the compilation rules in Figure 3 specify a bijective function between $\mathcal{L}_{HH}$ and the fragment of $\mathcal{L}^c_{HH}$ that satisfies our restrictions.

Programs

$$\frac{}{\cdot \gg \cdot}\;\textbf{pc\_empty} \qquad \frac{\Gamma \gg \Psi \quad A \gg \alpha \setminus C}{\Gamma, A \gg \Psi, \Lambda\alpha.\, C}\;\textbf{pc\_clause}$$

Clauses

$$\frac{}{a \gg \alpha \setminus a \doteq \alpha}\;\textbf{cc\_Atom} \qquad \frac{}{\mathbf{t} \gg \alpha \setminus \mathbf{f}}\;\textbf{cc\_True} \qquad \frac{A \gg \alpha \setminus C_1 \quad B \gg \alpha \setminus C_2}{A \wedge B \gg \alpha \setminus C_1 \vee C_2}\;\textbf{cc\_And}$$

$$\frac{B \gg \alpha \setminus C \quad A \gg G}{A \supset B \gg \alpha \setminus C \wedge G}\;\textbf{cc\_Imp} \qquad \frac{A \gg \alpha \setminus C}{\forall x.\, A \gg \alpha \setminus \exists x.\, C}\;\textbf{cc\_Forall}$$

Goals

$$\frac{}{a \gg a}\;\textbf{gc\_Atom} \qquad \frac{}{\mathbf{t} \gg \mathbf{t}}\;\textbf{gc\_True} \qquad \frac{A \gg G_1 \quad B \gg G_2}{A \wedge B \gg G_1 \wedge G_2}\;\textbf{gc\_And}$$

$$\frac{A \gg \alpha \setminus C \quad B \gg G}{A \supset B \gg (\Lambda\alpha.\, C) \supset G}\;\textbf{gc\_Imp} \qquad \frac{A \gg C}{\forall x.\, A \gg \forall x.\, C}\;\textbf{gc\_Forall}$$

Figure 3: Compilation of $\mathcal{L}_{HH}$.

As an example, we consider the familiar (Horn) clauses computing the concatenation (or non-deterministic splitting) of two lists. We write "nil" for the empty list and $x :: l$ for a list with head x and tail l. For the sake of readability, we use $A_2 :- A_1$ as an alternate notation for $A_1 \supset A_2$.

$$\forall l.\, \text{append nil}\, l\, l$$
$$\forall k.\, \forall l.\, \forall m.\, \forall x.\, \text{append}\, (x :: k)\, l\, (x :: m) \;\; :- \text{append}\, k\, l\, m$$

Applying the compilation process we just described yields the following compiled clauses.

$$\Lambda\alpha.\, \exists l.\, (\text{append nil}\, l\, l) \doteq \alpha$$
$$\Lambda\alpha.\, \exists k.\, \exists l.\, \exists m.\, \exists x.\, (\text{append}\, (x :: k)\, l\, (x :: m)) \doteq \alpha \;\; \wedge \;\; (\text{append}\, k\, l\, m)$$

We can procedurally interpret the equality test in any compiled clause as a request to unify the head of the original clause ("append nil l l" for example), with the *current* (atomic) goal represented by the parameter α: in an unoptimized WAM, α corresponds to the register ($\mathtt{A_1}$) holding the atomic formula being called. The conjunction introduced in the second clause can similarly be interpreted as a sequencing operator. Therefore, an operational reading of this clause specifies to allocate space for the local variables k, l, m and x, unify the current goal with "append $(x :: k)$ l $(x :: m)$" and then call "append k l m" in case of success.

We have written a compiler and the corresponding abstract machine along the guidelines in this section for the language *LF* [19], a type theory closely related to $\mathcal{L}_{HH}$. By choosing appropriate names and spacing in order to print compiled clauses and no further processing, we could obtain the following WAM-like pseudo-code for the "append" program.

```
append1: ALLOCATE L
         UNIFY    append nil L L
```

```
append2: ALLOCATE K
         ALLOCATE L
         ALLOCATE M
         ALLOCATE X
         UNIFY    append (cons X K) L (cons X M)
         CALL     append K L M
```

The "real" WAM program is much more detailed since it provides code to handle the various forms of non-determinism we discussed in Section 2. However, its high-level structure is the same.

Compilation for first- and higher-order hereditary Harrop formulas has been extensively studied by Nadathur *et al.* [12, 16, 17] with the aim of building an abstract machine for λ*Prolog* [14]. Their work has focused on devising and optimizing a concrete WAM-like instruction set for this language.

We conclude this section by proving the soundness and completeness of the compilation process for $\mathcal{L}_{HH}$. An important fact to notice is the elegance and simplicity of these proofs (whose details are omitted for space reasons): they rely on nothing more than structural inductions. This contrasts greatly with the complexity of the hefty proofs of soundness and correctness currently available for the WAM [3, 21]. We do not believe that this huge difference depends solely on the fact that we are treating the compilation process at a very high abstraction level. We have furthermore transcribed our proofs in the *Elf* [20] implementation of the logical framework *LF* [19] and let this system check their validity. This contrasts again with the proofs in [3, 21], which have not been mechanically checked and should therefore be considered informal.

The soundness theorem states that whenever a goal is derivable from a program in $\mathcal{L}_{HH}$, then the compiled goal is derivable from the compiled program in $\mathcal{L}^c_{HH}$. A close analysis of the proof reveals that not only derivability but also actual derivations are preserved, so that it is possible to extract a function that maps derivations in the source ALPL to derivations in $\mathcal{L}^c_{HH}$.

Theorem 1 (*Soundness of the compilation of* $\mathcal{L}_{\mathrm{HH}}$)

i. If $\Gamma \xrightarrow{u} A,\ \Gamma \gg \Psi$ *and* $A \gg G$*, then* $\Psi \xrightarrow{c} G.$

ii. If $\Gamma \xrightarrow{u} A \gg a,\ \Gamma \gg \Psi$ *and* $A \gg \alpha \setminus C$*, then* $\Psi \xrightarrow{cg} [a/\alpha]C.$

Proof. By simultaneous induction of the structure of the given derivations. ☑

Completeness implies that it is not the case that a compiled goal is provable when the corresponding source goal is not. Again, the proof is constructive and a function mapping derivations in $(\mathcal{L}^c_{HH}, \xrightarrow{c})$ to derivations in $(\mathcal{L}_{HH}, \xrightarrow{u})$ can easily be extracted.

Theorem 2 (*Completeness of the compilation of* $\mathcal{L}_{\mathrm{HH}}$)

i. If $\Psi \xrightarrow{c} G,\ \Gamma \gg \Psi$ *and* $A \gg G$*, then* $\Gamma \xrightarrow{u} A.$

Uniform provability

$$\frac{\Delta \overset{?}{;} \Delta' \xrightarrow{u} A \gg a}{\Delta \hat{;} A \overset{?}{;} \Delta' \xrightarrow{u} a}\,\textbf{lu\_LinAtom} \qquad \frac{\Delta, A \overset{?}{;} \Delta' \xrightarrow{u} A \gg a}{\Delta, A \overset{?}{;} \Delta' \xrightarrow{u} a}\,\textbf{lu\_IntAtom}$$

$$\frac{}{\Delta \xrightarrow{u} \top}\,\textbf{lu\_Top} \qquad \frac{\Delta \xrightarrow{u} A_1 \quad \Delta \xrightarrow{u} A_2}{\Delta \xrightarrow{u} A_1 \,\&\, B_2}\,\textbf{lu\_With}$$

$$\frac{\Delta \hat{;} A_1 \xrightarrow{u} A_2}{\Delta \xrightarrow{u} A_1 \multimap A_2}\,\textbf{lu\_Lolli} \qquad \frac{\Delta, A_1 \xrightarrow{u} A_2}{\Delta \xrightarrow{u} A_1 \supset A_2}\,\textbf{lu\_Imp}$$

Immediate entailment

$$\frac{}{!\Delta \xrightarrow{u} a \gg a}\,\textbf{li\_Atom} \qquad \text{(No rule for } \top\text{)}$$

$$\frac{\Delta \xrightarrow{u} A_1 \gg a}{\Delta \xrightarrow{u} A_1 \,\&\, A_2 \gg a}\,\textbf{li\_With}_\mathbf{1} \qquad \frac{\Delta \xrightarrow{u} A_2 \gg a}{\Delta \xrightarrow{u} A_1 \,\&\, A_2 \gg a}\,\textbf{li\_With}_\mathbf{2}$$

$$\frac{\Delta_1 \xrightarrow{u} A_1 \gg a \quad \Delta_2 \xrightarrow{u} A_2}{\Delta_1 \bowtie \Delta_2 \xrightarrow{u} A_2 \multimap A_1 \gg a}\,\textbf{li\_Lolli} \qquad \frac{\Delta \xrightarrow{u} A_1 \gg a \quad !\Delta \xrightarrow{u} A_2}{\Delta \xrightarrow{u} A_2 \supset A_1 \gg a}\,\textbf{li\_Imp}$$

Figure 4: Uniform Deduction System for $\mathcal{L}_{LHH}$.

ii. If $\Psi \xrightarrow{cg} R$, $\Gamma \gg \Psi$, $R = [a/\alpha]C$ *and* $A \gg \alpha \setminus C$, *then* $\Gamma \xrightarrow{u} A \gg a$.

Proof. By mutual induction on the structure of the given derivations. ☑

4 Adding Linearity

The language of *linear hereditary Harrop* formulas [9], denoted $\mathcal{L}_{LHH}$, is the largest freely-generated fragment of intuitionistic linear logic with the properties satisfying the definition of an ALPL. Its operators are similar to the constructs of $\mathcal{L}_{HH}$ which can be viewed as a fragment of $\mathcal{L}_{LHH}$, modulo a simple encoding. $\mathcal{L}_{LHH}$ is the logic underlying the logic programming language *Lolli* [9]. The syntax of $\mathcal{L}_{LHH}$ is given by the following grammar.

$$\begin{array}{rl} \textit{Formulas:} & A \;::=\; a \mid \top \mid A_1 \,\&\, A_2 \mid A_1 \multimap A_2 \mid A_1 \supset A_2 \mid \forall x.\, A \\ \textit{Contexts:} & \Delta \;::=\; \cdot \mid \Delta, A \mid \Delta \hat{;} A \end{array}$$

As for $\mathcal{L}_{HH}$, we shall consider this definition independent from the language of terms. The assumptions contained in a context Δ are now distinguished in *linear* and *intuitionistic*; they are introduced by the constructors "$\hat{;}$" and ",", respectively. We write "$\overset{?}{;}$" as an abbreviation of "$\hat{;}$" or ",". We moreover write $!\Delta$ for a context from which all linear assumptions have been deleted, and $\Delta_1 \bowtie \Delta_2$ the result of the order-preserving merge of the linear assumptions in the contexts Δ_1 and Δ_2, which are assumed to contain identical intuitionistic assumptions [6]. We will omit discussing the universal quantifier since it is handled as in the non-linear case.

Figure 4 displays a deduction system for $\mathcal{L}_{LHH}$ set up in such a way that only uniform derivations are constructed. The involved judgments are similar to the non-linear case. A proof that $\mathcal{L}_{LHH}$ is an ALPL can be found in [9]. This result shows that a presentation similar to ours is sound and complete with respect to the usual rules for intuitionistic linear logic, as soon as $A_1 \supset A_2$ is expanded as $!A_1 \multimap A_2$ and every intuitionistic assumption A is replaced with $!A$.

Observe that rule **li_Lolli** displays a form of non-determinism not present in $\mathcal{L}_{HH}$: in order to implement this language, we must provide a strategy to partition the linear assumptions in the context Δ in the conclusion of this rule into the subcontexts Δ_1 and Δ_2 used in the premises, so that $\Delta = \Delta_1 \bowtie \Delta_2$ [5, 9].

An intermediate ALPL $(\mathcal{L}^c_{LHH}, \xrightarrow{c})$ for $\mathcal{L}_{LHH}$ with right introduction rules only is devised similarly to the case of hereditary Harrop formulas treated in Section 3: we consider constructs from full intuitionistic linear logic whose right rules have the same structure as the immediate entailment rules of $\mathcal{L}_{LHH}$. The only complication derives from the constraints linear logic imposes on the ways the context can be manipulated. It is easy to check that the correct choices for $\top$, & and $\multimap$ are $\mathbf{0}$, $\oplus$ and $\otimes$, respectively. Notice that, differently from $\mathcal{L}_{HH}$, $\multimap$ is mapped to multiplicative conjunction $\otimes$ and not to its additive counterpart &, which belongs to $\mathcal{L}_{LHH}$. As in the non-linear case, atomic clauses are rendered by adding a syntactic equality connective $\doteq$.

The only connective that we still need to analyze is intuitionistic implication, $\supset$. The immediate entailment rule for this connective has two premises and it prescribes that the program be integrally propagated to its left-hand branch, while only its intuitionistic assumptions should be forwarded to the right premise. Therefore, we are apparently compelled to invent a new connective, $\odot$ say, together with the desired right-introduction rule. This is a reasonable approach: it would simply have the effect of adding one more connective to linear logic. However, in this particular case, we can do better by observing that the right-introduction rule for the exponential operator ! is applicable only when its linear context is empty. The synergetic action of this connective and $\otimes$ achieve the desired effect:

$$\frac{\Delta \xrightarrow{u} A_1 \gg a \quad !\Delta \xrightarrow{u} A_2}{\Delta \xrightarrow{u} A_2 \supset A_1 \gg a}\,\mathbf{li\_Imp} \qquad \frac{\Delta \longrightarrow A_1 \quad \dfrac{!\Delta \longrightarrow A_2}{!\Delta \longrightarrow !A_2}}{\Delta \longrightarrow A_1 \otimes !A_2}$$

Therefore, if A_1 and A_2 are compiled into the clause C and the goal G respectively, the implication $A_1 \supset A_2$ is translated as $C \otimes !G$.

These considerations summon us to consider the language $\mathcal{L}^c_{LHH}$ defined by the following grammar as the intermediate language of $\mathcal{L}_{LHH}$.

$$\begin{array}{rrcl} \textit{Goals:} & G & ::= & a \mid \top \mid G_1 \,\&\, G_2 \mid (\Lambda\alpha.\,C) \multimap G \mid (\Lambda\alpha.\,C) \supset G \mid \forall x.\,G \\ \textit{C-Goals} & R & ::= & a_1 \doteq a_2 \mid \mathbf{0} \mid R_1 \oplus R_2 \mid R \otimes G \mid R \otimes !G \mid \exists x.\,R \end{array}$$

Goals

$$\frac{\Psi \overset{?}{;} \Psi' \xrightarrow{cg} [a/\alpha]C}{\Psi \,\hat{;}\, \Lambda\alpha.\, C \overset{?}{;} \Psi' \xrightarrow{c} a}\ \textbf{lc\_LinAtom} \qquad \frac{\Psi, \Lambda\alpha.\, C \overset{?}{;} \Psi' \xrightarrow{cg} [a/\alpha]C}{\Psi, \Lambda\alpha.\, C \overset{?}{;} \Psi' \xrightarrow{c} a}\ \textbf{lc\_IntAtom}$$

$$\frac{}{\Psi \xrightarrow{c} \top}\ \textbf{lc\_Top} \qquad \frac{\Psi \xrightarrow{c} G_1 \quad \Psi \xrightarrow{c} G_2}{\Psi \xrightarrow{c} G_1 \,\&\, G_2}\ \textbf{lc\_With}$$

$$\frac{\Psi \,\hat{;}\, \Lambda\alpha.\, C \xrightarrow{c} G}{\Psi \xrightarrow{c} (\Lambda\alpha.\, C) \multimap G}\ \textbf{lc\_Lolli} \qquad \frac{\Psi, \Lambda\alpha.\, C \xrightarrow{c} G}{\Psi \xrightarrow{c} (\Lambda\alpha.\, C) \supset G}\ \textbf{lc\_Imp}$$

Compilation goals

$$\frac{}{!\Psi \xrightarrow{cg} a \doteq a}\ \textbf{lcg\_Eq} \qquad \text{(No rule for } \mathbf{0}\text{)}$$

$$\frac{\Psi \xrightarrow{cg} R_1}{\Psi \xrightarrow{cg} R_1 \oplus R_2}\ \textbf{lcg\_Plus}_\mathbf{1} \qquad \frac{\Psi \xrightarrow{cg} R_2}{\Psi \xrightarrow{cg} R_1 \oplus R_2}\ \textbf{lcg\_Plus}_\mathbf{2}$$

$$\frac{\Psi_1 \xrightarrow{cg} R \quad \Psi_2 \xrightarrow{c} G}{\Psi_1 \bowtie \Psi_2 \xrightarrow{cg} R \otimes G}\ \textbf{lcg\_Times} \qquad \frac{!\Psi \xrightarrow{c} G}{!\Psi \xrightarrow{cg} !G}\ \textbf{lcg\_Bang}$$

Figure 5: Search Semantics of $\mathcal{L}^c_{LHH}$.

$$\begin{array}{rrcl} \textit{Clauses:} & C & ::= & a \doteq \alpha \mid \mathbf{0} \mid C_1 \,\&\, C_2 \mid C \otimes G \mid C \otimes !G \mid \exists x.\, C \\ \textit{Programs:} & \Psi & ::= & \cdot \mid \Psi, \Lambda\alpha.\, C \mid \Psi \,\hat{;}\, \Lambda\alpha.\, C \end{array}$$

The terminology, remarks, and conventions made in the non-linear case in Section 3 apply in the current setting.

The operational semantics of this language is given in Figure 5. Observe that it consists only of right-introduction rules. The above considerations about how to compile an intuitionistic implication in a program position lead to two alternative presentations of the search semantics of a c-goal of the form $R \otimes !G$. The simplest, displayed in Figure 5, views ! as an independent construct and relies on its right-introduction rule. Another possibility arises by observing that ! can only occur as the topmost operator of the right conjunct of $\otimes$. We could therefore have devised a rule specialized to this case:

$$\frac{\Psi \xrightarrow{cg} R \quad !\Psi \xrightarrow{c} G}{\Psi \xrightarrow{cg} R \otimes !G}\ \textbf{lcg\_TimesBang}$$

This is exactly the right sequent rule for the hypothetical connective $\odot$ we discussed earlier.

The rules describing the compilation of an expression in $\mathcal{L}_{LHH}$ into an object in $\mathcal{L}^c_{LHH}$ are given in Figure 6. Similarly to the non-linear case in Section 3, we can prove that they specify a bijection between $\mathcal{L}_{LHH}$ and $\mathcal{L}^c_{LHH}$, modulo obvious syntactic restrictions. The soundness and completeness for the compilation process of $\mathcal{L}_{LHH}$ is proved similarly to the non-linear case. We expect these proofs to be amenable to a direct encoding in the linear logical framework *LLF* [6]. We omit their statement for space reasons.

Programs

$$\frac{}{\cdot \gg \cdot}\,\textbf{lpc\_empty} \qquad \frac{\Delta \gg \Psi \quad A \gg \alpha \backslash C}{\Delta, A \gg \Psi, \Lambda\alpha.\, C}\,\textbf{lpc\_clause}$$

Clauses

$$\frac{}{a \gg \alpha \backslash a \doteq \alpha}\,\textbf{lcc\_Atom} \qquad \frac{}{\top \gg \alpha \backslash \mathbf{0}}\,\textbf{lcc\_Top} \qquad \frac{A_1 \gg \alpha \backslash C_1 \quad A_2 \gg \alpha \backslash C_2}{A_1 \,\&\, A_2 \gg \alpha \backslash C_1 \oplus C_2}\,\textbf{lcc\_With}$$

$$\frac{A_1 \gg \alpha \backslash C \quad A_2 \gg G}{A_2 \multimap A_1 \gg \alpha \backslash C \otimes G}\,\textbf{lcc\_Lolli} \qquad \frac{A_1 \gg \alpha \backslash C \quad A_2 \gg G}{A_2 \supset A_1 \gg \alpha \backslash C \otimes !G}\,\textbf{lcc\_Imp}$$

Goals

$$\frac{}{a \gg a}\,\textbf{lgc\_Atom} \qquad \frac{}{\top \gg \top}\,\textbf{lgc\_Top} \qquad \frac{A_1 \gg G_1 \quad A_2 \gg G_2}{A_1 \,\&\, A_2 \gg G_1 \,\&\, G_2}\,\textbf{lgc\_With}$$

$$\frac{A_1 \gg \alpha \backslash C \quad A_2 \gg G}{A_1 \multimap A_2 \gg (\Lambda\alpha.\, C) \multimap G}\,\textbf{lgc\_Lolli} \qquad \frac{A_1 \gg \alpha \backslash C \quad A_2 \gg G}{A_1 \supset A_2 \gg (\Lambda\alpha.\, C) \supset G}\,\textbf{lgc\_Imp}$$

Figure 6: Compilation of $\mathcal{L}_{LHH}$.

5 Conclusion and Future Work

In this paper, we have proposed an abstract, logic-independent and modular definition of the notion of compilation for logic programming languages, founded on proof-theory. We have described an effective method based on the duality between left and right sequent rules in order to find an intermediate language and to construct a compiler and a matching abstract machine from any abstract logic programming language. We have applied it to define abstract compilation systems for the language of hereditary Harrop formulas and for its linear refinement. We have furthermore sketched soundness and completeness proofs for the first of these examples.

Our compiled programs are clearly closer to the WAM than their sources, but there is still a wide gap to bridge in order to achieve the efficiency expected from compilation. We observed however that many of the optimizations found in the WAM can be achieved by proof-theoretic principles. In particular, it is easy to refine our system so that only pertinent clauses are selected (e.g., only those clauses defining atoms with the same predicate symbol as the goal). In a non-propositional setting, we can unfold the appropriate definition of $\doteq$ in a manner that is very reminiscent of the term instructions of the WAM. By systematically applying further simple logical manipulations, we obtain the following code in the case of "append" for example:

$$\begin{array}{l}\text{append} : \Lambda\alpha_1.\, \Lambda\alpha_2.\, \Lambda\alpha_3. \\ \quad (\exists l.\, \text{nil} \doteq \alpha_1 \;\wedge\; l \doteq \alpha_2 \;\wedge\; l \doteq \alpha_3) \\ \vee\, (\exists k.\, \exists l.\, \exists m.\, \exists x.\, (x :: k) \doteq \alpha_1 \;\wedge\; l \doteq \alpha_2 \;\wedge\; (x :: m) \doteq \alpha_3 \;\wedge\; (\text{append}\, k\, l\, m))\end{array}$$

Observe that this corresponds exactly to the *Clark completion* [8] of the original clauses for "append": just prefix "(append α_1 α_2 α_3) $\equiv$" in the scope of the abstractions. We indeed noticed that, in the case of *Prolog*, intermediate levels of compilation correspond to Clark completions of logic programs.

The approach described in this paper has been applied in a new implementation of the logical framework *LF* [19] as the constraint higher-order logic programming language *Twelf*, a successor to *Elf* [20]. In addition to constructs discussed in this paper, this formalism is characterized by (dependent) types and proof-terms. Handling the former is straightforward. Representing the latter in a *spine calculus* [7] enables an elegant treatment in our framework. At the time of writing, statistics to evaluate the benefits of compilation are not available, although *Twelf* has been observed to run significantly faster than the interpreted *Elf*. Indeed, it is not possible to measure the effects of compilation alone since *Twelf* is the result of a thorough redesign of its predecessor.

We intend to develop this research by showing that specific strategies to handle non-deterministic choices can be included in our compilation scheme and apply them to *Twelf*. We also intend to use the above techniques in order to compile *LLF* [6], a linear refinement of *LF*.

Acknowledgements

This research was supported by ONR grant N00014-97-1-0505. The author would like to thank Frank Pfenning for inspiring this work and for his comments on earlier versions of this paper.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: a Tutorial Reconstruction.* MIT Press, 1991.

[2] Christoph Beierle and Egon Börger. Correctness proof for the WAM with types. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter, editors, *Proceedings of the Fifth Workshop on Computer Science Logic — CSL'91*, pages 15–34. Springer-Verlag LNCS 626, 1992.

[3] Egon Börger and Dean Rosenzweig. The WAM — definition and compiler correctness. In C. Beierle and L. Pluemer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Computer Science and Artificial Intelligence*, pages 21–90. North-Holland, 1995.

[4] Egon Börger and Rosario F. Salamone. CLAM specification for provably correct compilation of CLP programs. In E. Börger, editor, *Specification and Validation Methods*, pages 97–130. Oxford University Press, 1995.

[5] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81, Leipzig, Germany, March 1996. Springer-Verlag LNAI 1050.

[6] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer*

Science, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[7] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, April 1997.

[8] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*. Plenum Press, 1978.

[9] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[10] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. An abstract machine for *CLP*($\mathcal{R}$). In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation — PLDI'92*, San Francisco, CA, 1992.

[11] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Thirteenth Symposium on Principles of Programming Languages*, pages 86–96. ACM, January 1986.

[12] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1):25–42, 1994.

[13] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[14] Dale Miller. Lambda Prolog: An introduction to the language and its logic. Current draft available from `http://cse.psu.edu/~dale/lProlog`, 1996.

[15] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[16] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, 1995.

[17] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Department of Computer Science, Duke University, June 1993.

[18] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[19] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[20] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[21] David M. Russinoff. A verified Prolog compiler for the Warren abstract machine. *Journal of Logic Programming*, 13:367–412, 1992.

[22] David H. D. Warren. an abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.