TLP **12** (4–5): 639–657, 2012. © Cambridge University Press 2012
doi:10.1017/S1471068412000221

An Improved Proof-Theoretic Compilation of Logic Programs

ILIANO CERVESATO

Department of Computer Science, Carnegie Mellon University
(*e-mail:* iliano@cmu.edu)

Abstract

In prior work, we showed that logic programming compilation can be given a proof-theoretic justification for generic abstract logic programming languages, and demonstrated this technique in the case of hereditary Harrop formulas and their linear variant. Compiled clauses were themselves logic formulas except for the presence of a second-order abstraction over the atomic goals matching their head. In this paper, we revisit our previous results into a more detailed and fully logical justification that does away with this spurious abstraction. We then refine the resulting technique to support well-moded programs efficiently.

KEYWORDS: Compilation, Abstract Logic Programming, Hereditary Harrop Formulas, Well-Moded Logic Programs.

1 Introduction

In (Cervesato 1998), we presented a general methodology for developing a compiler and associated intermediate language for any abstract logic programming language (ALPL) (Miller et al. 1991) that satisfies some basic proof-theoretic properties. We applied it abstractly to the language of hereditary Harrop formulas and its linear variant, and also based the concrete implementations of the Twelf (Pfenning and Schürmann 1999) and LLF (Cervesato and Pfenning 2002) systems directly on it. This methodology identified right sequent rules that behave like the left rules that can appear in a uniform proof and used the corresponding connectives as the compilation targets of the constructs in program clauses. The intermediate language was therefore just another ALPL and its abstract machine relied on proof-search, like the source ALPL. Because the transformation was based on the proof-theoretic duality between left and right rules, proving the correctness of the compilation process amounted to a simple induction. Finally, for Horn clauses the connectives in the target ALPL corresponded to key instructions in the Warren Abstract Machine (WAM) (Warren 1983). The WAM is an essential component of commercial Prolog systems since many compiled programs run over an order of magnitude faster than when interpreted.

Up to then, the notoriously procedural instruction set of the WAM was regarded as a wondrous piece of engineering without any logical status, in sharp contrast with the deep logical roots of Prolog. In the words of (Börger and Rosenzweig

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

1995) "[the WAM] resembles an intricate puzzle, whose many pieces fit tightly together in a miraculous way". As a result, understanding it was complex in spite of the availability of excellent tutorials (Aït-Kaci 1991), proving its correctness was a formidable task (Börger and Rosenzweig 1995; Russinoff 1992), and adapting it to other logic programming languages a major endeavor — it was done for $CLP(\mathscr{R})$ (Jaffar et al. 1992) and $\lambda Prolog$ (Nadathur and Mitchell 1999). By contrast, the methodology in (Cervesato 1998) is simple, (mostly) logic-based, easily verifiable, and of general applicability.

The technique in (Cervesato 1998) had however one blemish: it made use of equality over atomic formulas together with a second-order binder over atomic goals, which lacked logical status. In this paper, we remedy this drawback by carefully massaging the head of clauses. This allows us to replace those constructs with term-level equality and regular universal quantifications over the arguments of a clause head. The result is an improved proof-theoretic account of compilation for logic programs that sits squarely within logic. It also opens the doors to specializing the compilation process to well-moded programs, which brings out the potential of doing away with unification in favor of matching, a more efficient operation in many languages. We present these results for the language of hereditary Harrop formulas and only at the highest level of abstraction. Just like (Cervesato 1998), they are however general, both in terms of the source ALPL and of the level of the abstraction considered. We are indeed in the process of using them to implement a compiler for CLF (Watkins et al. 2003; Cervesato et al. 2003), a higher-order concurrent linear logic programming language that combines backward and forward chaining.

The paper is organized as follows: Section 2 recalls the compilation process of (Cervesato 1998). In Section 3, we present our improved compilation process. In Section 4, we refine it to support moded programs. We lay out future developments in Sections 5 and 6.

2 Background and Recap

In this section, we recall the compilation process presented in (Cervesato 1998). For succinctness, we focus on a smaller source language — it corresponds to the language underlying the Twelf system (Pfenning and Schürmann 1999), on which this technique was first used. We will comment on larger languages, including those examined in (Cervesato 1998), in Section 5.

2.1 Source Language

We take the language freely generated from atomic propositions (a), intuitionistic implication ($\supset$) and universal quantification ($\forall$) as our source language. We expand the open-ended atomic propositions of (Cervesato 1998), into a *predicate symbol* p followed by zero or more terms t. A program is a sequence of closed formulas. This language, which we call $\mathscr{L}^s$, is given by the following grammar:

$$\begin{array}{lll} \textit{Formulas:} & A ::= a \mid A_1 \supset A_2 \mid \forall x.\, A & \qquad \textit{Programs:} \quad \Gamma ::= \cdot \mid \Gamma, A \\ \textit{Atoms:} & a ::= p \mid a\, t & \end{array}$$

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

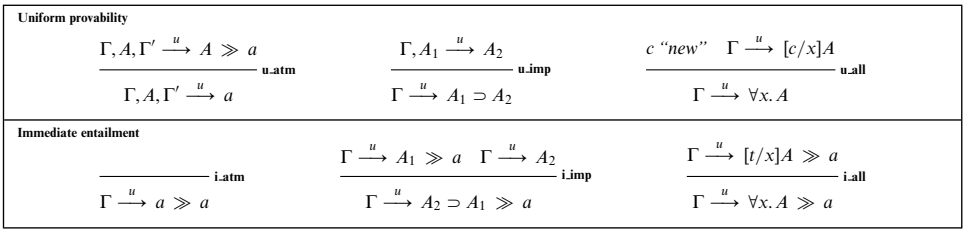


Fig. 1. Uniform Deduction System for $\mathscr{L}^s$.

As in (Cervesato 1998), we leave the language of terms open, but require that it be predicative (substituting a term for a variable cannot alter the outer structure of a formula). We will often write an atom a as $p\ \underline{t}$, where p is its predicate symbol and $\underline{t}$ is the sequence of terms it is applied to. We implicitly assume that a predicate symbol is consistently applied to the same number of terms throughout a program — its arity. We write $[t'/x]t$ (resp. $[t'/x]A$) for the capture-avoiding substitution of term t' for all free occurrences of variable x in term t (resp. in formula A). Simultaneous substitution is denoted $[\underline{t'}/\underline{x}]t$ and $[\underline{t'}/\underline{x}]A$.

$\mathscr{L}^s$ is an abstract logic programming language (Miller et al. 1991) and, for appropriate choices of the term language, has indeed the same expressive power as λProlog (Miller and Nadathur 1986) or Twelf (Pfenning and Schürmann 1999). It differs from the first language discussed in (Cervesato 1998) for the omission of conjunction and truth (see Section 5).

The operational semantics of $\mathscr{L}^s$ is given by the two judgments

$$\Gamma \xrightarrow{u} A \qquad A \textit{ is uniformly provable from } \Gamma$$
$$\Gamma \xrightarrow{u} A \gg a \qquad a \textit{ is immediately entailed by } A \textit{ in } \Gamma$$

Their defining rules, given in Figure 1, produce uniform proofs (Miller et al. 1991): the uniform provability judgment includes the right sequent rules for $\mathscr{L}^s$ and, once the goal is atomic, rule **u_atm** calls the immediate entailment judgment, which focuses on a program formula A and decomposes it as prescribed by the left sequent rules. This strategy is complete with respect to the traditional sequent rules of this logic (Miller et al. 1991). From a logic programming perspective, the connectives appearing in the goal — handled by right rules — are search directives, while the left rules carry out a run-time preparatory phase.

2.2 Target Language

In (Cervesato 1998), the target language of the compilation process distinguished compiled goals (G) from compiled clauses (C). A compiled goal was either an atomic proposition, or a hypothetical goal (a goal to be solved in the presence of an additional clause) or a universal goal (a goal to be solved in the presence of a new constant). A compiled clause had the form $\Lambda\alpha.\, C$, where the second-order variable α stood for the atomic goal to be resolved against the present clause, while C could either match α with the head a of this clause ($a \doteq \alpha$), invoke a goal ($C \wedge G$), or request that a variable x be instantiated with a term ($\exists x.\, C$). A compiled program Ψ

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

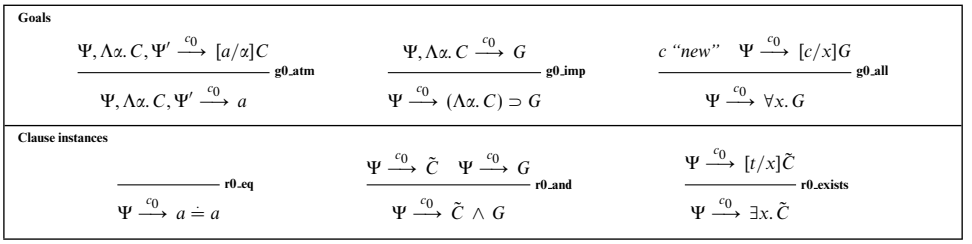


Fig. 2. Search Semantics of $\mathscr{L}_0^c$.

was then a sequence of compiled clauses. The grammar for the resulting language, which we call $\mathscr{L}_0^c$, is as follows:

$$\begin{array}{rl} \textit{Goals:} & G ::= a \mid (\Lambda\alpha.\, C) \supset G \mid \forall x.\, G \\ \textit{Clauses:} & C ::= a \doteq \alpha \mid C \wedge G \mid \exists x.\, C \end{array} \qquad \textit{Programs:} \quad \Psi ::= \cdot \mid \Psi, \Lambda\alpha.\, C$$

The operational semantics of a compiled program, as given by the above grammar, is defined on the basis of the following two judgments:

$$\begin{array}{ll} \Psi \xrightarrow{c_0} G & G \textit{ is uniformly provable from } \Psi \\ \Psi \xrightarrow{c_0} \tilde{C} & \tilde{C} \textit{ is uniformly provable from } \Psi \end{array}$$

Here, clause instances $\tilde{C}$ are C's whose variable α has been instantiated with an atomic formula a'. The operational semantics of $\mathscr{L}_0^c$ is shown in Figure 2. Observe that, with the partial exception of **g0_atm**, it consists solely of right rules. This means that every connective is seen as a search directive: the dynamic clause preparations embodied by the left rules has now been turned into right search rules through a static compilation phase.

2.3 Compilation

Compilation, the process that transforms a logic program in $\mathscr{L}^s$ into a compiled program in $\mathscr{L}_0^c$, is expressed by means of the following three judgments:

$$\begin{array}{ll} \Gamma \gg \Psi & \textit{Program } \Gamma \textit{ is compiled to } \Psi \\ A \gg \alpha \setminus C & \textit{Clause } A \textit{ with } \alpha \textit{ is compiled to } C \\ A \gg G & \textit{Goal } A \textit{ is compiled to } G \end{array}$$

These judgments are defined by the rules in Figure 3 — see (Cervesato 1998) for details.

As our ongoing example, consider the following two clauses, taken from a type checking specification for a Church-style simply typed λ-calculus. For clarity, we write program clauses Prolog-style, using the reverse implication $\subset$ instead of $\supset$ in positive formulas.

1.
$$\begin{array}{l} \forall E_1.\, \forall E_2.\, \forall T_1.\, \forall T_2. \\ \quad \text{of (app } E_1\ E_2)\ T_2 \\ \subset \quad \text{of } E_1 \text{ (arr } T_1\ T_2) \\ \subset \quad \text{of } E_2\ T_1 \end{array} \gg \begin{array}{l} \Lambda\alpha. \\ \exists E_1.\, \exists E_2.\, \exists T_1.\, \exists T_2. \\ \quad \text{(of (app } E_1\ E_2)\ T_2) \doteq \alpha \\ \wedge \quad \text{of } E_1 \text{ (arr } T_1\ T_2) \\ \wedge \quad \text{of } E_2\ T_1 \end{array}$$

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

Programs

$$\frac{}{\cdot \gg \cdot}\ \textbf{p0c_empty} \qquad \frac{\Gamma \gg \Psi \quad A \gg \alpha \setminus C}{\Gamma, A \gg \Psi, \Lambda\alpha.\, C}\ \textbf{p0c_clause}$$

Clauses

$$\frac{}{a \gg \alpha \setminus a \doteq \alpha}\ \textbf{c0c_atm} \qquad \frac{B \gg \alpha \setminus C \quad A \gg G}{A \supset B \gg \alpha \setminus C \wedge G}\ \textbf{c0c_imp} \qquad \frac{A \gg \alpha \setminus C}{\forall x.\, A \gg \alpha \setminus \exists x.\, C}\ \textbf{c0c_all}$$

Goals

$$\frac{}{a \gg a}\ \textbf{g0c_atm} \qquad \frac{A \gg \alpha \setminus C \quad B \gg G}{A \supset B \gg (\Lambda\alpha.\, C) \supset G}\ \textbf{g0c_imp} \qquad \frac{A \gg C}{\forall x.\, A \gg \forall x.\, C}\ \textbf{g0c_all}$$

Fig. 3. Compilation of $\mathscr{L}^s$ into $\mathscr{L}^c_0$.

2.

$$\begin{array}{l} \forall E.\, \forall T_1.\, \forall T_2. \\ \quad \mathsf{of}\ (\mathsf{lam}\ T_1\ E)\ (\mathsf{arr}\ T_1\ T_2) \\ \subset\ (\forall x.\, \mathsf{of}\ x\ T_1 \\ \quad \supset \mathsf{of}\ (E\ x)\ T_2) \end{array} \quad \gg \quad \begin{array}{l} \Lambda\alpha. \\ \exists E.\, \exists T_1.\, \exists T_2. \\ \quad (\mathsf{of}\ (\mathsf{lam}\ T_1\ E)\ (\mathsf{arr}\ T_1\ T_2)) \doteq \alpha \\ \wedge\ (\forall x.\quad \Lambda\beta.\, ((\mathsf{of}\ x\ T_1) \doteq \beta) \\ \quad \supset \mathsf{of}\ (E\ x)\ T_2) \end{array}$$

The compiled language $\mathscr{L}^c_0$ is sound and complete for $\mathscr{L}^s$. See (Cervesato 1998) for the formal statements. The proof of both directions proceeds by straightforward induction, which contrasts greatly with the complex proofs of soundness and correctness previously devised for the WAM (Börger and Rosenzweig 1995; Russinoff 1992).

3 Fully Logical Compilation

Because clauses are compiled to expressions of the form $\Lambda\alpha.\, C$, the language $\mathscr{L}^c_0$ is not fully logical. In this section we consider a different compilation target, the language $\mathscr{L}^c_1$, which lies entirely within logic.

In the previous section, a generic Horn clause of the form

$$\forall \underline{y}.\, (p\ \underline{t} \subset a_1 \subset \ldots \subset a_n) \tag{1}$$

was compiled into $\Lambda\alpha.\, \exists \underline{y}.\, (p\ \underline{t} \doteq \alpha \wedge a_1 \wedge \ldots \wedge a_n)$. During execution, rule **c0_atm** reduced the current atomic goal a to the clause instance $\exists \underline{y}.\, (p\underline{t} \doteq a \wedge a_1 \wedge \ldots \wedge a_n)$. Note that $\underline{t}$ may depend on $\underline{y}$, but a does not. We will now compile that Horn clause into

$$\forall \underline{x}.\, (p\ \underline{x} \subset \exists \underline{y}.\, (\underline{x} \doteq \underline{t} \wedge a_1 \wedge \ldots \wedge a_n)) \tag{2}$$

where $\underline{x}$ is a sequence of fresh variables, all distinct from each other, and equal in number to the arity of p, and $\underline{x} \doteq \underline{t}$ stands for a conjunction of equalities between each variable x_i in $\underline{x}$ and the term t_i in $\underline{t}$ in the corresponding position (or $\top$ if the arity of p is zero). Notice that the non-logical second-order binder "$\Lambda\alpha$." is gone. At run time, formula (2) will resolve an atomic goal $p\ \underline{t'}$ into the clause $p\ \underline{t'} \subset \exists \underline{y}.\, (\underline{t'} \doteq \underline{t} \wedge a_1 \wedge \ldots \wedge a_n)$, which immediately reduces to $\exists \underline{y}.\, (\underline{t'} \doteq \underline{t} \wedge a_1 \wedge \ldots \wedge a_n)$. Like earlier, $\underline{t}$ may depend on $\underline{y}$, but $\underline{t'}$ does not. The variables $\underline{x}$ correspond directly to the "argument registers" ($\mathtt{A}n$) of the WAM (Aït-Kaci 1991), while the $\underline{y}$'s are closely related to its "permanent variables" ($\mathtt{Y}n$).

Formula (2) can be understood as an uncurried form of (1): outer implications are transformed into conjunctions and universals into existentials. Doing so literally would yield the formula $p\ \underline{t} \subset \exists \underline{y}.\, (a_1 \wedge \ldots \wedge a_n)$, which is incorrect because

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

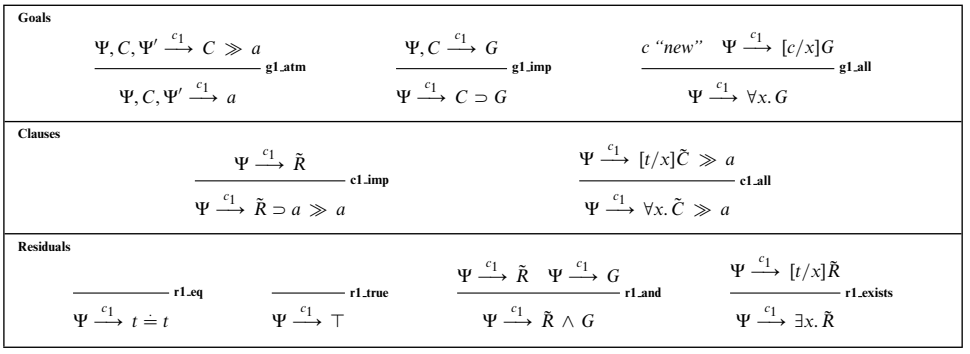


Fig. 4. Search Semantics of $\mathscr{L}_1^c$.

occurrences of variables in $\underline{y}$ within $\underline{t}$ have escaped their scope. Instead, formula (2) installs fresh variables $\underline{x}$ as the arguments of the head predicate p and adds the equality constraints $\underline{x} \doteq \underline{t}$ in the body.

3.1 Target Language

We now generalize the above intuition to any formula in $\mathscr{L}^s$, not just Horn clauses. Our second target language, $\mathscr{L}_1^c$, is given by the following grammar.

$$\begin{array}{rrcl} \textit{Goals:} & G & ::= & a \mid C \supset G \mid \forall x.\, G \\ \textit{Clauses:} & C & ::= & R \supset p\,\underline{x} \mid \forall x.\, C \\ \textit{Residuals:} & R & ::= & x \doteq t \mid \top \mid R \wedge G \mid \exists x.\, R \end{array} \qquad \textit{Programs:}\quad \Psi ::= \cdot \mid \Psi, C$$

Compiled goals (G) are just like in Section 2.2: atoms, hypothetical goals, or universal goals. Compiled clauses (C) have the form $\forall \underline{x}.(R \supset p\,\underline{x})$, i.e., a (possibly empty) outer layer of universal quantifiers enclosing an implication $R \supset p\,\underline{x}$ whose head $p\,\underline{x}$ always consists of a predicate name (p) applied to a (possibly empty) sequence of distinct variables ($\underline{x}$). Its body is a *residual* (R). A residual can be either an equality constraint ($x \doteq t$), the trivial constraint $\top$ (logical truth), or like in Section 2.2 a goal invocation or an instantiation request. Notice that C is now the full result of compiling a clause.

The operational semantics of $\mathscr{L}_1^c$ is specified by the following three judgments:

$$\begin{array}{ll} \Psi \xrightarrow{c_1} G & G \textit{ is uniformly provable from } \Psi \\ \Psi \xrightarrow{c_1} \tilde{C} \gg a & a \textit{ is immediately entailed by } \tilde{C} \textit{ in } \Psi \\ \Psi \xrightarrow{c_1} \tilde{R} & \tilde{R} \textit{ is uniformly provable from } \Psi \end{array}$$

where $\tilde{C}$ and $\tilde{R}$ differ from C and R by the instantiation of some variables in a clause head and on the left-hand side of equalities, respectively.

Their operational semantics is given in Figure 4. Goals are handled exactly in the same way as uniform provability in $\mathscr{L}^s$ (top part of Figure 1). The operational reading of compiled clauses is an instance of that of immediate entailment: rule **c1_imp** is a special case of **i_imp** while **c1_all** is isomorphic to **i_all**. Note that rule **c1_imp** reduces immediately to the residual R if the head of the clause matches the atomic goal a being proved. The rules for residuals correspond closely to the rules for clause

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

instances for our original target language at the bottom of Figure 2: rule **r1_eq** requires that the two sides of an equality be indeed equal and rule **r1_true** is always satisfied.

The rules in Figure 4 build uniform proofs (Miller et al. 1991), characteristic of abstract logic programming languages: the operational semantics decomposes a goal to an atomic formula (top segment of Figure 4), then selects a clause and focuses on it until it finds a matching head (middle segment) and then decomposes its body (bottom segment), which may eventually expose some goals, and the cycle repeats. In particular, once an atomic goal $p\ \underline{t}$ has been exposed, a successful derivation will necessarily contain an instance of rule **g1_atm** that picks a clause C with head $p\ \underline{x}$, as many instances of rule **c1_all** as the arity of p, and an instance of rule **c1_imp**. This necessary sequence of steps is captured by the following derived "macro-rule" (the *backchaining* rule):

$$\frac{\Psi, \forall \underline{x}.\,(R \supset p\ \underline{x}), \Psi' \xrightarrow{c_1} [\underline{t}/\underline{x}]R}{\Psi, \forall \underline{x}.\,(R \supset p\ \underline{x}), \Psi' \xrightarrow{c_1} p\ \underline{t}}\ \mathbf{g1_atm'}$$

Replacing rules **g1_atm**, **c1_all** and **c1_imp** with rule **g1_atm′** yields a system that is equivalent to that in Figure 4. Taking it as primitive amounts to replacing the construction for compiled clauses, $\forall \underline{x}.\,(R \supset p\ \underline{x})$, with a synthetic connective, call it $\Lambda_p \underline{x}.\,R$. Therefore, by accounting for the structure of atomic propositions and proper quantification patterns, $\mathscr{L}_1^c$ provides a fully logical justification for clause compilation that $\mathscr{L}_0^c$'s $\Lambda\alpha.\,C$ lacked.

3.2 Compilation

Compilation transforms logic programs in $\mathscr{L}^s$ into compiled logic programs in $\mathscr{L}_1^c$. In order to define it, the auxiliary notion of pseudo clause will come handy:

$$\textit{Pseudo Clauses:}\quad \mathscr{C} ::= \square \supset p\ \underline{x} \mid \forall x.\,\mathscr{C}$$

A pseudo clause retains the outer structure of a clause, but has a hole ($\square$) in place of the residual R. In general, a pseudo clause $\mathscr{C}$ has the form $\forall \underline{x}.\,\square \supset p\ \underline{x'}$. In a fully compiled clause, variables $\underline{x}$ will coincide with $\underline{x'}$.

Pseudo clauses are generated while processing the head of a clause. The hole then needs to be replaced with the compiled body, a residual. We write this operation, pseudo clause instantiation, as $\mathscr{C}[R]$. It is formally defined as follows:

$$\begin{cases} (\square \supset p\ \underline{x})[R] &= R \supset p\ \underline{x} \\ (\forall x.\,\mathscr{C})[R] &= \forall x.\,(\mathscr{C}[R]) \end{cases}$$

As is often the case with such contextual operations, pseudo clause instantiation can, and generally will, lead to variable capture: in $(\forall \underline{x}.\,\square \supset p\ \underline{x})[R]$, there may be free occurrences of variables in $\underline{x}$ within R. In the result, these occurrences are bound by the outer quantifiers.

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

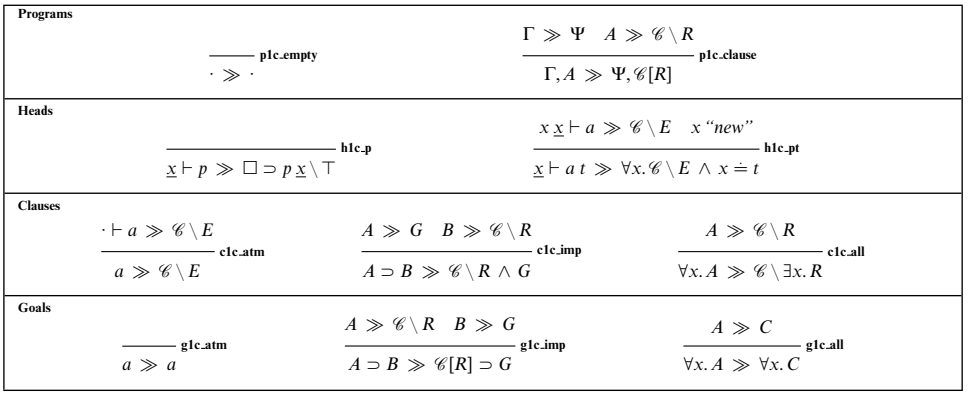


Fig. 5. Compilation of $\mathscr{L}^s$ into $\mathscr{L}^c_1$.

Compilation is expressed by means of the following four judgments

$\Gamma \gg \Psi$	*Program* Γ *is compiled to* Ψ
$\underline{x} \vdash a \gg \mathscr{C} \setminus E$	*Head* a *with* $\underline{x}$ *is compiled to* $\mathscr{C}$ *and* E
$A \gg \mathscr{C} \setminus R$	*Clause* A *is compiled to* $\mathscr{C}$ *and* R
$A \gg G$	*Goal* A *is compiled to* G

and defined by the rules in Figure 5, where we wrote E for conjunctions of equalities. The judgment $A \gg \mathscr{C} \setminus R$ compiles an $\mathscr{L}^s$ clause A into a pseudo clause $\mathscr{C}$ and a residual R. They are assembled into an $\mathscr{L}^c_1$ clause in rules **p1c_clause** and **g1c_imp**. Programs and goals are otherwise compiled just as for $\mathscr{L}^c_0$ in Figure 3. Clause heads are handled differently: rule **c1c_atm** invokes the auxiliary head compilation judgment to compile the goal $p\,\underline{t}$ into a pseudo clause $\forall \underline{x}. \Box \supset p\,\underline{x}$ and the equalities $\underline{x} \doteq \underline{t}$, which will form the seed of the clause's residual.

Consider the first example clause in Section 2.3. Its head $(\mathsf{of}\ (\mathsf{app}\ E_1\ E_2)\ T_2)$ is compiled into the pseudo clause $\forall x_1. \forall x_2. (\Box \supset \mathsf{of}\ x_1\ x_2)$ and the equality constraints $\top \wedge (x_1 \doteq \mathsf{app}\ E_1\ E_2) \wedge (x_2 \doteq T_2)$, where x_1 and x_2 are new variables. These core equalities are then extended with the compiled body of that clause, $(\mathsf{of}\ E_1\ (\mathsf{arr}\ T_1\ T_2)) \wedge (\mathsf{of}\ E_2\ T_1)$, and existential quantifications over the original variables of the clause, E_1, E_2, T_1 and T_2, are finally wrapped around the result before embedding it in the hole of the pseudo clause. The resulting $\mathscr{L}^c_1$ clause is displayed in the top part of Figure 3.2.

The target language $\mathscr{L}^c_1$ is sound and complete with respect to $\mathscr{L}^s$. In order to show it, we need the following auxiliary results. The first statement is proved by induction on the structure of a. The second by induction on the given derivation.

Lemma 3.1

- If $\underline{x} \vdash a \gg \mathscr{C} \setminus E$, then for all $\underline{t}$ of the same length as $\underline{x}$ and all Ψ we have $\Psi \xrightarrow{c_1} [\underline{t}/\underline{x}](\mathscr{C}[E]) \gg a\,\underline{t}$.
- If $\Psi \xrightarrow{c_1} \mathscr{C}[R] \gg a$, then $\Psi \xrightarrow{c_1} R$.

The statements of soundness and completeness are as follows. For each of them, the proof proceeds by mutual induction on the first derivation in the antecedent.

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

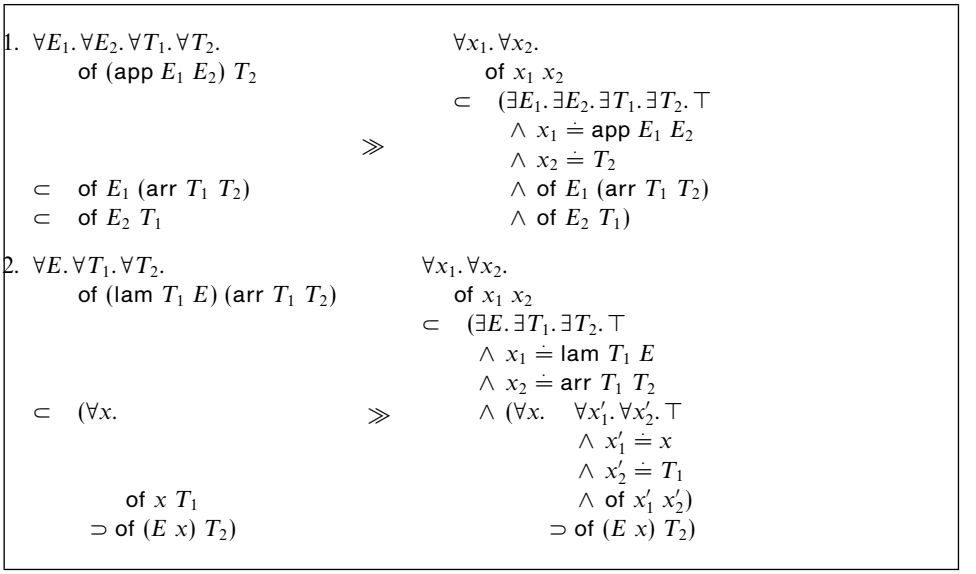


Fig. 6. $\mathscr{L}_1^c$ Compilation Example

Theorem 3.2 (*Soundness of the compilation to* $\mathscr{L}_1^c$)

- If $\Gamma \xrightarrow{u} A$, $\Gamma \gg \Psi$ and $A \gg G$, then $\Psi \xrightarrow{c_1} G$.
- If $\Gamma \xrightarrow{u} A \gg a$, $\Gamma \gg \Psi$ and $A \gg \mathscr{C} \setminus R$, then $\Psi \xrightarrow{c_1} \mathscr{C}[R] \gg a$.

Theorem 3.3 (*Completeness of the compilation to* $\mathscr{L}_1^c$)

- If $\Psi \xrightarrow{c_1} G$, $\Gamma \gg \Psi$ and $A \gg G$, then $\Gamma \xrightarrow{u} A$.
- If $\Psi \xrightarrow{c_1} C \gg a$, $\Gamma \gg \Psi$, $C = \mathscr{C}[R]$ and $A \gg \mathscr{C} \setminus R$, then $\Gamma \xrightarrow{u} A \gg a$.

We conclude this section by showing in Figure 3.2 the output of our compilation procedure for the two examples seen in Section 2.3. We stretch the source clauses (left) to align corresponding atoms. As can be gleaned from these clauses, there are ample opportunities for optimizations in our compilation process. In particular, a constraint $x \doteq y$ mentioning variables on both sides can often be eliminated by replacing the existential variable y with the universal variable x in the rest of the clause (and removing the existential quantifier) — the exception is when there are multiple constraints of this form for the same y. The leading logical constant $\top$ makes for a succinct presentation of the compilation process, but plays no actual role: it can also be eliminated.

It is interesting to rewrite these clauses using the synthetic connective Λ_p discussed earlier (we have omitted occurrences of $\top$ for readability):

$$\Lambda_{\mathsf{of}}\ x_1\ x_2.\quad \exists E_1.\exists E_2.\exists T_1.\exists T_2.\ x_1 \doteq \mathsf{app}\ E_1\ E_2 \wedge x_2 \doteq T_2 \wedge \mathsf{of}\ E_1\ (\mathsf{arr}\ T_1\ T_2) \wedge \mathsf{of}\ E_2\ T_1$$

$$\Lambda_{\mathsf{of}}\ x_1\ x_2.\quad \exists E.\exists T_1.\exists T_2.\ x_1 \doteq \mathsf{lam}\ T_1\ E \wedge x_2 \doteq \mathsf{arr}\ T_1\ T_2 \wedge \forall x.(\Lambda_{\mathsf{of}}\ x'_1\ x'_2.\ x'_1 \doteq x \wedge x'_2 \doteq T_1) \supset \mathsf{of}\ (E\ x)\ T_2$$

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

4 Support for Moded Programs

In this section, we will specialize the compilation process just outlined to the case where the source program is well-moded. In a well-model program, the argument positions of each predicate symbol are designated as either input or output. Input arguments are guaranteed to be ground terms at the time a goal is called. Dually, output arguments are guaranteed to have been made ground by the time the call returns.

There are operational benefits to working with well-moded programs: while an interpreter for a generic program must implement term-level unification, well-moded programs can be executed by relying uniquely on pattern matching and variable instantiation. This is desirable because matching often behaves better than general unification. For example, it is more efficient for first-order term languages were it only because it does away with the occurs-check, and it is decidable for higher-order term languages while general unification is not (Stirling 2009).

The development in this section is motivated by well-moding, but is sound independently of whether a program is well-moded or not. Statically enforcing well-moding brings the operational advantages just discussed, but the results in this section do not depend on it.

4.1 Source Language

In this section, we assume that each predicate symbol in $\mathscr{L}^s$ comes with a *mode* which declares each of its arguments as input, written $\check{}$, or output, written $\hat{}$. For simplicity of exposition, we decorate the actual arguments of all atomic propositions with these symbols, so that a term t in input position in an atomic proposition is written $\check{t}$ (read "in t"). Similarly t in output position is written $\hat{t}$ (pronounced "out t"). This amounts to revising the grammar of atomic propositions as follows:

$$\textit{Atoms:}\quad a ::= p \mid a\,\check{t} \mid a\,\hat{t}$$

Just like we assume that the arity of a predicate symbol p remains constant in a program, we require that all atomic propositions for p have their input/output marks in the same positions. This pattern is the mode of p — an actual language would rely on explicit mode declarations.

For typographic convenience and without loss of generality, our examples assume that input positions precede output positions so that an atomic formula a can be written as $p\,\underline{\check{t}}\,\underline{\hat{t}}$ where $\underline{\check{t}}$ and $\underline{\hat{t}}$ are the (possibly empty) sequences of terms in input (resp. output) positions for p. To avoid notational proliferation, we use the markers $\check{}$ and $\hat{}$ both as mode designators and as symbol decorations (like primes and subscripts) when working with generic terms. Therefore, $\check{t}$ and $\hat{t}$ indicate possibly different terms in $p\,\check{t}\,\hat{t}$, and similarly for term sequences, as in $p\,\underline{\check{t}}\,\underline{\hat{t}}$ above.

At our level of abstraction, the rules in Figure 1 capture the operational semantics of this variant of $\mathscr{L}^s$: mode annotations are simply ignored. However, moded execution requires that two of the operational choices left open by those rules be resolved using some algorithmic strategy: the order in which rule **i_imp** searches for derivations of its two premises, and the substitution term that rule **i_all** picks.

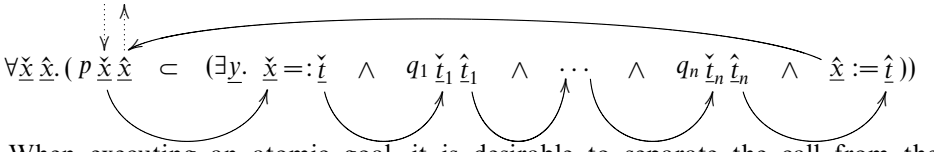
https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

For both, we will assume the same strategy as Prolog: implement rule **i_imp** left to right and implement rule **i_all** lazily by replacing each variable x with a "logical variable" X which is instantiated incrementally through unification. This allows us to view an atomic goal as a (non-deterministic) procedure call. In a well-moded program (Debray and Warren 1988), terms in input position are seen as the actual arguments of this procedure, and terms in output position yield return values.

In this section, we will not formalize the notion of well-modedness — see (Debray and Warren 1988) for Prolog and (Sarnat 2010) for Twelf — nor refine our operational semantics to make goal evaluation order and unification explicit — see (Pientka 2003). We will instead refine our compilation process to account for mode information and produce compiled programs that, if well-moded, can be executed without appealing to unification.

4.2 Target Language

In $\mathscr{L}_1^c$, a (well-moded) Horn clause $\forall \underline{y}.\, p\ \check{\underline{t}}\ \hat{\underline{t}} \subset a_1 \subset \ldots \subset a_n$ was compiled into $\forall \check{\underline{x}}\ \hat{\underline{x}}.\, (p\ \check{\underline{x}}\ \hat{\underline{x}} \subset \exists \underline{y}.\, (\check{\underline{x}} \doteq \check{\underline{t}} \wedge \hat{\underline{x}} \doteq \hat{\underline{t}} \wedge a_1 \wedge \ldots \wedge a_n))$. Here, the left-to-right execution order forces us to guess the final values of the output variables $\hat{\underline{x}}$ before the goals in its body have been fully executed. In $\mathscr{L}_2^c$, we will move the equality $\hat{\underline{x}} \doteq \hat{\underline{t}}$ after the last goal a_n. Since $\hat{\underline{x}}$ appear nowhere else in the residual, this equality is no more than an assignment of the computed instance of $\hat{\underline{t}}$ to $\hat{\underline{x}}$. Accordingly, we will write it as $\hat{\underline{x}} := \hat{\underline{t}}$. Furthermore, in a well-moded program, this clause will be invoked with ground terms in input position, so that $\check{\underline{x}}$ will be bound to ground terms. Then, the input equality $\check{\underline{x}} \doteq \check{\underline{t}}$ will match the variables in $\check{\underline{t}}$ with appropriate subterms. For this reason, we will write it as $\check{\underline{x}} =: \check{\underline{t}}$. Expanding each goal a_i into $q_i\ \check{\underline{t}}_i\ \hat{\underline{t}}_i$, the above clause will be compiled (almost) as follows, where the arrows represent the data flow of a well-moded execution (note that it parallels the control flow):



When executing an atomic goal, it is desirable to separate the call from the verification that the output terms returned by the caller match the expected output terms in this goal. We will do so by rewriting any atomic goal $q\ \check{\underline{t}}\ \hat{\underline{t}}$ in a compiled clause into the formula $\exists \underline{z}.\, (q\ \check{\underline{t}}\ \underline{z} \wedge \underline{z} =: \hat{\underline{t}})$ for fresh variables $\underline{z}$. This transformation preserves the left-to-right control and data flow. No special provision needs to be made for the input arguments of q as variables in it will have been instantiated to ground terms at the moment the call is made.

Next, we again generalize this intuition to any formula in $\mathscr{L}^s$, not just Horn clauses. Our third target language, $\mathscr{L}_2^c$, is defined by the following grammar.

$$
\begin{array}{rrcl}
\textit{Goal Matches:} & M & ::= & \top \mid M \wedge z =: \hat{t} \\
\textit{Atomic Goals:} & F & ::= & p\ \check{\underline{t}}\ \hat{\underline{z}} \wedge M \mid \exists z.\, F \\
\textit{Goals:} & G & ::= & F \mid C \supset G \mid \forall x.\, G \\
\textit{Clauses:} & C & ::= & R \supset p\ \check{\underline{x}}\ \hat{\underline{x}} \mid \forall x.\, C \\
\textit{Residuals:} & R & ::= & \check{x} =: t \mid \hat{x} := t \mid \top \mid R \wedge G \mid \exists x.\, R
\end{array}
\qquad
\textit{Programs:}\quad \Psi ::= \cdot \mid \Psi, C
$$

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

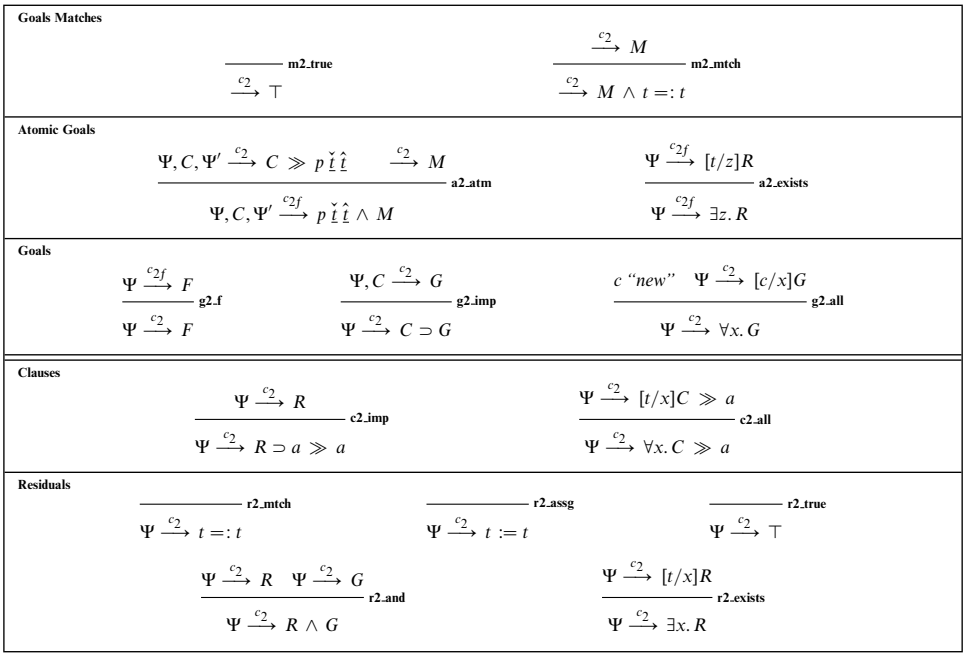


Fig. 7. Search Semantics of $\mathscr{L}_2^c$.

Residuals (R) refine the equality predicate $x \doteq t$ of $\mathscr{L}_1^c$ into a matching predicate $x =: t$ and an assignment predicate $x := t$. At our level of abstraction, they behave just like equality. During well-moded execution, the match predicate will have the form $t_g =: t_v$ where t_g is a ground term while t_v may contain variables. It will bind these variables to ground subterms of t_g, thereby realizing matching. However, presented with programs that are not well-moded, the terms t_g cannot be assumed to be ground and $=:$ performs unification. The assignment predicate will be called as $x := t$ where x is a variable and t a term — a ground term for well-moded programs. It simply binds x to t. Compiled clauses and programs are just like in $\mathscr{L}_1^c$.

Following the motivations above, an atomic goal $p\ \check{t}\ \hat{t}$ is not compiled any more to itself as in $\mathscr{L}_1^c$, but to a formula F of the form $\exists \underline{z}.(q\ \check{t}\ \hat{z} \wedge \check{z} =: \hat{t})$. In the grammar above, we isolated the match predicates $\check{z} =: \hat{t}$ as the non-terminal M.

We specify the operational semantics of $\mathscr{L}_2^c$ by means of the following five judgments:

$\xrightarrow{c_2} M$	*M is provable*
$\Psi \xrightarrow{c_{2f}} F$	*F is uniformly provable from Ψ*
$\Psi \xrightarrow{c_2} G$	*G is uniformly provable from Ψ*
$\Psi \xrightarrow{c_2} C \gg a$	*a is immediately entailed by C in Ψ*
$\Psi \xrightarrow{c_2} R$	*R is uniformly provable from Ψ*

which parallel the grammar just presented. The resulting operational semantics is shown in Figure 7. The rules for clauses are unchanged with respect to $\mathscr{L}_1^c$ while that language's residual rule for equality has been duplicated into isomorphic rules for

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

matching and assignment. The rules for compiled goals have instead proliferated due to our handling of terms in output position in atomic goals. Observe that rule **a2_atm** is essentially a combination of rule **g1_atm** in $\mathscr{L}_1^c$ and the rule for conjunction. Rules **a2_exists** and **m2_true** are just the standard rules for existential quantification and truth. Rule **m2_mtch** combines the rules for conjunction and matching.

Just like in the case of $\mathscr{L}_1^c$, the rules in Figure 7 construct proofs that are uniform (Miller et al. 1991), which makes $\mathscr{L}_2^c$ an abstract logic programming language. In a successful derivation, this operational semantics decomposes a goal to formulas of the form $F = \exists \underline{z}.\,(p\,\check{\underline{t}}\,\underline{\hat{z}} \wedge \check{\underline{z}} =: \hat{\underline{t}})$ (rules in the "Goals" segment). Then, rules **a2_exists**, **m2_mtch** and **m2_true** necessarily reduce it in a few steps into the atomic formula $p\,\check{\underline{t}}\,\hat{\underline{t}}$. Similarly to $\mathscr{L}_1^c$, the left premise of rule **a2_atm** selects a clause and focuses on it until it finds a potentially matching head ("Clauses" segment). It then proceeds to decomposing its body ("Residuals" segment) and the cycle repeats with whatever goals it finds in there.

As just noticed, any atomic goal F of the form $\exists \underline{\hat{z}}.\,(p\,\check{\underline{t}}\,\underline{\hat{z}} \wedge \underline{\hat{z}} =: \hat{\underline{t}})$ is necessarily reduced to $p\,\check{\underline{t}}\,\hat{\underline{t}}$ by as many applications of rule **a2_exists** as there are variables in $\underline{\hat{z}}$, a pass-through instance of **a2_atm** via its right branch, and a similar number of uses of rules **m2_mtch** and **m2_true** respectively. This entails that the macro-rule **a2_atm′**, on the left-hand side of the following display, is derivable:

$$\frac{\Psi \xrightarrow{c_2} p\,\check{\underline{t}}\,\hat{\underline{t}}}{\Psi \xrightarrow{c_2} \exists \underline{z}.\,(p\,\check{\underline{t}}\,\underline{z} \wedge \underline{z} =: \hat{\underline{t}})}\ \textbf{a2_atm}' \qquad \frac{\Psi, C, \Psi' \xrightarrow{c_2} C \gg p\,\check{\underline{t}}\,\hat{\underline{t}}}{\Psi, C, \Psi' \xrightarrow{c_{2f}} p\,\check{\underline{t}}\,\hat{\underline{t}}}\ \textbf{a2_atm}''$$

Having factored rule **a2_atm′** out, the work performed by **a2_atm** degenerates to rule **a2_atm″** on the right-hand side of the above display, which is akin to **u_atm**. The system obtained by replacing the **m2_*** and **a2_*** rules as well as **g2_f** with rules **a2_atm′** and **a2_atm″** is indeed equivalent to the rule set in Figure 7.

Rule **a2_atm′** entices us to interpret the compiled formula $\exists \underline{z}.\,(p\,\check{\underline{t}}\,\underline{z} \wedge \underline{z} =: \hat{\underline{t}})$ for an atomic goal $p\,\check{\underline{t}}\,\hat{\underline{t}}$ as a synthetic operator $\mathsf{call}\ p\,\check{\underline{t}} =: \hat{\underline{t}}$ which invokes a clause for p with its (ground) input arguments $\check{\underline{t}}$ and matches the returned values against its terms $\hat{\underline{t}}$ in output position.

Having recovered atomic goals $p\,\check{\underline{t}}\,\hat{\underline{t}}$ through rules **a2_atm′** and **a2_atm″**, we can carry out a sequence of reasoning steps similar to what led us to the backchaining rule for $\mathscr{L}_1^c$. Exposing the trailing assignments, a generic compiled clause C has the form $\forall \check{\underline{x}}\,\hat{\underline{x}}.\,(\exists \underline{y}.\,R \wedge \hat{\underline{x}} := \hat{\underline{s}}) \supset p\,\check{\underline{x}}\,\hat{\underline{x}}$. In a successful derivation, all rule **a2_atm″** does is to pick such a clause. Then, applications of rule **c2_all** will instantiate variables $\check{\underline{x}}\,\hat{\underline{x}}$ with the terms $\check{\underline{t}}\,\hat{\underline{t}}$, and next rule **c2_imp** will invoke the instantiated residual $[\check{\underline{t}}/\check{\underline{x}}, \hat{\underline{t}}/\hat{\underline{x}}](\exists \underline{y}.\,R \wedge \hat{\underline{x}} := \hat{\underline{s}})$. Now, because $\hat{\underline{x}}$ does not occur in R and $\check{\underline{x}}\,\hat{\underline{x}}$ cannot appear in $\hat{\underline{s}}$, this formula reduces to $\exists \underline{y}.\,([\check{\underline{t}}/\check{\underline{x}}]R \wedge \hat{\underline{t}} := \hat{\underline{s}})$ by pushing the substitution in. Rule **r2_exists** will then instantiate the variables $\underline{y}$ with terms $\underline{u}$ (which cannot mention variables $\check{\underline{x}}\,\hat{\underline{x}}$). Pushing this substitution in yields the formula $[\check{\underline{t}}/\check{\underline{x}}, \underline{u}/\underline{y}]R \wedge \hat{\underline{t}} := [\underline{u}/\underline{y}]\hat{\underline{s}}$ since variables in $\underline{y}$ can occur in neither $\check{\underline{t}}$ nor $\hat{\underline{t}}$. Finally, by rule **r2_assg**, $\hat{\underline{t}}$ and $[\underline{u}/\underline{y}]\hat{\underline{s}}$ must be equal in a successful derivation. This necessary sequence of steps is captured

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

by the following derived backchaining macro-rule,

$$\frac{\Psi, C, \Psi' \xrightarrow{c_2} [\check{t}/\check{x}, u/y]R}{\Psi, \underbrace{\forall \check{x}\, \hat{x}.\, (\exists y.\, R \wedge \hat{x} := \hat{s}) \supset p\, \check{x}\, \hat{x}}_{C}, \Psi' \xrightarrow{c_2} p\, \check{t}\, [u/y]\hat{s}}\ \mathbf{g2_atm'}$$

where we have carried out the assignment $\hat{t} := [u/y]\hat{s}$ in the conclusion. This rule can be seen as a refinement of **g1_atm′** in $\mathscr{L}_1^c$ that makes use of the trailing assignment in the compiled clauses of $\mathscr{L}_2^c$. With this derived inference, rules **a2_atm″**, **c2_imp** and **c2_all** become unnecessary: the system consisting of rules **a2_atm′**, **g2_atm′**, the goal rules for implication and universal quantification, and the residual rules is equivalent to that in Figure 7.

Taking rule **g2_atm′** as primitive amounts to replacing compiled clauses with the following synthetic connective, which refines $\mathscr{L}_1^c$'s $\Lambda_p y.\, R$.

$$\underbrace{\forall \check{x}\, \hat{x}.\, p\, \check{x}\, \hat{x} \subset}_{\Lambda_p \check{x}.} \quad \exists y.\, (R \quad \wedge \quad \underbrace{\hat{x} := \hat{t}}_{\mathsf{return}\ \hat{t}})$$
$$\exists y.\, (R \quad ; \quad$$

The variables y are then interpreted as local variables for the execution of this clause. In this, they are akin to the $\mathtt{Y}n$ permanent variables of the WAM (Aït-Kaci 1991).

In a valid proof in this system, an occurrence of **a2_atm′** is always immediately followed by an instance of **g2_atm′**: the conclusion of the latter must match the premise of the former. This fact realizes the requirement that, upon returning from a call, the output terms, here $[u/y]\hat{s}$, must be checked against the terms in output position of the caller.

4.3 Compilation

Compilation transforms logic programs in $\mathscr{L}^s$ to compiled programs in $\mathscr{L}_2^c$. The input does not have to be well-moded at the level of detail considered here, but this would be operationally advantageous in a refinement of the semantics in Figure 7 that handles quantifiers lazily. We will make use of two auxiliary notions in this section: pseudo clauses that we encountered already in Section 3.2 and the analogous notion of pseudo atomic goal. They are defined as follows:

$$\textit{Pseudo Clauses:}\quad \mathscr{C} ::= \square \supset p\, x \mid \forall x.\mathscr{C}$$
$$\textit{Pseudo Atomic Goals:}\quad \mathscr{F} ::= p\, \check{t}\, \hat{z} \wedge \square \mid \exists z.\mathscr{F}$$

Just like pseudo clauses retain the outer structure of a clause replacing the embedded residual with a hole ($\square$), pseudo atomic goals have a hole in place of their trailing matches. The general form of pseudo clauses and pseudo atomic formulas, accounting for input and output positions, are $\forall \check{x}\, \hat{x}.\, \square \supset p\, \check{x}'\, \hat{x}'$ and $\exists \hat{z}.\, (p\, \check{t}\, \hat{z}' \wedge \square)$. In Section 3.2, wrote $\mathscr{C}[R]$ for the replacement of the hole of $\mathscr{C}$ with the residual R and noted that variable capture could (and generally will) occur. Similarly, we write $\mathscr{F}[M]$ for replacement of the hole of $\mathscr{F}$ with matches M.

The compilation process is modeled by the following five judgments, which are reminiscent of the compilation judgments $\mathscr{L}_1^c$. They are more complex because clause

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

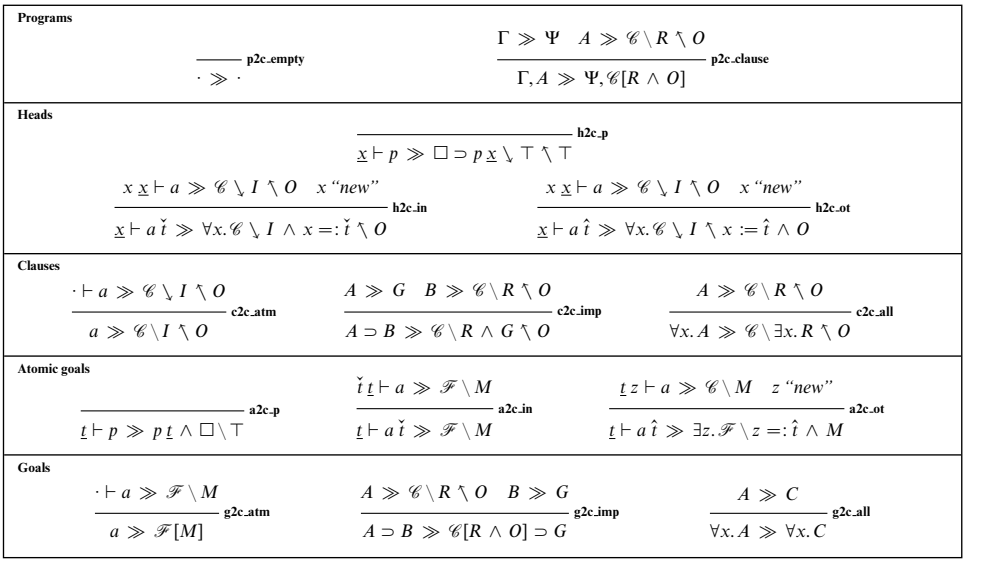


Fig. 8. Compilation of $\mathscr{L}^s$ into $\mathscr{L}_2^c$.

compilation now needs to handle both matching and assignment as opposed to a generic equality. Furthermore, a new judgment is needed to compile atomic goals.

$\Gamma \gg \Psi$	*Program* Γ *is compiled to* Ψ
$\underline{x} \vdash a \gg \mathscr{C} \searrow I \nwarrow O$	*Head* a *with* $\underline{x}$ *is compiled to* $\mathscr{C}$, I *and* O
$A \gg \mathscr{C} \setminus R \nwarrow O$	*Clause* A *is compiled to* $\mathscr{C}$, R *and* O
$\underline{t} \vdash a \gg \mathscr{F} \setminus M$	*Atomic goal* a *with* $\underline{t}$ *is compiled to* $\mathscr{F}$ *and* M
$A \gg G$	*Goal* A *is compiled to* G

We write I and O for a conjunction of matches (compilation of terms in input position) and assignments (compilation of output terms), respectively, in the body of a compiled clause. In compiled atomic goals, we write M for a conjunction of matches.

The rules for compilation, which define these judgments, are shown in Figure 8. Compiling a clause A, modeled by the judgment $A \gg \mathscr{C} \setminus R \nwarrow O$, returns a pseudo clause $\mathscr{C}$, the residual R (inclusive of input matches) and the output assignments O that will fill its hole. The rules in the "Clauses" segment build up this residual starting with the compilation of its head, which is displayed in the "Heads" segment. The rules therein differ from the similar inference for $\mathscr{L}_1^c$ by the fact that they dispatch terms in input and output positions in the I and O zones of the judgment as matches and assignments respectively. Residuals and assignments are plugged in the hole of the pseudo clause once this clause has been fully compiled, as can be seen in the "Programs" segment and in rule **g2c_imp**.

The compilation of goals differs from $\mathscr{L}_1^c$ for the treatment of atomic formulas: upon encountering an atom a, the compilation appeals to the new judgment $\cdot \vdash a \gg \mathscr{F} \setminus M$. It generates a pseudo atomic formula $\mathscr{F}$ and matches M, which are integrated in rule **g2c_atm**. The zone to the left of the turnstile serves as an accumulator, very much like when compiling heads.

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

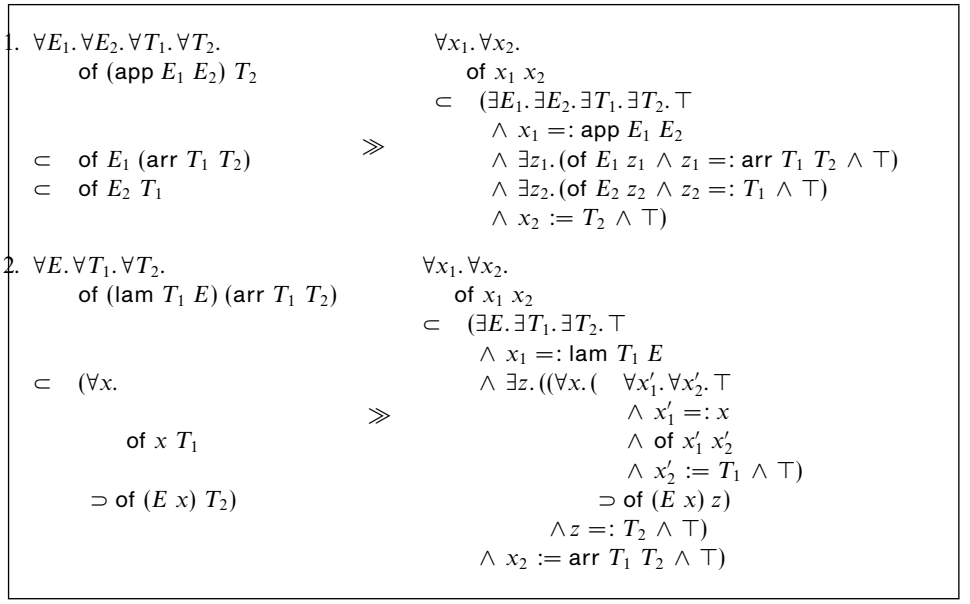


Fig. 9. $\mathscr{L}_2^c$ Compilation Example

Target language, $\mathscr{L}_2^c$, is sound and complete with respect to $\mathscr{L}^s$. The following lemma collects some auxiliary results needed to prove this property. The first two statements are proved by induction on the structure of a; the third by induction on the given derivation.

Lemma 4.1

- If $\underline{x} \vdash a \gg \mathscr{C} \setminus I \nwarrow O$, then for any term sequence $\underline{t}$ of the same length as $\underline{x}$ and program Ψ we have $\Psi \xrightarrow{c_2} [\underline{t}/\underline{x}](\mathscr{C}[I \wedge O]) \gg a\,\underline{t}$.
- If $\underline{t} \vdash a \gg \mathscr{F} \setminus M$, then for all Ψ we have $\Psi \xrightarrow{c_2} \mathscr{F}[M] \gg a\,\underline{t}$.
- If $\Psi \xrightarrow{c_2} \mathscr{C}[R] \gg a$, then $\Psi \xrightarrow{c_1} R$.

We have the following soundness and completeness theorems for $\mathscr{L}_2^c$. In both cases, the proof proceeds by mutual induction over the first derivation in the antecedent.

Theorem 4.2 (*Soundness of the compilation to* $\mathscr{L}_2^c$)

- If $\Gamma \xrightarrow{u} A$, $\Gamma \gg \Psi$ and $A \gg G$, then $\Psi \xrightarrow{c_2} G$.
- If $\Gamma \xrightarrow{u} A \gg a$, $\Gamma \gg \Psi$ and $A \gg \mathscr{C} \setminus R \nwarrow O$, then $\Psi \xrightarrow{c_2} \mathscr{C}[R \wedge O] \gg a$.

Theorem 4.3 (*Completeness of the compilation to* $\mathscr{L}_2^c$)

- If $\Psi \xrightarrow{c_2} G$, $\Gamma \gg \Psi$ and $A \gg G$, then $\Gamma \xrightarrow{u} A$.
- If $\Psi \xrightarrow{c_2} C \gg a$, $\Gamma \gg \Psi$, $C = \mathscr{C}[R \wedge O]$ and $A \gg \mathscr{C} \setminus R \nwarrow O$, then $\Gamma \xrightarrow{u} A \gg a$.

To conclude this section, we revisit our ongoing examples. Here, we assume that the mode of the predicate of is of ˇ ˆ — the first argument is input and the second output. The result of compiling our two familiar clauses into $\mathscr{L}_2^c$ is shown in Figure 9. As in Section 3.2, the moded compilation process offers ample opportunities for optimization: matches and assignments with variables on both

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

side and the corresponding existential quantification can often be elided, and all occurrences of $\top$ can be optimized away.

It is instructive to rewrite these clauses with the two synthetic connectives introduced earlier for $\mathscr{L}_2^c$, again omitting $\top$ for readability:

$$\begin{aligned}\Lambda_{\mathsf{of}}\ x_1.\quad & \exists E_1.\exists E_2.\exists T_1.\exists T_2.\ \ x_1 =: \mathsf{app}\ E_1\ E_2\\ & \wedge\ \ \mathsf{call}\,(\mathsf{of}\ E_1)\ =:\ (\mathsf{arr}\ T_1\ T_2)\ \wedge\ \mathsf{call}\,(\mathsf{of}\ E_2)\ =:\ T_1;\\ & \quad\ \mathsf{return}\ T_2\\ \Lambda_{\mathsf{of}}\ x_1.\quad & \exists E.\exists T_1.\exists T_2.\ \ x_1 =: \mathsf{lam}\ T_1\ E\\ & \wedge\ \ \forall x.(\Lambda_{\mathsf{of}}\ x_1'.\ \ x_1' =: x\ ;\mathsf{return}\ T_1)\ \supset\ \mathsf{call}\,(\mathsf{of}\ (E\ x))\ =:\ T_2;\\ & \quad\ \mathsf{return}\,(\mathsf{arr}\ T_1\ T_2)\end{aligned}$$

5 Larger Source Languages

In (Cervesato 1998), we illustrated our original abstract logical compilation method on the language of hereditary Harrop formulas. This language differs from $\mathscr{L}^s$ for the presence of conjunction (formulas of the form $A \wedge B$) and truth ($\top$). While our original treatment could handle them easily (in a clause position, they were compiled to disjunctions and falsehood respectively), the approach taken in Sections 3 and 4 does not support them directly. The problem is that, as soon as we allow these connectives, clauses can have multiple heads (or even none). Consider for example:

$$\forall x.\forall y.q\ x\ y \supset (p_1\ x\ y\ \wedge\ (r\ x\ y \supset p_2\ x))$$

This clause has two heads: $p_1\ x\ y$ and $p_2\ x$. What should it be compiled to? To ensure immediacy (embodied in the macro-rule **g1_atm′**), our compilation strategy produces a pseudo clause applied to a residual, thereby exposing the (flattened) head of a compiled clause as close to the top level as possible. How to achieve this now that there may be more than one head?

One approach to dealing with this problem is to observe that $\wedge$ distributes over (the antecedent of) $\supset$ and $\forall$. By doing so to the above example, we obtain the formula

$$(\forall x.\forall y.q\ x\ y \supset p_1\ x\ y)\ \wedge\ (\forall x.\forall y.q\ x\ y \supset r\ x\ y \supset p_2\ x)$$

Observe that it is a conjunction of $\mathscr{L}^s$ clauses. Each of them can now be compiled as in Section 3 and the results can be combined by means of a disjunction. This approach generalizes to the full language of hereditary Harrop formulas. It pushes the conjunctions to the outside, leaving inner formulas resembling the clauses of $\mathscr{L}_0^c$ (conjunction and truth in a goal position are left alone as they are not problematic). Clauses with no head (e.g., $A \supset \top$) are reduced to $\top$. These preprocessing steps can be implemented as a source-code transformation or integrated in the compilation process.

The other abstract logic programming language examined in (Cervesato 1998) is the language of linear hereditary Harrop formulas, found at the core of Lolli (Hodas and Miller 1994) and LLF (Cervesato and Pfenning 2002). The improved compilation process discussed in this paper extends directly in the presence of linearity. Because

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

linear hereditary Harrop formulas feature a form of conjunction and truth, the technical device just outlined is needed to obtain workable compiled clauses.

6 Future Work

The discussion in Section 4 sets the stage for a nearly functional operational semantics of well-moded programs. Indeed, given an atomic goal with ground terms in its input positions, proof search will instantiate its output positions to ground terms, if it succeeds. Being in a logic programming setting, more than one answer could be returned. Indeed, for well-moded programs, the clauses for a predicate implement a partial, non-deterministic function. This observation informed the choice of the notation for the synthetic operators we exposed: $\mathsf{call}\ p\ \check{\underline{t}} =: \hat{\underline{t}}$ and $\Lambda_p \underline{\check{x}}.\, \exists \underline{y}.\, (R; \mathsf{return}\ \hat{\underline{t}})$.

Now we believe that, in the case of well-moded programs, a more detailed operational semantics that exposes variable manipulations using logical variables and explicit substitutions (and restricts the execution order) can bring this functional interpretation to the surface. This would provide a logical justification for the natural impulse to give well-moded programs a semantics that is typical of functional programming languages, where atomic predicates carry just input terms and from which the terms in output position emerge by a process of reduction.

In future work, we intend to carry out this program by giving such a detailed operational semantics to $\mathscr{L}^s$ as well as well-moding rules. The goal will then be to perform logical transformations, akin to what we did in this paper, that expose this functional semantics for well-moded programs. It would also allow us to prove formally that the operator $=:$ of Section 4 can indeed be implemented as matching rather than general unification.

Acknowledgments

This work was supported by the Qatar National Research Fund under grant NPRP 09-1107-1-168. We are grateful to Frank Pfenning, Carsten Schürmann, Robert J. Simmons and Jorge Sacchini for the many fruitful discussions, as well as to the anonymous reviewers.

References

Aït-Kaci, H. 1991. *Warren's Abstract Machine: a Tutorial Reconstruction.* MIT Press.

Börger, E. and Rosenzweig, D. 1995. The WAM — definition and compiler correctness. In *Logic Programming: Formal Methods and Practical Applications*, C. Beierle and L. Pluemer, Eds. Computer Science and Artificial Intelligence, vol. 11. North-Holland, 21–90.

Cervesato, I. 1998. Proof-Theoretic Foundation of Compilation in Logic Programming Languages. In *1998 Joint International Conference and Symposium on Logic Programming — JICSLP'98*, J. Jaffar, Ed. MIT Press, Manchester, UK, 115–129.

Cervesato, I. and Pfenning, F. 2002. A Linear Logical Framework. *Information & Computation 179,* 1, 19–75.

Cervesato, I., Pfenning, F., Walker, D., and Watkins, K. 2003. A Concurrent Logical Framework II: Examples and Applications. Technical Report CMU-CS-02-102,

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press

Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA. March 2002, revised May.

Debray, S. K. and Warren, D. S. 1988. Automatic mode inference for logic programs. *Journal of Logic Programming 5*, 207–229.

Hodas, J. S. and Miller, D. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation 110,* 2, 327–365.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. 1992. An abstract machine for *CLP*($\mathscr{R}$). In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation — PLDI'92*. San Francisco, CA.

Miller, D. and Nadathur, G. 1986. Higher-order logic programming. In *Proceedings of the Third International Logic Programming Conference*, E. Shapiro, Ed. London, 448–462.

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. 1991. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic 51*, 125–157.

Nadathur, G. and Mitchell, D. J. 1999. System description: Teyjus — a compiler and abstract machine based implementation of lambda prolog. In *Sixteenth Conference on Automated Deduction (CADE'99)*, H. Ganzinger, Ed. 287–291.

Pfenning, F. and Schürmann, C. 1999. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *Proceedings of the 16th International Conference on Automated Deduction — CADE-16*. Springer-Verlag LNAI 1632, Trento, Italy, 202–206.

Pientka, B. 2003. Tabled higher-order logic programming. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University.

Russinoff, D. M. 1992. A verified Prolog compiler for the Warren abstract machine. *Journal of Logic Programming 13*, 367–412.

Sarnat, J. 2010. Syntactic finitism in the metatheory of programming languages. Ph.D. thesis, Department of Computer Science, Yale University.

Stirling, C. 2009. Decidability of higher-order matching. *Logical Methods in Computer Science 5,* 3.

Warren, D. H. D. 1983. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA. Oct.

Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. 2003. A Concurrent Logical Framework I: Judgments and Properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA. March 2002, revised May.

https://doi.org/10.1017/S1471068412000221 Published online by Cambridge University Press