

# Mode and Termination Checking for Higher-Order Logic Programs

Ekkehard Rohwedder and Frank Pfenning\*

Department of Computer Science  
Carnegie Mellon University  
Pittsburgh, Pennsylvania 15213-3890, U.S.A.  
er+@cs.cmu.edu, fp+@cs.cmu.edu

**Abstract.** We consider how *mode* (such as input and output) and *termination* properties of typed higher-order constraint logic programming languages may be declared and checked effectively. The systems that we present have been validated through an implementation and numerous case studies.

## 1 Introduction

Just like other paradigms logic programming benefits tremendously from types. Perhaps most importantly, types allow the early detection of errors when a program is checked against a type specification. With some notable exceptions most type systems proposed for logic programming languages to date (see [18]) are concerned with the declarative semantics of programs, for example, in terms of many-sorted, order-sorted, or higher-order logic. Operational properties of logic programs which are vital for their correctness can thus neither be expressed nor checked and errors will remain undetected.

In this paper we consider how the declaration and checking of *mode* (such as input and output) and *termination* properties of logic programs may be extended to the typed higher-order logic case. While we do not cast our proposal as a type system in the traditional sense, our design choices were motivated by the desirable characteristics of type systems. In particular, it should be uniform, intuitive, concise and efficiently decidable. Furthermore, relatively few natural and correct programs should be rejected as ill-moded or non-terminating.

We present a system for mode and termination properties of Elf programs. Elf [17] is a higher-order constraint logic programming language based on the LF logical framework. Although Elf encompasses pure Prolog, it has been designed as a meta-language for the specification, implementation, and meta-theory of programming languages and logics. We have validated our system through an implementation and post-hoc analysis of numerous existing case studies from this domain.

Elf includes dependently typed higher-order functions and proof objects to represent the abstract syntax and semantic judgments of many object languages

---

\* This work was supported by NSF Grant CCR-930383

in a concise and natural manner. The presence of these features presents a challenge, but also provides an opportunity. The challenge is to extend previous work on modes (see, *e.g.*, [10, 4, 7, 25, 27]) and termination (see, *e.g.*, [24, 1]) to deal with types and higher-order constraint simplification. On the other hand it turns out that we can take advantage of the already very expressive underlying type structure in our analysis. In order to concentrate our effort on higher-order terms and dependent types, we employ very basic but practical notions for modes and termination criteria.

The principal contributions of this paper are practical systems for mode and termination analysis of higher-order logic programs in Elf. Their correctness proofs are only sketched in this paper. In addition we outline a success continuation passing semantics for Elf and present a subterm ordering for higher-order terms which may be of independent interest. We expect a minor variation of these systems to be applicable to  $\lambda$ Prolog [15].

The remainder of the paper is organized as follows. We introduce the Logical Framework, Elf, and a sketch of its operational semantics based on success continuations in Section 2. Mode analysis, including a mode-checking system for Elf programs is presented in Section 3. Next we consider a subterm order for higher-order terms and outline a termination checker for Elf programs in Section 4. We discuss pragmatic aspects of our implementation and provide an assessment in Section 5. In the conclusion we discuss some related and future work.

## 2 The Logical Framework and Elf

We give a brief introduction to the Logical Framework, the theory on which Elf is based. After an overview of Elf we present some sample programs and a formal execution model for the Elf interpreter.

**Logical Framework.** The *Logical Framework (LF)* [6] is a calculus of dependent types consisting of three staged syntactic levels.

$$\begin{array}{ll}
\text{Kinds :} & K ::= \text{type} \mid \Pi x:A.K \\
\text{Types :} & A ::= a \ M_1 \dots M_n \mid \Pi x:A_1.A_2 \\
\text{Objects :} & M ::= c \mid x \mid \lambda x:A.M \mid M_1 \ M_2 \\
\text{Signatures :} & \Sigma ::= \cdot \mid \Sigma, a : K \mid \Sigma, c : A \\
\text{Contexts :} & \Gamma ::= \cdot \mid \Gamma, x : A
\end{array}$$

Here  $\Pi x:A_1.A_2$  denotes the dependent function type or dependent product: the type  $A_2$  may depend upon an object  $x$  of type  $A_1$ . Whenever  $x$  does not occur free in  $A_2$  we may abbreviate  $\Pi x:A_1.A_2$  as  $A_1 \rightarrow A_2$ . In the grammar above,  $a$  and  $c$  stand for type families and object constants, respectively. They are introduced through a *signature*. Below we assume that we have fixed a signature  $\Sigma$ . The types of free variables in a term  $M$  are provided by a *context*.

The following principal judgments characterize the LF type theory [6]:

$$\begin{array}{l}
\Gamma \vdash_{\Sigma} M \equiv M' : A \text{ and } \Gamma \vdash_{\Sigma} A \equiv A' : \text{type} \text{ — type and object equivalences;} \\
\vdash \Sigma, \vdash_{\Sigma} \Gamma, \text{ and } \Gamma \vdash_{\Sigma} K \text{ — the validity of signatures, contexts and kinds;} \\
\Gamma \vdash_{\Sigma} A : K \text{ and } \Gamma \vdash_{\Sigma} M : A \text{ — assigning kinds to types and types to objects.}
\end{array}$$

The equivalence  $\equiv$  is equality modulo  $\beta\eta$ -conversion. We will rely on the fact that canonical (*i.e.*, long  $\beta\eta$ -normal) forms of LF objects are computable and that equivalent LF objects have the same canonical form up to  $\alpha$ -conversion. We assume that a constant may be declared at most once in a signature and a variable at most once in a context, employing implicit renaming of bound variables in cases where this assumption would be violated. We also generally assume that all signatures and contexts are valid. Similarly, we write  $[N/x]M$  and  $[N/x]A$  for capture-avoiding substitution in an object or type. We define the *head* of a type  $\mathbf{hd}(\Pi x_1:A_1 \dots \Pi x_m:A_m.a M_1 \dots M_n)$  as  $a$ . Since types of valid objects are unique up to  $\beta\eta$ -conversion we sometimes write  $A_M$  for the canonical type of  $M$ .

**Elf.** Using the propositions-as-types and derivations-as-objects correspondences, the LF type theory can also be viewed as a logic calculus, for which—in form of the language Elf [20, 17]— we have an implementation in the spirit of constraint logic programming. The Elf interpreter type-checks programs (*i.e.*, LF signatures) presented to it and searches for derivations of goals in the manner of Prolog, replacing unification by simplification of constraints involving higher-order functions.

Elf employs the following concrete syntax for LF terms:<sup>2</sup>  $A \rightarrow B$  for  $A \rightarrow B$ ,  $\{x:A\} B$  for  $\Pi x:A.B$ , and  $[x:A] M$  for  $\lambda x:A.M$ . Instead of  $A \rightarrow B$  we may write  $B <- A$ . Note that the former arrow is right-associative and the latter left-associative. A declaration is terminated with “.” and capitalized identifiers that occur free in it are interpreted as logic variables and implicitly  $\Pi$ -quantified. When a constant is used, its implicit  $\Pi$ -abstractions do not need to be supplied with arguments—they are determined by type reconstruction. These features are responsible in large part for the conciseness and practicality of the Elf language.

**Example.** We introduce a signature with several simple relations to be used in examples throughout. These examples also highlight similarities and differences between Prolog and Elf. Elf kinds are employed for type declarations, such as `nat : type` or `ack : nat -> nat -> nat -> type`, whereas Elf object constants are used as syntax constructors and as clause labels. Lambda expressions `exp` are a prototypical example of higher-order abstract syntax [13], where binding at the object level is represented as a higher-order term at the meta-level.

```

nat  : type.                                % Natural numbers
  0: nat.
  s: nat -> nat.

ack   : nat -> nat -> nat -> type.          % Ackermann function
  ack1: ack 0 Y Y.
  ack2: ack (s X) 0 (s 0).
  ack3: ack (s X) (s Y) R <- ack (s X) Y Q <- ack X Q R.

exp  : type.                                % Untyped lambda expressions
  app: exp -> exp -> exp.

```

---

<sup>2</sup> We will mix Elf and LF syntax below.

```

lam: (exp -> exp) -> exp.

cp: exp -> exp -> type.                % Copying lambda expressions
  cpapp: cp (app E1 E2) (app F1 F2)
         <- cp E1 F1 <- cp E2 F2.
  cplam: cp (lam E) (lam F)
         <- ({x:exp} cp x x -> cp (E x) (F x)).

eval : exp -> exp -> type.             % Evaluating lambda expressions
  evapp: eval (app E1 E2) V
         <- eval E1 (lam E1')
         <- eval E2 V2
         <- eval (E1' V2) V.
  evlam: eval (lam E) (lam E).

```

The full LF type for `cplam` is  $\Pi E:\text{exp}\rightarrow\text{exp}.\Pi F:\text{exp}\rightarrow\text{exp}.\Pi G:(\Pi x:\text{exp}.\Pi D:\text{cp } x x.\text{cp } (E x) (F x)).\text{cp } (\text{lam } E) (\text{lam } F)$ .

**Dynamic and static.** In order to control Elf’s search, the programmer specifies for every type family  $a$  whether a variable of type  $A = \Pi x_1:A_1.\dots\Pi x_n:A_n.a M_1\dots M_n$  represents a goal or a logic variable. In the first case we call the type  $A$  *dynamic*, and in the second case  $A$  is *static*. In our example above we could define that `nat`, `exp` are static and `ack`, `cp`, and `eval` are dynamic. Since a dynamic type  $a M_1\dots M_n$  plays the rôle of an atomic goal  $a(M_1,\dots,M_n)$  in Prolog, we will use the terms *type family* and *predicate* interchangeably. Similarly, we refer to the declaration of a constant with dynamic type as a *clause*.

**Elf execution model.** In [17] a nondeterministic state transition system is given for Elf execution. This was sufficient to obtain the necessary soundness and non-deterministic completeness properties. For our modes and termination system the subgoal ordering is critical, so we make it explicit in the form of an operational semantics based on *success continuations*. Backtracking and failure, on the other hand, are not modeled explicitly. Specifically, we do not reflect clause ordering within the signature  $\Sigma$  or the context  $\Delta$ , since our mode and termination system is independent of the clause selection mechanism. The transition system for Elf execution uses states of the following form, the components of which are explained below.

$$\left(\Delta' \Vdash_{\theta;\Delta;\kappa} F\right) \&K$$

**Goal formulas.** We need goal formulas  $F$  to represent success ( $\top$ ), conjunction ( $F_1 \wedge F_2$ ), subgoals ( $M \in A$ , provided the type  $A$  is dynamic) and unification between objects ( $M \doteq N : A$ ) or types ( $A \doteq B : \text{type}$ ). Constraints  $\kappa$  are characterized through a subclass of formulas. They arise during Elf “unification” which is implemented as constraint simplification [17]. Every solution to the constraints  $\kappa$  returned after simplification will unify the indicated equation. The only equations permitted in constraints are so-called flex-flex and flex-rigid pairs where at least one of the flexible terms is not a generalized variable (see section 3).

As long as the terms  $M_1$  and  $M_2$  are first-order or higher-order patterns after application of  $\theta$  (see [12]), this guarantees most general unifiers. It should be noted that constraint simplification either fails or succeeds with a unique answer constraint.

$$\begin{aligned} \text{Goal formulas } F &::= \top \mid M \doteq N : A \mid A \doteq B : \text{type} \mid M \in A \mid F_1 \wedge F_2 \\ &\quad \mid \forall x:A.F \mid \exists x:A.F \\ \text{Constraints } \kappa &::= \top \mid M \doteq N : A \mid \kappa_1 \wedge \kappa_2 \mid \forall x:A.\kappa \mid \exists x:A.\kappa \end{aligned}$$

**Mixed-prefix contexts.** Variables  $\Delta$  and  $\Delta'$  denote mixed-prefix contexts in the sense of Miller [14]. They allow us to model existentially and universally quantified variables (*i.e.*, logic variables and parameters, respectively) as well as dependencies between them. The substitution  $\theta$  relates these contexts:  $\Delta' \vdash \theta : \Delta$ . It maps the goal formula  $F$  defined in the context  $\Delta$  to a formula over  $\Delta'$ . We obtain the LF context  $\overline{\Delta}$  corresponding to a mixed-prefix context  $\Delta$  by dropping all quantifiers.

$$\begin{aligned} \text{Mixed prefix } \Delta &::= \cdot \mid \Delta, \forall x:A \mid \Delta, \exists x:A \\ \text{Substitutions } \theta &::= \cdot \mid \theta, \forall x/x \mid \theta, \exists M/x \end{aligned}$$

Substitutions  $\Delta' \vdash \theta : \Delta$  on mixed-prefix contexts are built with the axiom  $\Delta' \vdash \cdot : \cdot$  and the two following rules.

$$\frac{\Delta' \vdash \theta : \Delta \quad \overline{\Delta'} \vdash_{\Sigma} M : \theta A}{\Delta' \vdash (\theta, \exists M/x) : (\Delta, \exists x:A)} \quad \frac{\Delta' \vdash \theta : \Delta}{\Delta', \forall x:\theta A \vdash (\theta, \forall x/x) : (\Delta, \forall x:A)}$$

**Success continuation.** A success continuation  $K$  is used to enforce an execution order among the subgoals of a clause. For a given state  $(\Delta' \Vdash_{\theta, \Delta; \kappa} F) \& K$ , the continuation  $K$  is a (meta-level) function that takes as arguments a new mixed-prefix  $\Delta''$ , a new substitution  $\Delta'' \vdash \theta' : \Delta$ , and new constraints  $\kappa'$  and yields a new state. The initial continuation  $\mathbf{initial}_{\Delta'', \theta'; \kappa'}$  takes the same arguments and returns them as the final answer.

$$\begin{aligned} \text{State } \mathcal{S}_{\Delta'; \theta; \kappa} &::= (\Delta' \Vdash_{\theta, \Delta; \kappa} F) \& K \quad \text{where } \Delta' \vdash \theta : \Delta \\ \text{Continuations } K &::= \mathbf{initial}_{\Delta; \theta; \kappa} \mid \lambda \Delta. \lambda \theta. \lambda \kappa. \mathcal{S}_{\Delta; \theta; \kappa} \end{aligned}$$

**Transition system.** The possible Elf execution sequences are given by a transition system  $\mathcal{S} \longrightarrow \mathcal{S}'$  on states. Although we do not have space for the full transition system here, we summarize the effects on different goal formulas and examine in detail the rule responsible for backchaining. Formulas  $\top$  and  $M \in A$  with  $A$  static succeed immediately and invoke the success continuation  $K$ . For  $\forall x:A.F$  and  $\exists x:A.F$  the respective quantifier is added to the mixed-prefix context  $\Delta$  and later removed before the remainder of the search is started with  $K$ . The success continuation is also used to enforce that conjunctions  $F_1 \wedge F_2$  are solved from left to right. Unification  $M \doteq N : A$  is performed through Elf constraint simplification. A higher-order goal  $M \in \Pi x:A.B$  can be reduced to  $\forall x:A.(M \ x \in B)$ , while atomic goals are solved by backchaining.

**Backchaining.** For a goal  $M \in a \ M_0 \dots M_n$ , where  $a$  is dynamic, we nondeterministically pick a residuation clause  $h : \Pi x_1:A_1. \dots \Pi x_m:A_m. a \ N_1 \dots N_n$  from  $\Sigma$  or ( $\forall$ -quantified) from  $\Delta$ . The corresponding transition is

$$\left(\Delta' \Vdash_{\theta; \Delta; \kappa} M \in a M_1 \dots M_n\right) \&K \longrightarrow \left(\Delta' \Vdash_{\theta; \Delta; \kappa} F\right) \&K$$

where  $F$  stands for the following *residuation formula*

$$\begin{aligned} &\exists x_1:A_1. (\exists x_2:A_2. \dots (\exists x_m:A_m. \\ &\quad (a N_1 \dots N_n \doteq a M_1 \dots M_n : \text{type} \wedge h x_1 \dots x_m \doteq M : a M_1 \dots M_n) \\ &\quad \wedge x_m \in A_m) \dots \wedge x_2 \in A_2) \wedge x_1 \in A_1 \end{aligned}$$

which is responsible for unifying goal and clause heads, for building a derivation object, and for solving the newly introduced subgoals  $x_m \in A_m, \dots, x_1 \in A_1$  from the inside out.

Example. The execution of  $\left(\exists P \Vdash_{\exists P/P; \exists P; \top} P \in \text{cp} (\text{lam } \lambda x.x) (\text{lam } \lambda y.y)\right) \&$   
**initial** $_{\Delta'; \theta; \kappa}$  results in  $\Delta' \equiv \emptyset$ ,  $\theta \equiv \exists (\text{cp lam } (\lambda x.x) (\lambda y.y) (\lambda x.\lambda D.D)) / P$ , and  $\kappa \equiv \top$ .

### 3 Mode Analysis

**Modes.** Modes have been proposed for expressing aspects of the operational semantics of logic programs (see, *e.g.*, [7, 25, 27]). The simplest and most useful modes declare the *input* and *output* arguments of a predicate. The input arguments to a predicate should be ground when it is called. Upon successful return, the output arguments should be ground. This is often strengthened by requiring the output arguments to a predicate to be free logic variables when the predicate is called. In this paper we employ the first, most basic notion of modes — it offers sufficient characterization of the meta-theoretic relations that we are interested in and can be obtained in a direct manner even in the presence of higher-order terms, dependent types, and constraints. Also, since Elf does not offer the cut control operator, an identification of free variables is not as essential to us.

Thus we assign *polarities*  $p ::= + \mid - \mid *$  for input, output, and don't care arguments, respectively, and a *mode*  $m_a = \langle p_0, \dots, p_n \rangle$  for every dynamic type family  $a : \Pi x_1:A_1. \dots \Pi x_n:A_n. \text{type} \in \Sigma$ .<sup>3</sup> We also use the following abbreviation for the input positions of the predicate  $a$ :  $m_a^+ \stackrel{\text{def}}{=} \{i \mid m_a = \langle p_0, \dots, p_n \rangle \wedge p_i = +\}$ . Similarly defined are  $m_a^-$  and  $m_a^*$ . For our example signature we might declare modes such as:

$$m_{\text{ack}} \stackrel{\text{def}}{=} \langle -, +, +, - \rangle, \quad m_{\text{cp}} \stackrel{\text{def}}{=} \langle -, +, - \rangle, \quad \text{etc.}$$

The corresponding `%mode`-pragmas that declare these modes in an Elf program are more perspicuous:

```
%mode -ack  +X +Y -Z
%mode -cp   +E -F
```

<sup>3</sup> The polarity  $p_i$  refers to the  $i$ -th argument of  $a$  and  $p_0$  refers to the polarity of the derivation object.

In order to characterize the consistent goal invocations that respect such mode declarations, we first need to define *ground* terms. The judgment  $\Delta \vdash M : A$  **ground** is straightforward—it holds if all variables in the canonical form of  $M$  are parameters, *i.e.*, universally quantified in  $\Delta$ .

**Consistency.** The consistency conditions before and after subgoal invocation are as follows.  $\Delta \vdash M \in a M_1 \dots M_n$  is *input consistent* (*output consistent*) wrt. mode  $m_a$  if  $\Delta \vdash M_i : A_{M_i}$  **ground** for  $i \in m_a^+$  (respectively  $\in m_a^-$ ) and  $\Delta \vdash M : a M_1 \dots M_n$  **ground** for  $0 \in m_a^+$  (respectively  $\in m_a^-$ ).

**Approximations.** In order to check a given signature against a set of mode specifications for its predicates we perform an abstract interpretation using abstract substitutions which note—in addition to the substitution domain—whether an existential variable  $\exists x:A$  is known to have been instantiated to a ground term (**gnd**  $x:A$ ) or whether its status is still indeterminate (**uk**  $x:A$ ).

Abstract substitution  $\eta ::= \cdot \mid \eta, \forall x:A \mid \eta, \mathbf{uk} x:A \mid \eta, \mathbf{gnd} x:A$

The domain of  $\eta$ , written  $\eta : \Delta$ , can be read off immediately. The approximation judgment  $\Delta' \vdash (\eta \sim \theta) : \Delta$  between an abstract substitution  $\eta : \Delta$  and a concrete substitution  $\theta$  with  $\Delta' \vdash \theta : \Delta$  is defined below. This judgment is intentionally nondeterministic (*i.e.* a  $\theta$  may be approximated by different  $\eta, \eta'$ ) and relies heavily on canonical forms.

$$\frac{}{\Delta' \vdash (\cdot \sim \cdot) : \cdot} \qquad \frac{\Delta' \vdash (\eta \sim \theta) : \Delta \quad \Delta' \vdash M : \theta A \text{ **ground**}}{\Delta' \vdash (\eta, \mathbf{gnd} x:A \sim \theta, \exists M/x) : \Delta, \exists x:A}$$

$$\frac{\Delta' \vdash (\eta \sim \theta) : \Delta}{\Delta' \vdash (\eta, \forall x:A \sim \theta, \forall x/x) : \Delta, \forall x:A} \qquad \frac{\Delta' \vdash (\eta \sim \theta) : \Delta}{\Delta' \vdash (\eta, \mathbf{uk} x:A \sim \theta, \exists M/x) : \Delta, \exists x:A}$$

An *initial approximation*  $\eta(\Delta)$  with  $\Delta \vdash (\eta(\Delta) \sim id_\Delta) : \Delta$  is given by  $\eta(\cdot) \stackrel{def}{=} \cdot$ ,  $\eta(\Delta, \forall x:A) \stackrel{def}{=} \eta(\Delta), \forall x:A$ , and  $\eta(\Delta, \exists x:A) \stackrel{def}{=} \eta(\Delta), \mathbf{uk} x:A$ .

**Mode checking.** To ensure that all Elf execution sequences (starting from an input consistent goal) obey modes, it is sufficient to show that the abstract execution of all clauses in the signature  $\Sigma$  and the goal context  $\Gamma$  respects modes, *i.e.* all subgoal invocations are input consistent and—upon return—output consistent.

Due to space constraints we omit the formal system for mode checking and just exhibit an example of the abstract interpretation. Consider the body of the clause **cp****lam** with respect to the mode **cp** **+E** **-F** under an empty abstract substitution.

$$\cdot \vdash \{\mathbf{E}\}\{\mathbf{F}\}(\{\mathbf{x:exp}\} \text{cp } \mathbf{x} \ \mathbf{x} \rightarrow \text{cp } (\mathbf{E} \ \mathbf{x}) \ (\mathbf{F} \ \mathbf{x})) \rightarrow \text{cp } (\mathbf{lam} \ \mathbf{E}) \ (\mathbf{lam} \ \mathbf{F})$$

We can view **cp****lam** as a clause whose head **cp** (**lam** **E**) (**lam** **F**) characterizes its call parameters. Assuming that it is invoked in a mode-consistent manner, we know that the input **lam** **E** and therefore the term **E** is ground, although at

this point we have no information about  $F$ . Once  $\text{cplam}$  returns, however, we need to establish that  $F$  has become ground.

$$\begin{array}{l} \text{gnd } E:\text{exp} \rightarrow \text{exp}, \text{uk } F:\text{exp} \rightarrow \text{exp} \\ \vdash (\{x:\text{exp}\} \text{cp } x \ x \rightarrow \text{cp } (E \ x) (F \ x)) \rightarrow \text{cp } (\text{lam } E) (\text{lam } F) \end{array}$$

The term  $\{x:\text{exp}\} \text{cp } x \ x \rightarrow \text{cp } (E \ x) (F \ x)$  represents the only subgoal of our clause. The goal head  $\text{cp } (E \ x) (F \ x)$  is executed once the local parameter  $x$  and the local program clause  $\text{cp } x \ x$  have been introduced. Obviously, we need to check the local clause  $\text{cp } x \ x$  for well-modedness before we can assume it: since  $x$  is a parameter and thus ground, the output argument of  $\text{cp } x \ x$  is always ground. Now consider the goal

$$\text{gnd } E:\text{exp} \rightarrow \text{exp}, \text{uk } F:\text{exp} \rightarrow \text{exp}, \forall x:\text{exp}, \forall D:\text{cp } x \ x \vdash \text{cp } (E \ x) (F \ x).$$

Since this may resolve with any clause for  $\text{cp}$  we now have to show that the call's input argument  $E \ x$  is ground (which is the case), while we may safely assume that its output  $F \ x$  has become ground once it returns.

Fortunately, we can obtain more information about  $F$  from this fact. The term  $F \ x$  is a *gvar* (*generalized variable*) under a mixed prefix  $\Delta$  (see [14]) since  $F$  is an existential variable that is applied to distinct universal variables declared to the right of  $F$ . Unification of a *gvar* with a ground term always returns a most general unifier (if it succeeds), instantiating the *gvar* to a ground term. Without the restriction to generalized variables this property may be violated — consider for example  $F \ G \doteq \text{lam } (\lambda x.x)$  which has a solution where  $F = \lambda y. \text{lam } (\lambda x.x)$  and  $G$  is arbitrary and not necessarily ground. On the other hand, the application of two ground terms (as in the first subgoal of the clause  $\text{evapp}$ ) can be recognized as ground. This is an example where a program outside the  $L_\lambda$  fragment [12] is verified as mode correct, which means that  $\beta_0$  unification and no constraints are generated for well-moded queries.

Returning to the example, since the *gvar*  $F \ x$  is known to be ground we may now also conclude that  $F$  is ground, thus finally demonstrating groundedness of the output argument  $\text{lam } F$  in the original program head  $\text{cp } (\text{lam } E) (\text{lam } F)$  of  $\text{cplam}$ .

**Mode-consistency for Elf.** A computation sequence  $\mathcal{S}_1 \rightarrow \dots \mathcal{S}_n \rightarrow \dots$  is *mode-consistent* whenever for every state  $\mathcal{S}_i$  of the form (*i.e.* for a subgoal call)

$$\left( \Delta' \Vdash_{\theta; \Delta; \kappa} M \in a \ M_1 \dots M_n \right) \&K$$

we have  $\Delta' \vdash \theta M \in \theta(a \ M_1 \dots M_n)$  is input consistent, and furthermore for every subsequence (*i.e.* for a subgoal return)

$$\mathcal{S}_i \rightarrow \left( \Delta' \Vdash_{\theta; \Delta; \kappa} F \right) \&K \rightarrow \dots \rightarrow K \ \Delta'' \ \theta' \ \kappa'$$

—with  $F$  a residuation formula for  $\mathcal{S}_i$ — we have  $\Delta'' \vdash \theta' M \in \theta'(a \ M_1 \dots M_n)$  is output consistent.

**Theorem.** If  $\Sigma$  and  $\Delta$  are mode-checked and  $\Delta \vdash M \in a M_1 \dots M_n$  is input consistent then all possible Elf execution sequences from

$$\left( \Delta \parallel_{-id_{\Delta}; \Delta; \top} M \in a M_1 \dots M_n \right) \&\mathbf{initial}_{\Delta'; \theta; \kappa}$$

are mode-consistent.

**Proof sketch:** For every state  $\mathcal{S}_{\Delta'; \theta; \kappa}$  (where  $\Delta' \vdash \theta : \Delta$ ) in an Elf execution sequence there is a corresponding abstract state in the mode checking system which is characterized by an abstract substitution  $\eta$  with  $\Delta' \vdash (\eta \sim \theta) : \Delta$ . We then show via induction over the computation sequences that the groundedness properties established during mode checking will also hold at the Elf level.

**Related work.** Many properties of logic programs can be derived by abstract interpretation [3], including the inference of mode declarations [10]. Contrary to Debray and Mellish [4] we view mode declarations as part of a logic program’s specification rather than as a property to be inferred. Our system distinguishes between ground and possibly non-ground terms, which makes mode information in a higher-order setting manageable while still being very useful. As a result our mode analysis requires neither fixed-point constructions nor a sharing analysis — a single pass over a program suffices, possibly with backtracking if several modes are permitted for a given predicate. Mode systems for logic programs often rely on type information and also consider more precise modes, such as partially instantiated terms [26]. In our case we directly exploit Elf’s type system.

## 4 Termination Analysis

For termination we need to demonstrate that arguments to (possibly mutually) recursive subgoals decrease in some well-founded ordering with respect to the original program call. For example, in the `ack` function we can show a decrease in `ack3` if we consider both input arguments lexicographically:  $\langle \mathbf{s} X, Y \rangle < \langle \mathbf{s} X, \mathbf{s} Y \rangle$  and  $\langle X, Q \rangle < \langle \mathbf{s} X, \mathbf{s} Y \rangle$ .

In order to obtain termination we assume that the termination conditions are defined only on input arguments to a well-moded predicate [2]. This avoids a much more complicated analysis—consider, for example, a program containing the clauses `p (s X) <- p X` and `p z` in this order. Elf (and Prolog) search for the goal `?- p Y` will not terminate. In this case our mode analysis would reject the goal since `Y` is not closed. Due to this and other current restrictions, the system for termination is less accurate than the mode system, yet still exceedingly useful, especially for establishing meta-theoretical properties of the object languages we encode in Elf (see Section 5 for further discussion).

In a first-order setting it is straightforward to determine whether a term is a subterm of another. Consider now higher-order terms in Elf, *e.g.*, in the clause `cp lam` for the copying function `cp` on  $\lambda$ -expressions. We want to show that in a suitable sense  $(\mathbf{E} \mathbf{x})$  is a subterm of  $(\mathbf{lam} \mathbf{E})$  when  $\mathbf{x}$  is a newly introduced parameter. Another important aspect of higher-order subterms can be demonstrated, *e.g.*, from a formalization of predicate logic as it is used

in the cut-elimination proof for the sequent calculus [22]. We have type families  $\mathbf{i}$  for individuals and  $\mathbf{o}$  for formulas, and—among others—a constructor  $\mathbf{forall} : (\mathbf{i} \rightarrow \mathbf{o}) \rightarrow \mathbf{o}$ . The proof requires  $\mathbf{A T}$  (which represents  $[t/x]A$ ) to be strictly smaller than  $\mathbf{forall A}$  (which represents  $\forall x.A$ ). In the informal proof we count the number of quantifiers and connectives, noting that a term  $t$  in first-order logic cannot contain any logical symbols. Thus we may consider  $\mathbf{A T}$  a *subterm* of  $\mathbf{forall A}$  as long as there is no way to construct an object of type  $\mathbf{i}$  from objects of type  $\mathbf{o}$ .

**Mutually recursive type families.** We define a type family  $a$  to be *subordinate* to a type family  $a'$  ( $a \triangleleft^* a'$ ) whenever a term  $M : A$  with  $\mathbf{hd}(A) = a$  may be used in constructing a term  $N : B$  with  $\mathbf{hd}(B) = a'$  (see [29]). If additionally  $a' \triangleleft^* a$  we say that  $a, a'$  are *mutually recursive*. We write  $a \triangleleft^* a'$  if  $a$  is subordinate to  $a'$ , but not mutually recursive with  $a$ .

Subordination of type families is the transitive closure of the immediate subordination relation ( $a \triangleleft a'$ ) which can be directly read off the signature  $\Sigma$ . *E.g.* our sample signature contains  $\mathbf{nat} \triangleleft \mathbf{nat}$  (from  $\mathbf{s}$ ),  $\mathbf{nat} \triangleleft \mathbf{ack}$  (from  $\mathbf{ack}$ ),  $\mathbf{ack} \triangleleft \mathbf{ack}$  (from  $\mathbf{ack3}$ ),  $\mathbf{exp} \triangleleft \mathbf{cp}$  (from  $\mathbf{cp}$ ),  $\mathbf{cp} \triangleleft \mathbf{cp}$  (from  $\mathbf{cpapp}$ ,  $\mathbf{cplam}$ ),  $\mathbf{exp} \triangleleft \mathbf{eval}$  (from  $\mathbf{eval}$ ), and  $\mathbf{eval} \triangleleft \mathbf{eval}$  (from  $\mathbf{evapp}$ ,  $\mathbf{evlam}$ ).

**Subterms.** We now define a judgment for the subterm relationship between higher-order terms. In the system below the use of abstract substitutions  $\eta$  appears superfluous. However, mixed-prefix contexts  $\Delta$  alone cannot supply the groundedness information obtained from mode-checking and necessary for defining a well-founded term measure in the termination proof. In the rules we use long  $\beta\eta$  normal forms throughout and we assume re-normalization after every substitution. Variables  $C, C'$  denote atomic types, *i.e.*, types of the form  $a M_1 \dots M_n$  and we have  $h : A_h \in \Sigma$  or  $\forall h : A_h \in \eta$ .

$$\frac{\eta \vdash M : A \prec N : B}{\eta \vdash M : A \preceq N : B} \quad \frac{\eta : \Delta \quad \overline{\Delta} \vdash_{\Sigma} A \equiv B : \text{type} \quad \overline{\Delta} \vdash_{\Sigma} M \equiv N : A}{\eta \vdash M : A \preceq N : B}$$

$$\frac{\eta, \forall x : A \vdash M : B \prec N : B'}{\eta \vdash \lambda x : A. M : \Pi x : A. B \prec N : B'} \quad \frac{\eta \vdash M : C \preceq N_i : A_{N_i} \text{ for some } 1 \leq i \leq m}{\eta \vdash M : C \prec h N_1 \dots N_m : C'}$$

$$\frac{\eta : \Delta \quad \forall y : A \in (\Delta, \forall x : A) \quad \eta, \forall x : A \vdash M : C \prec [y/x]N : [y/x]B \text{ where } \mathbf{hd}(C), \mathbf{hd}(A) \text{ mut. rec.}^4}{\eta \vdash M : C \prec \lambda x : A. N : \Pi x : A. B}$$

$$\frac{\eta : \Delta \quad \overline{\Delta}, x : A \vdash_{\Sigma} M' : A \quad \eta, \forall x : A \vdash M : C \prec [M'/x]N : [M'/x]B \text{ where } \mathbf{hd}(A) \triangleleft^* \mathbf{hd}(C)^4}{\eta \vdash M : C \prec \lambda x : A. N : \Pi x : A. B}$$

The side conditions in the last two rules enforce that  $\lambda$ -bound variables must be instantiated with a parameter  $y$  unless a term of type  $A$  can never contain a subterm of type  $C$ , in which case it may be instantiated with an *arbitrary* term  $M'$ . It should be noted that in the implementation the choice of  $M'$  or  $y$  is delayed and determined later via unification.

<sup>4</sup> We have analogous rules for  $\eta \vdash M : C \preceq \lambda x : A. N : \Pi x : A. B$ .

**Example.** With the subterm judgment in place we can now revisit our examples. The following valid judgments arise in the checking of termination.

$\text{gnd } E:\text{exp} \rightarrow \text{exp}, \forall x:\text{exp} \vdash (E x) : \text{exp} \prec (\text{lam } (\lambda y. E y)) : \text{exp}$

The derivation uses the crucial fact that  $x$  is a parameter (*i.e.*  $\forall$ -quantified).

Now consider the predicate logic example where we want to show

$\text{gnd } A:\text{i} \rightarrow \text{o}, \text{uk } T:\text{i} \vdash (A T) : \text{o} \prec (\text{forall } (\lambda x. A x)) : \text{o}.$

Here we do not even know whether the logic variable  $T$  has been instantiated to a ground term. However, since  $\text{i}$  is *not* mutually recursive with  $\text{o}$  we apply the second to last rule above to obtain the desired result.

**Termination Checking.** Given a mode-checked signature  $\Sigma$  we can perform termination checking by showing that calls to auxiliary predicates terminate and that input arguments decrease wrt.  $\prec$  in recursive subgoals. We merely need to declare which input arguments we consider, and in which lexicographic order they diminish, *e.g.*

```
%mode -ack +X +Y -Z
%lex X Y
```

The additional `%lex` pragma simply gives the *lexicographic* termination order for the preceding mode declaration: the  $X$  argument decreases, or —if  $X$  remains unchanged— the  $Y$ -argument will decrease. In addition, we also use the `%lex` pragma to relate arguments between mutually recursive predicates.

**Theorem.** Given a mode-checked and termination-checked signature  $\Sigma$  and an input-consistent, well-moded Elf goal  $\mathcal{S}_0$  that does not introduce any new type dependencies not found in  $\Sigma$ . Then all possible Elf execution sequences  $\mathcal{S}_0 \rightarrow \mathcal{S}_1 \rightarrow \dots \rightarrow \mathcal{S}_n \rightarrow \dots$  are finite. Since there are only finitely many choices at each step, Elf search will always terminate.

**Proof sketch:** First we define a term measure which is finite for ground terms and consistent with the subterm relationship. Since input (*i.e.*,  $+$ ) arguments to subgoals are ground, they have a finite measure.

We can linearize the LF type hierarchy by combining mutually recursive type families into a single node. Based on this linear vector, we define a multiset measure whose elements contain the lexicographic argument measures of all these nodes. We add a bookkeeping measure to ensure that the decomposition of non-atomic goal formulas also decreases their measure. Then we assign a well-founded measure to an Elf state  $\mathcal{S}$  which counts the current goal formula as well as all goals postponed in the success continuation and show that this measure decreases in each transition step.

**Related work.** Similarly to our mode analysis approach, we have extended a rather naïve first-order termination analysis based on a subterm property [16] to a higher-order setting. Although it would have been straightforward to implement, we do not automatically infer an actual lexicographic order — we would rather consider the termination orderings part of the specification of a logic program. For proofs of meta-theorems formalized in Elf that employ structural induction, these orderings correspond directly to the nesting of the inductive argument.

## 5 Pragmatics

The mode and termination analyses described in this paper were implemented for the current Elf interpreter [20] and have proven to be valuable tools in the development of Elf programs.

**Multiple modes.** Sometimes a predicate may be executed in multiple directions and we would like to assign it multiple modes, rather than copy its definition. Our mode-checker allows multiple mode declarations for the same type family by considering different modes of the same predicate as mutually recursive but distinct type families. The mode system remains decidable, although the mode assigned to an occurrence of a type family may not necessarily be unique. We choose the one which gives us the most information if it exists; otherwise we issue a warning and try each mode assignment in turn.

**Applications.** In case of a mode or termination error, the checker pinpoints the offending clause and subterm and reports all information needed by the user to remedy the problem. The checking of 6 previously defined Elf theories with some 50 theorems uncovered one mode error. In another instance, however, where an Elf novice had 1500 lines of code under active development, the checker revealed 35 locations with mode problems, of which 20 could be attributed to mistyped variable names. The other most common mistake is incorrect subgoal ordering.

It is not surprising that mode errors outnumber termination errors. Termination checks are only performed on well-moded predicates, whereas mode errors such as wrong subgoal ordering or variable name misspellings can lead to uninstantiated input arguments and nontermination.

The combination of mode and termination checking is particularly useful when we want to establish that an Elf program constitutes a decision procedure. This allows us to make a meta-mathematical statement about an object language formalized in Elf simply by exhibiting a checked program in the same framework. Some examples of this approach are:

- a formalization of linear logic [21], where the linearity of derivations is a decidable property,
- an implementation of the sequent calculus [22] with a terminating cut-elimination procedure,
- a formulation of refinement types [19] for which the subtype property is decidable, and
- a representation of Mini-ML [11] for which type inference is guaranteed to terminate.

**Limitations.** To date we have encountered a sole instance where the mode checker rejected an intuitively correct formalization, which, we believe, may be rewritten. We thus do not consider this a major limitation of the mode systems.

Our lexicographically extended higher-order subterm ordering works well for structural inductions (on which most meta-proofs in Elf are based on), but they fail in other cases such as course-of-value induction or recursion over sublists [5]. In some of these situations one can make the termination proofs apparent

through the introduction of additional “measure” arguments (such as the length of a list). We also encountered an instance where unfolding of mutually recursive predicates was necessary to automatically show termination.

A more fundamental limitation is that we do not relate measures of input arguments to output arguments of predicates, which is sometimes necessary if intermediate results are used in recursive calls. The restriction to lexicographic orderings is surprisingly flexible, but there are instances where others (such as multi-set orderings) would be helpful. It seems feasible to also allow multi-set orderings in the termination specification of predicates in future versions.

## 6 Conclusion

We have described a practical system for the specification of mode and termination properties of programs written in the higher-order language Elf. These properties are decidable within our mode system and the implementation checks them efficiently and provides useful feedback in case a property is violated.

While the basic notions of modes and termination have been known for some time [30, 1] they have not yet been applied to a higher-order setting that includes dependent types, higher-order terms and proof objects. As a pragmatic decision, especially since the underlying LF type theory does not prescribe one particular operational interpretation, we implemented modes separately from types, whereas Reddy [25] proposes to combine mode and type specifications. At present, the simplicity of our approach outweighs the benefits of a more flexible system such as Reddy’s.

We do not know of any framework logics that perform an analysis similar to ours. In the ALF framework [9] only total functions over disjoint patterns are definable and —since ALF does not employ higher-order abstract syntax— a first-order subterm ordering is sufficient for showing termination of recursive calls.

Termination proofs in a higher-order setting have been investigated, among others, in [28] and [8]. Although our system employs similar ideas, we need to additionally make use of type subordination to obtain the desired termination ordering.

We expect mode and termination properties of higher-order logic programs to play an important rôle in the compilation of such programs. We need to investigate how our ideas can be applied to  $\lambda$ -Prolog [15] which presents two additional complications: extra-logical primitives (such as cut, or primitives for input and output) and higher-order subgoals permitting predicates as arguments to other predicates. This means that we may not be able to statically determine the call graph of a program. However, this is not as important since predicates and types are syntactically separated and, in our approach, mutual recursion appears to be more important for types.

Perhaps the most important extension is to show the totality of predicates: not only will every execution sequence terminate, but every execution sequence starting from a well-moded goal will *succeed*. This allows us to verify that certain

higher-level judgments implement proofs [23], formally establishing many important meta-theoretic properties of the object languages under investigation.

Finally, work on negation has often relied on mode information [27]. We plan to take advantage of mode and termination information when considering negation in the context of higher-order logic programming.

**Acknowledgments.** We would like to thank the anonymous referees for their helpful comments and Brian Milnes for supplying us with a user’s perspective on the checker implementation.

## References

1. K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.
2. K. R. Apt and Alessandro Pellegrini. On the occur-check-free PROLOG programs. *ACM Transactions on Programming Languages and Systems*, 16(3):687–726, May 1994.
3. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.
4. S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, 1988.
5. John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.
6. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
7. Dean Jacobs. A pragmatic view of types for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 7, pages 217–227. MIT Press, 1992.
8. Stefan Kahrs. Towards a domain theory for termination proofs. In Jieh Hsiang, editor, *Sixth International Conference on Rewriting Techniques and Applications, RTA-95*, pages 241–255, Kaiserslautern, Germany, April 1995. Springer-Verlag.
9. Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf’s Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.
10. C. S. Mellish. The automatic generation of mode declarations for Prolog programs. DAI Research Report 163, Department of Artificial Intelligence, University of Edinburgh, 1981.
11. Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.
12. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.
13. Dale Miller. Abstract syntax and logic programming. In *Proceedings of the First and Second Russian Conferences on Logic Programming*, pages 322–337, Irkutsk and St. Petersburg, Russia, 1992. Springer-Verlag LNAI 592.

14. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.
15. Gopalan Nadathur and Dale Miller. An overview of  $\lambda$ Prolog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.
16. L. Naish. Automatic generation of control for logic programs. Technical Report 83/6, Department of Computer Science, The University of Melbourne, 1983.
17. 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.
18. Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.
19. Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.
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. Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.
22. Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.
23. Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.
24. Lutz Plümer. *Termination Proofs for Logic Programs*. Springer-Verlag LNAI 446, 1991.
25. Uday S. Reddy. A typed foundation for directional logic programming. In E. Lamma and P. Mello, editors, *Proceedings of the Third International Workshop on Extensions of Logic Programming*, pages 282–318, Bologna, Italy, February 1992. Springer-Verlag LNAI 660.
26. Z. Somogyi. A system of precise modes for logic programs. In J. L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming, Volume 2*, pages 769–787, Cambridge, Massachusetts, 1987. MIT Press.
27. Robert F. Stärk. The declarative semantics of the Prolog selection rule. In S. Abramsky, editor, *Proceedings of the Ninth Annual Symposium on Logic in Computer Science*, pages 252–261, Paris, France, July 1994. IEEE Computer Society Press.
28. J. van de Pol and H. Schwichtenberg. Strict functionals for termination proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, pages 350–364, Edinburgh, United Kingdom, April 1995. Springer-Verlag LNCS.
29. Roberto Virga. Higher-order superposition for dependent types. Technical Report CMU-CS-95-150, Carnegie Mellon University, 1995.
30. D. H. D. Warren. Implementing Prolog—compiling predicate logic programs, Volume 1. DAI Research Report 39, University of Edinburgh, 1977.