Termination and Reduction Checking for Higher-Order Logic Programs

Brigitte Pientka

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
bp@cs.cmu.edu

Abstract. In this paper, we present a syntax-directed termination and reduction checker for higher-order logic programs. The reduction checker verifies parametric higher-order subterm orderings describing relations between input and output of well-moded predicates. These reduction constraints are exploited during termination checking to infer that a specified termination order holds. To reason about parametric higher-order subterm orderings, we introduce a deductive system as a logical foundation for proving termination. This allows the study of proof-theoretical properties, such as consistency, local soundness and completeness and decidability. We concentrate here on proving consistency of the presented inference system. The termination and reduction checker are implemented as part of the Twelf system and enable us to verify proofs by complete induction.

1 Introduction

One of the central problems in verifying specifications and checking proofs about them is the need to prove termination. Several automated methods to prove termination have been developed for first-order functional and logic programs in the past years (for example [15, 1]). One typical approach is to transform the program into a term rewriting system (TRS) such that the termination property is preserved. A set of inequalities is generated and the TRS is terminating if there exists no infinite chain of inequalities. This is usually done by synthesizing a suitable measure for terms. To show termination in higher-order simply-typed term rewriting systems (HTRS) mainly two methods have been developed (for a survey see [13]): the first approach relies on strict functionals by van de Pol [12], and the second one is a generalization of recursive path orderings to the higher order case by Jouannaud and Rubio [4].

In this paper, we present a syntax-directed method for proving termination of higher-order logic programs. First, the reduction checker verifies properties relating input and output of higher-order predicates. Using a deductive system to reason about reduction constraints, the termination checker then proves that the inputs of the recursive call are smaller than the inputs of the original call with respect to higher-order subterm orderings. Our method has been developed for the higher-order logic programming language Twelf [8] which is based on the logical framework LF [3]. Although Twelf encompasses pure Prolog, it has been designed as a meta-language for the specification of deductive systems and proofs about them. In addition to Prolog it allows hypothetical and parametric subgoals. As structural properties play an important role in this setting, higher-order subterm orderings have been proven to be very powerful (see Section 5).

The principal contributions of this paper are two-fold: 1) We present a logical foundation for proving termination which is of interest in proving termination of first-order and higher-order programs. The logical perspective on reasoning about orders allows the study of proof-theoretical properties, such as consistency, local soundness and completeness and decidability. In this paper, we concentrate on proving consistency of the presented reasoning system by showing admissibility of cut. This implies soundness and completeness of the reasoning system. 2) We describe a practical syntax-directed system for proving termination of higher-order logic programs. Unlike most other approaches, we are interested in checking a given order for a program and not in synthesizing an order for a program. The advantage is that checking whether a given order holds is more efficient than synthesizing orders. In the case of failure, we can provide detailed error messages. These help the user to revise the program or to refine the specified order. The termination checker is implemented as part of the Twelf system and has been used successfully on examples from compiler verification (soundness and completeness proofs for stack semantics and continuation-based semantics), cut-elimination and normalization proofs for intuitionistic and classical logic, soundness and completeness proofs for the Kolmogorov translation of classical into intuitionistic logic (and vice versa).

The paper is organized as follows: In Section 2 we give a representative Twelf program taken from the domain of compiler verification. Using this example we illustrate the basic idea of the termination checker. We review the background (see Section 3) In Section 4 we outline the deductive system for reasoning about orders and prove consistency of the system. Finally, in Section 5 we discuss related work, summarize the results and outline future work.

2 Motivating Example

Our work on termination is motivated by induction theorem proving in the logical framework and its current limitations to handle proofs by complete induction. In this section, we consider a typical example from compiler verification [2] to illustrate our approach.

Compilation is the automatic transformation of a program written in a source language to a program in a target language. Typically, there are several stages of compilation. Starting with a high-level language, the computational description is refined in each step into low-level machine language. To prove correctness of a compiler, we need to show the correspondence between the source and target language. In this example, we consider Mini-ML as the source language, and the language of the abstract machine as the target language. We only consider a small subset of a programming language in this paper.

Mini-ML Syntax		Abstract Machine Syntax	
expressions	e ::= $e_1 e_2 \mid \mathsf{lam}\ x.e$	instructions	I ::= ret $v \mid$ ev $e \mid \mathrm{app}_1\ v\ e \mid \mathrm{app}_2\ v_1 v_2$
values	v ::= $x \mid \mathsf{Lam}\ x.e$	stack	S ::= nil $\mid S; \lambda v.I$

The Mini-ML language consists of lambda-abstraction and application. To evaluate an application $e_1\ e_2$, we need to evaluate e_1 to some value $\mathsf{Lam}\ x.e'$, e_2 to some value v_2 and $[v_2/x]e'$ to the final value of the application. Note, the order of evaluation of these premises is left unspecified. The abstract machine has a more refined computation model which is reflected in the instruction set. We not only have instructions operating on expressions and values, but also intermediate mixed instructions such as $\mathrm{app}_1\ v_1\ e_2$ and $\mathrm{app}_2\ v_1\ v_2$. Computation in an abstract machine can be represented as a sequence of states. Each state is characterized by a stack S representing the continuation and an instruction I and written as $S\#I$. In contrast to the big-step semantics for Mini-ML, the small-step transition semantics precisely specifies that an application is evaluated from left to right.

Big step Mini-ML semantics:

$$\frac{}{\mathsf{lam}\ x.e \hookrightarrow \mathsf{Lam}\ x.e}\ \mathit{ev\_lam} \qquad \frac{e_1 \hookrightarrow \mathsf{Lam}\ x.e_1' \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e_1' \hookrightarrow v}{e_1 e_2 \hookrightarrow v}\ \mathit{ev\_app}$$

Small-step transition semantics (single step):

$$\begin{array}{ll}
\mathit{t_lam}: & S\#(\mathrm{ev}\ \mathsf{lam}\ x.e) \longmapsto S\#(\mathrm{ret}\ \mathsf{Lam}\ x.e) \\
\mathit{t_app}: & S\#(\mathrm{ev}\ e_1 e_2) \longmapsto (S; \lambda v.\mathrm{app}_1\ v\ e_2)\#(\mathrm{ev}\ e_1) \\
\mathit{t_app1}: & S\#(\mathrm{app}_1\ v_1 e_2) \longmapsto (S; \lambda v_2.\mathrm{app}_2\ v_1 v_2)\#(\mathrm{ev}\ e_2) \\
\mathit{t_app2}: & S\#(\mathrm{app}_2\ (\mathsf{Lam}\ x.e')v_2) \longmapsto S\#(\mathrm{ev}\ [v_2/x]e') \\
\mathit{t_ret}: & (S; \lambda v.I)\#(\mathrm{ret}\ v_1) \longmapsto S\#[v_1/v]I
\end{array}$$

A computation sequence

$$S\#(\mathrm{ev}\ e_1 e_2) \overset{t\_app}{\longmapsto} (S; \lambda v.\mathrm{app}_1\ v\ e_2)\#(\mathrm{ev}\ e_1) \overset{*}{\longmapsto} \mathrm{nil}\ \#(\mathrm{ret}\ w)$$

is represented in Twelf as `(t_app @ D1)` where `t_app` represents the first step of computation $S\#(\mathrm{ev}\ e_1 e_2) \overset{t\_app}{\longmapsto} (S; \lambda v.\mathrm{app}_1\ v\ e_2)\#(\mathrm{ev}\ e_1)$ while `D1` describes the tail of the computation $(S; \lambda v.\mathrm{app}_1\ v\ e_2)\#(\mathrm{ev}\ e_1) \overset{*}{\longmapsto} \mathrm{nil}\ \#(\mathrm{ret}\ w)$. We will sometimes mix multi-step transitions $\overset{*}{\longmapsto}$ with single step transitions $\longmapsto$ with the obvious meaning.

An evaluation tree in the big step semantics

$$\frac{\begin{array}{c}\mathcal{P}_1 \\ e_1 \hookrightarrow \mathsf{Lam}\ x.e_1'\end{array} \quad \begin{array}{c}\mathcal{P}_2 \\ e_2 \hookrightarrow v_2\end{array} \quad \begin{array}{c}\mathcal{P}_3 \\ [v_2/x]e_1' \hookrightarrow v\end{array}}{e_1 e_2 \hookrightarrow v}\ \mathit{ev\_app}$$

is implemented as `(ev_app P1 P2 P3)`. The leaves of the evaluation tree are formed by applications of the *ev_lam* axiom which is implemented as a constant `ev_lam` in Twelf.

To show that the compiler works correctly, we need to show soundness and completeness of the two semantics. We will concentrate on the first property. To prove soundness we show the following: if we start in an arbitrary state $S\#(\text{ev } e)$ with a computation $S\#(\text{ev } e) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)$ then there exists an intermediate state $S\#(\text{ret } v)$ such that $e \hookrightarrow v$ in the Mini-ML semantics and $S\#(\text{ret } v) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)$.

Theorem 1 (Soundness).
For all computation sequences $\mathcal{D} : S\#(ev\ e) \overset{*}{\longmapsto} nil\ \#(ret\ w)$ there exists an evaluation tree $\mathcal{P} : e \hookrightarrow v$ and a tail computation $\mathcal{D}' : S\#(ret\ v) \overset{*}{\longmapsto} nil\ \#(ret\ w)$ such that $\mathcal{D}'$ is smaller than $\mathcal{D}$.

The proof follows by complete induction on $\mathcal{D}$. We consider each computation sequence $\mathcal{D}$ in the small step semantics and translate it into an evaluation tree $\mathcal{P}$ in the Mini-ML semantics and some tail computation $\mathcal{D}'$ which is a sub-derivation of the original computation $\mathcal{D}$. This translation can be described by a meta-predicate `sound` which takes a computation sequence as input and returns an evaluation tree and a tail computation sequence.

As a computation sequence can either start with `t_lam` or `t_app` transition, we need to consider two cases. If the computation sequence starts with a `t_lam` transition `(t_lam @ D)` then there exists an evaluation of $\mathsf{lam}\ x.e$ to $\mathsf{Lam}\ x.e$ by the `ev_lam` rule and a tail computation `D`. The interesting case is when the computation sequence starts with an `t_app` transition `(t_app @ D1)`.

$$S\#(\text{ev } e_1\ e_2) \overset{t\_app}{\longmapsto} \underbrace{(S; \lambda v.\text{app}_1\ v\ e_2)\#(\text{ev } e_1) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)}_{\texttt{D1}}$$

We recursively apply the translation to `D1` and obtain `P1` which represents an evaluation starting in $e_1 \hookrightarrow v_1$ and $(S; \lambda v.\text{app}_1\ v\ e_2)\#(\text{ret } v_1) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)$ as the tail computation sequence $\mathcal{D}'$. By inversion using the *t_ret* and *t_app1* transition rules, we unfold $\mathcal{D}'$ and obtain the following tail computation sequence

$$(S; \lambda v.\text{app}_1\ v\ e_2)\#(\text{ret } v_1) \overset{t\_ret}{\longmapsto} S\#(\text{app}_1\ v_1 e_2) \overset{t\_app1}{\longmapsto} \underbrace{(S; \lambda v.\text{app}_2\ v_1 v)\#(\text{ev } e_2) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)}_{\texttt{D2}}$$

which is represented as `(t_ret @ t_app1 @ D2)`. By applying the translation again to `D2`, we obtain an evaluation tree for $e_2 \hookrightarrow v_2$ described by `P2` and some computation sequence $\mathcal{D}'' : (S; \lambda v.\text{app}_2\ v_1 v)\#(\text{ret } v_2) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)$. By inversion using rules *t_ret* and *t_app2*, we know that the value v_1 represents some function $\mathsf{Lam}\ x.e'$ and $\mathcal{D}''$ can be unfolded to obtain the tail computation

$$(S; \lambda v.\text{app}_2\ (\mathsf{Lam}\ x.e')v)\#(\text{ret } v_2) \overset{t\_ret}{\longmapsto} S\#(\text{app}_2\ (\mathsf{Lam}\ x.e')v_2) \overset{t\_app2}{\longmapsto} \underbrace{S\#(\text{ev } [v_2/x]e') \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)}_{\texttt{D3}}$$

which is represented as `(t_ret @ t_app2 @ D3)`. Now we apply the translation for a final time to `D3` and obtain an evaluation tree `P3` starting in $[v_2/x]e' \hookrightarrow v$

and some tail computation $S\#(\text{ret } v) \stackrel{*}{\longmapsto} \text{nil } \#(\text{ret } w)$ which we refer to as `D4`. The final results of translating a computation sequence `(t_app @ D1)` are the following: The first result is an evaluation tree for $e_1 e_2 \hookrightarrow v$ which can be constructed by using the *ev_app* rule and the premises $e_1 \hookrightarrow (\mathsf{Lam}\ x.e')$, $e_2 \hookrightarrow v_2$ and $[v_2/x]e' \hookrightarrow v$. This step is represented in Twelf by `(ev_app P1 P2 P3)`. As a second result, we return the tail computation sequence `D4`.

The following Twelf program implements the described translation. Throughout this example, we reverse the function arrows writing A_2 `<-` A_1, instead of A_1 `->` A_2 following logic programming notation. Since `->` is right associative, `<-` is left associative. A more detailed discussion of this example is given in [7].

```
sound : S # (ev E) =>* nil # (ret W) ->
        eval E V -> S # (ret V) =>* nil # (ret W) -> type.
%mode sound +D -P -D'.
s_lam : sound (t_lam @ D ) ev_lam D.
s_app : sound (t_app @ D1) (ev_app P3 P2 P1) D4
        <- sound D1 P1 (t_ret @ t_app1 @ D2)
        <- sound D2 P2 (t_ret @ t_app2 @ D3)
        <- sound D3 P3 D4.
```

First the type of the meta-predicate `sound` is defined. It has three arguments: the computation $S\#(\text{ev } E) \stackrel{*}{\longmapsto} \text{nil } \#(\text{ret } W)$ which is described as `S # (ev E) =>* nil # (ret W)`, the evaluation $e \hookrightarrow v$ which is represented as `eval E V` and the tail computation sequence $S\#(\text{ret } E) \stackrel{*}{\longmapsto} \text{nil } \#(\text{ret } W)$ which is defined as `S # (ret V) =>* nil # (ret W)`.

The mode declaration `%mode sound +D -P -D'` specifies inputs and outputs of the defined predicate. When executed this program translates computations on the abstract machine into Mini-ML evaluations. Dependent types underlying this implementation guarantee that only valid computation sequences and evaluations are generated. The mode checker [11] verifies that all inputs are known when the predicate is called and all output arguments are known after successful execution of the predicate. To check that this program actually constitutes a proof, meta-theoretic properties such as coverage and termination need to be established. Termination guarantees that the input of each recursive call (induction hypothesis) is smaller than the input of the original call (induction conclusion). For termination checking the program needs to be well-moded. In addition, the user specifies which input arguments to consider and in which order they diminish. In the given example, we specify that the predicate `sound` should terminate in the first argument by `%terminates D (sound D P D')`. For reduction checking we specify an explicit order relation between input and output elements. In the example we say `%reduces D' < D (sound D E D')`. In general, we allow atomic, lexicographic ($\{Arg_1, Arg_2\}$) or simultaneous ($[Arg_1, Arg_2]$) subterm orderings. To show that a given program satisfies a given reduction constraint pattern, we proceed for each clause in two stages: First we extract a set Δ of reduction constraints from the recursive calls which can be assumed and the reduction constraint P of the whole clause which needs to be satisfied. Second, we prove that the set Δ implies the reduction constraint P. For proving termination of a given program, we also proceed in two stages: For each clause, and for each recursive call we first extract a set Δ of reduction constraints which are valid and a termination constraint P which characterizes the relation between the recursive call and the original call. Second, we prove that the set Δ implies the termination constraint P. For example, to show that the predicate `sound` terminates, we show the following properties:

Reduction: `%reduces D' < D (sound D P D')`
if `D4` $\prec$ `D3`, (`t_ret @ t_app2 @ D3`) $\prec$ `D2` and (`t_ret @ t_app1 @ D2`) $\prec$ `D1` then `D4` $\prec$ (`t_app @ D1`).
Termination: `%terminates D (sound D P D')`
1. `D1` $\prec$ (`app @ D1`)
2. if (`ret @ app1 @ D2`) $\prec$ `D1` then `D2` $\prec$ (`app @ D1`)
3.if (`ret @ app2 @ D3`) $\prec$ `D2` and (`ret @ app1 @ D2`) $\prec$ `D1` then `D3` $\prec$ (`app @ D1`).

We use $\prec$ to represent the subterm order relation. In general we might have nested clauses which need to be checked recursively. Moreover, we generate parametric reduction constraints for parametric sub-clauses. In Section 5 we give another example for checking termination and reduction. A more detailed explanation for extracting the termination and reduction properties can be found in [9]. In the remainder of the paper we will briefly explain the background theory and then discuss a deductive system for reasoning about structural orderings.

3 Background

The higher-order logic programming language we are working with is based on the logical framework LF [3]. The meta-language of LF is the $\lambda\Pi$-calculus. It is a three-level hierarchical calculus for *objects*, *families*, and *kinds*. Families are classified by kinds, and objects are classified by types, that is, families of kind type.

Kinds	$K := \text{type} \mid \Pi x : A.K$	Signatures	$\Sigma := \cdot \mid \Sigma, h : K \mid \Sigma, c : A$
Types	$A := hM_1 \ldots M_n \mid \Pi x : A_1.A_2$	Context	$\Gamma := \cdot \mid \Gamma, x : A$
Objects	$M := c \mid x \mid \lambda x : A.M \mid M_1 M_2$		

We will use h for type family constants, c for object constants, and x for variables. Constants are introduced through a signature. $\Pi x : A_1.A_2$ denotes the dependent function type or dependent product: the type A_2 may depend on 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 $A_1 \rightarrow A_2$. Below we assume a fixed signature Σ. The types of free variables in a term M are provided by a context Γ. 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 object are computable and that equivalent LF objects have the same canonical form up to α-conversion. We assume that constants and variables are declared at most once in a signature and context, respectively. As usual we apply tacit renaming of bound variables to maintain this assumption and to guarantee capture-avoiding substitutions.

To illustrate the use of basic notation, we consider the representation of the abstract machine which was introduced in the last section. The operations application and lambda abstraction can be represented as canonical LF objects of type `exp`. Values, continuations, instructions and states are defined in a similar fashion. The evaluation derivation $e \hookrightarrow v$ is represented by the judgement `eval : exp -> val -> type.` in Twelf. Similarly, we can encode the one-step transition relation and the multi-step transition relation as a judgements in Twelf.

```
exp:     type.                      eval:    exp -> val -> type.
lam:     (val -> exp) -> exp.       ev_lam : eval (lam E) (lam* E).
app:     exp -> exp -> exp.         ev_app : eval (app E1 E2) V
                                             <- eval E1 (lam* E1')
val:     type.                               <- eval E2 V2
lam*:    (val -> exp) -> val.                <- eval (E1' V2) V.
```

The capitalized identifiers that occur free in each declaration are implicitly Π-quantified. The appropriate type is deduced from the context during type reconstruction. The fully explicit form of the first declaration would be `ev_lam:` Π `E: val -> exp. eval (lam E) (lam* E).`

4 A logical approach to termination

4.1 Reasoning about higher-order subterm orderings

In Section 2 we sketched the analysis of higher-order logic programs for termination and reduction properties. Termination and reduction analysis is separated from reasoning about higher-order subterm relations. The analysis collects valid reduction properties as assumptions and states the ordering which needs to be satisfied under the assumptions. In this section we develop a formal inference system to check whether a set of valid reduction constraints implies an ordering constraint. For now, we consider only first-order subterm reasoning. An ordering constraint is either the $\prec$ subterm relation, the $\preceq$ subterm relation or structural equivalence relation $\equiv$. A context Δ is a set of ordering constraints.

Context	Δ	$:= \cdot \mid \Delta, P$
Ordering constraints	P	$:= Arg_1 \prec Arg_2 \mid Arg_1 \preceq Arg_2 \mid Arg_1 \equiv Arg_2$
Arg	Arg	$:= M \mid \{Arg_1, Arg_2\} \mid [Arg_1, Arg_2]$

The reasoning system should exhibit a minimal set of desired properties such as transitivity reasoning, congruence closure for structural equality reasoning, and reasoning about λ-terms. The system for first-order subterm reasoning is given in Figure 1. It is similar to the sequent calculus formulation with right and left rules for each ordering relation. $\preceq$ is defined in terms of $\prec$ and $\equiv$. If the rule $L\text{-}\prec$ has no premises, i.e., N is a constant c with no arguments, the hypothesis is contradictory and the conclusion $\Delta, M \prec c \longrightarrow P$ is trivially true. Reasoning about structural orderings is inherently different from the usual reasoning with equality and inequality. Usually when reasoning about equalities/inequalities, we reason about the value of a term. For example, the value of $hM_1 \ldots M_n$ can be

$$\frac{}{\Delta, P \longrightarrow P}\ id$$

$$\frac{}{\Delta \longrightarrow M \equiv M}\ refl \qquad \frac{\Delta, M' \equiv M \longrightarrow P}{\Delta, M \equiv M' \longrightarrow P}\ sym$$

$$\frac{\Delta \longrightarrow M_1 \equiv N_1 \quad \dots \quad \Delta \longrightarrow M_n \equiv N_n}{\Delta \longrightarrow hM_1 \dots M_n \equiv hN_1 \dots N_n}\ R{\equiv} \qquad \frac{\Delta, M_1 \equiv N_1, \dots, M_n \equiv M_n \longrightarrow P}{\Delta, hM_1 \dots M_n \equiv hN_1 \dots N_n \longrightarrow P}\ L{\equiv_1}$$

$$\frac{\Delta[M], X \equiv M \longrightarrow P}{\Delta[X], X \equiv M \longrightarrow P}\ Lsubst \qquad \frac{h \neq g}{\Delta, hM_1 \dots M_n \equiv gN_1 \dots N_k \longrightarrow P}\ L{\equiv_2}$$

$$\frac{\Delta \longrightarrow M \prec M_1 \quad \Delta \longrightarrow M_1 \prec M'}{\Delta \longrightarrow M \prec M'}\ t{\prec} \qquad \frac{\Delta \longrightarrow M \prec M_1 \quad \Delta \longrightarrow M_1 \equiv M'}{\Delta \longrightarrow M \prec M'}\ t{\prec}{\equiv}$$

$$\frac{\Delta \longrightarrow M \equiv M_1 \quad \Delta \longrightarrow M_1 \prec M'}{\Delta \longrightarrow M \prec M'}\ t{\equiv}{\prec} \qquad \frac{\Delta \longrightarrow M \equiv M_1 \quad \Delta \longrightarrow M_1 \equiv M'}{\Delta \longrightarrow M \equiv M'}\ t{\equiv}{\equiv}$$

$$\frac{\Delta \longrightarrow M \preceq N_i}{\Delta \longrightarrow M \prec hN_1 \dots N_n}\ R{\prec_i} \qquad \frac{\Delta, M \preceq N_1 \longrightarrow P \quad \dots \quad \Delta, M \preceq N_n \longrightarrow P}{\Delta, M \prec hN_1 \dots N_n \longrightarrow P}\ L{\prec}$$

$$\frac{\Delta \longrightarrow M \prec N}{\Delta \longrightarrow M \preceq N}\ R{\preceq_1} \qquad \frac{\Delta \longrightarrow M \equiv N}{\Delta \longrightarrow M \preceq N}\ R{\preceq_2}$$

$$\frac{\Delta, M \prec N \longrightarrow P \quad \Delta, M \equiv N \longrightarrow P}{\Delta, M \preceq N \longrightarrow P}\ L{\preceq}$$

Fig. 1. First-order Subterm Relations ($\prec$, $\preceq$)

equal to the value of $gN_1 \dots N_k$ where h and g denote different function symbols. When reasoning about subterms, we are only interested in the syntactic structure of a term. Therefore, a term $hM_1 \dots M_n$ can never be structurally equivalent to $gN_1 \dots N_k$, if $g \neq h$. If $hM_1 \dots M_n \equiv gN_1 \dots N_k$ occurs in our assumptions, we can infer anything ($L{\equiv_2}$).

This system is already expressive enough to prove termination of the translation of small-step semantics into big-step Mini-ML semantics which is implemented by the `sound` predicate (see p. 4). One of the claims we need to prove during termination checking is the following:

(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ `D2` $\prec$ (`app @ D1`)

The proof written in a bottom-up linear notation is as follows:

2.	(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ (`ret @ app1 @ D2`) $\prec$ `D1`	id
	(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ `D2` $\equiv$ `D2`	$refl$
	(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ `D2` $\preceq$ `D2`	$R{\preceq_2}$
	(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ `D2` $\prec$ (`app1 @ D2`)	$R{\prec_2}$
	(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ `D2` $\preceq$ (`app1 @ D2`)	$R{\preceq_1}$
1.	(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ `D2` $\prec$ (`ret @ app1 @ D2`)	$R{\prec_2}$
	(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ `D2` $\prec$ `D1`	$t{\prec}$ using 1,2
	(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ `D2` $\preceq$ `D1`	$R{\preceq_1}$
	(`ret @ app1 @ D2`) $\prec$ `D1` $\longrightarrow$ `D2` $\prec$ (`app @ D1`)	$R{\prec_2}$

$$\frac{\Delta \longrightarrow O_1 \prec O_1{}'}{\Delta \longrightarrow \{O_1, O_2\} \prec \{O_1{}', O_2{}'\}}\ RLex{\prec_1} \qquad \frac{\Delta \longrightarrow O_1 \equiv O_1{}' \quad \Delta \longrightarrow O_2 \prec O_2{}'}{\Delta \longrightarrow \{O_1, O_2\} \prec \{O_1{}', O_2{}'\}}\ RLex{\prec_2}$$

$$\frac{\Delta \longrightarrow \{O_1, O_2\} \prec \{O_1{}', O_2{}'\}}{\Delta \longrightarrow \{O_1, O_2\} \preceq \{O_1{}', O_2{}'\}}\ RLex{\preceq_1} \qquad \frac{\Delta \longrightarrow \{O_1, O_2\} \equiv \{O_1{}', O_2{}'\}}{\Delta \longrightarrow \{O_1, O_2\} \preceq \{O_1{}', O_2{}'\}}\ RLex{\preceq_2}$$

$$\frac{\Delta \longrightarrow O_1 \equiv O_1{}' \quad \Delta \longrightarrow O_2 \equiv O_2{}'}{\Delta \longrightarrow \{O_1, O_2\} \equiv \{O_1{}', O_2{}'\}}\ RLex{\equiv}$$

$$\frac{\Delta, O_1 \prec O_1{}' \longrightarrow P \quad \Delta, O_1 \equiv O_1{}', O_2 \prec O_2{}' \longrightarrow P}{\Delta, \{O_1, O_2\} \prec \{O_1{}', O_2{}'\} \longrightarrow P}\ LLex{\prec}$$

$$\frac{\Delta, \{O_1, O_2\} \prec \{O_1{}', O_2{}'\} \longrightarrow P \quad \Delta, \{O_1, O_2\} \equiv \{O_1{}', O_2{}'\} \longrightarrow P}{\Delta, \{O_1, O_2\} \preceq \{O_1{}', O_2{}'\} \longrightarrow P}\ LLex{\preceq}$$

$$\frac{\Delta, O_1 \equiv O_1{}', O_2 \equiv O_2{}' \longrightarrow P}{\Delta, \{O_1, O_2\} \equiv \{O_1{}', O_2{}'\} \longrightarrow P}\ LLex{\equiv}$$

Fig. 2. Lexicographic Extensions

We can extend the system with rules for lexicographic orderings by defining left and right rules (see Figure 2). O_1 and O_2 are considered to be lexicographically smaller than O_1' and O_2' if either O_1 is smaller than O_1' or O_1 is structurally equivalent to O_1' and O_2 is smaller than O_2'. This disjunctive choice is reflected in the two rules $RLex{\prec_1}$ and $RLex{\prec_2}$. If we assume O_1 and O_2 to be lexicographically smaller than O_1' and O_2', then we need to be able to prove some ordering P under the assumption O_1 is smaller than O_1' and under the assumptions O_1 is structurally equivalent to O_1' and O_2 is smaller than O_2' (see $LLex{\prec}$). The rules for $\preceq$ and $\equiv$ are straightforward. Similarly, we can define extensions for simultaneous orderings. Although we do not pursue other more complex structural orderings for now, in general this approach can be also applied to define extensions for simplification orderings, multi-set orderings or recursive path orderings. In this paper, we focus on extending the system to higher-order subterm relations.

In the setting of a dependently typed calculus, we face two challenges: First, we need to reason about orders involving higher-order terms. Second, we might synthesize parametric order relations due to parametric subgoals. When considering higher-order terms, we need to find an appropriate interpretation for lambda-terms. This problem is illustrated by the following example. Assume the constructor `lam` is defined as `lam: (exp -> exp) -> exp`. We want to show that $E\ a$ is a subterm of `lam` $\lambda x.E\,x$ where a is a parameter. In the informal proof we might count the number of constructors and consider $E\ a$ an instance of $\lambda x.E\,x$. Therefore we consider a term M a subterm of $\lambda x.N\,x$ if there exists a parameter instantiation $\underline{a}$ for x s.t. M is smaller than $[\underline{a}/x]N$. We will use the convention that a will represent a new parameter, while $\underline{a}$ stands for an already defined parameter. To adopt a logical point of view, the λ-term on the left of a subterm relation can be interpreted as universally quantified and the λ-term on the right as existentially quantified.

Another example is taken from the representation of first-order logic [6]. We can represent formulas by the type family `o`. Individuals are described by the type family `i`. The constructor $\forall$ can be defined as `forall: (i -> o) -> o`. We might want to show that $A\ T$ (which represents $[t/x]A$) is smaller than `forall` $\lambda x.A\,x$ (which represents $\forall x.A$). Similarly, we might count the number of quantifiers and connectives in the informal proof, noting that a term t in first-order logic cannot contain any logical symbols. Thus we may consider $A\,T$ a subterm of `forall` $\lambda x.A\,x$ as long as there is no way to construct an object of type `i` from objects of type `o`. A term M is smaller than a λ-term $(\lambda x.N)$ if there exists an instantiation T for x s.t. M is smaller than $[T/x]N$ and the type of T is a subordinate to N. For a more detailed development of mutual recursion and subordination we refer the reader to R. Virga's PhD thesis [14].

$$\frac{}{\Delta, \lambda x.M \equiv hN_1 \ldots N_n \longrightarrow P}\ L{\equiv_3} \qquad \frac{}{\Delta, a \equiv hN_1 \ldots N_n \longrightarrow P}\ L{\equiv_4}$$

$$\frac{\Delta \longrightarrow [a/x]M \equiv [a/x]N}{\Delta \longrightarrow \lambda x.M \equiv \lambda x.N}\ R{\equiv}\,\lambda \qquad \frac{\Delta, [\underline{a}/x]M \equiv [\underline{a}/x]N \longrightarrow P}{\Delta, \lambda x.M \equiv \lambda x.N \longrightarrow P}\ L{\equiv}\,\lambda$$

$$\frac{\Delta \longrightarrow [a/x]M \prec N}{\Delta \longrightarrow \lambda x : A.M \prec N}\ RL{\prec}\,\lambda^a \qquad \frac{\Delta, [\underline{a}/x]M \prec N \longrightarrow P}{\Delta, \lambda x : A.M \prec N \longrightarrow P}\ LL{\prec}\,\lambda$$

$$\frac{\Delta \longrightarrow M \prec [\underline{a}/x]N}{\Delta \longrightarrow M \prec \lambda x : A.N}\ RR{\prec}\,\lambda \qquad \frac{\Delta, M \prec [a/x]N \longrightarrow P}{\Delta, M \prec \lambda x : A.N \longrightarrow P}\ LR{\prec}\,\lambda^a$$

$$\frac{\Delta \longrightarrow [a/x]M \preceq N}{\Delta \longrightarrow \lambda x : A.M \preceq N}\ RL{\preceq}\,\lambda^a \qquad \frac{\Delta, [\underline{a}/x]M \preceq N \longrightarrow P}{\Delta, \lambda x : A.M \preceq N \longrightarrow P}\ LL{\preceq}\,\lambda$$

$$\frac{\Delta \longrightarrow M \preceq [\underline{a}/x]N}{\Delta \longrightarrow M \preceq \lambda x : A.N}\ RR{\preceq}\,\lambda \qquad \frac{\Delta, M \preceq [a/x]N \longrightarrow P}{\Delta, M \preceq \lambda x : A.N \longrightarrow P}\ LR{\preceq}\,\lambda^a$$

$$\frac{\Delta \longrightarrow [a/x]P}{\Delta \longrightarrow \Pi x.P}\ R\Pi^a \qquad \frac{\Delta, [\underline{a}/x]P \longrightarrow P'}{\Delta, \Pi x.P \longrightarrow P'}\ L\Pi$$

Fig. 3. Higher-order Extensions

Reasoning about λ-terms cannot be solely based $\prec$ and $\equiv$, as neither $[a/x]M \equiv \lambda x.M$ nor $[a/x]M \prec \lambda x.M$ is true. Therefore, we introduce a set of inference rules to reason about $\preceq$ which are similar to the $\prec$ rules. Extensions to higher-order subterm reasoning are presented in Figure 3. As we potentially need different instantiations of the relation $\lambda x.M \prec N$ when reading the inference rules bottom-up, we need to copy $\lambda x.M \prec N$ in Δ even after it has been instantiated. For simplicity, we assume all assumptions persist. Note that we only show the case for mutual recursive type families, but the case where type family a is a subordinate to the type family a' can be added in straightforward manner. For handling parametric order relations we add $R\Pi^a$ and $L\Pi$ which are similar to universal quantifier rules in the sequent calculus. Similar to instantiations of $\lambda x.M \prec N$, we need to keep a copy of $\Pi x.P$ after it has been instantiated. The weakening and contraction property hold for the given calculus.

Reasoning about higher-order subterm relations is complex due to instantiating λ-terms and parametric orderings. Although soundness and decidability of the first-order reasoning system might still be obvious, this is non-trivial in the higher-order case. In this paper, we concentrate on proving consistency of the higher-order reasoning system. Consistency of the system implies soundness, i.e. any step in proving an order relation from a set of assumptions is sound. The proof also implies completeness i.e. anything which should be derivable from a set of assumptions is derivable.

4.2 Consistency of higher-order subterm reasoning

In general, the consistency of a logical system can be shown by proving cut admissible.

$$\frac{\Delta \longrightarrow P \qquad \Delta, P \longrightarrow P'}{\Delta \longrightarrow P'}\, cut$$

Δ usually consists of elements which are assumed to be true. Any P which can be derived from Δ is true and can therefore be added to Δ to prove P'. In our setting Δ consists of reduction orderings which have already been established. Hence, the reduction orderings are true independently from any other assumptions in Δ and they are assumed to be valid. The application of the cut-rule in the proof can therefore only introduce valid orderings as additional assumptions in Δ.

Theorem 2 (Admissibility of cut).

1. *If* $\mathcal{D} : . \longrightarrow M \equiv M'$ *and* $\mathcal{E} : \Delta, M \equiv M' \longrightarrow P'$ *then* $\mathcal{F} : \Delta \longrightarrow P'$.
2. *If* $\mathcal{D} : . \longrightarrow \sigma M \prec M'$ *and* $\mathcal{E} : \Delta, \lambda \overrightarrow{x}.M \prec M' \longrightarrow P'$ *then* $\mathcal{F} : \Delta \longrightarrow P'$.
3. *If* $\mathcal{D} : . \longrightarrow \sigma M \preceq M'$ *and* $\mathcal{E} : \Delta, \lambda \overrightarrow{x}.M \preceq M' \longrightarrow P'$ *then* $\mathcal{F} : \Delta \longrightarrow P'$.

The substitution σ maps free variables to new parameters. In general, we allow the cut between $\sigma M \prec N$ and $\lambda \overrightarrow{x}.M \prec N$ where σM is an instance of $\lambda \overrightarrow{x}.M$.

However, we will not be able to show admissibility of cut directly in the given calculus due to the non-deterministic choices introduced by λ-terms. Consider, for example, the cut between

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ . \longrightarrow \sigma \circ [a/x]M \prec N\end{array}}{. \longrightarrow \sigma \lambda x.M \prec N}\, RL\text{-}{\prec}\lambda. \qquad \begin{array}{c}\mathcal{E}\\ \Delta, \lambda x.M \prec N \longrightarrow P\end{array}$$

We would like to apply inversion on $\mathcal{E}$; therefore we need to consider all possible cases of previous inference steps which lead to $\mathcal{E}$. There are three possible cases we need to consider: $L{\prec}$, $LR{\prec}\lambda^a$ and $LL{\prec}\lambda$. Unfortunately, it is not possible to appeal to the induction hypothesis and finish the proof in the $L{\prec}$ and $LR{\prec}\lambda$ case. This situation does not arise in the first order case, because all the inversion steps were unique. In the higher-order case we have many choices and we are manipulating the terms by instantiating variables in λ-terms.

The simplest remedy seems to restrict the calculus in such a way, that we always first introduce all possible parameters, and then instantiate all Π quantified orders and $\lambda x.M$ which occur on the left side of a relation. This means, we push the instantiation with parameter variables as high as possible in the proof tree. This way, we can avoid the problematic case above, because we only instantiate a λ-term in $\lambda x.M \prec N$, if N is atomic.

Therefore, we proceed as follows: First, we define an inference system, in which we first introduce all new parameters. This means we restrict the application of the $R\preceq_1$, $R\preceq_2$, $R\prec_i$, $RR\prec\lambda$, $RR\preceq\lambda$ to only apply if the left hand side of the principal order relation $\prec$ or $\preceq$ is already of base type. Similarly, we restrict the application of $L\preceq$, $LL\preceq\lambda$, $LL\prec\lambda$, i.e. the rule only applies if the right hand side of the principal ordering relation is of base type. In addition, we show that the application of the identity rules can be restricted to atomic terms. Second, we show this restricted system is sound and complete with respect to the original inference system. Third, we show that cut is admissible in the restricted calculus. This implies that cut is also admissible in the original calculus. The proof proceeds by nested induction on the structure of P, the derivation $\mathcal{D}$ and $\mathcal{E}$. More precisely, we appeal to the induction hypothesis either with a strictly smaller order constraint P or P stays the same and one of the derivations is strictly smaller while the other one stays the same. For a more detailed development of the intermediate inference system and the proofs we refer to [9]

Using the cut-admissibility theorem, cut-elimination follows immediately. Therefore, our inference system is consistent. This implies that all derivation steps in the given reasoning system are sound. It also implies that the inference rules are strong enough to deduce as much as possible from the assumptions and hence the system is complete.

5 Related work and Conclusion

Most work in automating termination proofs has focused on first-order languages. The most general method for synthesizing termination orders for a given term rewriting system (TRS) is by Arts and Giesl [1]. One approach to proving termination of logic programs is to translate it into a TRS and show termination of the TRS instead. However this approach has several drawbacks. In general, a lot of information is lost during the translation. In particular, if termination analysis fails for the TRS, it is hard to provide feedback and re-use this failure information to point to the error in the logic program. Moreover important structural information is lost during the translation and constructors and functions are indistinguishable. One of the consequences is that proving termination of the TRS often requires more complicated orders. This is illustrated using an example from arithmetic. Using logic programming we implement a straightforward version of `minus` and the quotient predicate `quot`.

```
minus : nat -> nat -> nat -> type.       quot : nat -> nat -> nat -> type.
%mode minus +X +Y -Z.                    %mode quot +X +Y -Z.
m_z : minus X z X.                       q_z : quot z (s Y) z.
m_s : minus (s X) (s Y) Z                q_s : quot (s X) (s Y) (s Z)
       <- minus X Y Z.                          <- minus X Y X'
                                                <- quot X' (s Y) Z.
%reduces Z <= X (minus X Y Z).
%terminates X (minus X Y Z).             %terminates X (quot X Y Z).
```

Proving termination of `quot` is straightforward with the presented method. We first prove termination of `minus`. In addition we show that `minus X Y Z` satisfies the reduction constraint `Z <= X`. When we prove termination of `quot`, we can assume the reduction constraint $X' \preceq X$. As the reduction constraint $X' \preceq X$ implies $X' \prec (\mathtt{s}\, X)$, we proved termination of `quot`. Note that only subterm reasoning is required to prove termination of `quot` while other methods like Arts and Giesl's method for proving the corresponding term rewrite system needs a recursive path ordering. Another example is an algorithm to compute the negation normal form of a first-order logical formula and uses higher-order functions (see [9]). We implemented this algorithm using two mutual recursive predicates. Termination of this algorithm can be proven based on subterm ordering, while the corresponding term rewriting system given in [5] requires a more complicated ordering like recursive path ordering.

Although some of the underlying ideas in higher-order term rewriting system (HTRS) are shared with the logical framework, there are two principal differences: First, all arguments of a predicate are in canonical form and therefore are terminating. This additional restriction simplifies termination analysis in the logical framework. On the other hand, the dependently typed $\lambda\Pi$ calculus, on which the logical framework LF is based, allows the representation of hypothetical and parametric judgements which make termination and reduction analysis more challenging. Hypothetical and parametric judgements have in general no counterpart in HTRS and their translation to HTRS seems difficult.

One approach which analyzes logic programs directly has been developed by Plümer [10]. The idea is to construct a subgoal dependency graph and then show that this graph is acyclic according to some ordering. Although this approach works well for Prolog programs, it is not obvious how to extend this method in a higher-order setting with parametric and hypothetical subgoals. In this paper we propose a proof-theoretical foundation for termination checking of higher-order logic programs. To infer that a specified ordering holds under a set of assumptions, we introduced a deductive system to reason about structural orderings. We focused on consistency of the presented reasoning system. Consistency implies that anything we derive from the assumption is sound. Cut-elimination implies that the reasoning system is complete, i.e. everything which should be derivable from the assumptions is in fact derivable. A valuable advantage of this approach is its extensibility and its modularity. Similar to lexicographic extensions we can imagine extensions for simplification ordering, multi-set ordering and recursive path orderings. In addition our method allows us to combine different structural orderings for different predicates. This is unlike other termination methods which require one ordering for the whole dependency graph.

This paper builds on Rohwedder and Pfenning's work on mode and termination checking for higher-order logic programs [11]. Their termination checker requires a direct relationship between inputs of the recursive call and inputs of the original call without taking into account input and output relations. Reasoning about orderings allows us to check proofs by complete induction such as the soundness proof discussed in this paper. The emphasis of their work has been the correctness of the termination checker with respect to the operational semantics of Twelf programs. Although we have not proven the correctness of the extended termination checker, we are expecting the proof to be a straightforward extension of their proof.

One question not discussed in this paper is whether the system is decidable. This question is not trivial as we can potentially instantiate λ-terms and Π-quantified order relations which occur in the context multiple times. One approach for proving decidability would be to show that we can bound the number of instantiations needed.

Our system is implemented as part of Twelf, and efficiently checks programs and proofs. Currently multiplicity is restricted to one, i.e. we instantiate Π-quantified orderings and λ-terms occurring on the left hand side of a relation in the hypothesis just once. Although we can artificially construct examples which require multiplicity more than one, we have not encountered these cases in practice so far. If a higher multiplicity is needed, an appropriate warning is returned. As our algorithm analyzes program clauses directly, its behaviour is easy to understand. In the case of failure, our implementation will point to the clause and argument where the error occurred. This enables the user to either revise the program or strengthen the ordering. In practice we have used the termination and reduction checker on examples from compiler verification (soundness and completeness proofs for stack semantics and continuation-based semantics), cut-elimination and normalization proofs for intuitionistic and classical logic, soundness and completeness proofs for the Kolmogorov translation of classical into intuitionistic logic (and vice versa)[1]. Currently, Rohwedder and Pfenning's termination checker is used in the automatic induction theorem prover. In the future, we plan to incorporate the extended termination checker.

Acknowledgements

The author gratefully acknowledges numerous fruitful discussions with Frank Pfenning regarding the subject of this paper. His guidance and careful reading of this paper contributed greatly to its clarity and correctness.

[1] The code of all the examples mentioned in the paper can be found at `http://www.cs.cmu.edu/~bp/code`.

References

1. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
2. 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.
3. 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.
4. J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 402–411, Trento, Italy, July 1999. IEEE Computer Society Press.
5. Olav Lysne and Javier Piris. A termination ordering for higher order rewrite systems. In Jieh Hsiang, editor, *Proceedings of the Sixth International Conference on Rewriting Techniques and Applications*, pages 26–40, Kaiserslautern, Germany, April 1995. Springer-Verlag LNCS 914.
6. 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.
7. Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically.
8. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
9. Brigitte Pientka. Termination and reduction checking in the logical framework. Technical report cmu-cs-01-115, Carnegie Mellon University, 2001.
10. Lutz Plümer. *Termination Proofs for Logic Programs*. LNAI 446. Springer-Verlag, 1990.
11. Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.
12. J. van de Pol and H. Schwichtenberg. Strict functionals for termination proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 350–364, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.
13. Femke van Raamsdonk. Higher-order rewriting. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA '99)*, pages 220–239, Trento, Italy, July 1999. Springer-Verlag LNCS 1631.
14. Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 2000.
15. Christoph Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1), 1994.