Termination and Reduction Checking in the Logical Framework

Brigitte Pientka and Frank Pfenning
Department of Computer Science
Carnegie Mellon University

1 Introduction

The logical framework LF [HHP93] offers concise encodings of deductive systems and their meta-theory. Twelf [SP98] is a realization of LF. It provides a higher-order logic programming language for the implementation of deductive systems as well as a higher-order inductive theorem prover to automatically prove properties about these systems. The inductive theorem prover has been used successfully to prove several challenging theorems like cut-admissibility of intuitionistic logic and the Church-Rosser theorem. Under the proofs-as-programs paradigm the application of the induction hypothesis (IH) in a proof corresponds to the recursive call in a program. To check that the IH application is valid, we need to show that the induction hypothesis is smaller than the induction conclusion according to a well-founded order. This corresponds to proving that the arguments in the recursive call decrease according to a well-founded order, i.e., a program terminates. Twelf uses a termination checker based on structural ordering [RP96] to check termination of programs and to generate valid induction hypotheses according to a given order.

We are interested in extending the power of the induction component to enable complete induction or so called course-of-value ind. Complete induction plays an important role in proofs about sequences of computation. These proofs follow by induction on the structure of the computation sequence. Often we do not only want to apply the induction hypothesis to immediate subsequences, but to all smaller subsequences. In general, we also want to be able to apply the induction hypothesis to the outcome of a previous IH application. This can be done by first showing that the computation sequence resulting from the IH application is smaller than the sequence we applied the IH to. Then we verify that each IH application itself is valid, i.e. the sequence we apply the IH to is smaller than the original sequence. To show that a subcomputation is smaller than the original sequence of computation we need to reason about orders relating subsequences.

In this paper, we present a reduction and termination checker which reasons about orders. The reduction checker verifies properties relating input and output. The termination checker proves properties relating inputs of the original call to inputs of the recursive call. Both checkers take into account already derived reduction properties and reason about them. Our method can be used in a first-order and higher-order framework. To infer whether an order holds, we use a deductive system in the spirit of the sequent calculus. We show completeness of our approach by proving cut to be admissible. The reduction and termination checker is implemented in Twelf and has been used to check various programs and inductive proofs.

Most work in automating termination proofs has been done for first-order languages ([AG00], [GBG99]). To show termination in a higher-order setting mainly two methods have been developed (for a survey see [vR99]): the first approach relies on strict functionals by van de Pol [vdPS95], and the second one is a technique developed by Jouannaud and Rubio for proving termination of recursive path ordering [JR99]. Neither of these approaches provides an algorithmic method for proving termination.

Unlike most other approaches, we are interested in checking a given order for a program-proof and not in synthesizing an order for a program-proof. As a consequence the user has to specify an order. 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-proof or to refine the specified order.

The paper is organized as follows: We illustrate complete induction and the reasoning about orders by discussing a proof which arises during compiler verification [HP92] in section 2. By means of this example, we review the background (see 3) and explain the main idea of our approach. In section 4 we outline deductive system for reasoning about orders and discuss completeness of the system. Finally, in section 5 we discuss related work, summarize the results and outline future work.

2 Motivation

In this section, we motivate our approach by considering an example from compiler verification [HP92]. A compiler can be viewed as an abstract machine. The computation in an abstract machine (and a real machine) can be represented as a sequence of states.

We consider the operational semantics of an abstract machine for a small subset of a programming language. The only operations we are discussing are application and lambda abstraction. A state consists of a continuation K together with a machine instruction I. In each single step we transition from one state of the abstract machine to another. The instruction I represents the program to be executed. The continuation K represents the computation which needs to be performed on I and can be viewed as an environment.
Instructions model left to right execution of the program, i.e. if we evaluate the application $e_1 e_2$, then we first evaluate expressions e_1 and save the continuation $\lambda v.\mathrm{app}_1\ v\ e_2$. The continuation represents what remains to be done. When the

expressions :	e	$::= x \mid e_1 e_2 \mid \lambda x.e$
values :	v	$::= \Lambda x.e$
instructions :	I	$::= \mathrm{ret}\ v \mid \mathrm{ev}\ e \mid \mathrm{app}_1\ ve \mid \mathrm{app}_2\ v_1 v_2$
continuation :	K	$::= \mathrm{init} \mid K; \lambda v.I$
states :	S	$::= K\#I \mid \mathrm{ans}\ v$

value v_1 of expression e_1 is computed, we return it (ret v_1). Then, expression e_2 is evaluated and the continuation $\lambda v.\mathrm{app}_2\ v_1\ v$ is saved until the computation of expression e_2 is finished.

Figure 1 describes the operational semantics of the abstract machine and inference rules representing the evaluation of expressions. Deductions of the

One-step transitions:

$$\frac{}{K\#(\mathrm{ev}\ \lambda x.e) \longmapsto K\#(\mathrm{ret}\ \Lambda x.e)}\ lam$$

$$\frac{}{K\#(\mathrm{ev}\ e_1 e_2) \longmapsto (K; \lambda v_1.\mathrm{app}_1\ v_1 e_2)\#(\mathrm{ev}\ e_1)}\ app$$

$$\frac{}{K\#(\mathrm{app}_1\ v_1 e_2) \longmapsto (K; \lambda v_2.\mathrm{app}_2\ v_1 v_2)\#(\mathrm{ev}\ e_2)}\ app1$$

$$\frac{}{K\#(\mathrm{app}_2\ (\Lambda v2.e_1)v_2) \longmapsto K\#(\mathrm{ev}\ (\Lambda v_2.e_1)v_2)}\ app2$$

$$\frac{}{(K; I)\#(\mathrm{ret}\ v) \longmapsto K\#(Iv)}\ ret$$

$$\frac{}{\mathrm{init}\#(\mathrm{ret}\ v) \longmapsto \mathrm{ans}\ v}\ init$$

Multi-step transitions:

$$\frac{}{S \overset{*}{\longmapsto} S}\ id$$

$$\frac{S \longmapsto S_1 \quad S_1 \overset{*}{\longmapsto} S'}{S \overset{*}{\longmapsto} S'}\ <<$$

Evaluation of expressions

$$\frac{}{\lambda x.e \hookrightarrow \Lambda x.e}\ ev\_lam$$

$$\frac{e_1 \hookrightarrow \lambda x.e_1{}' \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e_1{}' \hookrightarrow v}{e_1 e_2 \hookrightarrow v}\ ev\_app$$

Figure 1: abstract machine

judgement $S \overset{*}{\longmapsto} S'$ have a very simple form: They all consist of a sequence of single steps terminated by an application of the *id* rule. We will follow standard practice and use a linear notation for sequences of steps:

$$S_1 \longmapsto S_2 \longmapsto S_3 \longmapsto \ldots \longmapsto S_n$$

Similarly, we will mix multi-step and single-step transitions in sequences with the obvious meaning.

We will discuss a central theorem when proving soundness of a compiler. The lemma states that a complete computation with an appropriate initial state can be translated into an evaluation followed by another complete computation. More formally, if an expression e evaluates in an environment K in multiple steps to some answer w then there exists an evaluation of expression e which results in a value v and a subcomputation in environment K starting with the value v which will return an answer w. For a more detailed discussion of this example we refer to [Pfe00].

Theorem 1 *If $\mathcal{D} :: K\#(ev\ e) \stackrel{*}{\longmapsto} (ans\ w)$ then $\mathcal{E} :: e \hookrightarrow v$ and $\mathcal{D}' :: K\#(ret\ v) \stackrel{*}{\longmapsto} (ans\ w)$ and $\mathcal{D}' \prec \mathcal{D}$.*

Proof: By structural induction on $\mathcal{D}$. Note we need the extra side condition $\mathcal{D}' \prec \mathcal{D}$ to apply the induction hypothesis on $\mathcal{D}'$ several times to finish the prove.

Case: $\mathcal{D}$ begins with *app*.

$$\mathcal{D} = K\#(\text{ev } e_1 e_2) \longmapsto \underbrace{(K; \lambda v_1.\text{app}_1\ v_1 e_2)\#(\text{ev } e_1) \stackrel{*}{\longmapsto} (\text{ans } w)}_{\mathcal{D}'}$$

By induction hypothesis on $\mathcal{D}'$ there exists a value v_1 and an evaluation $\mathcal{E}' : e_1 \hookrightarrow v_1$ and a subcomputation $\mathcal{D}_1 : (K; \lambda v_1.\text{app}_1\ v_1 e_2)\#(\text{ret } v_1) \stackrel{*}{\longmapsto} (\text{ans } w)$ s.t. $\mathcal{D}_1 \prec \mathcal{D}$. By unfolding the subcomputation $\mathcal{D}_1$ we yield the following computation sequence:

$$\begin{aligned}\mathcal{D}_1 = \quad & (K; \lambda v_1.\text{app}_1\ v_1 e_2)\#(\text{ret } v_1) \longmapsto K\#(\text{app}_1\ v_1 e_2) \longmapsto \\ & \underbrace{(K; \lambda v_2.\text{app}_2\ v_1 v_2)\#(\text{ev } e_2) \stackrel{*}{\longmapsto} (\text{ans } w)}_{\mathcal{D}''}\end{aligned}$$

By induction hypothesis on $\mathcal{D}''$ there exists a value v_2 and an evaluation $\mathcal{E}'' : e_2 \hookrightarrow v_2$ and a subcomputation $\mathcal{D}_2 : (K; \lambda v_2.\text{app}_2\ v_1 v_2)\#(\text{ret } v_2) \stackrel{*}{\longmapsto} (\text{ans } w)$ s.t. $\mathcal{D}_2 \prec \mathcal{D}''$.

By unfolding the subcomputation $\mathcal{D}''$:

$$\mathcal{D}_3 = (K; \lambda v_2.\text{app}_2\ v_1 v_2)\#(\text{ret } v_2) \longmapsto \underbrace{K\#(\text{app}_2\ v_1 v_2) \stackrel{*}{\longmapsto} (\text{ans } w)}_{\mathcal{D}_4}$$

We can unfold $\mathcal{D}_4$ and set v_1 to $\Lambda x.e'_1$:

$$\mathcal{D}_4 = K\#(\text{app}_2\ (\Lambda x.{e_1}')v_2) \longmapsto \underbrace{K\#(\text{ev } [v_2/x]{e_1}') \stackrel{*}{\longmapsto} (\text{ans } w)}_{\mathcal{D}'''}$$

By induction hypothesis on $\mathcal{D}'''$ there exists a value v and an evaluation$\mathcal{E}''' : ([v_2/x]{e_1}') \hookrightarrow v$ and a subcomputation $\mathcal{D}_5 : K\#(\text{ret } v) \stackrel{*}{\longmapsto} (\text{ans } w)$ s.t. $\mathcal{D}_5 \prec \mathcal{D}'''$ Recall, we needed to show the following two facts:

$$\mathcal{E} : e_1 e_2 \hookrightarrow v \text{ and } \mathcal{D}''' : K\#(\text{ret } v) \stackrel{*}{\longmapsto} (\text{ans } w) \text{ and } \mathcal{D}''' \prec \mathcal{D}.$$

We showed that we can derive $\mathcal{D}'''$ after several unfolding steps and three applications of the induction hypothesis. We can construct $\mathcal{E}$ by using *ev_app* rule and $\mathcal{E}', \mathcal{E}'', \mathcal{E}'''$ as premises.

To justify each of the IH applications we need to verify 1) if we apply the IH to $\mathcal{D}'$ ($\mathcal{D}'', \mathcal{D}'''$ resp.), the resulting sequence $\mathcal{D}_1$ ($\mathcal{D}_2$, $\mathcal{D}_5$ resp.) is shorter (reduction property) and 2) if we apply the IH to $\mathcal{D}'$ ($\mathcal{D}'', \mathcal{D}'''$ resp.), then $\mathcal{D}'$ ($\mathcal{D}'', \mathcal{D}'''$ resp.) is smaller than the original computation sequence $\mathcal{D}$ (termination property). To establish reduction and termination properties we need to apply transitivity reasoning.

□

This proof illustrates the need to apply the induction hypothesis to the outcome of the previous IH application. We formulated this reduction property explicitly in the theorem to emphasize the necessary reasoning steps. However, a more natural way of dealing with this situation is to strengthen the induction order. First, we check the reduction property (reduction checker) and then we show that the IH applications are valid (termination checker) by taking into account the derived reduction properties. In the next section we review the basic principles of the logical framework and show the basic idea of our approach.

3 Basic Notation

The Logical Framework (LF) was first presented in [HHP93] and forms the basis of Twelf. The LF calculus is a three-level 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	$::=$	type $\mid \Pi x : A.K$
Types	A	$::=$	$hM_1 \ldots M_n \mid \Pi x : A_1.A_2$
Objects	M	$::=$	$c \mid x \mid \lambda x : A.M \mid M_1 M_2$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a : K \mid \Sigma, c : A$
Context	Γ	$::=$	$\cdot \mid \Gamma, \forall x : A \mid \Gamma, \exists x : A$

We will use h for type family constants, c for object constants, and x for variables. $\Pi x : A_1.A_2$ 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 $A_1 \rightarrow A_2$.

The following principal judgments characterize the LF type theory.

$\Gamma \vdash_\Sigma M \equiv M' : A$, $\Gamma \vdash_\Sigma A \equiv A' : type$	object and type equivalence
$\vdash \Sigma$, $\vdash_\Sigma \Gamma$, $\Gamma \vdash_\Sigma K$	the validity of signatures, contexts, kinds
$\Gamma \vdash_\Sigma A : K$ and $\Gamma \vdash_\Sigma M : A$	assigning kinds to types, types to objects

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 system and the multi-step transition system as a judgements in Twelf.

```
exp :  type.
val :  type.
lam :  (val -> exp) -> exp.
app :  exp -> exp -> exp.
```

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

In this example we reversed the function arrows, writing $A_2 \leftarrow A_1$, instead of $A_1 \rightarrow A_2$ following logic programming notation. Since $\rightarrow$ is right associative, $\leftarrow$ is left associative. 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:` ΠE: `val -> exp.  eval (lam` E`) (lam*` E`)`.

The proof discussed in the previous section can be written as a judgement `csd`. The recursive call in the `csd_app` declaration corresponds to the application of the induction hypothesis. `(D' << app)` represents the unfolding step $\mathcal{D} = K\#(\mathrm{ev}\ e_1 e_2) \longmapsto \underbrace{(K; \lambda v_1.\mathrm{app}_1\ v_1 e_2)\#(\mathrm{ev}\ e_1) \overset{*}{\longmapsto} (\mathrm{ans}\ w)}_{\mathcal{D}'}$ in the informal proof.

```
csd :  K # (ev E) =>* (answer W) -> eval E V -> K # (ret V) =>* (answer W) -> type.
%mode csd +CS -E -CS'.
csd_lam :   csd (D' << lam) (ev_lam) D'.
csd_app :   csd (D' << app) (ev_app E''' E'' E') D5
            <- csd D' E' (D'' << app1 << ret)
            <- csd D'' E'' (D''' << app2 << ret)
            <- csd D''' E''' D5.
```

For checking termination the user has to specify which input arguments we need to consider and in which order they diminish. The specified input argument can either be atomic, or represent a lexicographic $(\{Arg_1, Arg_2\})$ or simultaneous $([Arg_1, Arg_2])$ relation of several input arguments. In the given example, we specify that the predicate `csd` should terminate in the first argument by `%terminates D (csd D E D')`. For reduction checking we specify an explicit order relation between input and output elements which relates either atomic arguments or lexicographic / simultaneous arguments to each other. In the example we could say `%reduces D' < D (csd D E D')`. To show that the implementation of the proof terminates, we need to prove the following two properties:

1. Reduction: if $(\mathcal{D}'' << \mathrm{app1} << \texttt{ret}) \prec \mathcal{D}'$, $(\mathcal{D}''' << \mathrm{app2} << \texttt{ret}) \prec \mathcal{D}''$ and $\mathcal{D}_5 \prec \mathcal{D}'''$ then $(\mathcal{D}_5 \prec (\mathcal{D}' << \mathrm{app}))$.
2. Termination:
   (a) $\mathcal{D}' \prec (\mathcal{D}' << \mathrm{app1})$
   (b) if $(\mathcal{D}'' << \mathrm{app1} << \texttt{ret}) \prec \mathcal{D}'$ then $\mathcal{D}'' \prec (\mathcal{D}' << \mathrm{app1})$
   (c) if $(\mathcal{D}'' << \mathrm{app1} << \texttt{ret}) \prec \mathcal{D}'$ and $(\mathcal{D}''' << \mathrm{app2} << \texttt{ret}) \prec \mathcal{D}''$ then $\mathcal{D}''' \prec (\mathcal{D}' << \mathrm{app})$.

The reduction property relates inputs and outputs. The reduction checker first analyzes a declaration and extracts i/o relations which are valid according to a given order. Hypothetical and parametric judgments are analyzed and checked recursively, possibly leading to parametric i/o relation. In the second step, we show that the (parametric) i/o relations from the recursive calls imply the i/o relation from the original call.

The termination checker extracts for each recursive call a termination property relating inputs of the original calls to inputs of the recursive calls possibly taking into account (parametric) i/o relations of previous calls. Note all extracted reduction properties are valid, i.e. the truth of the reduction properties does not depend on any assumptions. In the next section, we present a deductive system for reasoning about orders.

4 Reasoning about orders

In this section, we develop a deductive system to reason about orders. Arguments of an order can either be an object or it can be a lexicographic or simultaneous extension. We will concentrate on the case where arguments of the order are atomic, i.e. they are objects. However our system extends to lexicographic and simultaneous orders in a straightforward manner. An order predicate is either the $\prec$ subterm relation or the $\precsim$ subterm relation. As parametric judgements in the LF framework induce parametric orders, we allow to quantify over parameters which occur in an order relation $(\Pi x : A.P)$. In a

Pred. Context	Δ	$::=$	$\cdot \vert \Delta, P$
Order Predicate	P	$::=$	$\Pi x : A.P \vert Arg_1 \prec Arg_2 \vert Arg_1 \precsim Arg_2$
Arg	Arg	$::=$	$M \vert \{Arg_1, Arg_2\} \vert [Arg_1, Arg_2]$

first-order setting reasoning about orders is straightforward. The main task is to automate transitivity reasoning efficiently. In the higher-order framework reasoning about orders is a challenging task. First of all, we need to define an appropriate notion of higher-order subterm relations. For first-order terms it is straightforward to determine whether a term is a subterm of another. When considering higher-order terms, we need to find an appropriate interpretation for lambda-terms. For example, we want to show that $E\ x$ is a subterm of (`lam` E) when x is a newly introduced parameter. Another example is taken from the representation of first-order logic [Pfe95]. 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` A (which represents $\forall x.A$). In the informal proof we might 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 $A\,T$a subterm of `forall` A as long as there is no way to construct an object of type `i` from objects of type `o`. A term A' is less than a λ-term $(\lambda x.A)$ if there exists a parameter instantiation $\underline{a}$ for x s.t. A' is less than $[\underline{a}/x]A$. A λ-term $(\lambda x.A')$is less than a term A, if for any parameter a, $[a/x]A'$ is less than A. We refer to the head of a type $\mathbf{hd}(\Pi x_1 : A_1, \ldots, x_n : A_n : .aM_1 \ldots M_m)$ as a. If the type family a ($\mathbf{hd}$(A))is a subordinate of the type family a' ($\mathbf{hd}$(A')), i.e., a' is mutual recursive to a, but a is not mutually recursive to a', then a subterm of type A can never contain a subterm of type A'. In this case , we may instantiate a λ-term of type A with an arbitrary term M.

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. The inference system describing subterm reasoning is presented in figure 2. We assume that we always keep copies of $\lambda x.M < M'$, and $\Pi x.P$. There is also a set of inference rules to reason about $\precsim$ which are similar to the $\prec$ rules. If the rule $L \prec$ has no premises, i.e., N is a constant c with an empty spine, the hypothesis is contradictory and the conclusion $\Delta, M \prec c \longrightarrow P$ is trivially true. In general, the completeness of

$$\frac{}{\Delta, P \longrightarrow P}\ Id \qquad \frac{}{\Delta \longrightarrow M \precsim M}\ refl$$

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

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

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

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

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

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

Figure 2: Subterm Relations ($\prec$)

a logical system can be shown by cut-admissibility. It is important to recall that our assumptions are the extracted valid reduction properties, i.e., they are true without any assumptions. To show that our system is complete, it is therefore enough to prove cut-admissibility for valid order predicates.

Proving cut-admissibility for first-order subterm relations is straightforward. Reasoning about higher-order subterm relations introduces additional complexity. As λ-terms can occur on either side of the relation, this leads to non-determinism in the proof. Reasoning about λ-terms introduces new parameters. One of the consequence of this phenomenon is that the induction hypothesis used in the proof should be closed under substitution.

Theorem 2 (Cut admissibility)

1. *If* $\mathcal{D} : . \longrightarrow \sigma M \prec M'$ *and* $\mathcal{E} : \Delta, M \prec M' \longrightarrow P'$ *then* $\mathcal{F} : \Delta \longrightarrow P'$.
2. *If* $\mathcal{D} : . \longrightarrow \sigma M \precsim M'$ *and* $\mathcal{E} : \Delta, M \precsim M' \longrightarrow P'$ *then* $\mathcal{F} : \Delta \longrightarrow P'$.

The proof follows by induction on $\mathcal{E}$ and $\mathcal{D}$. Let P be the cut-predicate, i.e., either $M \prec M'$ or $M \precsim M'$. Either the order predicate P gets smaller or P stays the same and one of the derivations is strictly smaller while the other one stays the same. However, we will not be able to show cut-admissibility directly in the given calculus due to the non-deterministic choices introduced by λ-term. Consider, for example, the cut between

$$\mathcal{D} = \; . \longrightarrow \sigma\lambda x.M \prec N. \qquad\qquad \mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Delta, [\underline{a}/x]M \prec N \longrightarrow P\end{array}}{\Delta, \lambda x.M \prec N \longrightarrow P}\, LL\lambda$$

We would like to apply inversion on $\mathcal{D}$; therefore we need to consider all possible cases of previous inference steps which lead to $\mathcal{D}$. There are three possible cases we need to consider: $R <$, $RL\lambda^a$ and $RR\lambda$. Unfortunately, it is not possible to appeal to the induction hypothesis and finish the proof in the $R <$ and $RR\lambda$ case. This situation does not arise in the first order case, because all the inversion steps where deterministic. In the higher-order case we have many choices. Moreover, in the higher-order case 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 them. 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 and take apart the left hand side until all terms are normal. Then we transition to the system where variables in λ-terms are instantiated. Second, we show this 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. This way we can prove global completeness of our inference system. For a detailed discussion see [Pie00]. When extending the inference system and the proofs to lexicographical and simultaneous ordering, we might ask should the following conjecture should be true:

$$\Pi x.[(\lambda y.M)(M'x)] \prec [(\lambda y.N)(N'x)] \longrightarrow [(\lambda y.M)(\texttt{lam}\ M')] \prec [(\lambda y.N)(\texttt{lam}\ N')]$$

The problem is that simultaneous (and lexicographical) orderings introduce a disjunctive choice, i.e. to show $[(\lambda y.M)(\texttt{lam}\ M')] \prec [(\lambda y.N)(\texttt{lam}\ N')]$ we need to prove either $(\lambda y.M) \precsim (\lambda y.N)(\texttt{lam}\ N')$ and $\texttt{lam}\ M' \prec \texttt{lam}\ N'$ or $(\lambda y.M) \prec (\lambda y.N)(\texttt{lam}\ N')$ and $\texttt{lam}\ M' \precsim \texttt{lam}\ N'$. In order to be able to instantiate the Π-quantified order on the left, we need to descend to a lambda abstraction. Therefore, we first have to commit to a disjunctive choice. This will prevent us from proving the given conjecture. Therefore, from a logical perspective, this conjecture is not provable because the lambda abstraction might be buried underneath a disjunctive choice. The fact that this conjecture is not true may seem surprising. In practice we have not discovered any conjectures as the one discussed here and our system proves all previous termination properties and additional ones which involve reasoning about reduction properties.

5 Conclusion

In this paper we presented a termination and reduction checker to extend the power of the induction theorem prover to complete (or course-of-values) induction. At the heart of the termination and reduction checker lies a deductive system to reason about orders. We proved our system complete by showing cut to be admissible. The termination and reduction checker is implemented in Twelf based on the presented inference system. In the implementation we restrict the multiplicity to one. The algorithm first normalizes the orders, i.e. it unfolds all lexicographical/ simultaneous orders, then introduces all parameters originating from λ-terms. Then we instantiate possible Π-quantified predicates in the hypothesis and perform a second normalization phase until all orders are atomic and all parameters have been introduces. Finally we use transitivity reasoning and instantiate the parameters in λ-terms via unification. In practice this algorithm works on all previous examples and is able to check all new examples which require course-of-value induction.

To our knowledge this is the first method which allows reasoning about orders and gives an algorithm for termination / reduction checking for a higher-order framework. We expect our method to be applicable to other higher-order frameworks like Isabelle or λprolog. In the future, we plan to use the termination checker to generate valid induction hypothesis during the automatic inductive proof. This seems straightforward. As the user has to specify an induction order, we generate the additional hypotheses only when required. We also plan to extend the system to multi-set ordering and recursive path orderings, incorporating results about termination order for higher-order rewrite systems ([JR99, LP95]).

References

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 200.

[GBG99] Jeremy Gow, Alan Bundy, and Ian Green. Extesions to the estimation calculus. In A. Voronkov H. Ganzinger, D. McAllester, editor, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, LNAI 1705, pages 258–272, Tblisi, Georgia, 1999. Springer-Verlag.

[HHP93] 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.

[HP92] 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.

[JR99] 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.

[LP95] 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.

[Pfe95] 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.

[Pfe00] Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically.

[Pie00] Brigitte Pientka. Termination and reduction checking in the logical framework. Technical report cmu-cs-???, Carnegie Mellon University, 2000.

[RP96] 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.

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag LNCS 1421.

[vdPS95] 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.

[vR99] Femke von 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.