Verifying termination and reduction properties about higher-order logic programs

Brigitte Pientka
School of Computer Science
McGill University
Montreal, Canada

June, 2004

Abstract. We describe two checkers for verifying termination and reduction properties about higher-order logic programs. The reduction checker verifies that the result of a program execution is structurally smaller (or equal) than the inputs to the program. The termination checker guarantees that the inputs of the recursive calls are structurally smaller than the inputs of the original call taking into account reduction properties. At the heart of both checkers lies an inference system to reason about structural properties which are described by higher-order subterm relations. This approach provides a logical foundation for proving properties such as termination and reduction and factors the effort required for each one of them. Moreover, it allows the study of proof-theoretical properties, soundness and completeness and different optimizations. The termination and reduction checker are implemented as part of the *Twelf* system and has been used on a wide variety of examples, including proofs about typed assembly language and those in the area of proof-carrying code.

Keywords: Logical frameworks, termination

1. Introduction

Termination and reduction properties about programs and proofs are crucial invariants and represent an important step towards verifying correctness. Several automated methods to prove termination have been developed for first-order functional and logic programs in the past years (for example (Wal94; SSS97; EO00; AG00; JG02; NH03)). 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 (vR99)): the first approach relies on strict functionals by van de Pol (vdPS95), and the second generalizes recursive path orderings to the higher order case by Jouannaud and Rubio (JR99).

However, there are several drawbacks to applying these approaches to verify termination of higher-order logic programs. First, important structural information is lost during the translation of programs into TRS. In particular, if the termination analysis fails for the TRS, it is hard to provide feedback and re-use this failure information to point to the program location where the error occurred. Second, synthesizing a suitable measure to verify the TRS terminates may be expensive. In addition, failure to synthesize a suitable measure does not imply that the TRS is not terminating. In this paper, we develop a more direct approach of checking termination and reduction properties of higher-order logic programs and provide a logical foundation for verifying and reasoning with structural properties in general.

We focus on the higher-order logic programming language *Twelf*, a meta-language for specifying and implementing logical systems and proofs about them (Pfe91). *Twelf* extends first-order logic programming in two principal ways: First, we allow the user to define her own higher-order dependently typed data-types and support higher-order abstract syntax (PE88). Second, we not only have a static set of program clauses, but clauses may be introduced dynamically and used within a certain scope during execution.

This stands in sharp contrast to higher-order features supported in many traditional programming languages (for example (CKW93; SSS97)) and higher-order term rewriting systems (see (vdPS95; JR99)) where the use of functions is unrestricted. In particular, we can encapsulate functions within terms to later execute them. In contrast, functions in *Twelf* are used solely to support the use of higher-order data-types and model concisely many characteristics prevalent in logical systems, such as renaming and scoping of bound variables and the discharge of assumptions. These features make *Twelf* an ideal meta-language for specifying and implementing formal systems (such as type systems, operational semantics, logics etc.) and the proofs about these formal systems (such as type preservation, soundness, completeness etc.).

In this paper we describe reduction and termination checking for higher-order logic programs in *Twelf*. The principal contributions of this paper are three-fold: 1) We present a logical foundation for properties such as termination and reduction. The logical perspective allows us to draw on proof-theoretic methods to ensure soundness and completeness of the reasoning system. It also provides insights into optimizations and allows us to combine different structural orderings and reason about them. 2) We describe a syntax-directed checker for reduction properties which verifies that the output of program execution is structurally smaller or equal than the input. More generally, the reduction checker verifies relations between arguments of a well-moded predicate.

3) We present a syntax-directed checker for termination properties of higher-order logic programs. To show termination, the checker may rely on reduction properties to infer that the inputs to recursive calls are strictly smaller than the inputs to the original call. Both checkers analyze higher-order logic programs directly and provide precise error messages. The paper extends an earlier version (Pie01) and provides a detailed account of the termination and reduction analysis in the higher-order logic programming setting.

Both the termination and the reduction checker are implemented as part of the *Twelf* system and have 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 largest project so far undertaken is Crary's implementation of the typed-assembly language (TALT) (Cra03; CS03). TALT provides a foundational account of safety for a fully expressive typed assembly language. The project, a form of certified code, consists of about 30,000 lines of code and over 1,400 theorems which ultimately establish the soundness of the type-system for TALT. The termination and reduction checker in combination with the coverage checker (SP03) have been used to verify correctness of these proofs. The wide range and scope of these examples demonstrate the strength and success of this approach in practice.

The paper is organized as follows: In Section 2 we discuss an example taken from the domain of verifying the correctness of abstract machine translation. We review the background (see Section 3) and briefly discuss the implementation of the example in *Twelf*. Using this example we illustrate the basic idea of the reduction and termination checker. In Section 4 we outline the inference system for reasoning about orders and prove consistency of the system. In Section 5, we describe in detail the reduction and termination analysis of higher-order logic programs. Finally, in Section 6 we discuss related work, summarize the results and outline future work.

2. Motivating Example

In this section, we consider a typical example from verifying the correctness of abstract machines(HP92). To prove correctness, we need to show the correspondence between the high-level language and the low-level abstract machine. In other words, we can translate programs written in the high-level language into programs which run on the abstract machine. The proof is constructive and constitutes a program which translates derivations in the source language into derivations of the target language and vice versa. In this example, we consider Mini-ML as the source language, and the language of the machine instructions as the target language.

Mini-ML Syntax

$$\begin{array}{lrcl} \text{expressions} & e & ::= & x \mid \text{vl } v \mid \mathsf{z} \mid \mathsf{s}\ e \mid \mathsf{lam}\ x.e \mid e_1\ e_2 \\ \text{values} & v & ::= & x \mid \mathsf{z}^* \mid \mathsf{s}^*\ v \mid \mathsf{lam}^* x.e \end{array}$$

Abstract Machine Syntax

$$\begin{array}{lrcl} \text{instructions} & I & ::= & \text{ret } v \mid \text{ev } e \mid \text{app}_1\ v\ e \mid \text{app}_2\ v_1\ v_2 \\ \text{stack} & S & ::= & \text{nil } \mid S; \lambda v.I \end{array}$$

The Mini-ML language consists of numbers, constructed by `z` and successor `s`, lambda-abstraction and application. Evaluation rules are given via a big-step semantics. 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 other rules are straightforward.

Big step Mini-ML semantics:

$$\frac{}{\mathsf{z} \hookrightarrow \mathsf{z}^*}\ \mathit{ev\_z} \qquad \frac{e \hookrightarrow v}{\mathsf{s}\ e \hookrightarrow \mathsf{s}^*\ v}\ \mathit{ev\_s} \qquad \frac{}{\mathsf{lam}\ x.e \hookrightarrow \mathsf{lam}^* x.e}\ \mathit{ev\_lam}$$

$$\frac{}{\text{vl } v \hookrightarrow v}\ \mathit{ev\_vl} \qquad \frac{e_1 \hookrightarrow \mathsf{lam}^* x.e' \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e' \hookrightarrow v}{e_1\ e_2 \hookrightarrow v}\ \mathit{ev\_app}$$

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 $\text{app}_1\ v_1\ e_2$ and $\text{app}_2\ v_1\ v_2$. Computation in an abstract machine can be represented as a sequence of states.

Small-step transition semantics (single step):

$$\begin{array}{lcl} \mathit{t\_z} & : & S\#(\text{ev } \mathsf{z}) \longmapsto S\#(\text{ret } \mathsf{z}^*) \\ \mathit{t\_s} & : & S\#(\text{ev } (\mathsf{s}\ e)) \longmapsto (S; \lambda v.\text{ret } (\mathsf{s}^*\ v))\#(\text{ev } e) \\ \mathit{t\_lam} & : & S\#(\text{ev } \mathsf{lam}\ x.e) \longmapsto S\#(\text{ret } \mathsf{lam}^* x.e) \\ \mathit{t\_app} & : & S\#(\text{ev } e_1\ e_2) \longmapsto (S; \lambda v_1.\text{app}_1\ v_1\ e_2)\#(\text{ev } e_1) \\ \mathit{t\_app1} & : & S\#(\text{app}_1\ v_1\ e_2) \longmapsto (S; \lambda v_2.\text{app}_2\ v_1\ v_2)\#(\text{ev } e_2) \\ \mathit{t\_app2} & : & S\#(\text{app}_2\ (\mathsf{lam}^* x.e')\ v_2) \longmapsto S\#(\text{ev } [v_2/x]e') \\ \mathit{t\_vl} & : & S\#(\text{ev } (\text{vl } v)) \longmapsto S\#(\text{ret } v) \\ \mathit{t\_ret} & : & (S; \lambda v.I)\#(\text{ret } v_1) \longmapsto S\#([v_1/v]I) \end{array}$$

Each state T 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.

Multi-Step transition semantics:

$$\frac{}{T \overset{*}{\longmapsto} T}\ id \qquad\qquad \frac{T \longmapsto T_1 \quad T_1 \overset{*}{\longmapsto} T'}{T \overset{*}{\longmapsto} T'}\ @$$

Deductions of the judgment $T \overset{*}{\longmapsto} T'$ 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:

$$T_1 \longmapsto T_2 \longmapsto T_3 \longmapsto \ldots \longmapsto T_n$$

Similarly, we will mix multi-step and single-step transitions in sequences with the obvious meaning. To illustrate computation of the abstract machine, we give a sample computation sequence for computing the value of the expression $(\mathsf{lam}\ x.\mathrm{vl}\ x)\ (\mathsf{s}\ \mathsf{z})$.

$$\begin{array}{l}
S\#(\mathrm{ev}\ (\mathsf{lam}\ x.\mathrm{vl}\ x)\ (\mathsf{s}\,\mathsf{z}))\\
\mathcal{D}_1\left\{\begin{array}{l}\longmapsto (S; \lambda v_1.\mathrm{app}_1\ v_1\ (\mathsf{s}\,\mathsf{z}))\#\mathrm{ev}\ (\mathsf{lam}\ x.\mathrm{vl}\ x)\\ \longmapsto (S; \lambda v_1.\mathrm{app}_1\ v_1\ (\mathsf{s}\,\mathsf{z}))\#\mathrm{ret}\ (\mathsf{lam}^*x.\mathrm{vl}\ x)\end{array}\right.\\
\mathcal{D}_2\left\{\begin{array}{l}\longmapsto S\#\mathrm{app}_1\ (\mathsf{lam}^*x.\mathrm{vl}\ x)\ (\mathsf{s}\,\mathsf{z}) \longmapsto (S; \lambda v_2.\mathrm{app}_2\ (\mathsf{lam}^*x.\mathrm{vl}\ x)\ v_2)\#\mathrm{ev}\ (\mathsf{s}\,\mathsf{z})\\ \quad\ldots\\ \longmapsto (S; \lambda v_2.\mathrm{app}_2\ (\mathsf{lam}^*x.\mathrm{vl}\ x)\ v_2)\#\mathrm{ret}\ (\mathsf{s}^*\ \mathsf{z}^*)\end{array}\right.\\
\mathcal{D}_3\left\{\begin{array}{l}\longmapsto S\#\mathrm{app}_2\ (\mathsf{lam}^*x.\mathrm{vl}\ x)\ (\mathsf{s}^*\ \mathsf{z}^*)\\ \longmapsto S\#(\mathrm{ev}\ (\mathrm{vl}\ (\mathsf{s}^*\ \mathsf{z}^*)) \longmapsto S\#\mathrm{ret}\ (\mathsf{s}^*\ \mathsf{z}^*)\end{array}\right.
\end{array}$$

We can translate a computation sequence ($\longmapsto$) into an evaluation derivation ($\hookrightarrow$) by recursively translating each sub-sequence. In the above example, the computation sequence can be split into several consecutive sub-sequences $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$ where each of these sub-sequences corresponds to evaluation derivation $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ in the following derivation:

$$\frac{\mathsf{lam}\ x.\mathrm{vl}\ x \overset{\mathcal{P}_1}{\hookrightarrow} \mathsf{lam}^*x.\mathrm{vl}\ x \quad (\mathsf{s}\ \mathsf{z}) \overset{\mathcal{P}_2}{\hookrightarrow} (\mathsf{s}^*\ \mathsf{z}^*) \quad \mathrm{vl}\ (\mathsf{s}^*\ \mathsf{z}^*) \overset{\mathcal{P}_3}{\hookrightarrow} (\mathsf{s}^*\ \mathsf{z}^*)}{(\mathsf{lam}\ x.\mathrm{vl}\ x)\ (\mathsf{s}\ \mathsf{z}) \hookrightarrow (\mathsf{s}^*\ \mathsf{z}^*)}\ \mathit{ev\_app}$$

To show that the abstract machine works correctly, we prove that all computation sequences can be translated into evaluation derivations. To be more precise, we need to show that for all computation sequences on the abstract machine starting with the empty stack which evaluate some expression e to some value w in multiple steps, there exists an evaluation in the Mini-ML operational semantics s.t. expression e evaluates to value w.

THEOREM 1 (Soundness).
If $\mathcal{D}$:$nil\ \#(ev\ e) \overset{*}{\longmapsto} nil\ \#(ret\ w)$ *then* $\mathcal{P}$:$e \hookrightarrow w$.

This guarantees that the translation is sound. For correctness, we also need to prove that every evaluation derivation ($\hookrightarrow$) can be translated into a computation sequence ($\longmapsto$). We will concentrate on the soundness of the translation. Note that the stack grows during the computation on the abstract machine which will prevent the application of the induction hypothesis in the proof. Therefore we will prove the following generalized statement: 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 $S\#(\text{ret } v) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)$. The lemma states that a complete computation with an appropriate initial state can be translated into an evaluation followed by another complete computation. For a more detailed discussion of this example we refer to (Pfe00).

LEMMA 2. *If* $\mathcal{D} : S\#(ev\ e) \overset{*}{\longmapsto} nil\ \#(ret\ w)$ *then* $\mathcal{P} : e \hookrightarrow v$ *and* $\mathcal{D}' : S\#(ret\ v) \overset{*}{\longmapsto} nil\ \#(ret\ w)$ *and* $\mathcal{D}'$ *is smaller than* $\mathcal{D}$, *i.e.* $\mathcal{D}'$ *is a sub-sequence of* $\mathcal{D}$.

Proof: By course-of-value induction on $\mathcal{D}$. We abbreviate the extra side condition $\mathcal{D}'$ is smaller than $\mathcal{D}$ by $\mathcal{D}' \prec \mathcal{D}$. We show the case where $\mathcal{D}$ begins with *t_app*.

$$\mathcal{D} = S\#(\text{ev } e_1\ e_2) \longmapsto \underbrace{(S; \lambda v_1.\text{app}_1\ v_1\ e_2)\#(\text{ev } e_1) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)}_{\mathcal{D}_1}$$

By induction hypothesis on $\mathcal{D}_1$ there exists a value v_1 and an evaluation $\mathcal{P}_1 : e_1 \hookrightarrow v_1$ and a subcomputation

$$\mathcal{D}' : (S; \lambda v_1.\text{app}_1\ v_1\ e_2)\#(\text{ret } v_1) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w) \qquad \text{s.t. } \mathcal{D}' \prec \mathcal{D}_1$$

By inversion on the subcomputation $\mathcal{D}'$ we obtain the following computation sequence:

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

By induction hypothesis on $\mathcal{D}_2$ there exists a value v_2 and an evaluation $\mathcal{P}_2 : e_2 \hookrightarrow v_2$ and a subcomputation $\mathcal{D}'' : (S; \lambda v_2.\text{app}_2\ v_1\ v_2)\#(\text{ret } v_2) \overset{*}{\longmapsto}$

nil $\#(\text{ret } w)$ s.t. $\mathcal{D}'' \prec \mathcal{D}_2$. By inversion on the subcomputation $\mathcal{D}''$, v_1 must be of the form $\mathsf{lam}^* x.e'$ and we obtain the following computation sequence:

$$\mathcal{D}'' = (S; \lambda v_2.\text{app}_2\ (\mathsf{lam}^* x.e')\ v_2)\#(\text{ret } v_2) \longmapsto S\#(\text{app}_2\ (\mathsf{lam}^* x.e')\ v_2) \longmapsto \underbrace{S\#(\text{ev } [v_2/x]e') \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)}_{\mathcal{D}_3}$$

By induction hypothesis on $\mathcal{D}_3$ there exists a value v and an evaluation $\mathcal{P}_3$: $([v_2/x]e') \hookrightarrow v$ and a subcomputation $\mathcal{D}''' : S\#(\text{ret } v) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w)$ s.t. $\mathcal{D}''' \prec \mathcal{D}_3$. Recall, we needed to show the following two facts:

$$\mathcal{P} : e_1\ e_2 \hookrightarrow v \text{ and } \mathcal{D}''' : S\#(\text{ret } v) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } w) \text{ and } \mathcal{D}''' \prec \mathcal{D}.$$

We showed that we can derive $\mathcal{D}'''$ after several inversion steps and three applications of the induction hypothesis. We can construct $\mathcal{P}$ by using *ev_app* rule and $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ as premises.

To justify each of the IH applications we need to verify 1) if we apply the IH to $\mathcal{D}_1$ ($\mathcal{D}_2, \mathcal{D}_3$ resp.), the resulting sequence $\mathcal{D}'$ ($\mathcal{D}''$, $\mathcal{D}'''$ resp.) is shorter (reduction property) and 2) if we apply the IH to $\mathcal{D}_1$ ($\mathcal{D}_2, \mathcal{D}_3$ resp.), then $\mathcal{D}_1$ ($\mathcal{D}_2, \mathcal{D}_3$ resp.) is smaller than the original computation sequence $\mathcal{D}$ (termination property). To establish reduction and termination we need to apply transitivity reasoning. $\square$

Designing a correct translation of computation in a low-level language into evaluation of a high-level language is a non-trivial example illustrating course-of-value recursion. We recursively unfold the computation sequence of the low-level language and translate sub-sequences into evaluations. This is reflected in the soundness proof, where we needed to apply the induction hypothesis to the outcome of the previous IH application. Note that we formulated the reduction property that the output is size-decreasing explicitly in the theorem, to emphasize the necessary steps in justifying each induction hypothesis application.

In the next section, we explain basic notation and give an implementation of the proof in *Twelf* using higher-order logic programming.

3. Background

3.1. Basic Notation

Higher-order logic programming extends first-order logic programming in two ways: 1) Terms can be constructed using higher-order dependently typed data-types and are subject to higher-order unification during execution. 2) Clauses may contain implication and universal quantification. Hence clauses may be introduced dynamically during execution and used within a certain scope. Several higher-order logic programming languages exist, such as λProlog (NM88) or Isabelle (Pau86) and we expect in general that this work is also applicable to their systems with minor modifications. We focus here on the higher-order logic programming language *Twelf* which is based on the logical framework LF (HHP93).

$$\begin{array}{llcl}
\text{Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}A.K \\
\text{Types} & A & ::= & a\ M_1 \ldots M_n \mid A_1 \rightarrow A_2 \mid \Pi x{:}A_1.A_2 \\
\text{Terms} & M & ::= & c \mid x \mid \lambda x{:}A.M \mid M_1\ M_2 \\
\text{Context} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\text{Signature} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A
\end{array}$$

We will use c for term constants, a for type family constants and x for variables. Sometimes we also just use a to refer to an atomic type. $\Pi x{:}A_1.A_2$ denotes the dependent function type: the type A_2 may depend on term x of type A_1. Whenever x does not occur free in A_2 we may abbreviate $\Pi x{:}A_1.A_2$ as $A_1 \rightarrow A_2$. Types can be interpreted as formulas, such that the function arrow $A_1 \rightarrow A_2$ represents an implication and $\Pi x{:}A_1.A_2$ denotes universal quantification. Using the types-as-formulas paradigm, we can assign types a logic programming interpretation (Pfe91). Below we assume a fixed signature where the types of term constants c are specified. The free variables in a term M are provided by a context Γ. The equivalence between terms is equality modulo $\beta\eta$-conversion. We will rely on the fact that canonical (i.e. long $\beta\eta$-normal) forms of terms are computable and that equivalent terms 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 Mini-ML which was introduced in the last section. The applications and lambda-abstractions can be represented as canonical terms of type `exp`. We use higher-order abstract syntax (PE88) to represent expressions $\mathsf{lam}\ x.e$ and $e_1\ e_2$. The expression $\mathsf{lam}\ x.e$ of the Mini-ML language is represented by `lam` $\lambda x.E\ x$. Similarly the value $\mathsf{lam}^* x.e'$ is modeled as `lam*` $\lambda x.E'\ x$. The key idea is to represent the scope of the variable x in $\mathsf{lam}\ x.e$ (and by $\mathsf{lam}^* x.e'$) by λ-abstraction. Substitution is modeled via β-reduction. To improve readability we write `lam E` or `lam* E'` where we assume that `E` (or `E'` resp.) can be η-expanded. To represent the $[v_2/x]e'$ in Mini-ML, we use meta-level LF application `(E' V2)` where `E'` stands for the η-expanded expression $\lambda x.e'\ x$ and

(E' V2) will be in $\eta\beta$-normal form. Values, continuations, instructions and states are defined in a similar fashion. The evaluation judgment $e \hookrightarrow v$ is represented by the type family `eval : exp -> val -> type` in *Twelf*. Similarly, we can encode the one-step transition relation and the multi-step transition relation as judgments 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* E')
val: type.                                <- eval E2 V2
lam*:(val -> exp) -> val.                 <- eval (E' 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. Note that the head of the clause lies to the right when using $\rightarrow$ notation. 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 explicit form of the second declaration is:

```
ev_app:ΠE1:val -> exp.ΠE2:exp.ΠV:val.ΠE':val -> exp.ΠV2:val.
eval (E' V2) V -> eval E2 V2 -> eval E1 (lam* E')
                                         -> eval (app E1 E2) V.
```

3.2. Soundness proof as higher-order logic program

Twelf allows elegant encodings of source and target language and the translation of computation from one language to another. In the following, we develop the representation of the translation of computation sequences to evaluations in Mini-ML in *Twelf*. In the translation, 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 smaller than the original computation $\mathcal{D}$. A computation sequence

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

is represented in *Twelf* as `(t_app @ D1)` where `t_app` represents the first step of computation $S\#(\text{ev } e_1 e_2) \xmapsto{t\_app} (S; \lambda v.\text{app}_1\ v\ e_2)\#(\text{ev } e_1)$ while `D1` stands for $(S; \lambda v.\text{app}_1\ v\ e_2)\#(\text{ev } e_1) \xmapsto{*} \text{nil }\#(\text{ret } w)$, the tail of the computation. In other words, `(t_app @ D1)` means, we first apply the rule *t_app* followed by the tail computation sequence `D1`. `@`

represents the application of the multi-step transition rule and is used as an infix operator. An evaluation tree in the big step semantics

$$\dfrac{\begin{array}{ccc} \mathcal{P}_1 & \mathcal{P}_2 & \mathcal{P}_3 \\ e_1 \hookrightarrow \mathsf{lam}^* x.e' & e_2 \hookrightarrow v_2 & [v_2/x]e' \hookrightarrow v \end{array}}{e_1\ e_2 \hookrightarrow v}\ \mathit{ev\_app}$$

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*.

In Section 2, we proved the critical lemma which is needed for establishing soundness of the compiler: if we start in an arbitrary state $S\#(\mathrm{ev}\ e)$ with a computation $S\#(\mathrm{ev}\ e) \stackrel{*}{\longmapsto} \mathrm{nil}\ \#(\mathrm{ret}\ w)$ then there exists an intermediate state $S\#(\mathrm{ret}\ v)$ such that $e \hookrightarrow v$ in the Mini-ML semantics and $S\#(\mathrm{ret}\ v) \stackrel{*}{\longmapsto} \mathrm{nil}\ \#(\mathrm{ret}\ w)$. The translation of computation sequences to evaluation trees and tail computation sequences can be described by a meta-predicate `trans` 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)`.

$$S\#(\mathrm{ev}\ e_1\ e_2) \stackrel{\mathit{t\_app}}{\longmapsto} \underbrace{(S; \lambda v.\mathrm{app}_1\ v\ e_2)\#(\mathrm{ev}\ e_1) \stackrel{*}{\longmapsto} \mathrm{nil}\ \#(\mathrm{ret}\ w)}_{\mathtt{D1}}$$

We recursively apply the translation to `D1` and obtain `P1` which represents the evaluation starting in $e_1 \hookrightarrow v_1$ and the following tail computation sequence `(ret @ t_app1 @ D2)`:

$$\begin{array}{l} (S; \lambda v.\mathrm{app}_1\ v\ e_2)\#(\mathrm{ret}\ v_1) \stackrel{\mathit{ret}}{\longmapsto} S\#(\mathrm{app}_1\ v_1\ e_2) \stackrel{\mathit{t\_app1}}{\longmapsto} \\ \qquad\qquad\qquad\qquad \underbrace{(S; \lambda v.\mathrm{app}_2\ v_1\ v)\#(\mathrm{ev}\ e_2) \stackrel{*}{\longmapsto} \mathrm{nil}\ \#(\mathrm{ret}\ w)}_{\mathtt{D2}} \end{array}$$

By applying the translation again to `D2`, we obtain an evaluation tree for $e_2 \hookrightarrow v_2$ described by `P2` and some sequence `D''`. By inversion on `D''` we obtain some sequence `(ret @ t_app2 @ D3)` and know that v_1 is $\mathsf{lam}^* x.e'$.

$$\begin{array}{l} (S; \lambda v.\mathrm{app}_2\ (\mathsf{lam}^* x.e')\ v)\#(\mathrm{ret}\ v_2) \stackrel{\mathit{ret}}{\longmapsto} S\#(\mathrm{app}_2\ (\mathsf{lam}^* x.e')\ v_2) \stackrel{\mathit{t\_app2}}{\longmapsto} \\ \qquad\qquad\qquad\qquad \underbrace{S\#(\mathrm{ev}\ (\mathsf{lam}^* x.e')\ v_2) \stackrel{*}{\longmapsto} \mathrm{nil}\ \#(\mathrm{ret}\ w)}_{\mathtt{D3}} \end{array}$$

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 `D'''`. 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 $e_1 \hookrightarrow (\mathsf{lam}^* x.e')$, $e_2 \hookrightarrow v_2$ and $[v_2/x]e' \hookrightarrow v$ as premises. This step is represented in *Twelf* by `(ev_app P1 P2 P3)`. As a second result, we return a tail computation sequence. This is just the sequence `D'''`.

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. For a more detailed discussion of this example we refer to (Pfe00). The complete implementation can be found in the appendix.

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

First the type of the meta-predicate `trans` is defined. It has three arguments: the computation $S\#(\text{ev } E) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } W)$ 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 $S\#(\text{ret } E) \overset{*}{\longmapsto} \text{nil } \#(\text{ret } W)$ which is defined as `S # (ret V) =>* nil # (ret W)`.

The mode declaration `%mode trans +CS -E -CS'` specifies input and output relations 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 (RP96) 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. Coverage ensures that all cases in the proof have been considered (SP03). 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. This is not a limitation in our setting, as the user usually knows why a particular program or proof should terminate, if implemented correctly. In the given example, we specify that the predicate `trans` terminates in the first argument by `%terminates D (trans D P D')`. This captures the statement in the proof of the lemma where we specify the induction order by saying "By course-of-value induction on $\mathcal{D}$". For reduction checking we specify an explicit order relation between input and output elements. In the example we say `%reduces D' < D (trans D P D')`. This captures the statement "$\mathcal{D}'$ is smaller than $\mathcal{D}$, i.e. $\mathcal{D}'$ is a sub-sequence of $\mathcal{D}$" which is part of the lemma. In general, we allow atomic, lexicographic ($\{Arg_1, Arg_2\}$) or simultaneous ($[Arg_1, Arg_2]$) subterm orderings. To show that the previous predicate `trans` terminates, we show the following:

Reduction: `%reduces D' < D (trans D P D')`

if `D'''` $\prec$ `D3`, (`t_ret @ t_app2 @ D3`) $\prec$ `D2` and

(`t_ret @ t_app1 @ D2`) $\prec$ `D1` then `D'''` $\prec$ (`t_app @ D1`).

Termination: `%terminates D (trans D P D')`

1. `D1` $\prec$ (`app @ D1`)
2. if (`t_ret @ app1 @ D2`) $\prec$ `D1` then `D2` $\prec$ (`app @ D1`)
3. if (`t_ret @ app2 @ D3`) $\prec$ `D2` and (`t_ret @ app1 @ D2`) $\prec$ `D1` then `D3` $\prec$ (`app @ D1`).

$\prec$ denotes the subterm order relation. In general, we proceed for each clause in two stages to show that a given program satisfies a given reduction property pattern. First we extract a set Δ of reduction properties from the recursive calls which can be assumed and the reduction property P of the whole clause which needs to be satisfied. Second, we prove that the set Δ implies the reduction property 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 properties which are valid and a termination property P which characterizes the relation between the recursive call and the original call. Second, we prove that the set Δ implies the termination property P. In general we might have nested clauses which need to be checked recursively. Moreover, we generate parametric reduction properties for parametric sub-clauses. Note, while the first phase of analyzing higher-order logic programs and extracting a set of assumptions Δ together with the relation P may differ, the inference system to reason and infer P is implied by Δ is the same.

In Section 6 we give several other example for checking termination and reduction which illustrates the idea. In the remainder of the paper, we present a formal system which analyzes higher-order logic programs and develop a deductive system for reasoning about structural orderings.

4. A logical foundation for verifying structural properties

To describe structural properties of programs, we use higher-order subterm orderings. An order relation is either the $\prec$ subterm relation, the $\preceq$ subterm relation or structural equivalence relation $\equiv$. In addition, we allow compound orderings such as lexicographic and simultaneous orderings. We use $[O_1\ O_2]$ to denote simultaneous ordering and $\{O_1\ O_2\}$ for lexicographic ordering. A context Δ is either empty or contains valid order relations.

$$\begin{array}{lrcl} \text{Context} & \Delta & ::= & \cdot \mid \Delta, P \\ \text{Order relation} & P & ::= & O_1 \prec O_2 \mid O_1 \preceq O_2 \mid O_1 \equiv O_2 \mid \Pi x{:}A.P \\ \text{Order} & O & ::= & M \mid \{O_1, O_2\} \mid [O_1, O_2] \end{array}$$

Since higher-order logic programming admits nested clauses, which may include universal quantifiers, we also include universally quantified order relations. Order relations may therefore refer to logic variables and parameters where parameters denote variables bound by λ-abstraction or Π-quantifier. In the higher-order setting, it is crucial to be able to distinguish between parameters and logic variables. In this presentation, we use mixed-prefix context to model dependencies between logic and bound variables (Mil92). Logic variables are modeled as existentials, parameters as universals and subgoals are marked with $\exists!$. We can think of the mixed-prefix context as a stack of logic variables, parameters and subgoals. It models dependencies among variables, as well as the sequence of subgoals in a clause. Our notion of a mixed-prefix context slightly generalizes Miller's notion of a mixed prefix context since we also keep track of subgoals. This simplifies the conversion of a mixed prefix context into a well-typed ordinary LF context. An ordinary well-typed LF context can simply be obtained by dropping the quantifiers $\forall$, $\exists$ and $\exists!$.

$$\text{Mixed-prefix Context} \quad \Psi \ ::= \ \cdot \mid \Psi, \forall x{:}A \mid \Psi, \exists x{:}A \mid \exists! u{:}A$$

The substitution θ relates two mixed-prefix contexts in the following way:

$$\frac{\Psi \vdash \theta : \Psi\prime \qquad \Gamma \vdash M : \theta A}{\Psi \vdash (\theta, \exists M/x) : (\Psi\prime, \exists x : A)} \qquad \frac{\Psi \vdash \theta : \Psi\prime}{\Psi, \forall x : \theta A \vdash (\theta, \forall x/x) : (\Psi\prime, \forall x : A)}$$

$$\frac{\Psi \vdash \theta : \Psi\prime}{\Psi \vdash \theta : (\Psi\prime, \exists! u : A)} \qquad \frac{}{\Psi \vdash \cdot : \cdot}$$

In this section we develop a formal inference system to check whether a set of valid order relations implies an order relation.

4.1. First-order subterm orderings

We start by considering only first-order subterm reasoning. An order is either the $\prec$ subterm relation, the $\preceq$ subterm relation or structural equivalence relation $\equiv$. A context Δ is either empty or contains valid order relation. We use a formulation based on the sequent calculus with right and left rules for each operation on order relations. $\preceq$ is defined in terms of $\prec$ and $\equiv$. The main judgment is defined as follows:

$$\Psi; \Delta \rightarrow P \quad \Delta \text{ implies order relation } P \text{ in the context } \Psi$$

The mixed-prefix context Ψ ensures that any order relation in Δ and the order relation P are closed in Ψ. We start by giving straightforward inference rules for identity, reflexivity, and symmetry. For transitivity, we give a more algorithmic version than the more usual transitivity rule which states: if $\Psi; \Delta \longrightarrow M \prec N'$ and $\Psi; \Delta \longrightarrow N' \prec N$ then $\Psi; \Delta \longrightarrow M \prec N$ (and similar for $\equiv$ and $\preceq$). However, we can show that the given set of transitivity rules already suffices to show that general versions of transitivity are admissible.

$$\frac{}{\Psi; \Delta, P \longrightarrow P}\, id \qquad \frac{}{\Psi; \Delta \longrightarrow M \equiv M}\, ref \qquad \frac{\Psi; \Delta, M' \equiv M \longrightarrow P}{\Psi; \Delta, M \equiv M' \longrightarrow P}\, sym$$

$$\frac{\Psi; \Delta, M_2 \prec M_3 \longrightarrow M_1 \prec M_2}{\Psi; \Delta, M_2 \prec M_3 \longrightarrow M_1 \prec M_3}\, t{\prec} \qquad \frac{\Psi; \Delta, M_2 \equiv M_3 \longrightarrow M_1 \prec M_2}{\Psi; \Delta, M_2 \equiv M_3 \longrightarrow M_1 \prec M_3}\, t{\prec}{\equiv}$$

$$\frac{\Psi; \Delta, M_2 \prec M_3 \longrightarrow M_1 \equiv M_2}{\Psi; \Delta, M_2 \prec M_3 \longrightarrow M_1 \prec M_3}\, t{\equiv}{\prec} \qquad \frac{\Psi; \Delta, M_3 \equiv M_2 \longrightarrow M_1 \equiv M_3}{\Psi; \Delta, M_3 \equiv M_2 \longrightarrow M_1 \equiv M_2}\, t{\equiv}{\equiv}$$

Next, we give the remaining rules.

$\mathbf{M} \preceq \mathbf{N}$:

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

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

$\mathbf{M} \equiv \mathbf{N}$:

$$\frac{\Psi;\Delta \longrightarrow M_1 \equiv N_1 \;\ldots\; \Psi;\Delta \longrightarrow M_n \equiv N_n}{\Psi;\Delta \longrightarrow h\ M_1 \ldots M_n \equiv h\ N_1 \ldots N_n}\; R{\equiv}$$

$$\frac{\Psi;\Delta, M_1 \equiv N_1, \ldots, M_n \equiv M_n \longrightarrow P}{\Psi;\Delta, h\ M_1 \ldots M_n \equiv h\ N_1 \ldots N_n \longrightarrow P}\; L{\equiv_1}$$

$$\frac{h_1 \neq h_2}{\Psi;\Delta, h_1\ M_1 \ldots M_n \equiv h_2\ N_1 \ldots N_k \longrightarrow P}\; L{\equiv_2}$$

$\mathbf{M} \prec \mathbf{N}$:

$$\frac{\Psi;\Delta \longrightarrow M \preceq N_i}{\Psi;\Delta \longrightarrow M \prec h\ N_1 \ldots N_n}\; R{\prec_i}$$

$$\frac{\Psi;\Delta, M \preceq N_1 \longrightarrow P \;\ldots\; \Psi;\Delta, M \preceq N_n \longrightarrow P}{\Psi;\Delta, M \prec h\ N_1 \ldots N_n \longrightarrow P}\; L{\prec}$$

We write here h for the head of a term which is either a constant c or a variable (or parameter) y which is declared as universal in the mixed-prefix context $\Psi(y) = \forall y{:}A$. If the rule $L \prec$ has no premises, i.e., N is a constant c with no arguments, the hypothesis is contradictory and the conclusion $\Psi;\Delta, M \prec h \longrightarrow P$ 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 $h_1\ M_1 \ldots M_n$ can be equal to the value of $h_2\ N_1 \ldots N_k$ where h_1 and h_2 denote different constants. When reasoning about subterms, we are only interested in the syntactic structure of a term. Therefore, a term $h_1\ M_1 \ldots M_n$ can never be structurally equivalent to $h_2\ N_1 \ldots N_k$, if $h_2 \neq h_1$. If $h_1\ M_1 \ldots M_n \equiv h_2\ N_1 \ldots N_k$ 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 `trans` predicate (see p. 10). One of the claims we need to prove during termination checking is the following:

$(\texttt{t\_ret @ app1 @ D2}) \prec \texttt{D1} \longrightarrow \texttt{D2} \prec (\texttt{app @ D1})$

We omit here the mixed-prefix context to denote that `D1`, `D2` are logic variables. The proof written in a bottom-up linear notation.

$$
\begin{array}{ll}
(\texttt{t\_ret @ t\_app1 @ D2}) \prec \texttt{D1} \longrightarrow \texttt{D2} \equiv \texttt{D2} & \mathit{ref} \\
(\texttt{t\_ret @ t\_app1 @ D2}) \prec \texttt{D1} \longrightarrow \texttt{D2} \preceq \texttt{D2} & R{\preceq_2} \\
(\texttt{t\_ret @ t\_app1 @ D2}) \prec \texttt{D1} \longrightarrow \texttt{D2} \prec (\texttt{t\_app1 @ D2}) & R{\prec_2} \\
(\texttt{t\_ret @ t\_app1 @ D2}) \prec \texttt{D1} \longrightarrow \texttt{D2} \preceq (\texttt{t\_app1 @ D2}) & R{\preceq_1} \\
(\texttt{t\_ret @ t\_app1 @ D2}) \prec \texttt{D1} \longrightarrow \texttt{D2} \prec (\texttt{t\_ret @ t\_app1 @ D2}) & R{\prec_2} \\
(\texttt{t\_ret @ t\_app1 @ D2}) \prec \texttt{D1} \longrightarrow \texttt{D2} \prec \texttt{D1} & t{\prec} \\
(\texttt{t\_ret @ t\_app1 @ D2}) \prec \texttt{D1} \longrightarrow \texttt{D2} \preceq \texttt{D1} & R{\preceq_1} \\
(\texttt{t\_ret @ t\_app1 @ D2}) \prec \texttt{D1} \longrightarrow \texttt{D2} \prec (\texttt{t\_app @ D1}) & R{\prec_2}
\end{array}
$$

4.2. Lexicographic subterm orderings

We can extend the system with rules for lexicographic orderings by defining left and right rules. 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.

Lexicographic ordering: $\prec$

$$\frac{\Psi; \Delta \longrightarrow O_1 \prec {O_1}'}{\Psi; \Delta \longrightarrow \{O_1, O_2\} \prec \{{O_1}', {O_2}'\}}\ RLex{\prec_1}$$

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

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

Lexicographic ordering: $\equiv$

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

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

Lexicographic ordering: $\preceq$

$$\frac{\Psi; \Delta \longrightarrow \{O_1, O_2\} \prec \{{O_1}', {O_2}'\}}{\Psi; \Delta \longrightarrow \{O_1, O_2\} \preceq \{{O_1}', {O_2}'\}}\ RLex{\preceq_1}$$

$$\frac{\Psi; \Delta \longrightarrow \{O_1, O_2\} \equiv \{{O_1}', {O_2}'\}}{\Psi; \Delta \longrightarrow \{O_1, O_2\} \preceq \{{O_1}', {O_2}'\}}\ RLex{\preceq_2}$$

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

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.

4.3. Higher-order subterm ordering

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 universally quantified 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 $E\ y$ is a subterm of `lam` $\lambda x.E\,x$ where y is a parameter. In the informal proof we might count the number of constructors and consider $E\ y$ an instance of $\lambda x.E\,x$. Therefore we consider a term M a subterm of $\lambda x.N$ if there exists a parameter instantiation $\underline{y}$ for x s.t. M is smaller than $[\underline{y}/x]N$. We will use the convention that y will represent a new parameter, while $\underline{y}$ stands for an already defined parameter which is used for instantiation. 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. Furthermore, we assume all terms are η-expanded.

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` $\lambda x.A\,x$ (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`.

To consider also mutual recursive type families, we define the notion of subordination. Let $\mathbf{hd}(A)$ denote the head of a clause A. A type family a is a subordinate to a type family a' ($a \trianglelefteq^* a'$) whenever a canonical term $M{:}A$ with $\mathbf{hd}(A) = a$ may be used in constructing a canonical term $N{:}B$ with $\mathbf{hd}(B) = a'$. If additionally $a' \trianglelefteq^* a$, we say that a, a' are mutually recursive. We write $a \triangleleft^* a'$ if a is a subordinate to a', but not mutually recursive. Subordination of type families is the transitive closure of the immediate subordination relation ($a \trianglelefteq^* a'$) which can be directly read off the signature. If the type family a ($\mathbf{hd}(A)$) is a subordinate of the type family a' ($\mathbf{hd}(A')$), then a canonical subterm of type A can never contain a subterm of type A'. Therefore, a term M is considered smaller than a λ-term ($\lambda x{:}A.N$) if there exists a arbitrary instantiation T for x s.t. M is smaller than $[T/x]N$ and the type of T is a subordinate to N.

If the type family a ($\mathbf{hd}(A)$) is *not* a subordinate of the type family a' ($\mathbf{hd}(A')$), then M is only considered smaller than $\lambda x.N$ if there exists a parameter y such that $[\underline{y}/x]N$ is smaller than M. For a more detailed development of subordination we refer to R. Virga's PhD thesis (Vir99). Next, we present the higher-order extensions.

λ-abstraction: introduce a new parameter y

$$\frac{\Psi, \forall y{:}A; \Delta \longrightarrow [y/x]M \equiv [y/x]N}{\Psi; \Delta \longrightarrow \lambda x{:}A.M \equiv \lambda x.N}\; R{\equiv}\,\lambda \qquad \frac{\Psi, \forall y{:}A; \Delta \longrightarrow [y/x]M \prec N}{\Psi; \Delta \longrightarrow \lambda x{:}A.M \prec N}\; RL{\prec}\,\lambda$$

$$\frac{\Psi, \forall y{:}A; \Delta,\; M \prec [y/x]N \longrightarrow P}{\Psi; \Delta, M \prec \lambda x{:}A.N \longrightarrow P}\; LR{\prec}\,\lambda \qquad \frac{\Psi, \forall y{:}A; \Delta \longrightarrow [y/x]M \preceq N}{\Psi; \Delta \longrightarrow \lambda x{:}A.M \preceq N}\; RL{\preceq}\,\lambda$$

$$\frac{\Psi, \forall y{:}A; \Delta,\; M \preceq [y/x]N \longrightarrow P}{\Psi; \Delta, M \preceq \lambda x{:}A.N \longrightarrow P}\; LR{\preceq}\,\lambda$$

$$\frac{}{\Psi; \Delta, \lambda x{:}A.M \equiv h\; N_1 \dots N_n \longrightarrow P}\; L{\equiv_3}$$

λ-abstraction: instantiate with parameter $\underline{y}$ such that $\Psi(y) = \forall y{:}A$

$$\frac{\Psi; \Delta, [\underline{y}/x]M \equiv [\underline{y}/x]N \longrightarrow P}{\Psi; \Delta, \lambda x{:}A.M \equiv \lambda x.N \longrightarrow P}\; L{\equiv}\,\lambda \qquad \frac{\Psi; \Delta,\; [\underline{y}/x]M \prec N \longrightarrow P}{\Psi; \Delta, \lambda x{:}A.M \prec N \longrightarrow P}\; LL{\prec}\,\lambda$$

$$\frac{\Psi; \Delta \longrightarrow M \prec [\underline{y}/x]N}{\Psi; \Delta \longrightarrow M \prec \lambda x{:}A.N}\; RR{\prec}\,\lambda \qquad \frac{\Psi; \Delta,\; [\underline{y}/x]M \preceq N \longrightarrow P}{\Psi; \Delta, \lambda x{:}A.M \preceq N \longrightarrow P}\; LL{\preceq}\,\lambda$$

$$\frac{\Psi; \Delta \longrightarrow M \preceq [\underline{y}/x]N}{\Psi; \Delta \longrightarrow M \preceq \lambda x{:}A.N}\; RR{\preceq}\,\lambda$$

Pi-Quantifier

$$\frac{\Psi, \forall y{:}A; \Delta \longrightarrow [y/x]P}{\Psi; \Delta \longrightarrow \Pi x{:}A.P}\ R\Pi \qquad \frac{\Psi; \Delta, [\underline{y}/x]P \longrightarrow P' \quad \Psi(y) = \forall y{:}A}{\Psi; \Delta, \Pi x{:}A.P \longrightarrow P'}\ L\Pi$$

Reasoning about λ-terms cannot be solely based $\prec$ and $\equiv$, as neither $[y/x]M \equiv \lambda x{:}A.M$ nor $[y/x]M \prec \lambda x{:}A.M$ is true. Therefore, we introduce a set of inference rules to reason about $\preceq$ which are similar to the $\prec$ rules. As we potentially need different instantiations of the relation $\lambda x{:}A.M \prec N$ when reading the inference rules bottom-up, we need to copy $\lambda x{:}A.M \prec N$ 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 two rules $R\Pi$ and $L\Pi$. In the $R\Pi$ rule we replace x with a new parameter y and add $\forall y{:}A$ to the mixed-prefix context Ψ. In the rule $L\Pi$, we instantiate x with an existing parameter y where $\Psi(y) = \forall y{:}A$. Similar to instantiations of $\lambda x{:}A.M \prec N$, 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. Finally, it is an important step towards achieving a practical implementation of the presented system.

4.4. Proof-theoretic properties

In general, the consistency of a logical system can be shown by cut-admissibility.

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

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 properties which have already been established. Hence, the reduction properties 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 3 (Cut-admissibility).

1. *If* $\mathcal{D} : \Psi; . \Longrightarrow M \equiv M'$ *and* $\mathcal{P} : \Psi; \Delta, M \equiv M' \Longrightarrow P'$ *then* $\mathcal{F} : \Psi; \Delta \Longrightarrow P'$.
2. *If* $\mathcal{D} : \Psi, \forall y_1{:}A_1, \ldots, \forall y_n{:}A_n; . \Longrightarrow \sigma M \prec M'$ *and* $\mathcal{P} : \Psi; \Delta, \lambda x_n{:}A_n, \ldots, \lambda x_1{:}A_1.M \prec M' \Longrightarrow P'$ *then* $\mathcal{F} : \Psi; \Delta \Longrightarrow P'$.
3. *If* $\mathcal{D} : \Psi, \forall y_1{:}A_1, \ldots, \forall y_n{:}A_n; . \Longrightarrow \sigma M \preceq M'$ *and* $\mathcal{P} : \Psi; \Delta, \lambda x_n{:}A_n, \ldots, \lambda x_1{:}A_1.M \preceq M' \Longrightarrow P'$ *then* $\mathcal{F} : \Psi; \Delta \Longrightarrow P'$.

The substitution σ maps free variables $x_1, \ldots, x_n$ to new parameters $y_1, \ldots, y_n$. $\sigma \circ [y/x]$ represents the extension of σ which the substitution $[y/x]$ where y is a new parameter. In general, we allow the cut between $\sigma M \prec N$ and $\lambda x_n{:}A_n, \ldots, \lambda x_1{:}A_1.M \prec N$, where M is the suffix of $\lambda x_n{:}A_n, \ldots, \lambda x_1{:}A_1.M$ and σ is a substitution which maps free variables $x_1, \ldots, x_n$ in M to new parameters $y_1, \ldots, y_n$. For example, $[y_1/x_1](\lambda x_2.x_1 M x_2)$ represents the (partial) instantiation of the term $\lambda x_1{:}A_1 \lambda x_2{:}A_2.x_1 M x_2$ with new parameters and $\lambda x_2{:}A_2.x_1 M x_2$ is a suffix of $\lambda x_1{:}A_1 \lambda x_2{:}A_2.x_1 M x_2$.

The proof follows by induction on $\mathcal{P}$ and $\mathcal{D}$. Let P be the cut-predicate, i.e., either $M \prec M'$ or $M \preceq M'$ or $\Pi x.P$. Either the order relation 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 the rules for λ-terms. Consider, for example, the cut between

$$\mathcal{D} = \cfrac{\overset{\mathcal{D}_1}{\Psi, \forall y{:}A\,; . \longrightarrow \sigma \circ [y/x] M \prec N}}{\Psi;\ . \longrightarrow \sigma \lambda x{:}A.M \prec N} \qquad \mathcal{P} = \overset{\mathcal{P}_1}{\Psi; \Delta, \lambda x{:}A.M \prec N \longrightarrow P}$$

We would like to apply inversion on $\mathcal{P}$ therefore we need to consider all possible cases of previous inference steps which lead to $\mathcal{P}$. There are four possible cases we need to consider: $L{\prec}$, $LR{\prec}\lambda$, $LL{\prec}\lambda$ and transitivity. 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 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 $\lambda x{:}A.M \prec N$, if N is of base type.

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 atomic. 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. 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 $\mathcal{P}$. More precisely, we appeal to the induction hypothesis either with a strictly smaller order relation P or P stays the same and one of the derivations is strictly smaller while the other one stays the same. This way we can prove global completeness of our inference system.

Note, R is of base type and cannot be a λ-abstraction, when $\overrightarrow{x}$ is empty. We now restate the theorem already stated earlier.

THEOREM 4 (Cut-admissibility).

1. *If* $\mathcal{D} :\ \Psi; . \Longrightarrow M \equiv M'$ *and* $\mathcal{P} : \Psi; \Delta, M \equiv M' \Longrightarrow P'$
   then $\mathcal{F} : \Psi; \Delta \Longrightarrow P'$.

2. *If* $\mathcal{D} :\ \Psi, \forall y_1{:}A_1, \ldots, \forall y_n{:}A_n; . \Longrightarrow \sigma M \prec M'$ *and*
   $\mathcal{P} : \Psi; \Delta, \lambda x_n{:}A_n, \ldots, \lambda x_1{:}A_1.M \prec M' \Longrightarrow P'$
   then $\mathcal{F} : \Psi; \Delta \Longrightarrow P'$.

3. *If* $\mathcal{D} :\ \Psi, \forall y_1{:}A_1, \ldots, \forall y_n{:}A_n; . \Longrightarrow \sigma M \preceq M'$ *and*
   $\mathcal{P} : \Psi; \Delta, \lambda x_n{:}A_n, \ldots, \lambda x_1{:}A_1.M \preceq M' \Longrightarrow P'$
   then $\mathcal{F} : \Psi; \Delta \Longrightarrow P'$.

Proof: The proof follows by induction on $\mathcal{P}$ and $\mathcal{D}$. Let P be the cut-predicate, i.e., either $M \prec M'$ or $M \preceq M'$ or $\Pi x{:}A.P$. Either the order relation P gets smaller or P stays the same and one of the derivations is strictly smaller while the other one stays the same. The order relation $M \prec N$ is smaller than $M \prec N'$ iff N is smaller than N' or $M' \prec N$ is smaller than $M \prec N$ iff M' is smaller than M. $\square$

Using the cut-admissibility theorem, cut-elimination elimination follows immediately. Therefore, our inference system is consistent. In other words, not every termination order has a proof. The inference system presented implicitly gives us a bottom-up search procedure to prove that a given termination ordering assuming a set of reduction properties. The admissibility of cut, implies that anything which should be derivable from our reduction properties in Δ, can be derived without any need for additional lemmas. In other words, adding the cut-rule does not yield a more powerful engine to prove termination and everything needed to prove termination of a clause should be contained within this clause.

5. Termination and reduction checking

5.1. Analyzing reduction properties

In Section 3.2 we sketched the analysis of higher-order logic programs for proving reduction. In this section, we develop a syntax-directed inference system to extract conditions for reduction. Each program clause is analyzed separately. As mentioned earlier, the reduction analysis is complicated by the fact that higher-order logic programming allows implications and universal quantifiers in subgoals. Therefore, we need to recursively check implications which may occur as subgoals assuming all valid reduction properties of the surrounding subgoals. In addition, we need to distinguish between variables which can be instantiated, and variables which introduce a new parameter, depending on where the Π-quantifier occurs in the implication. This motivates the two different procedures, one for checking that a reduction property is satisfied, and another one for collecting valid reduction properties. Next, we give the judgment for collecting valid reduction properties.

$$\Psi \stackrel{c}{\Longrightarrow} A/R$$

Given a mixed prefix context Ψ, R is the valid reduction property associated with the subgoal A. In the first-order setting where all sub-goals are atomic, we can always directly analyze the sub-goal and extract a reduction property. In the higher-order setting however, subgoals may be more complex and can be of the form

$$\Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.B_k \to \ldots \to B_1 \to B.$$

The reduction property associated with this subgoal is in essence the reduction property associated with B. However, we must ensure that the reduction property does not extrude its scope. By invariant, the valid reduction property R and the goal A are closed under the context Ψ. Hence when synthesizing valid reduction properties of a universally quantified subgoal $\Pi x{:}A_1.A_2$ (see $s\Pi$), we return a reduction predicate $\Pi x{:}A_1.R$ where all free occurrences of $x{:}A_1$ are bound by a universal quantifier Π.

$$\frac{\Psi, \forall x{:}A_1 \overset{c}{\Longrightarrow} A_2/R}{\Psi \overset{c}{\Longrightarrow} \Pi x{:}A_1.A_2/\Pi x{:}A_1.R}\, s\Pi \qquad \frac{\Psi \overset{c}{\Longrightarrow} A_2/R}{\Psi \overset{c}{\Longrightarrow} (A_1 \to A_2)/R}\, s \to$$

$$\frac{}{\Psi \overset{c}{\Longrightarrow} c\, M_1\ \ldots M_n/\mathsf{redOrder}(c\, M_1\ \ldots M_n)}\, satom$$

We use $\mathsf{redOrder}\ c; M_1 \ldots M_n$ in the *satom* rule to describe the retrieval of the actual ordering according to the reduction order specified by the user. In the example above the reduction order was specified by `%reduces D > D' (trans D E D')`. $\mathsf{redOrder}$ (`trans D1 P1 (t_ret @ t_app1 @ D2)`) will then return the reduction ordering `(t_ret @ t_app1 @ D2)`$\prec$ `D1`.

Finally, we give the judgments for checking that a clause satisfies a specified reduction property.

$\Psi \mid \Delta \overset{rc}{\Longrightarrow} A$	check clauses A
$\Psi \mid \Delta \overset{ri}{\Longrightarrow} (A_2, A_1)$	check subgoal A_1 with respect to clause A_2
$\Psi \mid \Delta \overset{rg}{\Longrightarrow} A$	check goal A

$\Psi \mid \Delta \overset{rc}{\Longrightarrow} A$ means "assuming a set Δ of valid reduction properties and a clause A, we check that A satisfies some specified reduction relation and Δ and A are closed under Ψ". If A is atomic, then we prove that the assumptions in Δ imply the reduction ordering corresponding to A using the inference system given in Section 4. To show that the clause $A_1 \to A_2$ satisfies some reduction property in Δ (see rule $rc \to$), we assume the reduction property corresponding to A_1 and show that the clause A_2 satisfies the reduction property in (Δ, R). In addition, we need to recursively check that all subgoals are in fact reducing (third premise of the rule $rc \to$).

Clause

$$\frac{\Psi; \Delta \longrightarrow \mathsf{redOrder}(aM_1 \dots M_n)}{\Psi|\Delta \overset{rc}{\Longrightarrow} aM_1 \dots M_n}\ rc\_atom \qquad \frac{\Psi, \exists x{:}A_1|\Delta \overset{rc}{\Longrightarrow} A_2}{\Psi|\Delta \overset{rc}{\Longrightarrow} \Pi x{:}A_1.A_2}\ rc\Pi$$

$$\frac{\Psi \to A_1/R_1 \quad \Psi|\Delta, R_1 \overset{rc}{\Longrightarrow} A_2 \quad \Psi|\Delta, R_1 \overset{ri}{\Longrightarrow} (A_2, A_1)}{\Psi|\Delta \overset{rc}{\Longrightarrow} A_1 \to A_2}\ rc \to$$

$\Psi|\Delta \overset{ri}{\Longrightarrow} (A_2, A_1)$ means "assuming a set Δ of valid reduction properties and additional valid reduction properties from the clause A_2, we check that the goal A_1 satisfies some specified reduction property and Δ, A_2 and A_1 are closed under Ψ." To check nested implications, we are allowed to collect and then assume valid reduction properties of the clause A_2 (see rule $ri \to$). Once A_2 is atomic, we proceed to check that A_1 is reducing assuming all the valid reduction properties in Δ.

Implications

$$\frac{\Psi|\Delta \overset{rg}{\Longrightarrow} A}{\Psi|\Delta \overset{ri}{\Longrightarrow} (aM_1 \dots M_n, A)}\ ri\_atom \qquad \frac{\Psi, \exists x{:}A_1|\Delta \overset{ri}{\Longrightarrow} (A_2, A)}{\Psi|\Delta \overset{ri}{\Longrightarrow} (\Pi x{:}A_1.A_2, A)}\ ri\Pi$$

$$\frac{\Psi \to A_1/R_1 \quad \Psi|\Delta, R_1 \overset{ri}{\Longrightarrow} (A_2, A)}{\Psi|\Delta \overset{ri}{\Longrightarrow} (A_1 \to A_2, A)}\ ri \to$$

$\Psi|\Delta \overset{rg}{\Longrightarrow} A$ means "assuming a set Δ of valid reduction properties and a goal A, we check that A satisfies some specified reduction relation and Δ and A are closed under Ψ". Note that A may contain implications which need to be checked recursively (see rule $rg \to$). When the goal A is atomic, checking a specified reduction property is trivially true.

Goals

$$\frac{}{\Psi|\Delta \overset{rg}{\Longrightarrow} aM_1 \dots M_n}\ rg\_atom \qquad \frac{\Psi, \forall x{:}A_1|\Delta \overset{rg}{\Longrightarrow} A_2}{\Psi|\Delta \overset{rg}{\Longrightarrow} \Pi x{:}A_1.A_2}\ rg\Pi$$

$$\frac{\Psi|\Delta \overset{rc}{\Longrightarrow} A_1 \quad \Psi|\Delta \overset{rg}{\Longrightarrow} A_2}{\Psi|\Delta \overset{rg}{\Longrightarrow} A_1 \to A_2}\ rg \to$$

5.2. Analyzing termination properties

For verifying termination, we need to compare all recursive calls with the original call, and ensure that inputs to the recursive call are smaller than the original call. This termination analysis needs to take into account the underlying operational semantics. Assume we have a clause $\Pi x{:}B_n \ldots \Pi x{:}B_1.A_k \rightarrow \ldots A_1 \rightarrow A$. Recall that $\rightarrow$ is right-associative and the head of the clause can be found in the innermost implication. To show that subgoal A_i is smaller than the head A, we are only allowed to use valid properties of subgoals A_j where $j < i$. Since subgoals may contain implications, we need to recursively check the subgoals themselves for termination. This makes the termination analysis non-trivial. For simplicity, we will concentrate here on the case where we have no mutual recursion. A discussion of how to handle mutual recursion in termination proofs can be found in (RP96) and can be applied also in this setting. The rules of inference for termination analysis uses three judgments:

$(\Psi_0; \Delta) \vert \Psi \xrightarrow{tc} A$	analyze clause A
$\Psi \vert \Delta \xrightarrow{ti} a\ M_1 \ldots M_n \mid \Psi'$	compare head $a\ M_1 \ldots M_n$ with all subgoals in Ψ'
$\Psi \vert \Delta \xrightarrow{tg} a\ M_1 \ldots M_n \gg A_1$	compare head $a\ M_1 \ldots M_n$ with subgoal A_1

$(\Psi_0; \Delta) \mid \Psi \xrightarrow{tc} A$ means assuming a set Δ of valid orderings (reduction properties), we check that a clause A satisfies some specified termination ordering with respect to Ψ. In addition Δ and A are closed under Ψ_0, Ψ. Note that Ψ can be view as the stack of subgoals which is built up during the $\xrightarrow{tc}$ rules. Once we reach the head a of the clause, we transition to $\xrightarrow{ti}$ and compare each subgoal with the head calling the judgment $\xrightarrow{tg}$. After checking termination of subgoal A, we may collect valid reduction properties associated with A which may be used to check termination of subsequent subgoal which are listed in Ψ'. $\Psi \vert \Delta \xrightarrow{tg} a\ M_1 \ldots M_n \gg A_1$ means assuming a set Δ of valid orderings, we check that the subgoal A_1 is smaller than the head a. Since A_1 may contain sub-clauses, we need to recursively check them for termination (see $tc \rightarrow$). Once A_1 is atomic (see rule tg_atom), we show that the termination order associated with the subgoal is smaller than the termination order associated with the clause head using the inference system presented in Section 4.

Build up stack of subgoals

$$\frac{(\Psi_0, \Psi); \Delta \xrightarrow{ti} a\ M_1 \dots M_n \mid \Psi}{(\Psi_0; \Delta) \mid \Psi \xrightarrow{tc} a\ M_1 \dots M_n}\ tc\_atom \qquad \frac{(\Psi_0; \Delta) \mid \Psi, \exists x{:}A_1 \xrightarrow{tc} A_2}{(\Psi_0; \Delta) \mid \Psi \xrightarrow{tc} \Pi x{:}A_1.A_2}\ tc\Pi$$

$$\frac{(\Psi_0; \Delta) \mid \Psi, \exists! u{:}A_1 \xrightarrow{tc} A_2}{(\Psi_0; \Delta) \mid \Psi \xrightarrow{tc} A_1 \rightarrow A_2}\ tc \rightarrow$$

Compare head a to all subgoals in Ψ'

$$\frac{\Psi \overset{c}{\Longrightarrow} A/R \qquad \begin{array}{c}\Psi; \Delta \xrightarrow{tg} a\ M_1 \dots M_n \gg A \\ \Psi; \Delta, R \xrightarrow{ti} a\ M_1 \dots M_n \mid \Psi'\end{array}}{\Psi; \Delta \xrightarrow{ti} a\ M_1 \dots M_n \mid (\Psi', \exists! u{:}A)}\ ti\_sg$$

$$\frac{\Psi; \Delta \xrightarrow{ti} a\ M_1 \dots M_n \mid \Psi'}{\Psi; \Delta \xrightarrow{ti} a\ M_1 \dots M_n \mid (\Psi', \forall x{:}A)}\ ti\forall \qquad \frac{}{\Psi; \Delta \xrightarrow{ti} a\ M_1 \dots M_n \mid \cdot}\ ti\_empty$$

$$\frac{\Psi; \Delta \xrightarrow{ti} a\ M_1 \dots M_n \mid \Psi'}{\Psi; \Delta \xrightarrow{ti} a\ M_1 \dots M_n \mid (\Psi', \exists x{:}A)}\ ti\exists$$

Compare head a with subgoal A

$$\frac{\Psi; \Delta \longrightarrow \mathsf{termOrder}(a\ N_1 \dots N_n) \prec \mathsf{termOrder}(a\ M_1 \dots M_n)}{\Psi; \Delta \xrightarrow{tg} a\ M_1 \dots M_n \gg a N_1 \dots N_n}\ tg\_atom$$

$$\frac{a' \text{must be terminating}}{\Psi | \Delta \xrightarrow{tg} a\ M_1 \dots M_n \gg a' N_1 \dots N_m}\ tg\_atom'$$

$$\frac{(\Psi, \forall x{:}A_1); \Delta \xrightarrow{tg} a\ M_1 \dots M_n \gg A_2}{\Psi; \Delta \xrightarrow{tg} a\ M_1 \dots M_n \gg \Pi x{:}A_1.A_2}\ tg\Pi$$

$$\frac{(\Psi; \Delta) \mid \cdot \xrightarrow{tc} A_1 \quad \Psi; \Delta \xrightarrow{tg} a\ M_1 \dots M_n \gg A_2}{\Psi; \Delta \xrightarrow{tg} a\ M_1 \dots M_n \gg (A_1 \rightarrow A_2)}\ tg \rightarrow$$

The function termOrder extracts the termination argument according to the termination specification. In the example above, the termination order was given by `%terminates D (trans D P D')`. When analyzing the recursive calls, termOrder will always extract the first argument of the recursive calls. In rule *tg_atom*$'$, we enforce that any auxiliary predicate a' which may be used in the body of the predicate a, must also be terminating.

6. Related work and discussion

Most work in automating termination proofs has focused on languages with first-order data-types. The most general method for synthesizing termination orders for a given term rewriting system is by Arts and Giesl (AG00) and much recent work has built on this approach (EO00; JG02; NH03). They first analyze the term rewriting system and construct dependency pairs. A dependency pair consists of left-hand sides of a term rewrite rule and subterms of the right-hand side that may possibly start a new rewriting. To prove termination, we show that there is not an infinite chain of dependency pairs. Dependency pairs restrict the examination to subterms that can possibly reduced further. The reduction properties in our approach play a similar role. They allow us to trace input arguments throughout a clause and reason about input-output flow within a predicate. 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. As a result constructors and functions are indistinguishable. One of the consequences is that proving termination of the TRS often requires more complicated orders. The first example is taken from the arithmetic.

```
%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).
```

Using logic programming we implement a straightforward version of `minus` and the quotient predicate `quot` where `z` represents zero and `s` the successor constructor. 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 relation `Z <= X`. When we prove termination of `quot`, we can assume the reduction relation $X' \preceq X$. As the reduction relation $X' \preceq X$ 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 for proving the corresponding term rewrite system need recursive path ordering.

The second example is an algorithm to compute the negation normal form of a first-order logical formula and uses higher-order functions.

```
%mode nnf +A -A'                   %mode nnf' +A -A'
n1: nnf (not A) A'                 n1':nnf' (not A) A'
    <- nnf' A A'.                      <- nnf A A'.
n2: nnf (A and B) (A' and B')      n2':nnf' (A and B) (A' or B')
    <- nnf A A'                        <- nnf' A A'
    <- nnf B B'.                       <- nnf' B B'.
n3: nnf (forall A) (forall A')     n3':nnf' (forall A) (exists A')
    <- {x:i} nnf (A x) (A' x).         <- {x:i} nnf' (A x) (A' x).
```

We can implement an algorithm for negation normal form using two mutual recursive predicates. Termination of this algorithm can be proved based on subterm ordering, while other formalizations which do not exploit mutual recursion and verify a higher-order term rewriting system require more complicated orders like recursive path orderings (see for example (LP95)).

To show termination in higher-order simply-typed term rewriting systems (HTRS) 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 generalization of recursive path orderings to the higher order case by Jouannaud and Rubio (JR99).

Although some of the underlying ideas in higher-order term rewrite systems (HTRS) are shared with the logical framework, there are mainly two principal differences: First, all arguments of predicate are in canonical form and therefore are terminating. This is only possible since we restrict the use of λ-abstraction to higher-order data-type definitions. This additional restriction simplifies reduction and termination analysis in the logical framework. On the other hand, the higher-order logic programming interpretation of the logical framework LF allows subgoals which may contain implications and universal quantifiers. This makes termination and reduction analysis non-trivial. The translation of clauses with nested implications and quantifiers seems difficult, since they have in general no counterpart in HTRS.

One approach which analyzes logic programs directly has been developed by Plümer (Plü90). This approach has been refined during the last decade and is for example being used in the typed logic programming language Mercury (SSS97; NL97). During the first phase, we assign an integer to each procedure, giving the upper bound on the difference between the total size of the output argument and the total size of the input arguments. This involves producing and solving a set of linear inequalities. This first phase, corresponds to our reduction checking. Second, the solution is used to check that in every cycle in the program's call graph, the input arguments decrease in size. Although this approach works well for Prolog programs, transferring this approach to the higher-order setting is not straightforward. First, it is not obvious how to handle and reason about higher-order data-types and second, it is not obvious how to build and check a call graph in the higher-order setting since we have parametric and hypothetical subgoals. Hence, we found it important to clarify the higher-order issues by developing a declarative logical foundation for termination and reduction analysis.

Other termination methods in first-order logic programming have relied on a semantic approach (Bau92; CT99), exploring the fact that first-order logic programming has a simple model-theoretic or fixed-point semantics and relying on the fact that first-order unification is decidable. In higher-order logic programming no simple declarative semantics exist. This is due to the fact that higher-order unification is undecidable in general and we have to consider subgoals which may contain universal quantifiers and nested implications. Hence transferring the semantic approaches to termination to the higher-order setting is difficult.

Many research efforts in first-order logic programming have focused powerful engines which synthesize termination orders. The main motivation behind this approach is that it may be too burdensome to specify termination orders, the users implementing the program may be different from the people verifying properties such as termination. However, it is important to keep in mind that the application of *Twelf* lies in specifying and implementing formal systems and the proofs about them. Other logical frameworks similar to *Twelf* are for example λProlog (NM88) or Isabelle (Pau86). In this domain, the user typically has thought extensively about its formal system and has a clear idea how a particular proof (such as type preservation, or soundness and completeness of transformation) proceeds and what the underlying induction ordering is. Moreover, the formal system will be developed together with a proof about it. Hence, requiring the user to specify the induction (termination) ordering is not a burden to the user but just provides a means of expressing it and checking the intuition. Similar to the approach by McAllester *et. al* (MA96), we impose a discipline on the programmer. This forces the programmer sometimes to define restricted versions of predicates which are always reducing its output. While the approach of McAllester *et. al* synthesizes a valid termination ordering based on subterm ordering, we require the user to specify a termination order.

Other recent approaches for checking termination properties have pushed towards extending type-checking. Xi (Xi01) uses dependent types to express size information of terms for the language DML (dependently typed ML). It is the only other work we know of which deals with a dependently typed language. His approach is complementary to our analysis of structural run-time properties and it would be interesting to combine both. Abel and Altenkirch (Abe00; AA02) extend the type system for a small functional language to also verify termination based on first-order subterm orderings. Although similar in spirit, our approach is more ambitious and more general, since we are able to handle more complex orderings such higher-order orderings and compound orderings like lexicographic orderings.

Finally, we would like to mention the work by N. Jones and collaborators on the use of the size-change principle to prove termination for first-order functional languages (LJBA01). The approach on size-change graphs seems flexible and powerful enough to handle permuting arguments and more complex mutual recursions. Although it is possible to build a size-change graph around any well-founded size measure, it is not obvious how to incorporate reasoning about higher-order data-types.

This paper builds on Rohwedder and Pfenning's work on mode and termination checking for higher-order logic programs (RP96). 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. The emphasis of their work has been the correctness of the termination checker with respect to the operational semantics of *Twelf* programs. They define first a measure which characterizes higher-order subterm ordering. Then they define a small step operational semantics using success continuations and show that in each step the measure decreases. This approach extends to reduction properties straightforwardly. The measure associated with higher-order subterm ordering does not change. To handle reduction properties during termination checking, we just need to pass valid reduction properties to the success continuations, so they can be used when showing the measure decreases in each step. To present a full account of the operational semantics, the mode analysis needed, the measure associated with higher-order subterm ordering and the actual correctness proof is beyond the scope of this paper.

7. Conclusion

In this paper, we give two checkers for proving reduction and termination properties about higher-order logic programs. Both checkers share a syntax-driven inference system for reasoning about structural properties, which can be characterized by higher-order subterm orderings. Structural orderings have been very powerful in logical frameworks where we have higher-order data-types. The support for mutual recursive definitions allows elegant formalizations which expose structural properties.

Our system is implemented as part of *Twelf*, and efficiently checks programs and proofs. Over the past years, the system has been used on a wide range of examples. 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]. In particular, the development of type-safety proofs for TALT in *Twelf* attests to the strength of the presented approach in practice. Although many examples can be handled solely by termination checking, reduction properties have been crucial in several places to reason about the size of the induction variables. Our users attest to the fact that the directives for termination and reduction checking are simple to use and have been useful to check the user's intuition and catch bugs.

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. If a higher multiplicity might be needed, an appropriate warning is returned. So far there has been one example which had to be re-written to circumvent this restriction.

In the future, we plan to incorporate reasoning about the size of induction variables into the *Twelf*'s induction theorem prover (PS99), where currently Rohwedder and Pfenning's termination checker is used. The main problem we foresee in this setting is the increased search space.

---

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

Appendix

A. Proving cut-admissibility

As mentioned earlier, the overall idea for proving cut-admissibility is to first 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 atomic. 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 the following development, we will abbreviate a term $x\ M_1 \dots M_n$ where x is an existential and $\Psi(x) = \exists x{:}A$ and just write X. Objects which are of base type and are not λ-abstraction are denoted with R. Hence objects R are either $x\ M_1\ \dots\ M_n$ (where x is declared in Ψ) or $c\ M_1\ \dots M_n$. We sometimes refer to an atomic object R_a which represents either an existential variable $x\ M_1 \dots M_n$ where $\Psi(x) = \exists x{:}A$, or a bound x where $\Psi(x) = \forall x{:}A$ or a constant c. Atomic objects R_a are maximally unfolded and cannot be decomposed further. Next, we present the specialized inference system which also is a step towards a developing a basis for implementing the reasoning about orders.

Reflexivity, Symmetry, Transitivity

$$\frac{}{\Psi; \Delta, R \prec X \Longrightarrow R \prec X}\ Id\prec \qquad \frac{}{\Psi; \Delta, R \equiv X \Longrightarrow R \equiv X}\ Id\equiv$$

$$\frac{\Psi; \Delta, X \equiv R \Longrightarrow P}{\Psi; \Delta, R \equiv X \Longrightarrow P}\ sym$$

$$\frac{\Psi; \Delta, R_2 \prec X \Longrightarrow R_1 \prec R_2}{\Psi; \Delta, R_2 \prec X \Longrightarrow R_1 \prec X}\ t\prec \qquad \frac{\Psi; \Delta, R_2 \equiv X \Longrightarrow R_1 \prec R_2}{\Psi; \Delta, R_2 \equiv X \Longrightarrow R_1 \prec X}\ t{\prec}{\equiv}$$

$$\frac{\Psi; \Delta, R_2 \prec X \Longrightarrow R_1 \equiv R_2}{\Psi; \Delta, R_2 \prec X \Longrightarrow R_1 \prec X}\ t{\equiv}{\prec} \qquad \frac{\Psi; \Delta, X \equiv R_2 \Longrightarrow R_1 \equiv X}{\Psi; \Delta, X \equiv R_2 \Longrightarrow R_1 \equiv R_2}\ t{\equiv}{\equiv}$$

$\mathbf{M \equiv N}$

$$\frac{\Psi; \Delta \Longrightarrow M_1 \equiv N_1 \quad \dots \quad \Psi; \Delta \Longrightarrow M_n \equiv N_n}{\Psi; \Delta \Longrightarrow hM_1 \dots M_n \equiv hN_1 \dots N_n}\ R\equiv$$

$$\frac{\Psi; \Delta, M_1 \equiv N_1, \dots, M_n \equiv N_n \Longrightarrow P}{\Psi; \Delta, hM_1 \dots M_n \equiv hN_1 \dots N_n \Longrightarrow P}\ L\equiv \qquad \frac{}{\Psi; \Delta, hM_1 \dots M_n \equiv gN_1 \dots N_k \Longrightarrow P}\ L{\equiv_1}$$

$\mathbf{M \prec N}$

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

M $\preceq$ N

$$\frac{\Psi; \Delta \Longrightarrow R \prec N}{\Psi; \Delta \Longrightarrow R \preceq N}\ R{\preceq}_1 \qquad \frac{\Psi; \Delta \Longrightarrow R \equiv N}{\Psi; \Delta \Longrightarrow R \preceq N}\ R{\preceq}_2$$

$$\frac{\Psi; \Delta, M \prec R \Longrightarrow P \quad \Psi; \Delta, M \equiv R \Longrightarrow P}{\Psi; \Delta, M \preceq R \Longrightarrow P}\ L{\preceq}$$

λ-abstraction: introduce a new parameter y

$$\frac{\Psi, \forall y{:}A; \Delta \Longrightarrow [y/x]M \equiv [y/x]N}{\Psi; \Delta \Longrightarrow \lambda x{:}A.M \equiv \lambda x{:}A.N}\ R{\equiv}\lambda \qquad \frac{\Psi, \forall y{:}A; \Delta \Longrightarrow [y/x]M \prec N}{\Psi; \Delta \Longrightarrow \lambda x{:}A.M \prec N}\ RL{\prec}\lambda$$

$$\frac{\Psi, \forall y{:}A; \Delta,\ M \prec [y/x]N \Longrightarrow P}{\Psi; \Delta, M \prec \lambda x{:}A.N \Longrightarrow P}\ LR{\prec}\lambda \qquad \frac{\Psi, \forall y{:}A; \Delta \Longrightarrow [y/x]M \preceq N}{\Psi; \Delta \Longrightarrow \lambda x{:}A.M \preceq N}\ RL{\preceq}\lambda$$

$$\frac{\Psi, \forall y{:}A; \Delta, M \preceq [y/x]N \Longrightarrow P}{\Psi; \Delta, M \preceq \lambda x{:}A.N \Longrightarrow P}\ LR{\preceq}\lambda$$

λ-abstraction: instantiate with parameter $\underline{y}$ such that $\Psi(y) = \forall y{:}A$

$$\frac{\Psi; \Delta, [\underline{y}/x]M \equiv [\underline{y}/x]N \Longrightarrow P}{\Psi; \Delta, \lambda x{:}A.M \equiv \lambda x{:}A.N \Longrightarrow P}\ L{\equiv}\lambda$$

$$\frac{\Psi; \Delta \Longrightarrow R \prec [\underline{y}/x]N}{\Psi; \Delta \Longrightarrow R \prec \lambda x{:}A.N}\ RR{\prec}\lambda \qquad \frac{\Psi; \Delta,\ [\underline{y}/x]M \prec R_a \Longrightarrow P}{\Psi; \Delta, \lambda x{:}A.M \prec R_a \Longrightarrow P}\ LL{\prec}\lambda$$

$$\frac{\Psi; \Delta \Longrightarrow R \preceq [\underline{y}/x]N}{\Psi; \Delta \Longrightarrow R \preceq \lambda x{:}A.N}\ RR{\preceq}\lambda \qquad \frac{\Psi; \Delta, [\underline{y}/x]M \preceq R_a \Longrightarrow P}{\Psi; \Delta, \lambda x{:}A.M \preceq R_a \Longrightarrow P}\ LL{\preceq}\lambda$$

$$\frac{}{\Psi; \Delta, \lambda x{:}A.M \equiv hN_1 \dots N_n \Longrightarrow P}\ L{\equiv}_3$$

Soundness of the specialized system is obvious as every derivation in the specialized system has a corresponding (more general) derivation in the original system.

THEOREM 5 (Soundness).

1. *If $\mathcal{D} : \Psi; \Delta \Longrightarrow P$ then $\mathcal{P} : \Psi; \Delta \longrightarrow P$.*

Proof: Induction on $\mathcal{D}$. □

For completeness, we need to show first the following admissibility lemmas. We first show that we can maximally unfold terms and restrict the application of the identity rule to maximally unfolded terms.

LEMMA 6 (Admissibility of general identity rule).

1. *For all terms M, N, $\mathcal{D} : \Psi; \Delta, M \prec N \Longrightarrow M \prec N$*
2. *For all terms M, N, $\mathcal{D} : \Psi; \Delta, M \equiv N \Longrightarrow M \equiv N$.*

3. For all terms M, N, $\mathcal{D} : \Psi; \Delta, M \preceq N \Longrightarrow M \preceq N$

Proof: Induction on the structure of M and N. We apply the induction hypothesis if either the term M or N are smaller. Note, all assumptions persist, although we might not explicitly keep copies. We only show the proof for admissibility of the general identity rule $\prec$ and admissibility of identity rule for $\preceq$.

Case $M = R$ and $N = X$.
There exists a derivation $\mathcal{D} : \Psi; \Delta, R \prec X \Longrightarrow R \prec X$ by $Id \prec$.

Case $M = R$ and $N = hN_1 \dots N_n$.
By induction hypothesis and weakening there exists i derivations $\mathcal{D}_i : \Psi; \Delta, R \prec hN_1 \dots N_n, R \preceq N_i \Longrightarrow R \preceq N_i$. Let $\overrightarrow{N}$ be $N_1 \dots N_n$.

$$\dfrac{\dfrac{\overset{\mathcal{D}_1}{\Psi; \Delta, R \preceq N_1 \Longrightarrow R \preceq N_i}}{\Psi; \Delta, R \preceq N_1 \Longrightarrow R \prec h\overrightarrow{N}}\, R{\prec_1} \qquad \dfrac{\overset{\mathcal{D}_n}{\Psi; \Delta, R \preceq N_n \Longrightarrow R \prec N_n}}{\Psi; \Delta, R \preceq N_n \Longrightarrow R \prec h\overrightarrow{N}}\, R{\prec_n}}{\Psi; \Delta, R \prec h\overrightarrow{N} \Longrightarrow R \prec h\overrightarrow{N}}\, L{\prec}$$

Case $M = R$ and $N = \lambda x{:}A.N$. Then by induction hypothesis and weakening there exists a derivation $\mathcal{D}_1 : \Psi'; \Delta, R \prec \lambda x{:}A.N, R \prec [y/x]N \Longrightarrow R \prec [\underline{y}/x]N$. where $\Psi'(y) = \forall y{:}A$.

$$\dfrac{\dfrac{\overset{\mathcal{D}_1}{\Psi, \forall y{:}A\ ;\ \Delta, R \prec [y/x]N \Longrightarrow R \prec [\underline{y}/x]N}}{\Psi, \forall y{:}A\ ;\ \Delta, R \prec [y/x]N \Longrightarrow R \prec \lambda x{:}A.N}\, RL\lambda \prec}{\Psi\ ;\ \Delta, R \prec \lambda x{:}A.N \Longrightarrow R \prec \lambda x{:}A.N}\, LR\lambda{\prec}$$

Case $M = \lambda x{:}A.M$ and $N = \lambda \overrightarrow{x}.R$ where $\overrightarrow{x} = x_1{:}A_1 \dots x_n{:}A_n$ and $\Psi' = \forall y_1{:}A_1, \dots, \forall y_n{:}A_n$

$\Psi, \Psi', y_{n+1}{:}A; \Delta, [\underline{y_{n+1}}/x]M \prec [\overrightarrow{y}/\overrightarrow{x}]R \Longrightarrow [y_{n+1}/x]M \prec [\underline{\overrightarrow{y}}/\overrightarrow{x}]R$ by IH

$\Psi, \Psi', y_{n+1}{:}A; \Delta, \lambda x{:}A.M \prec [\overrightarrow{y}/\overrightarrow{x}]R \Longrightarrow [y_{n+1}/x]M \prec [\underline{\overrightarrow{y}}/\overrightarrow{y}]R$ by $LL\lambda{\prec}$

$\Psi, \Psi'; \Delta, \lambda x{:}A.M \prec [\overrightarrow{y}/\overrightarrow{x}]R \Longrightarrow \lambda x{:}A.M \prec [\underline{\overrightarrow{y}}/\overrightarrow{x}]R$ by $RL\lambda{\prec}$

$\Psi, \Psi'; \Delta, \lambda x{:}A.M \prec [\overrightarrow{y}/\overrightarrow{x}]R \Longrightarrow \lambda x{:}A.M \prec \lambda \overrightarrow{x}.R$ by $RR\lambda{\prec}$ n-times

$\Psi; \Delta, \lambda x{:}A.M \prec \lambda \overrightarrow{x}.R \Longrightarrow \lambda x{:}A.M \prec \lambda \overrightarrow{x}.R$ by $LR\lambda{\prec}$ n-times

Proof of identity rule $\equiv$ and $\preceq$ similar to the one above.

$\square$

LEMMA 7 (Admissibility).

ref:	$\mathcal{D} : \Psi; \Delta \Longrightarrow M \equiv M$.
t≺:	*If* $\mathcal{D} : \Psi; \Delta, M_2 \prec M_3 \Longrightarrow M_1 \prec M_2$ *then* $\mathcal{P} : \Psi; \Delta, M_2 \prec M_3 \Longrightarrow M_1 \prec M_3$.
t≺≡:	*If* $\mathcal{D} : \Psi; \Delta, M_2 \equiv M_3 \Longrightarrow M_1 \prec M_2$ *then* $\mathcal{P} : \Psi; \Delta, M_2 \equiv M_3 \Longrightarrow M_1 \prec M_3$.
t ≡ ≺:	*If* $\mathcal{D} : \Psi; \Delta, M_2 \prec M_3 \Longrightarrow M_1 \equiv M_2$ *then* $\mathcal{P} : \Psi; \Delta, M_2 \prec M_3 \Longrightarrow M_1 \prec M_3$.
t≡ ≡:	*If* $\mathcal{D} : \Psi; \Delta, M_2 \equiv M_3 \Longrightarrow M_1 \equiv M_2$ *then* $\mathcal{P} : \Psi; \Delta, M_2 \equiv M_3 \Longrightarrow M_1 \equiv M_3$.
R⪯₁:	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \prec N$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \preceq N$.
R⪯₂:	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \equiv N$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \preceq N$.
LL⪯λ:	*If* $\mathcal{D} : \Psi; \Delta, [\underline{y}/x]M \preceq N \Longrightarrow P$ *then* $\mathcal{P} : \Psi; \Delta, \lambda x{:}A.M \preceq N \Longrightarrow P$.
RR⪯λ:	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \preceq [\underline{y}/x]N$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \preceq \lambda x{:}A.N$.
R≺ᵢ:	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \preceq M_i$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \prec hM_1 \ldots M_n$.
LL≺λ:	*If* $\mathcal{D} : \Psi; \Delta, [\underline{y}/x]M \prec N \Longrightarrow P$ *then* $\mathcal{P} : \Psi; \Delta, \lambda x{:}A.M \prec N \Longrightarrow P$.
RR≺λ:	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \prec [\underline{y}/x]N$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \prec \lambda x{:}A.N$.

Proof: Proof by induction on the term structure M resp. N. We show the proof of $R\preceq_1$ and $R\preceq_2$.
Proof of $R \preceq 1$:

Case $M = R$.
By $R \preceq_1$ rule there exists a derivation $\Psi; \Delta \Longrightarrow R \preceq N$

Case $M = \lambda x{:}A.M$ $\qquad \mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi, \forall y{:}A; \Delta \Longrightarrow [y/x]M \preceq N\end{array}}{\Psi; \Delta \Longrightarrow \lambda x{:}A.M \prec N}\, RL\lambda\prec$

$\Psi, \forall y{:}A; \Delta \Longrightarrow [y/x]M \preceq N$ by IH
$\Psi; \Delta \Longrightarrow \lambda x{:}A.M \preceq N$ by $RL\lambda\preceq^y$

Proof of $R \preceq_2$:

Case $M = R$.

By $R \preceq 2$ rule there exists a derivation $\Psi; \Delta \Longrightarrow R \preceq N$

Case $M = \lambda x{:}A.M$ $\qquad \mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi, \forall y{:}A; \Delta \Longrightarrow [y/x]M \equiv [y/x]N\end{array}}{\Psi; \Delta \Longrightarrow \lambda x{:}A.M \equiv \lambda x{:}A.N}\, R{\equiv}\lambda$

$\Psi, \forall y{:}A; \Delta \Longrightarrow [y/x]M \preceq [y/x]N$	by IH
$\Psi, \forall y{:}A; \Delta \Longrightarrow [y/x]M \preceq \lambda x{:}A.N$	by $RR\lambda\preceq$
$\Psi; \Delta \Longrightarrow \lambda x{:}A.M \preceq \lambda x{:}A.N$	by $RL\lambda\preceq$

□

In addition we show that the general version of the restricted rules are admissible.

LEMMA 8 (Admissibility).

ref:	$\mathcal{D} : \Psi; \Delta \Longrightarrow M \equiv M$.
t≺:	*If* $\mathcal{D} : \Psi; \Delta, M_2 \prec M_3 \Longrightarrow M_1 \prec M_2$ *then* $\mathcal{P} : \Psi; \Delta, M_2 \prec M_3 \Longrightarrow M_1 \prec M_3$.
t≺≡:	*If* $\mathcal{D} : \Psi; \Delta, M_2 \equiv M_3 \Longrightarrow M_1 \prec M_2$ *then* $\mathcal{P} : \Psi; \Delta, M_2 \equiv M_3 \Longrightarrow M_1 \prec M_3$.
t ≡ ≺:	*If* $\mathcal{D} : \Psi; \Delta, M_2 \prec M_3 \Longrightarrow M_1 \equiv M_2$ *then* $\mathcal{P} : \Psi; \Delta, M_2 \prec M_3 \Longrightarrow M_1 \prec M_3$.
t≡ ≡:	*If* $\mathcal{D} : \Psi; \Delta, M_2 \equiv M_3 \Longrightarrow M_1 \equiv M_2$ *then* $\mathcal{P} : \Psi; \Delta, M_2 \equiv M_3 \Longrightarrow M_1 \equiv M_3$.
$R\preceq_1$*:*	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \prec N$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \preceq N$.
$R\preceq_2$*:*	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \equiv N$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \preceq N$.
$LL\preceq\lambda$*:*	*If* $\mathcal{D} : \Psi; \Delta, [\underline{y}/x]M \preceq N \Longrightarrow P$ *then* $\mathcal{P} : \Psi; \Delta, \lambda x{:}A.M \preceq N \Longrightarrow P$.
$RR\preceq\lambda$*:*	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \preceq [\underline{y}/x]N$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \preceq \lambda x{:}A.N$.
$R\prec_i$*:*	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \preceq M_i$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \prec hM_1 \dots M_n$.
$LL\prec\lambda$*:*	*If* $\mathcal{D} : \Psi; \Delta, [\underline{y}/x]M \prec N \Longrightarrow P$ *then* $\mathcal{P} : \Psi; \Delta, \lambda x{:}A.M \prec N \Longrightarrow P$.
$RR\prec\lambda$*:*	*If* $\mathcal{D} : \Psi; \Delta \Longrightarrow M \prec [\underline{y}/x]N$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow M \prec \lambda x{:}A.N$.

Proof: Proof by induction on the term structure M resp. N. □

THEOREM 9 (Completeness).

If $\mathcal{D} : \Psi; \Delta \longrightarrow P$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow P$.

Proof: Induction on $\mathcal{D}$. □

Before we show cut admissible in the type-directed $\Longrightarrow$ inference system, we prove some useful lemmas for valid judgments. The system for deriving an order under no assumptions is given next.

$$\frac{}{\Psi;\ \cdot \Longrightarrow M \equiv M}\ refl$$

$$\frac{\Psi;\ \cdot \Longrightarrow M_1 \equiv N_1 \quad \ldots \quad \Psi;\ \cdot \Longrightarrow M_n \equiv N_n}{\Psi;\ \cdot \Longrightarrow hM_1 \ldots M_n \equiv hN_1 \ldots N_n}\ R{\equiv} \qquad \frac{\Psi;\ \cdot \Longrightarrow R \preceq N_i}{\Psi;\ \cdot \Longrightarrow R \prec hN_1 \ldots N_n}\ R{\prec_i}$$

$$\frac{\Psi;\ \cdot \Longrightarrow R \prec [\underline{y}/x]N \quad \Psi(y) = \forall y{:}A}{\Psi;\ \cdot \Longrightarrow R \prec \lambda x{:}A.N}\ RR{\prec}\lambda \qquad \frac{\Psi, \forall y{:}A;\ \cdot \Longrightarrow [y/x]M \prec N}{\Psi;\ \cdot \Longrightarrow \lambda x{:}A.M \prec N}\ RL{\prec}\lambda$$

$$\frac{\Psi;\ \cdot \Longrightarrow R_1 \prec R_2}{\Psi;\ \cdot \Longrightarrow R_1 \preceq R_2}\ R{\preceq_1} \qquad \frac{\Psi;\ \cdot \Longrightarrow R_1 \equiv R_2}{\Psi;\ \cdot \Longrightarrow R_1 \preceq R_2}\ R{\preceq_2}$$

$$\frac{\Psi;\ \cdot \Longrightarrow R \preceq [\underline{y}/x]N}{\Psi;\ \cdot \Longrightarrow R \preceq \lambda x{:}A.N}\ RR{\preceq}\lambda \qquad \frac{\Psi, \forall y{:}A;\ \cdot \Longrightarrow [y/x]M \preceq N}{\Psi;\ \cdot \Longrightarrow \lambda x{:}A.M \preceq N}\ RL{\preceq}\lambda$$

$$\frac{\Psi, \forall y{:}A;\ \cdot \Longrightarrow [y/x]P}{\Psi;\ \cdot \Longrightarrow \Pi x{:}A.P}\ R\Pi$$

Note, we specialize the $R{\preceq_1}$ and $R{\preceq_2}$ rules even further. The specialized rules for $R{\preceq_1}$ / $R{\preceq_2}$ require that $R \preceq N$ where only R is of base type, i.e., $R \neq \lambda x{:}A.M$. N is unrestricted, because we might first introduce new parameters while working on the hypothesis, which are then needed to instantiate N. In the system without assumptions, we can further specialize the rules for $\preceq$, requiring both terms to be of base type. Transitivity is admissible in this inference system.

LEMMA 10 (Transitivity).

1. *If* $\mathcal{D} : \Psi; \cdot \Longrightarrow M \prec M'$ *and* $\mathcal{P} : \Psi; \cdot \Longrightarrow M' \prec M_1$ *then* $\mathcal{F} : \Psi; \cdot \Longrightarrow M \prec M_1$.
2. *If* $\mathcal{D} : \Psi; \cdot \Longrightarrow M \prec M'$ *and* $\mathcal{P} : \Psi; \cdot \Longrightarrow M' \equiv M_1$ *then* $\mathcal{F} : \Psi; \cdot \Longrightarrow M \prec M_1$.
3. *If* $\mathcal{D} : \Psi; \cdot \Longrightarrow M \equiv M'$ *and* $\mathcal{P} : \Psi; \cdot \Longrightarrow M' \prec M_1$ *then* $\mathcal{F} : \Psi; \cdot \Longrightarrow M \preceq M_1$.
4. *If* $\mathcal{D} : \Psi; \cdot \Longrightarrow M \equiv M'$ *and* $\mathcal{P} : \Psi; \cdot \Longrightarrow M' \equiv M_1$ *then* $\mathcal{F} : \Psi; \cdot \Longrightarrow M \equiv M_1$.

Proof: Induction on $\mathcal{P}$. □

LEMMA 11. *If* $\mathcal{D} : \Psi; \Delta, M \equiv M \Longrightarrow P$ *then* $\mathcal{P} : \Psi; \Delta \Longrightarrow P$.

Proof: Induction on $\mathcal{D}$ □

Now we will prove cut-admissibility. Note, R is of base type and cannot be a λ-abstraction, when $\overrightarrow{x}$ is empty.

THEOREM 12 (Cut-admissibility).

1. *If* $\mathcal{D} : \Psi; . \Longrightarrow M \equiv M'$ *and* $\mathcal{P} : \Psi; \Delta, M \equiv M' \Longrightarrow P'$ *then* $\mathcal{F} : \Psi; \Delta \Longrightarrow P'$.

2. *If* $\mathcal{D} : \Psi, \forall y_1{:}A_1, \ldots, \forall y_n{:}A_n; . \Longrightarrow \sigma M \prec M'$ *and* $\mathcal{P} : \Psi; \Delta, \lambda x_n{:}A_n, \ldots, \lambda x_1{:}A_1.M \prec M' \Longrightarrow P'$ *then* $\mathcal{F} : \Psi; \Delta \Longrightarrow P'$.

3. *If* $\mathcal{D} : \Psi, \forall y_1{:}A_1, \ldots, \forall y_n{:}A_n; . \Longrightarrow \sigma M \preceq M'$ *and* $\mathcal{P} : \Psi; \Delta, \lambda x_n{:}A_n, \ldots, \lambda x_1{:}A_1.M \preceq M' \Longrightarrow P'$ *then* $\mathcal{F} : \Psi; \Delta \Longrightarrow P'$.

Proof: The proof follows by induction on $\mathcal{P}$ and $\mathcal{D}$. Let P be the cut-predicate, i.e., either $M \prec M'$ or $M \preceq M'$ or $\Pi x{:}A.P$. Either the order relation P gets smaller or P stays the same and one of the derivations is strictly smaller while the other one stays the same. The order relation $M \prec N$ is smaller than $M \prec N'$ iff N is smaller than N' or $M' \prec N$ is smaller than $M \prec N$ iff M' is smaller than M.

Note, we assume all assumptions persist, although we might not mention them explicitly[2].

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi, \forall y{:}A; \cdot \Longrightarrow \sigma \circ [y/x]M \prec N\end{array}}{\Psi; \cdot \Longrightarrow \sigma\lambda x{:}A.M \prec N} R\lambda{\prec}$

$\mathcal{P} = \Psi; \Delta, \lambda\overrightarrow{x}.\lambda x{:}A.M \prec N \Longrightarrow P'$	by assumption
$\mathcal{F} : \Psi; \Delta \Longrightarrow P'$	by IH $\mathcal{D}_1$ and $\mathcal{P}$

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi; \cdot \Longrightarrow \sigma R \prec [\underline{y}/x]N\end{array} \qquad \Psi(y) = \forall y{:}A}{\Psi; \cdot \Longrightarrow \sigma R \prec \lambda x{:}A.N} RR{\prec}\lambda$

$\mathcal{P} = \Psi; \Delta, \lambda\overrightarrow{x}.R \prec \lambda x{:}A.N \Longrightarrow P'$	by assumption
$\mathcal{P}_1 = \Psi, \forall y{:}A; \Delta, \lambda\overrightarrow{x}.R \prec \lambda x{:}A.N, \lambda\overrightarrow{x}.R \prec [y/x]N \Longrightarrow P'$	by inversion
$\mathcal{P}'$: $\Psi, \forall y{:}A; \Delta, \lambda\overrightarrow{x}.M \prec [y/x]N \Longrightarrow P'$	by IH on $\mathcal{D}$ and $\mathcal{P}_1$
$\Psi; \Delta \Longrightarrow P'$	by IH on $\mathcal{D}_1$ and $[\underline{y}/y]\mathcal{P}'$

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}'\\ \Psi; \cdot \Longrightarrow \sigma R \preceq N_i\end{array}}{\Psi; \Delta \Longrightarrow \sigma R \prec hN_1 \ldots N_n} R{\prec}_i$

$\mathcal{P} = \Psi; \Delta, \lambda\overrightarrow{x}.R \prec hN_1 \ldots N_n \Longrightarrow P'$	by assumption
$\mathcal{P}_i = \Psi; \Delta, \lambda\overrightarrow{x}.R \preceq N_i \Longrightarrow P'$	by inversion
$\mathcal{F}'$: $\Psi; \Delta, \lambda\overrightarrow{x}.R \preceq N_i \Longrightarrow P'$	by IH on $\mathcal{D}$ and $\mathcal{P}_i$

[2] The cases for transitivity are not included here, however they are straightforward

$\Psi; \Delta \Longrightarrow P'$ by IH on $\mathcal{D}'$ and $\mathcal{F}'$

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi, \forall y{:}A; \cdot \Longrightarrow \sigma \circ [y/x]M \preceq N\end{array}}{\Psi; \cdot \Longrightarrow \sigma\lambda x{:}A.M \preceq N}\, RL\lambda{\preceq}$

$\mathcal{P} = \Psi; \Delta, \lambda\overrightarrow{x}.\lambda x{:}A.M \preceq N \Longrightarrow P'$ by assumption
$\mathcal{F} : \Psi; \Delta \Longrightarrow P'$ IH $\mathcal{D}_1$ and $\mathcal{P}$

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi; \cdot \Longrightarrow \sigma R \preceq [\underline{y}/x]N\end{array}}{\Psi; \cdot \Longrightarrow \sigma R \preceq \lambda x{:}A.N}\, RR\lambda \preceq$

$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1\\ \Psi, \forall y{:}A; \Delta, \lambda\overrightarrow{x}.R \preceq \lambda x{:}A.N, \lambda\overrightarrow{x}.R \preceq [y/x]N \Longrightarrow P'\end{array}}{\Psi; \Delta, \lambda\overrightarrow{x}.R \prec \lambda x{:}A.N \Longrightarrow P'}\, LR{\preceq}\lambda$$

$\mathcal{F}'$: $\Psi; \Delta, \lambda\overrightarrow{x}.R \preceq [y/x]N \Longrightarrow P'$ IH on $\mathcal{D}$ and $\mathcal{P}_1$
$\Psi; \Delta \Longrightarrow P'$ IH on $\mathcal{D}_1$ and $[\underline{y}/y]\mathcal{F}'$

Case $\mathcal{D} = \dfrac{}{\Psi; \cdot \Longrightarrow \sigma R_1 \preceq R_2}$

$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1\\ \Psi; \Delta, [\underline{y}/x]\lambda\overrightarrow{x}.R_1 \preceq R_2, \lambda x{:}A.\lambda\overrightarrow{x}.R_1 \preceq R_2 \Longrightarrow P'\end{array}}{\Psi; \Delta, \lambda x{:}A.\lambda\overrightarrow{x}.R_1 \preceq R_2 \Longrightarrow P'}\, LL{\preceq}\lambda$$

$\mathcal{F}'$: $\Psi; \Delta, [\underline{y}/x]\lambda\overrightarrow{x}.R_1 \prec R_2 \Longrightarrow P'$ IH on $\mathcal{D}$ and $\mathcal{P}_1$
$\Psi; \Delta \Longrightarrow P'$ with $\sigma = \sigma'\ [y/x]$
IH on $[\underline{y}/y]\mathcal{D}$ and $\mathcal{F}'$

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi; \cdot \Longrightarrow \sigma R_1 \prec R_2\end{array}}{\Psi; \cdot \Longrightarrow \sigma R_1 \preceq R_2}\, R{\preceq_1}$

$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1\\ \Psi; \Delta, \lambda\overrightarrow{x}.R_1 \preceq R_2, \lambda\overrightarrow{x}.R_1 \prec R_2 \Longrightarrow P'\end{array} \quad \begin{array}{c}\mathcal{P}_2\\ \Psi; \Delta, \lambda\overrightarrow{x}.R_1 \preceq R_2, \lambda\overrightarrow{x}.R_1 \equiv R_2 \Longrightarrow P'\end{array}}{\Psi; \Delta, \lambda\overrightarrow{x}.R_1 \preceq R_2 \Longrightarrow P'}\, L{\preceq}$$

$\mathcal{F}'$: $\Psi; \Delta, R_1 \prec R_2 \Longrightarrow P'$ IH on $\mathcal{D}$ and $\mathcal{P}_1$
$\Psi; \Delta \Longrightarrow P'$ IH on $\mathcal{D}_1$ and $\mathcal{F}'$

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi; \cdot \Longrightarrow \sigma R_1 \equiv R_2\end{array}}{\Psi; \cdot \Longrightarrow \sigma R_1 \preceq R_2} R{\preceq}_2$

$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1\\ \Psi; \Delta, R_1 \preceq R_2, R_1 \prec R_2 \Longrightarrow P'\end{array} \qquad \begin{array}{c}\mathcal{P}_2\\ \Psi; \Delta, R_1 \preceq R_2, R_1 \equiv R_2 \Longrightarrow P'\end{array}}{\Psi; \Delta, R_1 \preceq R_2 \Longrightarrow P'} L{\preceq}$$

$\mathcal{F}'$: $\Psi; \Delta, R_1 \equiv R_2 \Longrightarrow P'$ IH on $\mathcal{D}$ and $\mathcal{P}_2$
$\Psi; \Delta \Longrightarrow P'$ IH on $\mathcal{D}_1$ and $\mathcal{F}'$

Case $\mathcal{D} = \dfrac{}{\Psi; \cdot \Longrightarrow \sigma h \equiv h} refl$ $\qquad \mathcal{P} = \dfrac{}{\Psi; \Delta, h \equiv h \Longrightarrow P}$

$\Psi; \Delta \Longrightarrow P$ by lemma.

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi; \cdot \Longrightarrow \sigma M_1 \equiv N_1\end{array} \quad \ldots \quad \begin{array}{c}\mathcal{D}_n\\ \Psi; \cdot \Longrightarrow \sigma M_n \equiv N_n\end{array}}{\Psi; \cdot \Longrightarrow \sigma h M_1 \ldots M_n \equiv h N_1 \ldots N_n} R{\equiv}$

$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}'\\ \Psi; \Delta, h M_1 \ldots M_n \equiv h N_1 \ldots N_n, \forall i. M_i \equiv N_i \Longrightarrow P'\end{array}}{\Psi; \Delta, h M_1 \ldots \equiv h N_1 \ldots N_n \Longrightarrow P'} L{\equiv}$$

$\mathcal{F}_1$: $\Psi; \Delta, \forall i. M_i \preceq N_i \Longrightarrow P'$ IH on $\mathcal{D}$ and $\mathcal{P}_i$
$\mathcal{F}_{i+1} : \Psi; \Delta \Longrightarrow P'$ i-times IH on $\mathcal{D}_i$ and $\mathcal{F}_i$

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi, \forall y{:}A; \cdot \Longrightarrow [y/x]M \equiv [y/x]N\end{array}}{\Psi; \cdot \Longrightarrow \lambda x{:}A.M \equiv \lambda x{:}A.N} R{\equiv}\lambda$

$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}'\\ \Psi; \Delta, \lambda x{:}A.M \equiv \lambda x{:}A.N, [\underline{y}/x]M \equiv [\underline{y}/x]N \Longrightarrow P'\end{array}}{\Psi; \Delta, \lambda x{:}A.M \equiv \lambda x{:}A.N \Longrightarrow P'} L{\equiv}\lambda$$

$\mathcal{F}_1$: $\Psi; \Delta, [\underline{y}/x]M \equiv [\underline{y}/x]N \Longrightarrow P'$ IH on $\mathcal{D}$ and $\mathcal{P}_1$
$\mathcal{F}$: $\Psi; \Delta \Longrightarrow P'$ IH on $[\underline{y}/y]\mathcal{D}_1$ and $\mathcal{F}_1$

□

B. Reasoning about lexicographic orders

We can extend the the proofs of soundness, cut-admissibility and cut-elimination can be extended to lexicographic orders. However, the interplay of disjunctive choices and quantified terms results in surprising consequences. To highlight some of the consequences, let us consider the following conjecture:

$$\begin{array}{l}\Pi x{:}A.\{(\lambda y.M\ y)(M'x)\} \prec \{(\lambda y.N\ y)(N'x)\} \\ \qquad\qquad \longrightarrow \{(\lambda y.M\ y)(\texttt{lam}\ (\lambda x.M'\ x)\} \prec \{(\lambda y.N\ y)(\texttt{lam}\ \lambda x.N'\ x)\}\end{array}$$

This conjecture is not provable because the lambda abstraction is buried underneath a disjunctive choice. The problem is that lexicographic orderings introduce a disjunctive choice, i.e. to show

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

we need to prove either

$$(\lambda y.M\ y) \preceq (\lambda y.N\ y) \text{ and } \texttt{lam}\ \lambda x.M'\ x \prec \texttt{lam}\ \lambda x.N'\ x$$

or

$$(\lambda y.M\ y) \prec (\lambda y.N\ y) \text{ and } \texttt{lam}\ \lambda x.M'\ x \preceq \texttt{lam}\ \lambda x.N'\ x$$

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 which will prevent us from proving the given conjecture. 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.

C. Decidability and implementation issues

In this section we discuss decidability and implementation issues of higher-order subterm reasoning. It is clear that reasoning about first-order subterm relations, is decidable. We impose a precedence order on the relations such that $\prec$ and $\equiv$ are smaller than $\preceq$. In each rule, we either transition from $\preceq$ to $\prec$ or $\equiv$, i.e. the precedence ordering is decreasing, or the term size is getting smaller. In the higher-order case however, the question of decidability is non-trivial, since we can potentially instantiate λ-terms and Π-quantified order relations which occur in the context multiple times. Let us first consider the system with higher-order subterm relations, but without Π-quantified order relations. In this case, we can use the earlier presented specialized system for higher-order subterm relations. The underlying idea is that we can decompose terms and introduce all parameters first, before we instantiate λ-terms which may occur in the $RR{\prec}\lambda$ (or $RR{\preceq}\lambda$) and $LL{\prec}\lambda$ (or $LL{\preceq}\lambda$) rule. This yields a straightforward sound and complete algorithm, as long as no Π-quantified order relations occur. In the presence of Π-quantified order relations, we can no longer first introduce all parameters and then instantiate λ-terms, since there may be λ-terms at universally quantified positions buried underneath the Π-quantifier. As a consequence, we can potentially instantiate the λ-terms and Π-quantifier at existentially quantified positions infinitely many times. The idea for proving decidability would be to show that we can bound the number of instantiations needed.

To limit the search space in the implementation, we use multiplicity one for the Π-quantified orders for now. Obviously, this may lead to incompleteness of the reasoning system, however, we have not found it to be a limitation in practice. None of our examples, requires that the Π-quantifier is instantiated. This allows the implementation of a simple decision procedure based on the rules given earlier. We first decompose terms and introduce all parameters. Then we instantiate λ-terms at existentially quantified positions. This algorithm terminates, since in each rule application, the order relation is getting smaller, i.e. if we have $M \prec N$ then either M or N is getting smaller.

D. Implementation of abstract machine

```
%instructions
inst : type. %name inst I.
ev : exp -> inst.
ret : val -> inst.
app1 : val -> exp -> inst.
app2 : val -> val -> inst.
```

```
Values
val : type. %name val V.
vl : val -> exp.
lam* : (val -> exp) -> val.]
```

```
% Continuation Stack
stack : type. %name stack S.
init : stack.
; : stack -> (val -> inst) ->
stack.
%infix left 8 ;.
```

```
% Continuation Machine States
state : type. %name state T.
# : stack -> inst -> state.
answer : val -> state.
%infix none 7 #.
```

```
% One step state transitions
=>: state -> state -> type
t_lam : S # (ev (lam E)) => S # (ret (lam* E)).
t_app : S # (ev (app E1 E2)) => (S ; [v1:val] app1 v1 E2) # (ev P1).
t_app1 : S # (app1 V1 E2) => (S ; [v2:val] app2 V1 v2) # (ev E2).
t_app2 : S # (app2 (lam* E1') V2) => S # (ev (E1' V2)).
t_ret : (S ; [v:val] I v) # (ret V) => S # (I V).
t_init : (init) # (ret V) => (answer V).

% Multi-Step Computation
=>* : state -> state -> type. %name =>* D.
%infix none 5 =>*.
id : T =>* T.
@: T =>* T'' <- T => T' <- T' =>* T''.
%infix left 5 .
```

Acknowledgements

I would like to thank Andreas Abel and Albert Rubio for clarifying discussions concerning termination of functional languages and termination in higher-order simply-typed rewriting systems. I also would like to thank my advisor Frank Pfenning for all his comments, insights and persistence. For his careful reading and feedback, I also would like to thank Joshua Dunfield and Karl Crary. In particular, I would like to thank Karl Crary for putting the termination and reduction checker to test in his proofs about TALT and providing a valuable user's perspective on using the termination and reduction checker.

References

Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002.

Andreas Abel. Specification and verification of a formal system for structurally recursive functions. In *TYPES'99*. Springer-Verlag, 2000.

Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

Marianne Baudinet. Proving termination properties of prolog programs: A semantic approach. *Journal of Logic Programming*, 14(1&2):1–29, 1992.

Weidong Chen, Michael Kifer, and David Scott Warren. HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

Karl Crary. Toward a foundational typed assembly language. In *30th ACM Symposiumn on Principles of Programming Languages (POPL)*, pages 198–212, New Orleans, Louisisana, January 2003. ACM-Press.

Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *19th International Conference on Automated Deduction*, Miami, Florida, USA, 2003. Extended version published as CMU technical report CMU-CS-03-108.

Michael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

Claude Marche Enno Ohlebusch, Claus Claves. Talp: A tool for the termination analysis of logic programs. In Leo Bachmair, editor, *Proceedings of the 11th International Conference on Rewriting Techniques and Applications (RTA'00), Norwich, UK*, volume 1833 of *Lecture Notes in Computer Science (LNCS)*, pages . 270–273. Springer, July 2000.

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.

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.

E. Ohlenbusch J. Giesl, T. Arts. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.

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.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2001)*, pages 81–92. ACM press, january 2001.

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 Lecture Notes in Computer Science (LNCS) 914.

David McAllester and Kostas Arkoudas. Walther recursion. In *Proceedings of the 13th International Conference on Automated Deduction, New Brunswick, NJ, July 1996*, pages 643–657. Springer-Verlag, 1996.

Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

Aart Middeldorp Nao Hirokawa. Automating the dependency pair method. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 32–46. Springer-Verlag, July 2003.

Yehoshua Sagiv Naomi Lindenstrauss. Automatic termination analysis of logic programs. In Lee Naish, editor, *14th International Conference on Logic Programming, Leuven, Belgium*, pages 63–77. MIT Press, 1997.

Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

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.

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.

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

Brigitte Pientka. Termination and reduction checking for higher-order logic programs. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proceedings of the first International Joint Conference on Automated Reasoning, Siena, Italy*, Lecture Notes in Artificial Intelligence (LNAI) 2083, pages 401–415. Springer-Verlag, 2001.

Lutz Plümer. *Termination Proofs for Logic Programs*. Lecture Notes in Artificial Intelligence (LNAI) 446. Springer-Verlag, 1990.

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 Lecture Notes in Artificial Intelligence (LNAI) 1632.

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 Lecture Notes in Computer Science (LNCS) 1058.

Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.

Chris Speirs, Zoltan Somogyi, and Harald Sondergaard. Termination analysis for mercury. In Pascal Van Hentenryck, editor, *4th International Static Analysis Symposium (SAS), Paris, France, September 8-10, 1997, Proceedings*, volume 1302 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1997.

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 Lecture Notes in Computer Science (LNCS) 902.

Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, Available as Technical Report CMU-CS-99-167, Sep 1999.

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 Lecture Notes in Computer Science (LNCS) 1631.

Christoph Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1), 1994.

Hongwei Xi. Dependent Types for Program Termination Verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, pages 231–242, Boston, June 2001.