Decidability Results For Categorical Notions Related To Monads By Rewriting Techniques

Wolfgang.Gehrke@risc.uni-linz.ac.at

PhD thesis

May 1995

RISC, Johannes Kepler University, Linz

Support

This research was partially supported by:

ÖAD doctoral fellowship (Österreichischer Akademischer Austauschdienst),

DAAD doctoral fellowship (Deutscher Akademischer Austauschdienst),

FWF ESPRIT BRP 6471, MEDLAR II (Austrian Science Foundation),

NSF grant No. CCR-9303383 (American National Science Foundation).

Abstract

We solve the universal word problem for the categorical notion of monads. First this is done by a reduction to the universal word problem of adjunctions. Another canonical rewrite system was found with the help of the Kleisli category of a monad. The latter system reformulated for co-monads was applied to the notion of computational co-monads showing their word problem to be decidable. Finally we establish a canonical higher-order rewrite system for the monadic laws as in purely functional programming. The proofs proceed mainly by a computer assisted check of critical pairs where a special coding technique is necessary to take the object information of morphisms into account. The difficult parts are special methods necessary to prove termination and the reformulation of the corresponding equational specification. In the higher-order case we apply recent results from higher-order rewriting. These systems can be used for automated reasoning about semantic problems and they can be usefully applied to verify commutativity of classes of morphism diagrams.

Kurzreferat

Wir lösen das universelle Wortproblem für den kategoriellen Begriff der Monaden. Zunächst kann das Problem auf das universelle Wortproblem für Adjunktionen reduziert werden. Ein weiteres kanonisches Termersetzungsystem konnte mit Hilfe der Kleisli-Kategorie einer Monade gefunden werden. Dieses in der Formulierung einer Co-Monade ist anwendbar auf den Begriff der "computational Co-Monade", was die Entscheidbarkeit deren Wortproblems ergibt. Schließlich präsentieren wir ein kanonisches Termersetzungsystem höherer Ordnung für die Monadengesetze vom puren funktionalen Programmieren. Die Beweise wurden durch den Rechner unterstützt geführt, welcher kritische Paare überprüfte. Dabei erforderte die Objektinformation der Morphismen eine spezielle Darstellungstechnik. Als schwierig erwiesen sich die notwendigen Methoden, um Terminierung zu zeigen, und die Umformulierung der jeweiligen Spezifikation. Im Falle höherer Ordnung benutzen wir neueste Resultate vom Gebiet der Termersetzung höherer Ordnung. Diese Systeme können für das automatische Beweisen von semantischen Problemen benutzt werden, und sie sind von Nutzen für das Überprüfen der Kommutativität bestimmter Klassen von Morphismendiagrammen.

Acknowledgments

There are several people who influenced the work on my thesis either directly or indirectly. Here I want to express my thanks to all of them for support, encouragement, and guidance.

First of all I have to thank Prof. Bruno Buchberger for creating the fruitful scientific environment in the "Research Institute for Symbolic Computation" (RISC) of the Johannes Kepler University at Linz, Austria, which provided a good frame for this rather long-term project of a PhD thesis.

I want to thank Prof. Günter Pilz for giving me the opportunity to present my thesis work in his seminar where I met a warm and friendly atmosphere.

The completion of my work would not have been possible without the constant enthusiasm of my advisor Dr. Jochen Pfalzgraf who directed my interest on the language of category theory with its visual description in form of commutative diagrams and who gave me the problem which was the starting point of my work.

An important source of ideas was the collaboration with colleagues at RISC. Explicitly I want to mention Dr. Hoon Hong whom I have to thank for introducing me to the field of term rewriting. Furthermore I thank my colleagues in the "computational category theory group" for discussing ideas with me.

At an early stage of my work it was a pleasure to discuss first steps with Prof. Dana Scott who focused my attention on work which has been done at CMU in Pittsburgh, USA, and who established the contact to scientists there.

I am indebted to Dr. Frank Pfenning for his hospitality during a short visit at CMU in February 1993 and a longer visit in autumn 1994 and for our many interesting and stimulating discussions. I had the opportunity to learn more about his logic programming language "Elf" which gave the right frame for a final implementation of my decision procedures.

At CMU I had the opportunity for several scientific contacts. I want to mention discussions with Dr. Stephen Brookes who pointed to the application of co-monads for the study of intensional semantics and discussions with Andrzej Filinski who gave me more insight into the use of monads for functional programming.

Finally Stefan Kahrs from the LFCS of the University of Edinburgh supported this work by detailed comments on his method for termination proofs at a very late stage of the thesis. Additionally I am grateful to all others which I could not mention explicitly but who also helped me on my scientific way.

To my parents

Contents

List of Figures

List of Tables

Outline of the Thesis

Summary

The thesis presents canonical term rewriting systems for categorical notions related to monads. The main problem is the appropriate formalization of a notion in order to make it accessible for rewriting techniques.

The orientation of the rules and the proof of termination is not always obvious. The main tool for testing confluence (if termination is known) is the check of the convergence for critical pairs. To do this the additional type information of morphisms has to be taken into account.

The proofs have been done with the support of the Larch Prover which checked critical pairs and verified polynomial interpretations. The constructed rewriting based decision procedures have been finally implemented in the logic programming language Elf designed by Pfenning. This language supports dependent types and thus we gain an easy way to present morphisms depending on their corresponding source and target objects.

The main notions we are studying are:

1. adjunction
2. monad
3. computational co-monad
4. monadic functional programming

where in the thesis the following main results are established and shown:

- verification of a canonical system for adjunctions, given in a book by Curien without proof, where we show termination of that system with a technique similar to Hardin's and Laville's work on the system SUBST,
- reduction of the universal word problem for monads to adjunctions exploiting theorems from category theory,
- a new canonical system for monads based on a presentation exploiting the notion of Kleisli categories,
- extension of the canonical system for a co-monad to the notion of a computational co-monad as used by Brookes and Geva to study intensional semantics,
- a first step towards rewriting for monadic functional programming using recent results about higher-order rewriting.

Flow of Ideas

Chapter 1, the introduction, motivates the theme of the thesis and puts it into a context mentioning the reasoning with diagrams. In Chapter 2 some basic facts about term rewriting are recalled and their application is demonstrated for fundamental categorical notions where canonical systems are presented.

Chapter 3 introduces the categorical notion of an adjunction and presents a canonical system taken from the literature. The termination of that system is shown with a special technique similar to a related system for a special adjunction.

The central notion of a monad is given in Chapter 4. In the first step the word problem for monads is reduced to the word problem of associated adjunctions. Another and easier canonical system can be established with the help of the Kleisli category for a monad.

The latter canonical system in its adaption to a co-monad can then be extended to the notion of a computational co-monad which is shown in Chapter 5. The main technique is the transformation of the presentation until a canonical system of the computational co-monad can be found.

In Chapter 6 the notion of a monad is related to the monadic functional style used for purely functional programming languages to express impure programming constructs. This requires higher-order rewriting (a first step is sketched). Finally Chapter 7 concludes the thesis where prospects and plans of future work is outlined.

Chapter 1

Introduction

Category theory can be seen from very different points of view. The main reference is still [ML71], other comprehensive books are [AHS90, HP91a, HS79], an easy introduction for computer scientists can be found in [Pie91]. In fact it developed as generalization of various fields in mathematics. For example it can be understood as:

- an abstract theory of functions,
- an abstract theory of structures and structure preserving maps, or
- an abstract theory of deductions.

Its importance is stressed by the fact that it can be used as an alternative foundation of mathematics cf. the pioneering work by Lawvere [Law66] in the 1960s. The development of category theory was also heavily influenced by geometric concepts [MLM92]. It plays an important role in the study of semantical questions like for models in logic or semantics of programming languages [AL91, BW90], or in the investigation of type systems [Cro93]. A general approach to semantical questions can be found in [Pfa95a] which is extended in [Pfa95b]. [LS86] presents logic in a categorical flavor.

The basic ingredients of a category are objects and morphisms. The emphasis is put mainly on properties of the morphisms. Objects are often only characterized up to isomorphism.

In this thesis we study the equality of morphisms in particular categories enriched with structure from an equational point of view. This approach is similar to the investigations in [Hue90a]. [RB88] goes even a step further in that it turns categorical constructions into programs. It is very common to describe equations between morphisms in the form of diagrams.

Equality between morphisms is visualized with the help of commutative diagrams where two different paths from one object to another one impose the equality of the composition of the morphisms along the paths. Proofs of categorical statements can then be done with techniques of so called "diagram chasing" which deals with gluing together commutative diagrams along common paths and checking the commutativity of diagrams.

A very simple example of this technique is shown in Fig. 1.1 where the diagrams $\mathbb{D}_1$ and $\mathbb{D}_2$ have been glued together along $q : B \to E$ resulting in

$$
\begin{tikzcd}
A \arrow[r, "k"] \arrow[d, "p"'] & B \arrow[d, "q"] \\
D \arrow[r, "m"'] & E
\end{tikzcd}
\quad \textit{and} \quad
\begin{tikzcd}
B \arrow[r, "l"] \arrow[d, "q"'] & C \arrow[d, "r"] \\
E \arrow[r, "n"'] & F
\end{tikzcd}
\quad \textit{yield} \quad
\begin{tikzcd}
A \arrow[r, "k"] \arrow[d, "p"'] & B \arrow[r, "l"] \arrow[d, "q"'] & C \arrow[d, "r"] \\
D \arrow[r, "m"'] & E \arrow[r, "n"'] & F
\end{tikzcd}
$$

$$\mathbb{D}_1 \qquad\qquad \mathbb{D}_2 \qquad\qquad \mathbb{D}_{1+2}$$

Figure 1.1: Example of Gluing Commutative Diagrams

$\mathbb{D}_{1+2}$. Algebraically this is equivalent to the statement that $r \circ l \circ k = n \circ m \circ p$ follows from $q \circ k = m \circ p$ and $r \circ l = n \circ q$ (in the context of associativity).

The visual presentation of equations for morphisms in the form of commutative diagrams is a very compact and intuitive encoding. [FS90] introduces a graphical language to present categorical definitions. Results about the pasting of diagrams are obtained in [Pow90a, Pow90b, Bur93]. A first relationship between diagrams and rewriting is established in [Laf92, Laf93].

Nevertheless in the thesis we take the algebraic point of view. Fokkinga even argues completely in favor of the algebraic presentation [Fok91]. Early work on a special case can be found in [WS82].

Starting point and motivation of our thesis work was the general problem of checking the commutativity of diagrams in "every-days" technical work dealing with arrows (morphisms) and arrow diagrams. We point in particular to well established disciplines like homological algebra, algebraic topology, algebraic geometry, K-theory, etc., where these techniques are indispensable and lead to very deep mathematical results.

We are aware of the great generality of this commutativity check problem and that we can run immediately into hard decidability problems. In our thesis we study this problem in a restricted class of categories namely monads and co-monads where we show how the commutativity check problem can be solved in terms of suitable equational rewrite techniques.

Our approach to a solution of the stated problem is to focus on the equational characterization of the studied categorical notions since we are interested in the decision whether equality between two given morphisms does hold or not. Looking at the equations we try to apply methods from term rewriting. Our goal is to achieve a canonical term rewriting system which guarantees normal forms and can be used for deciding equality as in [Hue90a].

From the body of research about term rewriting systems we mainly need the critical pair lemma by Knuth and Bendix [KB70]. Termination is shown by interpretation [Der87] which implies that local confluence and confluence coincide. A confluent and terminating system is called canonical and can be used as a decision procedure.

This approach to the problem has been already chosen by Huet [Hue90a] who mainly focused on categorical notions towards Cartesian closed categories. Another approach has been taken by Rydeheard and Burstall [RB88] but with a different goal. They studied categorical notions which can be turned into algorithms.

Although in a category the objects themselves are more in the background of consideration they have to be taken into account for rewriting. We will not need them for the proofs of termination since they only impose constraints on the applicability of a rule. But for the check of local confluence they have to be considered again.

Concerning the role of objects, the situation for Cartesian closed categories is different since only the type information (source and target object) implies the termination. For purpose of rewriting Cartesian closed categories have to be handled with the help of a typed lambda calculus. This will also be our main technique.

Main Technique: The equational specification of a notion has to be transformed until it is accessible for rewriting, i.e. Knuth-Bendix completion.

In the study of categorical concepts we focus on monads [BW85] and related notions. Monads are the categorical generalization of monoids. They have been used:

- to structure the extensional semantics of programming languages by Moggi [Mog89],
- to study the intensional semantics of programming languages by Brookes and Geva [BG92],
- to structure purely functional programs to allow an easy incorporation of impure programming concepts by Wadler [Wad93].

After showing the problems which occur with the original presentation of monads we undertake the following steps:

1. We reduce the universal word problem of monads to adjunctions.
2. We give an easier solution to the same problem with the help of the Kleisli category for a monad.
3. We extend the easier system for a co-monad to the notion of a computational co-monad.
4. Finally we achieve a partial result for monads as used in functional programming by techniques for higher-order rewrite systems.

The proofs have been carried out with the help of a computer. First the Larch Prover [GG91] helped to check critical pairs and partially the termination. Afterwards we re-implemented the rewriting based decision procedures in the logic programming language Elf. This language is designed by Pfenning [Pfe89, Pfe91, Pfe94].

The reason for the re-implementation was the availability of dependent types in Elf which is based on the Edinburgh Logical Framework LF [HHP93]. Furthermore Elf as a programming language allows different rewriting strategies. Proofs of equality can be inspected since they are represented as terms in Elf.

At the end we will point to remaining open problems and directions of further research. The index gives the reference to main notions and theorems presented in the thesis. Also references to notation used in the thesis are included.

Chapter 2

Preliminaries

In this chapter the necessary preliminaries are presented. First we recall main facts about rewriting. We focus on the notions and theorems which are needed later on. Then we introduce the most basic categorical notions together with their canonical systems.

2.1 Facts about Rewriting

2.1.1 Reduction Relations

At the beginning we closely follow [Klo92].

Notion 1 (Reduction Relation) *Let S be an arbitrary set and $\rightarrow \subseteq S \times S$ be an arbitrary binary relation on S. In the following*

$\rightarrow$ *is called a* **reduction relation***: $a \rightarrow b :\Leftrightarrow (a,b) \in \rightarrow$,*

$\leftrightarrow$ *its* **symmetric** *closure: $a \leftrightarrow b :\Leftrightarrow a \rightarrow b \vee b \rightarrow a$,*

$\rightarrow^{+}$ *its* **transitive** *closure:*

$$
\begin{aligned}
a \rightarrow_1 b \quad &:\Leftrightarrow \quad a \rightarrow b \\
a \rightarrow_{n+1} b \quad &:\Leftrightarrow \quad \exists c \in S : a \rightarrow c \rightarrow_n b \\
a \rightarrow^{+} b \quad &:\Leftrightarrow \quad \exists n \in \mathbb{N}^{+} : a \rightarrow_n b
\end{aligned}
$$

$\rightarrow^{*}$ *its* **transitive reflexive** *closure:*

$$
\begin{aligned}
a \rightarrow_0 b \quad &:\Leftrightarrow \quad a \equiv b \\
a \rightarrow^{*} b \quad &:\Leftrightarrow \quad \exists n \in \mathbb{N}_0 : a \rightarrow_n b
\end{aligned}
$$

$\leftrightarrow^{*}$ *its* **symmetric, transitive, reflexive** *closure:*

$$a \leftrightarrow^{*} b :\Leftrightarrow a \rightarrow^{+} b \vee b \rightarrow^{+} a \vee a \equiv b$$

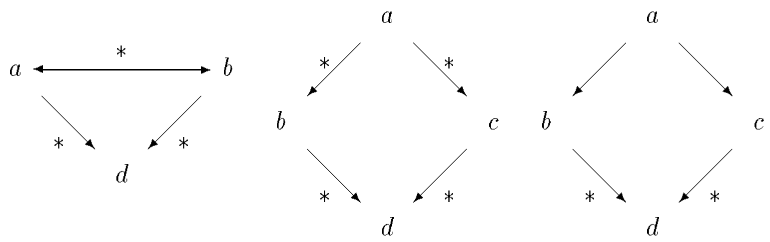

Figure 2.1: Church-Rosser Property, Confluence, Local Confluence

Remark. $(a, b) \in \rightarrow$ is also written as $a \rightarrow b$ and b is called a *one-step* $\rightarrow$ *reduct* of a. If $a \rightarrow^* b$ then b is called a *reduct* of a. Furthermore $a \in S$ is a *normal form* if there is no b with $a \rightarrow b$.

Notion 2 (Converse, Union, Composition) *For arbitrary reduction relations* $\rightarrow, \rightarrow_1, \rightarrow_2$ *there are*

$\rightarrow^{-1}$ *the converse of* $\rightarrow$ *(also denoted* $\leftarrow$*):* $a \rightarrow^{-1} b :\Leftrightarrow b \rightarrow a$*,*

$\rightarrow_1 \cup \rightarrow_2$ *the union of* $\rightarrow_1, \rightarrow_2$*:* $a(\rightarrow_1 \cup \rightarrow_2)b :\Leftrightarrow a \rightarrow_1 b \vee a \rightarrow_2 b$ *and*

$\rightarrow_1 \circ \rightarrow_2$ *the composition of* $\rightarrow_1, \rightarrow_2$*:*

$$a(\rightarrow_1 \circ \rightarrow_2)b :\Leftrightarrow \exists c \in S : a \rightarrow_1 c \wedge c \rightarrow_2 b$$

Definition 3 (Church-Rosser, (local) Confluence, Termination) *A reduction relation* $\rightarrow$ *is called*

Church-Rosser *if* $\leftrightarrow^* \subseteq \rightarrow^* \circ \leftarrow^*$*,*

confluent *if* $\leftarrow^* \circ \rightarrow^* \subseteq \rightarrow^* \circ \leftarrow^*$*,*

locally confluent *if* $\leftarrow \circ \rightarrow \subseteq \rightarrow^* \circ \leftarrow^*$*,*

terminating *if there is no infinite sequence of elements of* S *with* $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$

Remark. A terminating reduction relation sometimes is also called a *Noetherian* relation or a *strongly normalizing* relation. The Church-Rosser property, confluence and local confluence can also be described graphically cf. Fig. 2.1. If two elements $a, b \in S$ have a common reduct $c \in S$, i.e. $a \rightarrow^* c \leftarrow^* b$ they are said to *converge* $(a \downarrow b)$.

Proposition 1 (Equivalence Church-Rosser/Confluence) $\rightarrow$ *has the Church-Rosser property iff it is confluent.*

$$a \longleftarrow b \rightleftarrows c \longrightarrow d$$

Figure 2.2: Local Confluence $\neq$ Confluence

Proof. (idea cf. [New42])
$\Rightarrow$) obvious
$\Leftarrow$) induction over the length of $\leftrightarrow^*$

Example. Before the next proposition is stated it has to be pointed out that local confluence and confluence are not the same. This can be seen from Fig. 2.2. This is an example of a reduction relation which is locally confluent but not confluent (since $a \leftarrow b \rightarrow^* d$ but $a \downarrow d$ does not hold).

Method. ***Noetherian induction (c.f. [BO93])***
Let $\rightarrow$ be a reduction relation on S. The following is called *principle of Noetherian induction* for a predicate P on S:

$$\forall a \in S : ((\forall b \in S : a \rightarrow b \Rightarrow P(b)) \Rightarrow P(a)) \Rightarrow \forall c \in S : P(c)$$

Proposition 2 (Newman's lemma) *If $\rightarrow$ is terminating then it is confluent iff it is locally confluent.*

Proof. (idea cf. [New42])
$\Rightarrow$) obvious
$\Leftarrow$) Noetherian induction applied to $P(a) :\Leftrightarrow \forall b, c \in S : ((a \rightarrow^* b \wedge a \rightarrow^* c) \Rightarrow \exists d \in S : b \rightarrow^* d \leftarrow^* c)$ where the non-trivial inductive step of the proof is shown in Fig. 2.3.

Remark. Newman's lemma is sometimes also referred to as the *diamond lemma* because of the shape of the proof.

Proposition 3 (Unique normal forms) *If $\rightarrow$ is terminating and locally confluent then every $a \in S$ has a unique normal form $\check{a}$.*

Proof. (sketch)
i) because of Newman's lemma $\rightarrow$ is confluent
ii) because of termination all reductions starting at s terminate
iii) because of confluence all reductions terminate with the same $\check{s}$

Remark. Terminating and confluent reduction relations are also called *complete* or *canonical* .

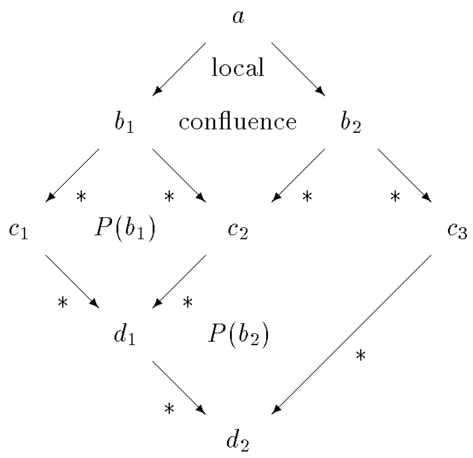

Figure 2.3: Diamond Lemma

2.1.2 Term Rewriting Systems

For all the details about syntax and semantics of term rewriting systems the reader is pointed to the literature e.g. the overview in [DJ90]. In the following only the essential facts are sketched.

Notion 4 (Signature) *A* **signature** $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ *is given by:*

variables *a countably infinite set* $\mathcal{V}$*,*

function symbols *a non-empty set* $\mathcal{F}$ *where* $\mathcal{V} \cap \mathcal{F} = \emptyset$*, and*

arity *a function* $ar : \mathcal{F} \to \mathbb{N}_0$*.*

Remark. 0-ary function symbols are called *constants*. Sometimes the existence of at least one constant is assumed [DJ90, page 249].

Notion 5 (Terms) *The set of* **terms** $\mathcal{T}$ *over a signature* $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ *is the smallest set with:*

1. $\mathcal{V} \subseteq \mathcal{T}$ *and*
2. *if* $f \in \mathcal{F}$ *and* $ar(f) = n$ *and* $t_1 \ldots t_n \in \mathcal{T}$ *then* $f(t_1, \ldots, t_n) \in \mathcal{T}$*.*

Remark. Terms which do not contain any variable are called *ground terms.*

Notion 6 (Syntactical equality, Equation, Rewrite rule) *Let* $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ *be a signature. Then there are the following notions:*

syntactical equality $\equiv$ *as the equality of elements in* $\mathcal{T}$*,*

an equation *as an unordered pair of two terms* $\{s, t\}$*,*

equations E *as a set of equations,*

a rewrite rule *as an ordered pair of two terms* (s, t)*,*

rewrite rules R *as a set of rewrite rules.*

Remark. $s =_E t$ means that there is an equation $\{s, t\} \in E$ (or $\{t, s\} \in E$) and $s \rightarrow_R t$ means there is a rewrite rule $(s, t) \in R$. A pair (Σ, E) is called *equational specification* and a pair (Σ, R) is called *term rewriting system*. Usually rules $l \rightarrow_R r$ satisfy the two constraints:
(1) $l \notin \mathcal{V}$ (i.e. l is not a variable) and
(2) $Var(l) \supseteq Var(r)$ (i.e. every variable in r occurs in l).

Notion 7 (Substitution, Instantiation, Context) *Let* $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ *be a signature. Then*

$\sigma : \mathcal{V} \rightarrow \mathcal{T}$ *denotes a substitution as a mapping leaving almost every variable fixed,*

t^σ *denotes the instantiation of* $t \in \mathcal{T}$ *with the substitution* σ *where* σ *is applied to all the variables in the term resulting in a new term,*

$C[t]$ *denotes the context* C *of the term* $t \in \mathcal{T}$ *where we will use* λ*-abstraction to denote a context such that it can be applied to the term* t *as* $C[t] \equiv (C\ t)$*.*

Remark. Substitutions acting only on one variable $v \in \mathcal{V}$ are also denoted as $[v := t]$. The *composition of two substitutions* σ and θ is denoted by concatenation $\sigma\theta$. A *renaming* substitution is a bijection $\rho : \mathcal{V} \rightarrow \mathcal{V} \subset \mathcal{T}$ which is a special kind of substitution.

Definition 8 (Equational Deductive System, Derivable Equality) *Let* E *be equations over a signature* Σ*. Then the axiom of* **reflexivity** $t = t$ *together with the axioms* $s = t$ *for all equations* $s =_E t$ *and the following rules of inference determine* **the equational deductive system** *for* E*:*

$$\frac{E \vdash t_1 = t_2}{E \vdash t_2 = t_1} \qquad \textit{symmetry}$$

$$\frac{E \vdash t_1 = t_2\ ,\ E \vdash t_2 = t_3}{E \vdash t_1 = t_3} \qquad \textit{transitivity}$$

$$\frac{E \vdash t_1 = t_1'\ ,\ E \vdash t_2 = t_2'}{E \vdash t_1[x := t_2] = t_1'[x := t_2']} \qquad \textit{replacement}$$

Two terms s *and* t *are syntactically derivable equal if the statement* $s = t$ *can be derived within the equational deductive system which is denoted as* $E \vdash s = t$ *(or more precisely* $(\Sigma, E) \vdash s = t$*).*

Remark. In the following theorem $\mathcal{M}od(E) \models s \sim t$ means that the semantical equivalence $s \sim t$ is valid in all models of E. Since we will concentrate on the syntactic side we do not give the details of semantics. They can be found in e.g. [Ebb86].

Theorem 4 (Birkhoff's Completeness Theorem) *For any set of equations E and terms $s, t \in \mathcal{T}$ over a signature Σ holds: $\mathcal{M}od(E) \models s \sim t$ iff $E \vdash s = t$.*

Remark. Birkhoff's theorem [Bir35] states the equivalence between syntactic and semantic proving. Although it is not possible to handle all models in a computer it is possible to do simple syntactic proofs.

Problem. ***Validity or uniform word problem for*** (Σ, E)
Given: the statement $s = t$ concerning two terms from $\mathcal{T}$
Decide: $E \vdash s = t$ or $E \not\vdash s = t$

Remark. This problem is not in general solvable recalling the *halting problem of Turing machines*. Here the attention is focused on validity problems which are decidable. Furthermore a better solution than enumerating all equational consequences is needed.

Definition 9 (Rewrite relation) *Let R be a set of rewrite rules over a signature Σ. The* **rewrite relation** $\longrightarrow_R$ *is a reduction relation on $\mathcal{T}$, such that $s \longrightarrow_R t$ iff there is a context C and a substitution σ and a rewrite rule $l \rightarrow_R r$ such that $C[l^\sigma] \equiv s \longrightarrow_R t \equiv C[r^\sigma]$.*

Remark. $(\Sigma, \longrightarrow_R)$ is called a *term rewriting system* or shortly TRS.

Method. ***Transform an equational specification into a TRS***
Given: an equational specification (Σ, E)
Try to find: a term rewriting system (Σ, R) such that:
(a) R is a finite set,
(b) $\longrightarrow_R$ is terminating,
(c) $\longrightarrow_R$ is Church-Rosser,
(d) $\longrightarrow_R$ is sound and adequate ($=$ is the same as $\longleftrightarrow_R^*$).
This gives immediately a decision procedure for $=$, since in this case we have obviously $s = t \iff \check{s} \equiv \check{t}$ where $\check{s}$ and $\check{t}$ are the normal forms of s and t with respect to $\longrightarrow_R$.

Remark. From these conditions (a) is easy to check and in the following we will assume that condition (d) is always satisfied (i.e. that we remain within conservative extensions of E). Actually one already starts with a finite set E of equations. This method is not the only one since sometimes it is convenient to modify the signature Σ, too. The attention is focused on (b) and (c).

Remark. Now a method for proving termination is presented. But it is undecidable in general whether a given term rewriting system terminates.

Definition 10 (Terminating function) *A* **terminating function** ϕ *from the set of terms $\mathcal{T}$ to a partially ordered set $(\mathcal{W}, \prec)$ is defined by*

1. *a set of functions $f_\phi : \mathcal{W}^n \rightarrow \mathcal{W}$ where there is one f_ϕ for every function symbol f and $ar(f) = n$ with*

2. *$\phi(f(t_1,\ldots,t_n)) = f_\phi(\phi(t_1),\ldots,\phi(t_n))$ for every term $f(t_1,\ldots,t_n) \in \mathcal{T}$ fulfilling*

3. *the* monotonicity condition *that $x \prec y$ implies $f_\phi(\cdots x \cdots) \prec f_\phi(\cdots y \cdots)$.*

Proposition 5 (Termination criterion) *A term rewriting system (Σ, R) where:*

1. *there is a set $(\mathcal{W}, \prec)$ with a well-founded ordering and*

2. *there is a terminating function $\phi : \mathcal{T} \to \mathcal{W}$ such that*

3. *$\phi(r^\sigma) \prec \phi(l^\sigma)$ for each rule $l \to_R r$ and for any substitution σ of ground terms for the variables of that rule*

is terminating.

Remark. With the help of this criterion several methods for proving termination can be justified. More details can be found in [Der87, pages 77–78] or [Lan75]. Here only one of those is presented: the method of polynomial interpretations (see also [Lan79]).

Method. ***Polynomial interpretations***
Given: a finite term rewriting system (Σ, R)
Try to find: a polynomial interpretation $\mathcal{I}$ assigning integer polynomials in n variables to every n-ary function symbol such that:
(i) $\mathcal{I}(t) \in \mathbb{N}^+$ and
(ii) the monotonicity condition is satisfied, i.e.
$z < z'$ implies $\mathcal{I}(f)(\cdots z \cdots) < \mathcal{I}(f)(\cdots z' \cdots)$.
In that case $\mathcal{I}$ is a terminating function from $\mathcal{T}$ to $(\mathbb{N}^+, <)$. The monotonicity condition is easily satisfied when all coefficients are positive. To check termination it has to be verified that the polynomial $\mathcal{I}(l) - \mathcal{I}(r)$ is positive for all rules $l \to_R r$ and all values of variables greater than the minimal value of the interpretation of a ground term.

Example. Let $R = \{(x;y);z \to x;(y;z)\}$ over the signature with only one function symbol ";" having arity 2. A possible polynomial interpretation to prove the termination of R is $\mathcal{I}$ where $\mathcal{I}(t_1;t_2) := 2 * \mathcal{I}(t_1) + \mathcal{I}(t_2)$. In that case we can compute:

$$\begin{aligned}
\mathcal{I}((x;y);z) - \mathcal{I}(x;(y;z)) &= 2 * \mathcal{I}(x;y) + \mathcal{I}(z) - [2 * \mathcal{I}(x) + \mathcal{I}(y;z)] \\
&= 4 * \mathcal{I}(x) + 2 * \mathcal{I}(y) + \mathcal{I}(z) \\
&\quad -[2 * \mathcal{I}(x) + 2 * \mathcal{I}(y) + \mathcal{I}(z)] \\
&= 2 * \mathcal{I}(x)
\end{aligned}$$

Since $\mathcal{I}(x) > 0$ due to the chosen domain of interpretation this expression is always greater than 0 which proves termination.

Remark. As the last step a method for proving the Church-Rosser property is presented. Also this is not a decidable property of term rewriting systems. But it becomes decidable in case of terminating term rewriting systems.

Remark. Before the important notion of a critical pair can be introduced another technical notion is required: the most general unifier.

Definition 11 (Unifier and most general unifier) **A unifier** *of two terms s and t is a substitution $\varsigma = \rho\sigma$ where:*

1. *ρ is a renaming substitution such that $Var(s^\rho) \cap Var(t^\rho) = \emptyset$ and*
2. *σ is a substitution with $s'^\sigma \equiv t'^\sigma$ and $s' \equiv s^\rho$ and $t' \equiv t^\rho$,*

A most general unifier *of two terms s and t is a substitution ς where:*

1. *ς is a unifier and*
2. *for all unifiers ς' there is a substitution θ such that $\varsigma' = \varsigma\theta$.*

Remark. A most general unifier ς of two terms s and t is denoted as $mgu(s,t)$ which is actually an equivalence class of substitutions up to renaming. More details can be found in [Llo87, pages 20–26].

Definition 12 (Critical pair) *Let $l_1 \rightarrow_R r_1$ and $l_2 \rightarrow_R r_2$ be renamed versions of two rewrite rules (possibly the same) of a term rewriting system (Σ, R) having no variables in common. If*

1. *$l_1 \equiv C[st]$ where $st \in \mathcal{T} \setminus \mathcal{V}$ is a subterm of l_1 and*
2. *st and l_2 are unifiable with $\varsigma = mgu(st, l_2)$*

then the pair of reducts $(C[r_2]^\varsigma, r_1^\varsigma)$ is called a **critical pair**.

Remark. This is the key notion to formulate the crucial result which allows a finite test of the Church-Rosser property for a given terminating term rewriting system. The source reference is [KB70] whereas the full proof of the following lemma can be found in [Hue81]. A general review of this method is given in [Buc87].

Proposition 6 (Critical Pair Lemma) *A term rewriting system is locally confluent iff all critical pairs converge.*

Remark. A *critical overlap* is shown in Fig. 2.4 leading to the critical pair consisting of the terms to the left and the right. Since we assumed finitely many rewriting rules there is only a finite number of critical pairs because every left hand side of a rewrite rule has only a finite number of subterms. Therefore this is a finite test.

Example. As an example we check the critical pairs of $R = \{(x;y);z \rightarrow x;(y;z)\}$. This shows that a rule can be overlapped by itself. In correspondence with the previous notation we have:

$$\begin{aligned} l_1 &:= (x;y);z \\ r_1 &:= x;(y;z) \\ l_2 &:= (x';y');z' \\ r_2 &:= x';(y';z') \end{aligned}$$

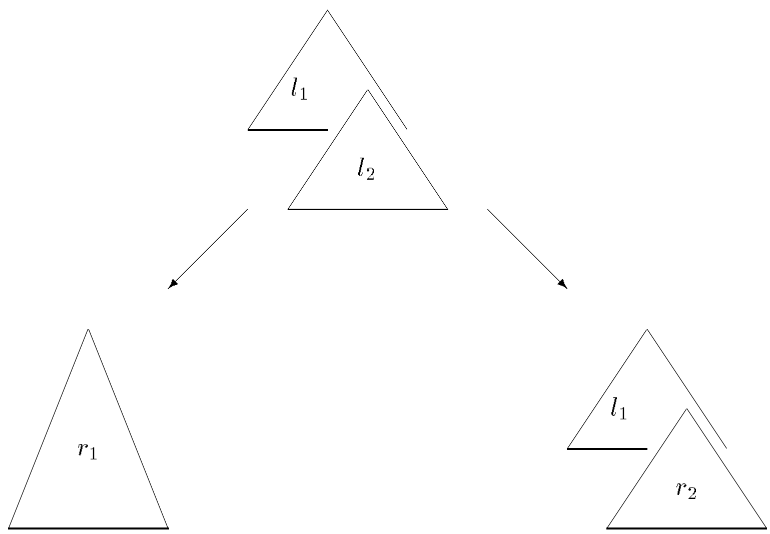

Figure 2.4: Critical Overlap

as renamed versions of two rewrite rules. Furthermore we have:

$$\begin{array}{rcl} C & := & (\lambda v.v; z) \\ st & := & (x; y) \\ l_1 & \equiv & C[st] = (\lambda v.v; z)(x; y) \\ mgu(st, l_2) & = & [x := x'; y'][y := z'] \end{array}$$

So that finally the pair of reducts is:

$$\begin{array}{rcl} C[r_2]^{mgu(st,l_2)} & = & (x'; (y'; z')); z \\ r_1^{mgu(st,l_2)} & = & (x'; y'); (z'; z) \end{array}$$

This pair of reducts is convergent as can be seen from:

$$\begin{array}{lcccl} (x'; (y'; z')); z & \longrightarrow_R & x'; ((y'; z'); z) & \longrightarrow_R & x'; (y'; (z'; z)) \\ & & & & \equiv \\ (x'; y'); (z'; z) & & \longrightarrow_R & & x'; (y'; (z'; z)) \end{array}$$

There is only one further critical pair which results when $C := \lambda v.v$. This type of overlap of a rule with itself is also called *overlay*. But convergence in that case is trivial. Altogether this means that R actually is a canonical system because we proved termination before.

Method. ***Knuth-Bendix completion***
Input: an equational specification (Σ, E) together with a terminating function ϕ
Output: an equivalent term rewriting system (Σ, R) or failure
Description of the algorithm as inference system:

$$
\begin{array}{lcl}
\text{Deduction} & \dfrac{E; R}{E \cup CP(R); R} & \text{add critical pairs} \\
\text{Orientation} & \dfrac{E \cup \{s = t\}; R}{E; R \cup \{s \to t\}} & \text{if } \phi(t) \prec \phi(s) \\
\text{Deletion} & \dfrac{E \cup \{s = s\}; R}{E; R} & \\
\text{Simplification} & \dfrac{E \cup \{s = u\}; R}{E \cup \{t = u\}; R} & \text{if } s \to t \\
\text{Composition} & \dfrac{E; R \cup \{u \to s\}}{E; R \cup \{u \to t\}} & \text{if } s \to t
\end{array}
$$

The algorithm may not necessarily terminate. Non-termination may be due to a successive generation of infinitely many critical pairs. Failure occurs when a remaining equation cannot be oriented into a rewrite rule with the help of the terminating function. The kind of presentation as an inference system is due to [Bac91].

2.1.3 Rewriting for Categorical Notions

We describe how we approach categorical notions with the help of term rewriting. Our attempts are assisted by the Larch Prover [GG91] and the logic programming language Elf [Pfe89, Pfe91, Pfe94]. The idea how to treat the type information of morphisms in the frame of rewriting is due to Huet [Hue90a].
Remark. Morphisms in a category come equipped with an additional type information. This are the source and target object also called domain and codomain. For a first test it is sufficient to try the rewriting without this type information. A final decision procedure in form of a canonical rewrite system has to be checked also with respect to the types.

Problem. ***Universal word problem for a categorical notion***
Given: an equational specification for a categorical notion describing the equalities of morphisms
Try to find: a canonical term rewriting system such that the universal word problem of this categorical notion can be solved

Method. ***Achieving a canonical system for categorical notions***
We carry out the following steps:

1. ignore the type information and code the equations which hold for morphisms
2. find a suitable orientation of the equations into rules
3. check the local confluence of that system or try to complete the system
4. try to prove termination of the locally confluent system without the additional type information
5. code additionally the type information and check the local confluence again of the resulting system

Remark. To achieve local confluence is the easier part since with the help of the Knuth-Bendix completion this is rather a purely syntactical check. The harder part is often the proof of termination which sometimes requires special techniques where the method by interpretation is a semantical one. Now we give an example how the type information can be coded in a term rewriting system.

```
declare sort Mor                        % morphisms
declare variables f, g: Mor
declare operators
        id: -> Mor                      % identity
        *: Mor, Mor -> Mor              % composition
```

Table 2.1: Untyped Declarations in LP

Method. ***Coding type information in a term rewriting system***
This method was already sketched by Huet in [Hue90a]. The trick is to introduce an auxiliary function symbol e.g. "*mor*" which makes the typing explicit in that it takes three arguments: an untyped morphism together with source and target object. We present this method by an example.

```
declare sort Mor                        % typed morphisms
declare sort Mor´                       % untyped morphisms
declare sort Obj                        % objects
declare variables f, g: Mor´
declare variables o1, o2, o3: Obj
declare operators
        id: -> Mor´                     % identity
        *: Mor, Mor -> Mor´             % composition
        mor: Obj, Mor´, Obj -> Mor      % typing
```

Table 2.2: Typed Declarations in LP

Example. We wish to encode the additional type information for morphisms, their composition, and the identity morphisms for an object. The comparison between the untyped and the typed coding is presented in Tab. 2.3 and the corresponding declarations are shown in Tab. 2.1 and 2.2 We use the language of the Larch Prover for the presentation.

Remark. With the help of this method all our systems are treated. First we work with the support of the Larch Prover and all the traces appear in the Appendix A. The Larch Prover provides methods to check critical pairs and to do completion. Additionally it is able to verify polynomial interpretations. Since the coding of the object information in the frame of term rewriting is rather complicated we provide a final implementation of the decision procedures in the programming language Elf which is based on the "Logical Framework" (LF)

morphism	untyped	typed
$o1 \xrightarrow{f} o2$	f	$mor(o2, f, o1)$
$o1 \xrightarrow{id} o1$	id	$mor(o1, id, o1)$
$o1 \xrightarrow{f} o2 \xrightarrow{g} o3$	$g * f$	mor(o3, mor(o3, g, o2) * mor(o2, f, o1), o1)

Table 2.3: Coding of the Type Information

[HHP93] which supports dependent types cf. Appendix B.

Example. For comparison we give the static definition for the type checking of morphisms in Elf cf. Tab. 2.4. Since Elf allows dependent types morphisms can be represented as a type indexed by two objects. This representation forces already well typed input.

```
%%% morphisms as dependent types
obj : type.
mor : obj -> obj -> type.
id  : {O1 : obj} mor O1 O1.
*   : mor O2 O3 -> mor O1 O2 -> mor O1 O3.
```

Table 2.4: Type Information in Elf

Remark. It will turn out that for the work with categorical notions related to monads it is not easy to find a canonical rewriting system. It is essential to reformulate given equational specifications. This fact is not surprising. Cartesian closed categories [Hue90a] are a well know example of a decidable notion where no canonical rewriting system is known in the original presentation. One possible decision procedure proceeds by a translation into a simply typed lambda calculus. Here the type information is absolutely essential in that it guarantees termination.

2.2 Basic Categorical Notions

2.2.1 Categories

There are several definitions of a category in the literature. For an easily understandable introduction see [Pie91]. Here this fundamental notion is introduced only briefly and we refer to the literature for further details.

Notion 13 (Category) *A* **category** C *is*

- *a class of* **objects** $O = Obj(C)$ *and* **morphisms** $M = Mor(C)$.
- *Every morphism* m *has an object as its domain* $dom(m)$ *and co-domain* $cod(m)$ *which pictorially is expressed as* $m : dom(m) \to cod(m)$.

- *For all objects A there is an* **identity morphism** id_A *with* $dom(id_A) = cod(id_A) = A$.
- *Morphisms f, g can be composed by an operation "$\circ_C$" if $dom(f) = cod(g)$, for the* **composition** *if holds* $dom(f \circ_C g) = dom(g)$ *and* $cod(f \circ_C g) = cod(f)$

where the following axioms are satisfied (assuming $f : A \to B$, and the compatibility of all the morphisms):

$$f \circ_C id_A = f \tag{2.1}$$
$$id_B \circ_C f = f \tag{2.2}$$
$$(f \circ_C g) \circ_C h = f \circ_C (g \circ_C h) \tag{2.3}$$

Remark. We will work with *small categories* where O and M are sets. Another way of expressing $dom(f) = A$ and $cod(f) = B$ is $f \in Hom(A, B)$ or $f \in Hom_C(A, B)$ to make the category explicit. Besides the graphical representation $f : A \to B$ there is also $A \xrightarrow{f} B$.

Example. In the category of sets Set objects are sets and morphisms are set theoretic functions. The composition is the usual composition of functions. Identities are the identity functions.

Example. A deductive system is a category where objects are formulas and morphisms are proofs. The composition corresponds to an inference rule. Identities are axioms which state that a proposition is provable from itself.

Example. Partially ordered sets form a category Pos where objects are partially ordered sets and morphisms are monotone mappings. The composition is the composition of mappings. Identities are the identity functions.

Remark. In the following considerations the fundamental problem will always be to decide the equality of two morphisms in all categories satisfying some given laws. Of course this problem is not always decidable. The emphasis from now on will be on equations more specifically on typed equations where the types are introduced by the domain and co-domain of the morphisms.

Lemma 7 (CAT1) *There is the following canonical system for a category:*

$$\begin{aligned} f \circ id_A &\longrightarrow f \\ id_B \circ f &\longrightarrow f \\ (f \circ g) \circ h &\longrightarrow f \circ (g \circ h) \end{aligned}$$

Proof.
i) Local confluence can be checked by the Knuth-Bendix procedure.
ii) Termination follows from the following polynomial interpretation Pi_{cat1} over $(\mathbb{N}^+, <)$:

$$\begin{aligned} Pi_{cat1}(id_A) &= 1 \\ Pi_{cat1}(f \circ g) &= 2 * Pi_{cat1}(f) + Pi_{cat1}(g) \end{aligned}$$

This can be verified in Appendix A.1.1. □

Remark. Instead of grouping parenthesis to the right they also can be grouped to the left:

Lemma 8 (CAT2) *There is the following canonical system for a category:*

$$\begin{array}{rcl} f \circ id_A & \longrightarrow & f \\ id_B \circ f & \longrightarrow & f \\ f \circ (g \circ h) & \longrightarrow & (f \circ g) \circ h \end{array}$$

Proof.
i) Local confluence can be checked by the Knuth-Bendix procedure.
ii) Termination follows from the following polynomial interpretation Pi_{cat2} over $(\mathbb{N}^+, <)$:

$$\begin{array}{rcl} Pi_{cat2}(id_A) & = & 1 \\ Pi_{cat2}(f \circ g) & = & Pi_{cat2}(f) + 2 * Pi_{cat2}(g) \end{array}$$

This can be verified in Appendix A.1.2. □

Remark. One of the most popular references for category theory is [ML71] which provides lots of examples in greater detail than it is possible here.

2.2.2 Functors

Definition 14 (Functor) *Let C and D be categories. A* **functor** $F : C \to D$ *is an assignment*

- *taking C-objects A to D-objects $F(A)$ and*
- *taking C-morphisms $f : A \to B$ to D-morphisms $F(f) : F(A) \to F(B)$,*

where the following axioms are satisfied (assuming the compatibility of morphisms):

$$F(id_A) = id_{F(A)} \tag{2.4}$$

$$F(f \circ_C g) = F(f) \circ_D F(g) \tag{2.5}$$

Remark. Sometimes this notion is also called a *covariant* functor in order to be distinguished from a *contravariant* functor. A contravariant functor $G : C \to D$ is a covariant functor $G : C^{op} \to D$ where C^{op} is the *opposite* or *dual* category (where all morphisms have reversed domain and co-domain) i.e. G takes C-morphisms $f : A \to B$ to D-morphisms $G(f) : G(B) \to G(A)$ and the following two axioms hold:

$$\begin{array}{rcl} G(id_A) & = & id_{G(A)} \\ G(f \circ_C g) & = & G(g) \circ_D G(f) \end{array}$$

Example. One of the most simple functors is the identity functor $Id_C : C \to C$ leaving everything unchanged. This is an example for an *endofunctor* from a category C to itself.

Example. The category of all small categories has small categories as objects and functors as morphisms. The composition is defined to be $F \circ G(\_) := F(G(\_))$ for objects and morphisms. Identities are the identity functors.

Example. Another important example of a functor is the forgetful functor U (like underlying). One instance of it is the functor between partially ordered sets and sets ($U : Pos \to Set$). Partially ordered sets are mapped to the underlying sets and morphisms are mapped to the corresponding underlying functions between sets.

Lemma 9 (FUN1) *There is the following canonical system for a functor $F : C \to C'$:*

$$\begin{array}{rcl} f \circ id_A & \longrightarrow & f \\ id_B \circ f & \longrightarrow & f \\ (f \circ g) \circ h & \longrightarrow & f \circ (g \circ h) \\ f \circ' id_{A'} & \longrightarrow & f' \\ id_{B'} \circ' f' & \longrightarrow & f' \\ (f' \circ' g') \circ' h' & \longrightarrow & f' \circ' (g' \circ' h') \\ F(id_A) & \longrightarrow & id_{F(A)} \\ F(f \circ g) & \longrightarrow & F(f) \circ' F(g) \end{array}$$

Proof.
i) Local confluence can be checked by the Knuth-Bendix procedure.
ii) Termination follows from the following polynomial interpretation Pi_{fun1} over $(\mathbb{N}^+, <)$:

$$Pi_{fun1}(id_A) = Pi_{fun1}(id_{A'}) = 1$$

$$\begin{array}{rcl} Pi_{fun1}(F(h)) & = & 2 * Pi_{fun1}(h) \\ Pi_{fun1}(f \circ g) & = & 2 * Pi_{fun1}(f) + Pi_{fun1}(g) + 1 \\ Pi_{fun1}(f' \circ' g') & = & 2 * Pi_{fun1}(f') + Pi_{fun1}(g') + 1 \end{array}$$

This can be verified in Appendix A.2.1. □

Remark. Instead of splitting all compositions inside a functor one can also combine as many compositions as possible under a common functor symbol as the following system shows.

Lemma 10 (FUN2) *There is the following canonical system for a functor $F : C \to C'$:*

$$f \circ id_A \longrightarrow f$$

$$\begin{array}{rcl} id_B \circ f & \longrightarrow & f \\ (f \circ g) \circ h & \longrightarrow & f \circ (g \circ h) \\ f \circ' id_{A'} & \longrightarrow & f' \\ id_{B'} \circ' f' & \longrightarrow & f' \\ (f' \circ' g') \circ' h' & \longrightarrow & f' \circ' (g' \circ' h') \\ F(id_A) & \longrightarrow & id_{F(A)} \\ F(f) \circ' F(g) & \longrightarrow & F(f \circ g) \\ F(f) \circ' (F(g) \circ' h') & \longrightarrow & F(f \circ g) \circ' h' \end{array}$$

Proof.
i) Local confluence can be checked by the Knuth-Bendix procedure.
ii) Termination follows from the following polynomial interpretation Pi_{fun2} over $(\mathbb{N}^+, <)$:

$$Pi_{fun2}(id_A) = Pi_{fun2}(id_{A'}) = 1$$

$$\begin{array}{rcl} Pi_{fun2}(F(h)) & = & Pi_{fun2}(h) + 1 \\ Pi_{fun2}(f \circ g) & = & Pi_{fun2}(f) * Pi_{fun2}(g) + P_{fun2}(f) \\ Pi_{fun2}(f' \circ' g') & = & Pi_{fun2}(f') * Pi_{fun2}(g') + P_{fun2}(f') \end{array}$$

This can be verified in Appendix A.2.2. □

Remark. The same systems work for an endofunctor since in that case the rewrite rules as well as the interpretation coincide. Furthermore one easily obtains two similar systems when associativity is oriented the other way.

2.2.3 Natural Transformations

Definition 15 (Natural Transformation) *Let C and D be categories and $F, G : C \to D$ functors. A* **natural transformation** *$\tau : F \to G$ is defined by assigning to every C-object A a D-morphism $\tau_A : F(A) \to G(A)$, where the following law is satisfied (assuming $f : A \to B$):*

$$\tau_B \circ_D F(f) \quad = \quad G(f) \circ_D \tau_A \tag{2.6}$$

If all τ_A are isomorphisms then τ is called natural isomorphism (or natural equivalence) .

Example. Take the functor category B^A of two given categories A and B where the objects are functors $F : A \to B$ and the morphisms are natural transformations. The composition is defined to be $(\tau \circ \tau')_A := \tau_A \circ \tau'_A$ for all objects of A. Identities are the identity natural transformations $ID(A) := id_{F(A)}$ for the functor F.

Example. The determinant is a natural transformation $det_{(R,n)} : Im_n(R) \to Un(R)$ where $Im_n(R)$ are the invertible $n \times n$ matrices with coefficients from the commutative ring R and $Un(R)$ are the units (i.e. invertible elements) of the

$$\begin{tikzcd} F(A) \arrow[r, "\tau_A"] \arrow[d, "F(f)"'] & G(A) \arrow[d, "G(f)"] \\ F(B) \arrow[r, "\tau_B"'] & G(B) \end{tikzcd}$$

Figure 2.5: Natural Transformation

ring R and $Im_n(R), Un(R) : CommutativeRing \rightarrow Group$.

Example. A finite-dimensional vector space is naturally equivalent to its double dual. This can be seen from the natural equivalence τ for the functors $Id : FinVec \rightarrow FinVec$ and $(Dual \circ Dual) : FinVec \rightarrow FinVec$ where $FinVec$ is the category of finite-dimensional vector spaces.

Remark. The concept of a natural transformation can be nicely visualized by a diagram cf. Fig. 2.5.

Lemma 11 (NAT1) *There is the following canonical system for a natural transformation* $\tau : F \rightarrow G$ *where* $F, G : C \rightarrow C'$*:*

$$\begin{array}{rcl}
f \circ id_A & \longrightarrow & f \\
id_B \circ f & \longrightarrow & f \\
(f \circ g) \circ h & \longrightarrow & f \circ (g \circ h) \\
f \circ' id_{A'} & \longrightarrow & f' \\
id_{B'} \circ' f' & \longrightarrow & f' \\
(f' \circ' g') \circ' h' & \longrightarrow & f' \circ' (g' \circ' h') \\
F(id_A) & \longrightarrow & id_{F(A)} \\
F(f \circ g) & \longrightarrow & F(f) \circ' F(g) \\
G(id_A) & \longrightarrow & id_{G(A)} \\
G(f \circ g) & \longrightarrow & G(f) \circ' G(g) \\
\tau_B \circ' F(f) & \longrightarrow & G(f) \circ' \tau_A \\
\tau_B \circ' (F(f) \circ' h') & \longrightarrow & G(f) \circ' (\tau_A \circ' h')
\end{array}$$

Proof.
i) Local confluence can be checked by the Knuth-Bendix procedure.
ii) Termination follows from the following polynomial interpretation Pi_{nat1} over $(\mathbb{N}^+, <)$:

$$Pi_{nat1}(id_A) = Pi_{nat1}(id_{A'}) = Pi_{nat1}(\tau_A) = 1$$

$$Pi_{nat1}(f \circ g) \quad = \quad 2 * Pi_{nat1}(f) + Pi_{nat1}(g) + 1$$

$$\begin{aligned}
Pi_{nat1}(f' \circ' g') &= 2 * Pi_{nat1}(f') + Pi_{nat1}(g') + 1 \\
Pi_{nat1}(F(f)) &= 6 * Pi_{nat1}(f) \\
Pi_{nat1}(G(g)) &= 2 * Pi_{nat1}(g)
\end{aligned}$$

This can be verified in Appendix A.3.1. □

Remark. Instead of splitting all compositions inside a functor one can also combine as many compositions as possible under a common functor symbol as the following systems shows.

Lemma 12 (NAT2) *There is the following canonical system for a natural transformation $\tau : F \to G$ where $F, G : C \to C'$:*

$$\begin{aligned}
f \circ id_A &\longrightarrow f \\
id_B \circ f &\longrightarrow f \\
(f \circ g) \circ h &\longrightarrow f \circ (g \circ h) \\
f \circ' id_{A'} &\longrightarrow f' \\
id_{B'} \circ' f' &\longrightarrow f' \\
(f' \circ' g') \circ' h' &\longrightarrow f' \circ' (g' \circ' h') \\
F(id_A) &\longrightarrow id_{F(A)} \\
F(f) \circ' F(g) &\longrightarrow F(f \circ g) \\
F(f) \circ' (F(g) \circ' h') &\longrightarrow F(f \circ g) \circ' h' \\
G(id_A) &\longrightarrow id_{G(A)} \\
G(f) \circ' G(g) &\longrightarrow G(f \circ g) \\
G(f) \circ' (G(g) \circ' h') &\longrightarrow G(f \circ g) \circ' h' \\
\tau_B \circ' F(f) &\longrightarrow G(f) \circ' \tau_A \\
\tau_B \circ' (F(f) \circ' h') &\longrightarrow G(f) \circ' (\tau_A \circ' h')
\end{aligned}$$

Proof.
i) Local confluence can be checked by the Knuth-Bendix procedure.
ii) Termination follows from the following polynomial interpretation Pi_{nat2} over $(\mathbb{N}^+, <)$:

$$Pi_{nat2}(id_A) = Pi_{nat2}(id_{A'}) = Pi_{nat2}(\tau_A) = 1$$

$$\begin{aligned}
Pi_{nat2}(f \circ g) &= Pi_{nat2}(f) * Pi_{nat2}(g) + Pi_{nat2}(f) \\
Pi_{nat2}(f' \circ' g') &= Pi_{nat2}(f') * Pi_{nat2}(g') + Pi_{nat2}(f') \\
Pi_{nat2}(F(f)) &= 3 * Pi_{nat2}(f) + 1 \\
Pi_{nat2}(G(g)) &= Pi_{nat2}(g) + 1
\end{aligned}$$

This can be verified in Appendix A.3.2. □

Remark. The same systems work for endofunctors. Furthermore the last two equations could be reversed resulting in another canonical system where the interpretations of F and G have to be swapped.

Chapter 3

Adjunctions

In this chapter we define the categorical notion of an adjunction. In a book by Curien [Cur93] a canonical system for adjunctions was given without a proof. We verify this system and prove termination with a special technique adapted from the one used by Hardin/Laville to prove the termination of system SUBST [HL86]. This system is related to a special adjunction between products and exponentiation.

Motivation. There are two reasons to have a look at the notion of an adjunction. On the one hand they are just the next more complicated categorical notion in the line presented in the previous chapter. On the other hand they will turn out to be useful for the study of monads due to some equivalence results.

3.1 Definition

Several equivalent definitions of an adjunction are given. For comparison see [ML71] or [Pie91]. We assume throughout that the morphisms between two objects form a set.

Definition 16 (Adjunction) *Let C and C' be categories and $L : C \to C'$ and $R : C' \to C$ be functors. L and R are* **adjoint** *$L \dashv R$ if there is a bijection $\zeta_{X,Y} : Hom_{C'}(L(X), Y) \to Hom_C(X, R(Y))$ for all objects $X \in Obj(C)$ and $Y \in Obj(C')$ where the following law is satisfied:*

$$\textbf{(Adj)} \quad \zeta(f \circ' g \circ' L(h)) \quad = \quad R(f) \circ \zeta(g) \circ h$$

assuming $f \in Hom_{C'}(A', B')$, $g \in Hom_{C'}(L(B), A')$, $h \in Hom_C(A, B)$.

Remark. This law can also be split into two equations which are equivalent:

$$\begin{aligned} \zeta(f \circ' g) &= R(f) \circ \zeta(g) \\ \zeta(g \circ' L(h)) &= \zeta(g) \circ h \end{aligned}$$

Furthermore, since $\zeta_{X,Y}$ is a bijection the following equations are implicitly obtained assuming $f \in Hom_{C'}(L(X), Y)$, $g \in Hom_C(X, R(Y))$.

$$\begin{aligned} \zeta^{-1}(\zeta(f)) &= f \\ \zeta(\zeta^{-1}(g)) &= g \end{aligned}$$

where $\zeta^{-1}_{X,Y} : Hom_C(X, R(Y)) \to Hom_{C'}(L(X), Y)$ denotes the inverse of ζ.

For describing an adjunction the following notation is in use:

$$\frac{A \to R(B)}{L(A) \to B}$$

Note. The constructors representing an adjunction on the level of morphisms are $(id, id', L, R, \zeta, \zeta^{-1}, \circ, \circ')$ in this definition. With the following lemma the number of constructors can be reduced by introducing a new constant $\varepsilon_Z := \zeta^{-1}(id_{R(Z)})$ for all $Z \in Obj(C')$.

Lemma 13 *An adjunction can be completely characterized by $(id, id', \varepsilon, L, \zeta, \circ, \circ')$ on the level of morphisms assuming it is known how R acts on objects.*

Proof. cf. [Cur93]
The missing components can be expressed as:

$$\begin{aligned} R(f) &:= \zeta(f \circ' \varepsilon_{A'}) \\ \zeta^{-1}(g) &:= \varepsilon_{B'} \circ' L(g) \end{aligned}$$

where $f \in Hom_{C'}(A', B')$ and $g \in Hom_C(A, R(B'))$. □

Remark. In the literature one often can find another characterization of an adjunction with the help of one natural transformation and a unique morphism closing a certain diagram or with the help of two natural transformations.

Lemma 14 *The following characterizations of an adjunction are equivalent with the definition above when given the categories C, C' and the functors L, R:*

1. *There is a natural transformation $\eta : I_C \to (R \circ L)$ called* unit *such that for each object $X \in Obj(C)$ and each morphism $f \in Hom_C(X, R(Y))$ there is a unique morphism $\bar{f} \in Hom_{C'}(L(X), Y)$ such that*

$$R(\bar{f}) \circ \eta_X = f \qquad \begin{tikzcd} X \arrow[r, "\eta_X"] \arrow[dr, "f"'] & R(L(X)) \arrow[d, "R(\bar{f})"] \\ & R(Y) \end{tikzcd}$$

2. *There is a natural transformation $\epsilon : (L \circ R) \to I_{C'}$ called* co-unit *such that for each object $Y \in Obj(C')$ and each morphism $g \in Hom_{C'}(L(X), Y)$ there is a unique morphism $\tilde{g} \in Hom_C(X, R(Y))$ such that*

$$\epsilon_Y \circ' L(\tilde{g}) = g \qquad \begin{tikzcd} L(X) \arrow[r, "L(\tilde{g})"] \arrow[dr, "g"'] & L(R(Y)) \arrow[d, "\epsilon_Y"] \\ & Y \end{tikzcd}$$

3. *There are two natural transformations $\eta : I_C \to (R \circ L)$ (the unit) and $\epsilon : (L \circ R) \to I_{C'}$ (the co-unit) such that*

$$\begin{aligned} \epsilon_{L(A)} \circ' L(\eta_A) &= id_{L(A)} \\ R(\epsilon_B) \circ \eta_{R(B)} &= id_{R(B)} \end{aligned}$$

are the identity natural transformations on L and R, respectively.

Proof. cf. [BW90]
$\Rightarrow$)
i) $\eta_X = \zeta(id_{L(X)})$ and $\bar{f} = \zeta^{-1}(f)$
ii) $\epsilon_X = \zeta^{-1}(id_{R(X)})$ and $\tilde{g} = \zeta(g)$
iii) $\eta_X = \zeta(id_{L(X)})$ and $\epsilon_X = \zeta^{-1}(id_{R(X)})$
$\Leftarrow$)
i) $\zeta(f) = R(f) \circ \eta_A$ and $\zeta^{-1}(g) = \bar{id}_{R(X)} \circ' L(g)$
(show that $\bar{id}_{R(X)}$ behaves like ϵ_X)
ii) $\zeta^{-1}(g) = \epsilon_{B'} \circ' L(g)$ and $\zeta(f) = R(f) \circ \tilde{id}_{L(Y)}$
(show that $\tilde{id}_{L(Y)}$ behaves like η_Y)
iii) $\zeta(f) = R(f) \circ \eta_A$ and $\zeta^{-1}(g) = \epsilon_{B'} \circ' L(g)$
The conditions are necessary to prove that ζ and ζ^{-1} are inverse to each other.

Remark. For rewriting we will use only the presentation with $(id, id', \varepsilon, L, \zeta, \circ, \circ')$ from Lemma 13.

3.2 A Canonical System

In this section a canonical system for an adjunction is presented together with the proof of termination and confluence.
Note. Assuming it is known how $R : C' \to C$ acts on objects there is the following rewriting system (ADJ) for an adjunction $L \dashv R$ with $L : C \to C'$ [Cur93]:

$$\begin{aligned}
id_B \circ f_{A \to B} &\longrightarrow f && (3.1)\\
f_{A \to B} \circ id_A &\longrightarrow f && (3.2)\\
(f_{A3 \to A4} \circ g_{A2 \to A3}) \circ h_{A1 \to A2} &\longrightarrow f \circ (g \circ h) && (3.3)\\
id'_{B'} \circ' f'_{A' \to B'} &\longrightarrow f' && (3.4)\\
f'_{A' \to B'} \circ' id'_{A'} &\longrightarrow f' && (3.5)\\
(f'_{A3' \to A4'} \circ' g'_{A2' \to A3'}) \circ' h'_{A1' \to A2'} &\longrightarrow f' \circ' (g' \circ' h') && (3.6)\\
L(id_A) &\longrightarrow id'_{L(A)} && (3.7)\\
L(f_{A2 \to A3}) \circ' L(g_{A1 \to A2}) &\longrightarrow L(f \circ g) && (3.8)\\
L(f_{A3 \to A4}) \circ' (L(g_{A2 \to A3}) \circ' h'_{A1' \to L(A2)}) &\longrightarrow L(f \circ g) \circ' h' && (3.9)\\
\zeta(\varepsilon_{B'} \circ' L(f_{A \to R(B')})) &\longrightarrow f && (3.10)\\
\zeta(\varepsilon_{A'}) &\longrightarrow id_{R(A')} && (3.11)\\
\varepsilon_{B'} \circ' L(\zeta(f'_{L(A) \to B'})) &\longrightarrow f' && (3.12)\\
\varepsilon_{B'} \circ' (L(\zeta(f'_{L(A) \to B'})) \circ' g'_{A' \to L(A)}) &\longrightarrow f' \circ' g' && (3.13)\\
\zeta(f'_{L(B) \to B'}) \circ g_{A \to B} &\longrightarrow \zeta(f' \circ' L(g)) && (3.14)
\end{aligned}$$

Proposition 15 *ADJ is a canonical rewriting system for adjunctions.*

Remark. The rules 1–3 and 4–6 express that C and C' are categories, respectively. Rules 7–9 express that L is a functor. The way of orienting these rules is important. The remaining equations are oriented consequences of the characterization of adjunctions due to Lemma 13. Therefore ADJ is adequate. From rule 11 one can see that it is indeed necessary to know how R acts on objects. It remains to show that this is a canonical system, i.e. it is terminating and confluent.

Proof.
First termination is shown in a similar way to the proof of the termination of system SUBST in [HL86]. The termination of the rules 1–13 can be seen from the following polynomial interpretation:

$$\begin{aligned}
\mathcal{I}(id) = \mathcal{I}(id') &= \mathcal{I}(\varepsilon) = 1 \\
\mathcal{I}(L(f)) &= 2 * \mathcal{I}(f) \\
\mathcal{I}(\zeta(f')) &= \mathcal{I}(f') + 1 \\
\mathcal{I}(f \circ g) &= \mathcal{I}(f) * \mathcal{I}(g) + \mathcal{I}(f) \\
\mathcal{I}(f' \circ' g') &= \mathcal{I}(f') * \mathcal{I}(g') + \mathcal{I}(f')
\end{aligned}$$

where the typing is not important for the interpretation.
Problem. How can one prove termination of all rules where rule 14 seems to be oriented in the "wrong" way?

Solution. The used method is as follows: Try to define a function from terms into natural numbers such that the number of applications of rule 14 is bounded by this function for all possible derivations starting from the term given.

Now we proceed with the proof by defining two functions ψ and ψ_ζ (ψ will be the desired function to bound redexes of rule 14, ψ_ζ bounds the potential number of ζ):

$$\begin{aligned}
\psi(id) = \psi(id') &= \psi(\varepsilon) = 0 \\
\psi(L(f)) &= \psi(f) \\
\psi(\zeta(f')) &= \psi(f') \\
\psi(f \circ g) &= \psi(f) + \psi(g) + \psi_\zeta(f) \\
\psi(f' \circ' g') &= \psi(f') + \psi(g') + \psi_\zeta(f') \\
\psi_\zeta(id) = \psi_\zeta(id') &= \psi_\zeta(\varepsilon) = 0 \\
\psi_\zeta(L(f)) &= \psi_\zeta(f) \\
\psi_\zeta(\zeta(f')) &= \psi_\zeta(f') + 1 \\
\psi_\zeta(f \circ g) &= \psi_\zeta(f) + \psi_\zeta(g) \\
\psi_\zeta(f' \circ' g') &= \psi_\zeta(f') + \psi_\zeta(g')
\end{aligned}$$

Using this definition one can see the following:

1. ψ and ψ_ζ have the subterm property (for all subterms s of the term t holds $B(s) \leq B(t)$) which is obvious from the definition.

2. ψ_ζ is compatible with the structure of terms $(\forall C[\_] : \psi_\zeta(s) \geq \psi_\zeta(t) \Rightarrow \psi_\zeta(C[s]) \geq \psi_\zeta(C[t]))$ which follows by induction over the structure of terms for the context $C[\_]$ (but one can see that ψ_ζ just counts the number of ζs).

3. $s \longrightarrow_R^* t \Rightarrow \psi_\zeta(s) \geq \psi_\zeta(t)$ which can be seen from inspecting all rules and using 2.

4. $s \longrightarrow_R^* t \Rightarrow [\psi(s) \geq \psi(t) \Rightarrow \psi(C[s]) \geq \psi(C[t])]$ follows by induction over terms forming the context $C[\_]$ using 3. The interesting case is ($\circ^{(\prime)}$ stands for $\circ$ and $\circ'$):

$$\psi(s \circ^{(\prime)} u) = \psi(s) + \psi(u) + \psi_\zeta(s) \geq \psi(t) + \psi(u) + \psi_\zeta(t) = \psi(t \circ^{(\prime)} u)$$

5. $s \longrightarrow_R^* t$ implies $\psi(s) > \psi(t)$ if rule 14 is used, otherwise $\psi(s) \geq \psi(t)$ holds which can be checked for all rules using 4. The interesting case is:

$$\begin{aligned} \psi(\zeta(f') \circ g) = & \ \psi(\zeta(f')) + \psi(g) + \psi_\zeta(\zeta(f')) = \\ \psi(f') + \psi(g) + \psi_\zeta(f') + 1 > & \ \psi(f') + \psi(g) + \psi_\zeta(f') \\ & = \psi(f') + \psi(L(g)) + \psi_\zeta(f') \\ & = \psi(\zeta(f' \circ' L(g))) \end{aligned}$$

Now termination follows when we consider the pair $(\psi(t), \mathcal{I}(t))$ ordered lexicographically. Since termination is known the check for confluence is equivalent to the check of local confluence which can be done by checking all critical pairs.

In the Appendix A.4.1 a trace of this check with the Larch Prover [GG91] can be found. First the termination of the first 13 rules is shown with the polynomial interpretation $\mathcal{I}$. Then the last rule is added and all rules are oriented from left to right. The last check is for local confluence. □

Example. The system SUBST in [HL86] is connected with the adjunction $\_ \times A \dashv A \Rightarrow \_$. Here we give a more detailed description of the functors and the characterization of this adjunction. The graphical representation of this adjunctions looks as follows:

$$\frac{C \rightarrow (A \Rightarrow B)}{C \times A \rightarrow B}$$

In the case of this adjunction we have (when A is an object of the given category C):

functors	$L := \_ \times A$	$R := A \Rightarrow \_$
objects	$B \mapsto B \times A$	$B \mapsto A \Rightarrow B$
morphisms	$F \mapsto < F \circ fst, snd >$	$F \mapsto \Lambda(F \circ app)$

In this example we have $\zeta := \Lambda$ and $\varepsilon := app$. $< \_, \_ >$ is pairing with the projections fst and snd, Λ is currying with the evaluation app.

Remark. For completeness we also add the system SUBST considered by Hardin and Laville. We will use their notation of the corresponding constants.

Note. The following system SUBST is canonical:

$$
\begin{array}{lrcl}
(\mathbf{Fst}) & F \circ \langle s, t\rangle & \longrightarrow & s \\
(\mathbf{Snd}) & S \circ \langle s, t\rangle & \longrightarrow & t \\
(\mathbf{IdL}) & I \circ t & \longrightarrow & t \\
(\mathbf{IdR}) & t \circ I & \longrightarrow & t \\
(\mathbf{SPair}) & \langle F \circ t, S \circ t\rangle & \longrightarrow & t \\
(\mathbf{Ass}) & (s \circ t) \circ u & \longrightarrow & s \circ (t \circ u) \\
(\mathbf{DPair}) & \langle s, t\rangle \circ u & \longrightarrow & \langle s \circ u, t \circ u\rangle \\
(\mathbf{D\Lambda}) & \Lambda(s) \circ t & \longrightarrow & \Lambda(s \circ \langle t \circ F, S\rangle) \\
(\mathbf{FSI}) & \langle F, S\rangle & \longrightarrow & I
\end{array}
$$

where F (first), S (second), I (identity), Λ, $\langle \_, \_\rangle$, $\circ$ are the usual constants from CCL (combinatory categorical logic). The termination is shown in [HL86] where the problematic rule here is $D\Lambda$. To demonstrate our method we include the proof local confluence in the Appendix A.4.2. This system is also referred to in [Hue90a].

Chapter 4

Monads

In this chapter we define the categorical notion of a monad. It is presented in great detail in [BW85]. First we solve the universal word problem for monads by a reduction to an associated adjunctions. Afterwards we derive an easier canonical system with the help of the Kleisli category of a monad.
Motivation. Our motivation to focus on monads is based on the work of Moggi in [Mog89] on structuring (extensional) semantics of programming languages. A canonical system for monads can help to automate reasoning about semantics. As a categorical generalization of monoids there was hope to find such a canonical system.

4.1 Definition

$$\begin{tikzcd} M\times M\times M \arrow[r, "\mu\times 1"] \arrow[d, "1\times\mu"'] & M\times M \arrow[d, "\mu"] \\ M\times M \arrow[r, "\mu"'] & M \end{tikzcd} \qquad \begin{tikzcd} 1\times M \arrow[r, "\eta\times 1"] \arrow[dr, phantom, "\cong"] & M\times M \arrow[d, "\mu"] & M\times 1 \arrow[l, "1\times\eta"'] \arrow[dl, phantom, "\cong"] \\ & M & \end{tikzcd}$$

associativity *identity laws*

Figure 4.1: Laws of a Monoid in Form of Diagrams

Remark. A *monoid* is a very useful mathematical notion which is described equationally and which has a decidable uniform word problem. The equations can be characterized by diagrams as in Fig. 4.1. A monad is the categorical generalization of the this concept [ML71]. This gives evidence that there can be a canonical system for monads, too.

Note. A recent application of monads in computer science can be found in [Mog89] where they are exploited to structure the semantics of programming languages which later was used to structure purely functional programs as in [Wad93].

Example. There are many examples of monoids:

1. natural numbers with multiplication and one as the unit element,
2. finite functions from $\mathbb{N}_0$ to $\mathbb{N}_0$ with the identity function as the unit element,
3. $n \times n$ matrices over a ring with multiplication and the identity matrix as the unit element.

Definition 17 (Monad) *Let C be a category. A* **monad** $M = (T, \eta, \mu)$ *on C is an endofunctor $T : C \to C$ with two natural transformations $\eta : I_C \to T$ and $\mu : (T \circ T) \to T$ where the following laws are satisfied:*

$$\begin{array}{lrcl}
\textbf{(Mon1)} & \mu_A \circ T(\mu_A) & = & \mu_A \circ \mu_{T(A)} \\
\textbf{(Mon2)} & \mu_A \circ T(\eta_A) & = & id_{T(A)} \\
\textbf{(Mon3)} & \mu_A \circ \eta_{T(A)} & = & id_{T(A)}
\end{array}$$

$$\begin{tikzcd}
T^3(A) \arrow[r, "T(\mu_A)"] \arrow[d, "\mu_{T(A)}"'] & T^2(A) \arrow[d, "\mu_A"] \\
T^2(A) \arrow[r, "\mu_A"'] & T(A)
\end{tikzcd}
\qquad
\begin{tikzcd}
T(A) \arrow[r, "T(\eta_A)"] \arrow[d, "\eta_{T(A)}"'] \arrow[dr, "id_{T(A)}"] & T^2(A) \arrow[d, "\mu_A"] \\
T^2(A) \arrow[r, "\mu_A"'] & T(A)
\end{tikzcd}$$

associativity *unit laws*

Figure 4.2: Monad Laws

Remark. The laws for a monad can be nicely visualized by diagrams cf. Fig. 4.2 which commute. These diagrams also show the similarity to monoids. Only the previous identity laws described by triangles have been glued together along the side marked with $\cong$ resulting in a square.

Example. Let CR be a commutative ring. An example of a monad is:

T is the endofunctor for the category of sets which takes an arbitrary set X to the set of polynomials in a finite number of variables from X and coefficients in CR,

η takes a member x of the set X to the polynomial described by x,

μ collects terms with respect to the operations and equalities which hold for polynomials.

Note. Let C be a category. A monad $M = (T, \eta, \mu)$ on C can be characterized by the following equational specification:

$$
\begin{align}
id_B \circ f_{A \to B} &= f \tag{4.1}\\
f_{A \to B} \circ id_A &= f \tag{4.2}\\
(f_{A3 \to A4} \circ g_{A2 \to A3}) \circ h_{A1 \to A2} &= f \circ (g \circ h) \tag{4.3}\\
T(id_A) &= id_{T(A)} \tag{4.4}\\
T(f_{A2 \to A3} \circ g_{A1 \to A2}) &= T(f) \circ T(g) \tag{4.5}\\
T(f_{A \to B}) \circ \eta_A &= \eta_B \circ f \tag{4.6}\\
\mu_B \circ T(T(f_{A \to B})) &= T(f) \circ \mu_A \tag{4.7}\\
\mu_A \circ T(\mu_A) &= \mu_A \circ \mu_{T(A)} \tag{4.8}\\
\mu_A \circ T(\eta_A) &= id_{T(A)} \tag{4.9}\\
\mu_A \circ \eta_{T(A)} &= id_{T(A)} \tag{4.10}
\end{align}
$$

Remark. The equations 1–3 express that C is a category, 4–5 describe the endofunctor T, 6 states that η is a natural transformation $I_C \to T$, 7 states that μ is a natural transformation $(T \circ T) \to T$, 8–10 are the monad laws.

Problem. We wish to decide the universal word problem for monads. Several attempts to orient these ten equations of the monad failed. The reason seems to be that T acts as an endofunctor and can be iterated since one can define:

$$
\begin{aligned}
T^1(f) &:= T(f)\\
T^{n+1}(f) &:= T(T^n(f))
\end{aligned}
$$

The iteration of T does not only occur on the morphism level. Additionally it occurs at the object level, too. Also the unification in the presence of a functor [Vog78] was not sufficient to establish a canonical system modulo an equational theory as in [Hue80] since the functor interacts closely with the law of associativity.

Remark. In [Mog89] a monad is exploited by considering a program as a morphism from A (the object of values of type A) to $T(B)$ (the object of computations of type B). Then intuitively η is an inclusion of values into computations and μ flattens a computation of a computation into a computation. Furthermore the notion of a computational model as defined in this paper has to fulfill an extra equalizing requirement.

4.2 A Canonical System via Adjunctions

Solution. One solution for the word problem of monads can be found with the help of the following two theorems from category theory [BW85] which allow to translate the problem to the level of an adjunction where a canonical system was given before.

Theorem 16 *Let $D \leftarrow C : L \dashv R : D \rightarrow C$ be an adjunction with the unit $\eta : I_C \rightarrow (R \circ L)$ and co-unit $\epsilon : (L \circ R) \rightarrow I_D$. Then $(R \circ L, \eta, R(\epsilon_L))$ is a monad on C.*

Proof. cf. [BW85]
The proof is done by verification of the monad laws.

Theorem 17 *Let $M = (T, \eta, \mu)$ be a monad on C. Then there is a category D and an adjunction $D \leftarrow C : L \dashv R : D \rightarrow C$ such that:*

1. *$T = (R \circ L)$,*
2. *η is the unit of the adjunction,*
3. *$R(\epsilon_L) = \mu$ where ϵ is the co-unit of the adjunction.*

Proof. cf. [BW85]
One possible proof is due to Kleisli with the following construction of the category D (also called "Kleisli category"):

objects	same objects as C
morphisms	$Hom_D(A, B) = Hom_C(A, T(B))$
identity of object A	η_A
composition of $f \in Hom_D(A2, A3)$ and $g \in Hom_D(A1, A2)$	$(f \circ_D g) := \mu_{A3} \circ_C T(f) \circ_C g$

This construction works due to the fact that for the full sub-category D' of D with $Obj(D') := L(Obj(C))$ the following has to hold:

$$Hom_{D'}(L(A), L(B)) \cong Hom_C(A, R(L(B))) = Hom_C(A, T(B))$$

The corresponding functors are:

functor	L	R
objects	$A \mapsto A$	$A \mapsto T(A)$
morphisms	$A \xrightarrow{f}_C B \mapsto T(f) \circ_C \eta_A$	$A \xrightarrow{f}_D B \mapsto \mu_B \circ_C T(f)$

Thus the "Kleisli category" can be completely defined in terms of the given category C.

Remark. Theorem 16 states that from a given adjunction one can construct a monad. Theorem 17 even demonstrates that every monad arises from an adjoint pair of functors. This fact can be exploited for solving the universal word problem for monads.

Corollary 18 *The universal word problem for monads is decidable.*

Proof.
Let $M = (T, \eta, \mu)$ be a monad on the category C. We have the following equivalences:

1. An equation E for the monad M is expressed by the constructors $(id, \eta, \mu, T, \circ)$ for morphisms.
2. The monad $M = (T, \eta, \mu)$ arises from an adjunction $L \dashv R$ with unit η and co-unit ϵ according to Theorem 17. Thus E can be transformed into an equation with the constructors $(id, id', \eta, \epsilon, L, R, \circ, \circ')$.
3. As we saw from the equivalent representations of an adjunction in Lemma 13 this can be expressed equivalently with the constructors $(id, id', \varepsilon, L, \zeta, \circ, \circ')$, where finally the canonical system from Proposition 15 can be applied.

The process of translations which preserve the validity of E is summarized in the following table:

original	first step	second step
$T(f_{A \to B})$	$R(L(f))$	$\zeta(L(f) \circ' \varepsilon_A)$
η_A	η_A	$\zeta(id'_{L(A)})$
μ_A	$R(\epsilon_L)$	$\zeta(\varepsilon_{L(A)} \circ' \varepsilon_{L(R(L(A)))})$

The verification of the entire system can be found in the Appendix A.5.1. □

Application. As an application of this canonical system we show in the Appendix C.3.1 that the Kleisli construction provides indeed a category. Thus the three laws of a category have to be verified. This is possible since the Kleisli category is defined in terms of the given data.

4.3 Another System via the Kleisli Category

Remark. Since we will need the notion of the Kleisli category again we make it explicit here although it already played a role in the proof of the correspondence between monads and adjunctions.

Definition 18 (Kleisli category) *Let $M = (T, \eta, \mu)$ be a monad on the category C. Then the corresponding* **Kleisli category** *K is defined by:*

objects	*same objects as C*
morphisms	$Hom_K(A, B) := Hom_C(A, T(B))$
identity of object A	η_A
composing $A2 \xrightarrow{f}_K A3, A1 \xrightarrow{g}_K A2$	$(f \circ_K g) := \mu_{A3} \circ_C T(f) \circ_C g$

Note. How can one take advantage of the Kleisli category which is formulated in terms of the given category? The law for multiplication in the Kleisli category suggests another auxiliary definition: $f^{\star}_{A \to T(B)} := \mu_B \circ_C T(f)$ (called the Kleisli star) such that $f \circ_K g = f^{\star} \circ_C g$. Formulating the categorical laws for the Kleisli category with this new function we get:

$$\begin{array}{rcl} in \quad K & & in \quad C \\ f = id_{(K)} \circ_K f & = & \eta^{\star} \circ_C f \\ f = f \circ_K id_{(K)} & = & f^{\star} \circ_C \eta \\ (f \circ_K g) \circ_K h & = & (f^{\star} \circ_C g)^{\star} \circ_C h \\ = f \circ_K (g \circ_K h) & = & f^{\star} \circ_C (g^{\star} \circ_C h) = \end{array}$$

With setting $\nu_A := \eta_A$ (called the unit of the Kleisli category) these equations can be reformulated as:

$$\begin{array}{lrcl}
\textbf{(Kl1)} & \nu_A^\star & = & id_{T(A)} \\
\textbf{(Kl2)} & f^\star_{A\to T(B)} \circ \nu_A & = & f \\
\textbf{(Kl3)} & g^\star_{A2\to T(A3)} \circ h^\star_{A1\to T(A2)} & = & (g^\star \circ h)^\star
\end{array}$$

Lemma 19 *A monad can be completely characterized by* $(id, \circ, \nu, {}^\star)$ *on the level of morphisms assuming it is known how* T *acts on objects.*

Proof. cf. [Man76]
The other components can be expressed as:
$\eta_A := \nu_A, \quad \mu_A := id^\star_{T(A)}, \quad$ and $\quad T(f_{A\to B}) := (\nu_B \circ f)^\star.$ □

Remark. This gives a more compact way of representing a monad since it was previously described in terms of $(id, \circ, T, \eta, \mu)$. Nevertheless the action of T on the object level has to be given since it plays a role in the typing information in the rules Kl1-3. This presentation can now be used to formulate a canonical system for monads.

Proposition 20 *Assuming it is known how* $T : C \to C$ *acts on objects there is the following canonical system MON for a monad:*

$$\begin{array}{rcll}
id_B \circ f_{A\to B} & \longrightarrow & f & (4.1) \\
f_{A\to B} \circ id_A & \longrightarrow & f & (4.2) \\
(f_{A3\to A4} \circ g_{A2\to A3}) \circ h_{A1\to A2} & \longrightarrow & f \circ (g \circ h) & (4.3) \\
\nu_A^\star & \longrightarrow & id_{T(A)} & (4.4) \\
f^\star_{A\to T(B)} \circ \nu_A & \longrightarrow & f & (4.5) \\
f^\star_{A2\to T(A3)} \circ g^\star_{A1\to T(A2)} & \longrightarrow & (f^\star \circ g)^\star & (4.6) \\
f^\star_{A2\to T(A3)} \circ (\nu_{A2} \circ g_{A1\to A2}) & \longrightarrow & f \circ g & (4.7) \\
f^\star_{A3\to T(A4)} \circ (g^\star_{A2\to T(A3)} \circ h_{A1\to T(A2)}) & \longrightarrow & (f^\star \circ g)^\star \circ h & (4.8)
\end{array}$$

Proof.
The correctness of this result was verified with the help of the Larch Prover. The check of the critical pairs can be seen in the Appendix A.5.2. It was done twice: firstly without taking the typing of morphisms into account and secondly including the typing. Here we only give the termination argument with the help of a polynomial interpretation $\mathcal{I}(f) := (\mathcal{I}_1(f), \mathcal{I}_2(f), \mathcal{I}_3(f))$:

$$\begin{array}{rcl}
\mathcal{I}_1(id) = \mathcal{I}_2(id) & = & \mathcal{I}_3(id) = 1 \\
\mathcal{I}_1(f \circ g) & = & \mathcal{I}_1(f) + \mathcal{I}_1(g) \\
\mathcal{I}_2(f \circ g) & = & 2 * \mathcal{I}_2(f) * \mathcal{I}_2(g) \\
\mathcal{I}_3(f \circ g) & = & 2 * \mathcal{I}_3(f) + \mathcal{I}_3(g) \\
\mathcal{I}_1(\nu) = \mathcal{I}_2(\nu) & = & \mathcal{I}_3(\nu) = 1 \\
\mathcal{I}_1(f^\star) & = & \mathcal{I}_1(f) + 2 \\
\mathcal{I}_2(f^\star) & = & \mathcal{I}_2(f) + 2 \\
\mathcal{I}_3(f^\star) & = & \mathcal{I}_3(f) + 2
\end{array}$$

These triples of natural numbers are ordered lexicographically where the first component has the highest priority. Also this ordering is expressible in the Larch Prover and is given in the Appendix A.5.2. $\square$

Application. To verify how this system is able to handle the iteration of the endofunctor T we check the equality for terms derived from $T^2(\mu_{T^2}) \circ T^6(F) \circ T(\eta_{T^4})$. The tests can be found in the Appendix C.3.2.

Chapter 5

Computational Co-monads

In this chapter we define the notion of a computational co-monad as it is used in [BG92] to study intensional semantics. For this notion we present a canonical system which is achieved by extending the system for co-monads in the representation with the help of the Kleisli category.

Motivation. Similar to the work of Moggi later on Brookes and Geva in [BG92] studied intensional semantics of programming languages. A central notion in their work is the "computational co-monad". In their paper several results about this notion had to be verified. With the help of a canonical system those were automated.

5.1 Definition

Remark. Here the definition of a co-monad [BW85] and a computational co-monad [BG92] are given. Furthermore the notion of the Kleisli category for a co-monad is introduced. Again we assume that the morphisms between two objects form a set.

Definition 19 (Co-Monad) *Let C be a category. A* **co-monad** $CM = (T, \epsilon, \delta)$ *on C is an endofunctor $T : C \to C$ with two natural transformations $\epsilon : T \to I_C$ and $\delta : T \to (T \circ T)$ where the following laws are satisfied:*

$$\begin{array}{lrcl} (\mathbf{Com1}) & T(\delta_A) \circ \delta_A & = & \delta_{T(A)} \circ \delta_A \\ (\mathbf{Com2}) & T(\epsilon_A) \circ \delta_A & = & id_{T(A)} \\ (\mathbf{Com3}) & \epsilon_{T(A)} \circ \delta_A & = & id_{T(A)} \end{array}$$

Remark. The laws for a co-monad can be visualized by diagrams as in Fig. 5.1 which is very similar to the diagrams of a monad. Since it is the dual categorical notion all the arrows are reversed.

Note. Let C be a category. A co-monad $CM = (T, \epsilon, \delta)$ can be characterized by the following equational specification:

$$id_B \circ f_{A \to B} \quad = \quad f \tag{5.1}$$

$$
\begin{tikzcd}
T(A) \arrow[r, "\delta_A"] \arrow[d, "\delta_A"'] & T^2(A) \arrow[d, "T(\delta_A)"] \\
T^2(A) \arrow[r, "\delta_{T(A)}"'] & T^3(A)
\end{tikzcd}
\qquad
\begin{tikzcd}
T(A) \arrow[r, "\delta_A"] \arrow[d, "\delta_A"'] \arrow[dr, "id_{T(A)}"] & T^2(A) \arrow[d, "T(\epsilon_A)"] \\
T^2(A) \arrow[r, "\epsilon_{T(A)}"'] & T(A)
\end{tikzcd}
$$

Figure 5.1: Co-monad Laws

$$
\begin{aligned}
f_{A\to B} \circ id_A &= f && (5.2)\\
(f_{A3\to A4} \circ g_{A2\to A3}) \circ h_{A1\to A2} &= f \circ (g \circ h) && (5.3)\\
T(id_A) &= id_{T(A)} && (5.4)\\
T(f_{A2\to A3} \circ g_{A1\to A2}) &= T(f) \circ T(g) && (5.5)\\
\epsilon_B \circ T(f_{A\to B}) &= f \circ \epsilon_A && (5.6)\\
T(T(f_{A\to B})) \circ \delta_A &= \delta_B \circ T(f) && (5.7)\\
T(\delta_A) \circ \delta_A &= \delta_{T(A)} \circ \delta_A && (5.8)\\
T(\epsilon_A) \circ \delta_A &= id_{T(A)} && (5.9)\\
\epsilon_{T(A)} \circ \delta_A &= id_{T(A)} && (5.10)
\end{aligned}
$$

Definition 20 (Kleisli category for co-monad) *Let $CM = (T, \epsilon, \delta)$ be a co-monad on the category C. Then the corresponding* **Kleisli category** *CK is defined by:*

objects	*same objects as C*
morphisms	$Hom_{CK}(A, B) := Hom_C(T(A), B)$
identity of object A	ϵ_A
composing $A2 \xrightarrow{f}_{CK} A3, A1 \xrightarrow{g}_{CK} A2$	$(f \circ_{CK} g) := f \circ_C T(g) \circ_C \delta_{A1}$

Remark. The same way the Kleisli category helped for monads it helps for co-monads. Since the notions of monad and co-monad are dual to each other we do not give an example of a co-monad. Instead we proceed with the notion of a computational co-monad as it is used by [BG92] to study intensional semantics.

Definition 21 (Computational Co-monad [BG92]) *Let C be a category and $CM = (T, \epsilon, \delta)$ be a co-monad on C. A* **computational co-monad** *$CCM = (T, \epsilon, \delta, \gamma)$ is a co-monad having additionally one further natural transformation $\gamma : I_C \to T$ fulfilling the laws:*

$$
\begin{aligned}
&\textbf{(CCom1)} & \epsilon_A \circ \gamma_A &= id_A\\
&\textbf{(CCom2)} & \delta_A \circ \gamma_A &= \gamma_{T(A)} \circ \gamma_A
\end{aligned}
$$

Note. Let C be a category. A computational co-monad $CCM = (T, \epsilon, \delta, \gamma)$ can be characterized by an equational specification with the further equations:

$$T(f_{A\to B}) \circ \gamma_A = \gamma_B \circ f \tag{5.11}$$
$$\epsilon_A \circ \gamma_A = id_A \tag{5.12}$$
$$\delta_A \circ \gamma_A = \gamma_{T(A)} \circ \gamma_A \tag{5.13}$$

Remark. In [BG92] a co-monad is exploited by considering a program as a morphism from $T(A)$ (the object of input computations of type A) to B (the object of output values of type B). In the notion of a computational co-monad γ can intuitively thought of as regarding every data value as a degenerate computation of the same type. This is expressed by making γ a natural transformation with some extra requirements.

5.2 A Canonical System

Note. Similar to the case of monads we can take advantage of the Kleisli category. The law for multiplication in the Kleisli category suggests another auxiliary definition here for co-monads: $g^\star_{T(A)\to B} := T(g) \circ_C \delta_A$ (called the Kleisli star) such that $f \circ_K g = f \circ_C g^\star$. Formulating the categorical laws for the Kleisli category with this new function we get:

$$\begin{array}{ccc}
in \quad CK & & in \quad C \\
f = f \circ_{CK} id_{(CK)} & = & f \circ_C \epsilon^\star \\
f = id_{(CK)} \circ_{CK} f & = & \epsilon \circ_C f^\star \\
(f \circ_{CK} g) \circ_{CK} h & = & (f \circ_C g^\star) \circ_C h^\star \\
= f \circ_{CK} (g \circ_{CK} h) & = & f \circ_C (g \circ_C h^\star)^\star =
\end{array}$$

With setting $\nu_A := \epsilon_A$ (called the unit of the Kleisli category) these equations can be reformulated as:

$$\begin{array}{lrcl}
\textbf{(CKl1)} & \nu^\star_A & = & id_{T(A)} \\
\textbf{(CKl2)} & \nu_B \circ f^\star_{T(A)\to B} & = & f \\
\textbf{(CKl3)} & g^\star_{T(A2)\to A3} \circ h^\star_{T(A1)\to A2} & = & (g \circ h^\star)^\star
\end{array}$$

Lemma 21 *A co-monad can be completely characterized by $(id, \circ, \nu, {}^\star)$ on the level of morphisms assuming it is known how T acts on objects.*

Proof. cf. [Man76]
The other components can be expressed as:
$\epsilon_A := \nu_A, \quad \delta_A := id^\star_{T(A)}, \quad$ and $\quad T(f_{A\to B}) := (f \circ \nu_A)^\star$. □

Remark. This gives a more compact way of presenting a co-monad since it was previously described in terms of $(id, \circ, T, \epsilon, \delta)$. Nevertheless the action of T on the object level has to be given since it plays a role in the typing information in the rules CKl1-3. This presentation can now be used to formulate a canonical system for co-monads.

Proposition 22 *Assuming it is known how $T : C \to C$ acts on objects there is the following canonical system COM for a co-monad:*

$$
\begin{array}{rcll}
id_B \circ f_{A\to B} & \longrightarrow & f & (5.1)\\
f_{A\to B} \circ id_A & \longrightarrow & f & (5.2)\\
(f_{A3\to A4} \circ g_{A2\to A3}) \circ h_{A1\to A2} & \longrightarrow & f \circ (g \circ h) & (5.3)\\
\nu_A^\star & \longrightarrow & id_{T(A)} & (5.4)\\
\nu_B \circ f^\star_{T(A)\to B} & \longrightarrow & f & (5.5)\\
f^\star_{T(A2)\to A3} \circ g^\star_{T(A1)\to A2} & \longrightarrow & (f \circ g^\star)^\star & (5.6)\\
\nu_{A3} \circ (f^\star_{T(A2)\to A3} \circ g_{A1\to T(A2)}) & \longrightarrow & f \circ g & (5.7)\\
f^\star_{T(A3)\to A4} \circ (g^\star_{T(A2)\to A3} \circ h_{A1\to T(A2)}) & \longrightarrow & (f \circ g^\star)^\star \circ h & (5.8)
\end{array}
$$

Proof.
The proof proceeds analogous to the case of monads and can be verified as the first part of the trace in Appendix A.6. □

Note. How can the additional equations for a computational co-monad be reformulated to fit into the new representation? Especially the references to T and δ have to be avoided. The key observation comes from the following equation:

$$f^\star_{T(A)\to B} \circ \gamma_A = T(f) \circ \delta_A \circ \gamma_A = T(f) \circ \gamma_{T(A)} \circ \gamma_A = \gamma_B \circ f \circ \gamma_A$$

Here both rules from the original presentation which involve T and δ have been applied leading to the single equation: $f^\star_{T(A)\to B} \circ \gamma_A = \gamma_B \circ f \circ \gamma_A$. This equation together with $\nu_A \circ \gamma_A = id_A$ suffices to describe a computational co-monad.

Lemma 23 *A computational co-monad can be completely characterized in terms of $(id, \circ, \nu, {}^\star, \gamma)$ on the level of morphisms assuming it is known how T acts on objects where γ is described by the equations:*

$$
\begin{array}{lrcl}
\textbf{(Gam1)} & \nu_A \circ \gamma_A & = & id_A\\
\textbf{(Gam2)} & f^\star_{T(A)\to B} \circ \gamma_A & = & \gamma_B \circ f \circ \gamma_A
\end{array}
$$

Proof.
It has to be shown that the previous three equations for γ can be derived:

$$
\begin{array}{rcl}
T(f_{A\to B}) \circ \gamma_A = (f \circ \nu_A)^\star \circ \gamma_A & = & \gamma_B \circ (f \circ \nu_A) \circ \gamma_A = \gamma_B \circ f\\
\epsilon_A \circ \gamma_A & = & \nu_A \circ \gamma_A = id_{T(A)}\\
\delta_A \circ \gamma_A = id^\star_{T(A)} \circ \gamma_A & = & \gamma_{T(A)} \circ id_{T(A)} \circ \gamma_A = \gamma_{T(A)} \circ \gamma_A
\end{array}
$$

□

Theorem 24 *Assuming it is know how $T : C \to C$ acts on objects there is the following canonical system CCM for a computational co-monad extending the canonical system COM*

$$\begin{aligned}
\nu_A \circ \gamma_A &\longrightarrow id_A && (5.9)\\
f^{\star}_{T(A)\to B} \circ \gamma_A &\longrightarrow \gamma_B \circ (f \circ \gamma_A) && (5.10)\\
\nu_B \circ (\gamma_B \circ f_{A\to B}) &\longrightarrow f && (5.11)\\
g^{\star}_{T(A2)\to A3} \circ (\gamma_{A2} \circ h_{A1\to A2}) &\longrightarrow \gamma_{A3} \circ (g \circ (\gamma_{A2} \circ h)) && (5.12)
\end{aligned}$$

Proof.
Again the check of the critical pairs was done in the Larch Prover which can be seen in the Appendix A.6. There also the extension of the polynomial interpretation can be verified. □

Remark. At this point it should be stressed that the choice of the representation for co-monads was not obvious to us. The introduction of the auxiliary $\_^{\star}$ was necessary to succeed. This example could be used to test rewrite tools which allow the extension of the signature as in [KZ89] such that one can see to which extent this may be automated.

Application. As an applications we derive statements from [BG92] automatically. In that paper a pair of functors is defined relating the category C and the Kleisli category CK. We have:

$$\begin{aligned}
alg : C \to CK \quad & defined\ by \quad alg(f_{A\to B}) := f \circ \epsilon_A\\
fun : CK \to C \quad & defined\ by \quad fun(f_{T(A)\to B}) := f \circ \gamma_A
\end{aligned}$$

The queries are the test whether alg and fun are indeed functors. Furthermore the following equalities were checked:

$$\begin{aligned}
fun \circ alg &= I_C\\
alg \circ fun &=^e I_{CK} \quad where \quad f =^e g \iff fun(f) = fun(g)
\end{aligned}$$

These tests can be found in the Appendix C.3.3.

Chapter 6

Related Topics

In this chapter related topics as the monadic style from purely functional programming and higher-order rewrite systems are presented. We focus on the monadic laws which describe the monadic style and treat them as a higher-order rewrite system. We achieve a canonical system which remains canonical even if we make use of dependent types.
Motivation. Once the monadic laws can be turned into a canonical higher-order rewrite system it can be used to provide an automated tool for reasoning about monadic functional programs or about the monadic functional programming style as e.g. considered by Filinski in [Fil94].

6.1 Monadic Functional Style

6.1.1 Examples

By means of examples taken from [Wad93] we will demonstrate the monadic style to structure purely functional programs. The examples are a variation on the theme of a simple evaluator.
Example. Consider the following simple evaluator coded in **SML**:

```
datatype term = Const of int | Quot of (term * term)

fun eval0 (Const i) = i
  | eval0 (Quot(j,k)) = (eval0 j) div (eval0 k)
```

If we want to add exception handling for the division by zero it becomes:

```
exception DivByZero

fun eval1 (Const i) = i
  | eval1 (Quot(j,k)) =
    if ((eval1 k) = 0) then raise DivByZero
    else (eval1 j) div (eval1 k)
```

If we want to add a counter to count the evaluation of a quotient it becomes:

```
val count = ref 0

fun eval2 (Const i) = i
  | eval2 (Quot(j,k)) =
    ((count := !count + 1);
     (eval2 j) div (eval2 k))
```

If we want to write a message any time a quotient is evaluated it becomes:

```
fun eval3 (Const i) = i
  | eval3 (Quot(j,k)) =
    (print "evaluate quot\n";
     (eval3 j) div (eval3 k))
```

Remark. Since SML is an *impure* functional language the changes were only minor and could be directly done at the point where another behavior was required. Thus the previous examples were coded in an *impure style*. Now we will express the same effects but in a *pure style*.

Example. We consider the same evaluator:

```
datatype term = Const of int | Quot of (term * term)

fun eval0 (Const i) = i
  | eval0 (Quot(j,k)) = (eval0 j) div (eval0 k)
```

To code exception handling one cannot take advantage of the built-in functions. Instead it has to be coded as a new data type:

```
datatype Answer = Raise of string | Return of int

fun eval1 (Const i) = (Return(i))
  | eval1 (Quot(j,k)) =
    (case (eval1 j) of
      (Raise(sj)) => (Raise(sj))
    | (Return(rj)) => (case (eval1 k) of
                         (Raise(sk)) => (Raise(sk))
                       | (Return(rk)) => if (rk = 0)
                                         then (Raise("DivByZero"))
                                         else (Return(rj div rk))))
```

The same happens for the counter. Since there are no global variables in a purely functional language this information has to be passed as an extra argument:

```
fun eval2 (Const i) s = (i, s)
  | eval2 (Quot(j,k)) s =
    let val (rj, sj) = eval2 j s in
        let val (rk, sk) = eval2 k sj in
            ((rj div rk), sk + 1)
        end
    end
```

Finally to produce the output it has to be accumulated over all the computations:

```
fun eval3 (Const i) = ("", i)
  | eval3 (Quot(j,k)) =
    let val (oj, rj) = eval3 j in
        let val (ok, rk) = eval3 k in
            ((oj ^ ok ^ "quotient\n"), (rj div rk))
        end
    end
```

Remark. Of course it is possible to program in such a pure style (like the pure functional programming language **Haskell**) inside SML. But the changes to the program appear to be quite drastic.

Problem. Is there a way to avoid all these global changes to a purely functional program in order to achieve **effects** like *exceptions*, *global variables*, *I/O*?

From the examples it seems that the program has to be changed everywhere where a modified function is called. This would be a big disadvantage compared with the impure style.

Solution. One answer to this question is to program in the so called *"monadic style"* due to Wadler who was inspired by the work of Moggi [Mog89] about structuring the semantics of programming languages. This monadic style comes as a type constructor together with two basic functions. Particular *"monads"* will require additional functions.

Example. Already the simple evaluator is expressed in a different way:

```
datatype term = Const of int | Quot of (term * term)

infix star0

type ´a Monad0 = ´a
fun unit0 (x : ´a) : (´a Monad0) = x
fun (x: ´a Monad0) star0 (f: ´a -> ´b Monad0) : (´b Monad0) = (f x)

fun eval0 (Const i) = unit0 i
  | eval0 (Quot(j,k)) =
    (eval0 j) star0
    (fn x => (eval0 k) star0 (fn y => unit0 (x div y)))
```

Changing to exceptions merely requires a change of the data type and the two basic functions:

```
infix star1

type Exception = string
datatype ´a Monad1 = Raise of Exception | Return of ´a
fun unit1 (x : ´a) : (´a Monad1) = Return x
```

```
fun (x : ´a Monad1) star1 (f : ´a -> ´b Monad1) : (´b Monad1) =
    (case x of
         Raise(s) => Raise(s)
       | Return(a) => (f a))

val raising : (Exception -> ´a Monad1) = (fn s => Raise s)

fun eval1 (Const i) = unit1 i
  | eval1 (Quot(j,k)) =
    (eval1 j) star1
    (fn x => (eval1 k) star1
     (fn y => if (y = 0) then raising "Division by Zero"
              else unit1 (x div y)))
```

If we need a global counter the change has mainly to be done to the monad ingredients - its data type and the basic functions:

```
infix star2

type State = int
type ´a Monad2 = (State -> (´a * State))
fun unit2 (x : ´a) : (´a Monad2) = (fn y => (x, y))
fun (x : ´a Monad2) star2 (f : ´a -> ´b Monad2) : (´b Monad2) =
    (fn y => let val (a, s) =  (x y)
             in (f a s) end)

val tick : (unit Monad2) = (fn x => ((), x + 1))

fun eval2 (Const(i)) = (unit2 i)
  | eval2 (Quot(j,k)) =
    (eval2 j) star2
    (fn x => (eval2 k) star2
     (fn y => (tick star2 (fn () => unit2 (x div y)))))
```

Finally to accumulate the output the monad has to be changed appropriately:

```
infix star3

type Output = string
type ´a Monad3 = Output * ´a
fun unit3 (x : ´a) : (´a Monad3) = ("", x)
fun (x : ´a Monad3) star3 (f : ´a -> ´b Monad3) : (´b Monad3) =
    let val (o1, a) = x in
        let val (o2, b) = (f a) in
            (o1 ^ o2, b)
        end
    end

val out : Output -> (unit Monad3) = (fn s => (s, ()))

fun eval3 (Const(i)) = (out "") star3 (fn () => (unit3 i))
```

```
| eval3 (Quot(j,k)) =
  (eval3 j) star3
  (fn x => (eval3 k) star3
   (fn y => ((out "evaluate quotient\n") star3
             (fn () => (unit3 (x div y))))))
```

Remark. From these examples it should be clear that the pure program has to be changed only at the definition of the monad (the type constructor and the two functions). The calls of these functions will remain the same. Note that this concept requires a *polymorphic type system.*

6.1.2 Monadic Laws

Here the notion of the "monadic laws" for functional programming is made formal.

Definition 22 (monadic laws cf. [Fil94]) *A* **monad** M *as in functional programming is a triple* $M = (T, \nu, \_^\star)$ *where* T *is a type constructor and* ν *and* $\_^\star$ *are polymorphic functions* $\nu : \iota \to T\iota$ *and* $\_^\star : (\iota \to T\kappa) \to T\iota \to T\kappa$ *fulfilling the* **monadic laws***:*

$$\begin{array}{lrcl} (\textbf{Ml1}) & \nu^\star & = & id_T \\ (\textbf{Ml2}) & f^\star \circ \nu & = & f \\ (\textbf{Ml3}) & f^\star \circ g^\star & = & (f^\star \circ g)^\star \end{array}$$

Remark. ι and κ are ranging over arbitrary types in this definition. These laws are quite similar to the three basic categorical laws of the Kleisli category for a monad.

Question. How can these laws be understood?

Answer. The following interpretation $\mathcal{I}$ gives a clear insight into the monadic laws (actually describing the "identity monad"):

$$\begin{array}{rcl} \mathcal{I}(T\iota) & := & \iota \\ \mathcal{I}(\nu) & := & \lambda x.x \\ \mathcal{I}(f^\star \circ g) & := & (f\ g) \end{array}$$

They behave like a reformulation of application in a functional program involving additionally the type constructor T. It is exactly this freedom of the type constructor which allows to program the different impure effects like state, exceptions, or I/O.

6.2 Higher-Order Rewriting

6.2.1 Higher-Order Rewriting Systems

Here only the absolutely necessary notions from typed lambda calculus are presented. Details about lambda calculus or simply typed lambda calculus can be found e.g. in the standard books [Bar84] or [HS86] and a more recent presentation of typed lambda calculus is [Bar92]. For the technical details we will follow [MN94].

Notion 23 (simple types, signature, simply typed terms) *The set $\mathcal{T}(\mathcal{B})$ of* **simple types** *over a nonempty set $\mathcal{B}$ of* **base types** *is the smallest set with:*

1. $\mathcal{B} \subseteq \mathcal{T}(\mathcal{B})$
2. *if $\iota, \kappa \in \mathcal{T}(\mathcal{B})$ then $\iota \to \kappa \in \mathcal{T}(\mathcal{B})$*

A **signature** Σ *is a triple $(\mathcal{B}, \mathcal{C}, \mathcal{V})$ where:*

1. *$\mathcal{B}$ is a set of base types,*
2. *$\mathcal{C}$ is a set of typed constants,*
3. *$\mathcal{V}$ is a set of typed variables,*

such that all sets are disjoint and the sets of variables and constants are formed with respect to the types in $\mathcal{T}(\mathcal{B})$ i.e.:

$\mathcal{C} := \bigcup_{\iota \in \mathcal{T}(\mathcal{B})} \mathcal{C}_\iota$ *all $\mathcal{C}_\iota$ disjoint*

$\mathcal{V} := \bigcup_{\iota \in \mathcal{T}(\mathcal{B})} \mathcal{V}_\iota$ *all $\mathcal{V}_\iota$ disjoint and infinite*

The sets $\Lambda_\iota^\to$ for $\iota \in \mathcal{T}(\mathcal{B})$ over a signature Σ are the smallest sets with:

1. $\mathcal{C}_\iota \subseteq \Lambda_\iota^\to$,
2. $\mathcal{V}_\iota \subseteq \Lambda_\iota^\to$,
3. *if $v \in \mathcal{V}_\iota$ and $t \in \Lambda_\kappa^\to$ then $(\lambda v.t) \in \Lambda_{\iota\to\kappa}^\to$ called abstraction,*
4. *if $s \in \Lambda_{\iota\to\kappa}^\to$ and $t \in \Lambda_\iota^\to$ then $(s\ t) \in \Lambda_\kappa^\to$ called application.*

The set $\Lambda^\to$ of all simply typed lambda terms over the signature Σ is defined as:

$$\Lambda^\to := \bigcup_{\iota \in \mathcal{T}(\mathcal{B})} \Lambda_\iota^\to$$

Note. Following [Bar84] terms only differing in the naming of bound variables are identified (also known as α-*conversion*). Furthermore we make use of the *variable convention* that for all terms $t \in \Lambda^\to$ holds $BV(t) \cap FV(t) = \emptyset$ (i.e. for all terms of the typed lambda calculus there is no naming conflict between free and bound variables) which can always be achieved by an appropriate renaming of the bound variables.

Definition 24 (linear term) *A term $t \in \Lambda^\to$ is called* **linear** *if every free variable occurs exactly once.*

Remark. We assume the usual α, β, η conversions between λ-terms. Furthermore in the following we will only consider well-typed terms, i.e. terms from $\Lambda^\to$.

Theorem 25 (cf. [Bar84]) *The following facts hold for the simply typed lambda calculus where $R \in \{\rightarrow_\beta, \rightarrow_\eta, \rightarrow_{\beta,\eta}\}$ is a relation on $\Lambda^{\rightarrow}$:*

1. *The relation R is confluent.*
2. *The relation R is terminating.*
3. *Every term has a unique normal form with respect to R.*

Remark. The normal forms of a term t with respect to R will be denoted by $t\downarrow_\beta$, $t\downarrow_\eta$ and $t\downarrow_{\beta,\eta}$. The shape of a β-normal form is always $\lambda\overline{v_m}a(\overline{s_n}) = \lambda v_1 \dots \lambda v_m(\dots(a\ s_1)\dots s_n)$ where a is an atom also called the *head* of that term.

Definition 25 (long $\beta\eta$-normal form) *The* **long $\beta\eta$-normal form** *of a term $t \in \Lambda^{\rightarrow}$ is the result of η-expanding the β-normal form of t, i.e. $t\downarrow_\beta\uparrow^\eta$ or more explicit:*

$$t\downarrow_\beta = \lambda\overline{v_m}a(\overline{s_n}) \Rightarrow t\downarrow_\beta\uparrow^\eta = \lambda\overline{v_{m+k}}a(\overline{s_n}\uparrow^\eta, v_{m+1}\uparrow^\eta, \dots, v_{m+k}\uparrow^\eta)$$

where $t : \overline{\iota_{m+k}} \rightarrow \iota$ for $v_{m+i} : \iota_{m+i}$, $\iota \in \mathcal{B}$ and $v_{m+i} \notin FV(\overline{s_n})$.

Remark. This normal form is also written as $t \updownarrow^\eta_\beta$. Note that ι has to be a base type. $\uparrow_\eta$ indicates a *single* η-expansion in contrast to $\downarrow_\beta$ which does a normalization. After applying a substitution a term is always normalized with respect to $\updownarrow^\eta_\beta$. Now some technically necessary notions are presented.

Definition 26 (positions and subterm) *A* **position** *in a term $t \in \Lambda^{\rightarrow}$ is a sequence of elements from $\{1, 2\}$, possibly empty. The* **subterm** *of t at position p is denoted by t/p and defined as:*

$$\begin{array}{rcl@{\qquad}rcl} t/nil & := & t & (s\ t)/(1.p) & := & s/p \\ (\lambda v.t)/(1.p) & := & t/p & (s\ t)/(2.p) & := & t/p \end{array}$$

Remark. $Pos(t)$ is the set of all positions p in the term t such that t/p is defined. $t[s]_p$ is the term t with t/p replaced by s. Note that a bound variable in the abstraction cannot be accessed by this notation.

Definition 27 (abstracted variables) *The set of* **abstracted variables** *in a term $t \in \Lambda^{\rightarrow}$ at position $p \in Pos(t)$ is defined by:*

$$\begin{array}{rcl@{\qquad}rcl} AV(t, nil) & := & \emptyset & AV((s\ t), 1.p) & := & AV(s, p) \\ AV((\lambda v.t), 1.p) & := & \{v\} \cup AV(t, p) & AV((s\ t), 2.p) & := & AV(t, p) \end{array}$$

Remark. Finally the notion of a lifter is introduced which will be needed to put a term into a context with certain bound variables.

Definition 28 (lifter) *A $\overline{v_n}$-lifter of a term t away from a set of variables V is a substitution $\sigma = \{v \mapsto (\rho(v)(\overline{v_n})) | v \in FV(t)\}$ where ρ is an injective map from variables to variables such that:*

1. *$Dom(\rho) = FV(t)$ and $Ran(\rho) \cap V = \emptyset$*
2. *if $v_i : \iota_i$ and $v : \iota$ then $\rho(v) : \iota_1 \to \cdots \to \iota_n \to \iota$*

Remark. Now the basic definitions for a higher-order rewrite system are presented together with the basic theorems. Also a simple example is included. Again we roughly follow [MN94].

Definition 29 (pattern) *A term $t \in \Lambda^{\to}$ in β-normal form is called a* **higher-order pattern** *if every occurrence of a free variable $v \in FV(t)$ in a subterm $st \equiv (v\ \overline{s_n})$ of t fulfills:*

1. *$\overline{s_n}$ is η-equivalent to a list of bound variables,*
2. *where all these bound variables are different.*

Remark. Also in the case $n = 0$ the condition is fulfilled. The notion of a pattern goes back to [Mil91]. The restriction to this subset of all simply typed terms allows decidable unification.

Theorem 26 (unification for patterns cf. [Mil91]) *The unification problem for higher-order patterns is decidable. If two patterns are unifiable a most general unifier can be effectively computed.*

Definition 30 (higher-order rewrite rule, HRS) *A* **higher-order rewrite rule** *is an ordered pair (l, r) of two simply typed lambda terms where:*

1. *l and r have the same type which is a base type,*
2. *l is a pattern but not a single free variable,*
3. *$FV(l) \supseteq FV(r)$.*

A **higher-order rewrite system (HRS)** *is a non-empty set of higher-order rewrite rules.*

Definition 31 (left linear system) *A higher-order rewrite system H is called left linear if for all its rewrite rules $(l, r) \in H$ the left-hand side l is linear.*

Remark. Instead of using the pair notation for (l, r) in the following a rewrite rule will be denoted by $l \to r$ or $l \to r \in H$.

Definition 32 (rewrite relation) *Let H be a higher-order rewrite system. Then H induces the following* **rewrite relation** *$\to_H$ on simply typed lambda terms $s \to_H t$ if there are $l \to r \in H$, $p \in Pos(s)$ and a substitution σ such that*

$$s/p = l^\sigma \quad \textit{and} \quad s[r^\sigma]_p = t$$

Remark. The decidability of this rewrite relation is due to the pattern condition.

Definition 33 (equality) *Let H be a higher-order rewrite system. Then H induces the following* **equality modulo** H *denoted by "$=_H$" and written as an inference system:*

conversion $\dfrac{(s,t) \in H}{s^\sigma =_H t^\sigma}(H)$, $\dfrac{s =_\beta t}{s =_H t}(\beta)$, $\dfrac{s =_\eta t}{s =_H t}(\eta)$

equality $\dfrac{}{s =_H s}(refl)$, $\dfrac{s =_H t}{t =_H s}(symm)$, $\dfrac{s =_H t' \quad t' =_H t}{s =_H t}(tran)$

congruence $\dfrac{s =_H t}{\lambda v.s =_H \lambda v.t}(abs)$, $\dfrac{s =_H s' \quad t =_H t'}{(s\ t) =_H (s'\ t')}(app)$

Remark. This definition does not assume any normal form. Rule (H) could also be presented without the substitution σ acting on s and t since this can always be simulated with (β), (abs), and (app).

Proposition 27 (equality and rewrite relation cf. [Nip91]) *Let H be a higher-order rewrite system. Then it holds:* $s =_H t \iff s \updownarrow_\beta^\eta \leftrightarrow_H^* t \updownarrow_\beta^\eta$.

Note. This correspondence between $=_H$ and $\rightarrow_H$ explains why it is useful to check properties like termination and (local) confluence of $\rightarrow_H$.

Example. The untyped lambda calculus is an easy example of a HRS. In this case we have $\mathcal{B} := \{term\}, \mathcal{C} := \{abs, app\}$ where $abs : (term \rightarrow term) \rightarrow term$ and $app : term \rightarrow term \rightarrow term$ with the rewrite rules:

$$\begin{array}{lrcl} (\textbf{beta}) & app(abs(\lambda v.(t\ v)), s) & \rightarrow_H & (t\ s) \\ (\textbf{eta}) & abs(\lambda v.app(t, v)) & \rightarrow_H & t \end{array}$$

where abs and app are written like operators with arity 1 and 2, respectively (in an un-curried form). Note how the use of meta-level applications removes the need for an explicit substitution in $(beta)$ and how the use of meta-level abstraction removes the need for a side condition in (eta).

Remark. It has to be stressed that a TRS is a special case of a HRS. The translation of terms from a TRS is simply done by "currying" i.e. a term $f(x_1, \ldots, x_n)$ is represented as $(\ldots(f\ x_1)\ldots x_n)$. All the translated terms are of one and the same single base type (in the case of a uni-sorted TRS). That means that all the following results carry over to TRS.

6.2.2 Facts about HRS

Here several criteria are introduced how the local confluence or confluence of a higher-order rewrite system can be checked. The key idea is a generalization of the critical pair lemma to the higher-order setting [Nip91]. More details are given in [MN94]. Furthermore a method to prove termination for higher-order rewrite systems will be presented following [Kah95a].

Definition 34 (critical pair) *Let H be a higher-order rewrite system. A pair (s,t) of simply typed lambda terms is called a* **critical pair** *if:*

1. *there are rewrite rules $l_1 \to r_1 \in H$ and $l_2 \to r_2 \in H$,*
2. *there is a position $p \in Pos(l_1)$ such that l_1/p is not a free variable in l_1,*
3. $\{\overline{v_n}\} := AV(l_1, p)$
4. *σ is a $\overline{v_n}$ lifter of l_2 away from $FV(l_1)$*
5. $\theta := mgu(\lambda\overline{v_n}.(l_1/p), \lambda\overline{v_n}.(\sigma(l_2)))$
6. *$s := \theta(r_1)$ and $t := \theta(l_1[\sigma(r_2)]_p)$*

Remark. This situation is also called: pattern l_2 *overlaps* pattern l_1 at position p. Furthermore $l_1 \to r_1$ and $l_2 \to r_2$ can be the same rewrite rule. The most general unifier can be computed because of Theorem 26.

Definition 35 (critical pairs) *The set of* **critical pairs** *of a HRS H is defined as: $cps(H) := \{(s,t) | (s,t)$ is a critical pair and either $l1 \neq l2$ or $p \neq nil\}$*

Remark. This definition excludes the consideration of *overlays* of a rule with itself, i.e. the overlapping at the top position.

Theorem 28 (cf. [Nip91]) *Let H be a higher-order rewrite system. Then H is locally confluent iff all critical pairs $(s,t) \in cps(H)$ converge i.e. $s \downarrow_H t$.*

Remark. If H is known to be terminating then local confluence implies confluence (Newman's lemma cf. 2).

Definition 36 (OHRS) *A higher-order rewrite system is called* **orthogonal** *if it is left linear and has no critical pairs.*

Theorem 29 (cf. [Nip93]) *OHRS are confluent.*

Remark. This result holds regardless of termination.

Definition 37 (WOHRS) *A higher-order rewrite system is called* **weakly orthogonal** *if it is left linear and has only* trivial *critical pairs of the form (s,s).*

Theorem 30 (cf. [vO94]) *WOHRS are confluent.*

Remark. Another confluence result of this type can be found in [MN94] where confluence modulo an additional equality is considered.

Example. Again we consider the pure lambda calculus as a HRS. The rules $(beta)$ and (eta) overlap in the following way:

$$
\begin{array}{lll}
app(t,s) =_\beta ((\lambda v.app(t,v))\ s) & \leftarrow_{(beta)} app(abs(\lambda v.app(t,v)),s) & \rightarrow_{(eta)} app(t,s) \\
abs(\lambda v.(\lambda v'.(t\ v')v)) \leftarrow_{(beta)} & abs(\lambda v.app(abs(\lambda v'.(t\ v')),v)) & \rightarrow_{(eta)} abs(\lambda v.(t\ v')) \\
=_\beta abs(\lambda v.(t\ v)) & &
\end{array}
$$

With the help of the previous theorems we can conclude the following:

$\lambda^{\rightarrow}_{\beta}, \lambda^{\rightarrow}_{\eta}, \lambda^{\rightarrow}_{\beta\eta}$ **are locally confluent** by Theorem 28,

$\lambda^{\rightarrow}_{\beta}$ **is confluent** by Theorem 29,

$\lambda^{\rightarrow}_{\eta}$ **is confluent** since (eta) alone is terminating or by Theorem 29,

$\lambda^{\rightarrow}_{\beta\eta}$ **is confluent** by Theorem 30.

The check of these higher-order critical pairs can be found in the Appendix C.2.1.

Remark. Now one method to prove the termination of HRS will be sketched. It is based on [Kah95a] which is well suitable for various enriched λ-calculi. Since this requires a wider especially categorical background we only summarize the method and show the application to typed lambda calculus.

Method. ***Termination for HRS***
The method by Kahrs proceeds with the following steps:

1. find an interpretation $\mathfrak{b}$ for any base type
2. find a symbol interpretation ϱ which is consistent with the interpretation of the base types $\mathfrak{b}$
3. give a proof that all the rules are decreasing with respect to the symbol interpretation ϱ

Then the higher-order rewrite system is terminating.
The underlying ideas of his method are:

- find a semantic interpretation $[\![\_]\!]$ (extending ϱ) such that $s \rightarrow_H t$ implies $[\![s]\!] > [\![t]\!]$ where $>$ is terminating, in particular it is needed that for all rules $l \rightarrow_H r$ holds $[\![l]\!] > [\![r]\!]$
- each type is interpreted as object in the category of partially well ordered sets WO (extending $\mathfrak{b}$)
- function symbols are interpreted as morphisms in WO

To achieve this in [Kah95a] the following ingredients are provided where interpretations involve ordinals o since they are closely related to WO:

$out_\iota : \iota \rightarrow o$ order preserving functions for erasing rules for every type

$\boxplus_\iota : o \times \iota \to \iota$ order preserving operations to combine ordinals and values of arbitrary type

$\sqcap, \sqcup, \otimes, \Longrightarrow$ functors to interpret the type constructors with the required properties

His method relies on the use of an indecomposable limit ordinal o (e.g. ω) and uses the operation $\#$ which is the *natural sum* or *Hessenberg sum* (in case of ω it coincides with the usual $+$).

Finally the interpretation of abstraction and application are:

$$[\![\lambda v.m]\!] := (z \mapsto out(z) \boxplus [\![m]\!]_{v \mapsto z}) \quad , \quad [\![(m\ n)]\!] := [\![m]\!][\![n]\!]$$

Remark. In [Kah95a] already several constructions together with algebraic laws are provided which help to achieve a suitable interpretation. Therefore we feel that this method is well suited for a possible automatic verification. Other techniques to prove termination of HRS are [vdP94, vdPS95] and [LP95]. We list some of the propositions from [Kah95a]:

Proposition 31 (cf. [Kah95a]) *The following equalities and inequalities hold:*

1. $x = 0 \boxplus x < 1 \boxplus x < \ldots,$
2. $m \boxplus (n \boxplus x) = (m \# n) \boxplus x,$
3. $out(n \boxplus x) = n \boxplus out(x),$
4. *β-reduction cannot increase the interpretation,*
5. $[\![m]\!](n \boxplus x) \geq n + [\![m]\!](x)$ *for ω (can be seen by induction).*

Example. Untyped lambda calculus is known to be non-terminating as can be seen from the term $(\lambda x.(x\ x))(\lambda x.(x\ x))$. On the other hand typed $\lambda^{\to}_{\beta\eta}$ in the typed case is terminating cf. Theorem 25. This will be verified with the presented method following [Kah95a]:

The type interpretation is:

$$\mathfrak{b}(term) := \omega, \quad \mathfrak{b}(\iota \to \kappa) := \mathfrak{b}(\iota) \Longrightarrow \mathfrak{b}(\kappa), \quad \mathfrak{b}(\iota \Rightarrow \kappa) := \mathfrak{b}(\iota) \Longrightarrow \mathfrak{b}(\kappa)$$

where $\to$ is the type constructor on the meta-level and $\Rightarrow$ is the type constructor on the object-level. $\Longrightarrow$ on the right hand side of the definitions corresponds to the functor defined in [Kah95a].

The symbol interpretation is:

$$\varrho(abs_{\iota,\kappa})\ f := 1 \boxplus_{\iota \Rightarrow \kappa} f, \quad \varrho(app)(f, g) := f(g)$$

Termination can then be seen by the following calculation:

$$\begin{array}{rcllcl}
[\![app(abs(\lambda v.(t\ v)), s)]\!] & = & [\![(abs(\lambda v.(t\ v)))]\!][\![s]\!] & & = & (1 \boxplus [\![(\lambda v.(t\ v))]\!])[\![s]\!] \\
& = & (1 \boxplus (z \mapsto out(z) \boxplus [\![(t\ v)]\!]_{[v \mapsto z]})[\![s]\!] & & & \\
& > & (z \mapsto out(z) \boxplus [\![t]\!][\![v]\!]_{[v \mapsto z]})[\![s]\!] & & = & out([\![s]\!]) \boxplus [\![t]\!][\![s]\!] \\
& & & & \geq & [\![(t\ s)]\!] \\
[\![abs(\lambda v.app(t, v))]\!] & = & 1 \boxplus [\![\lambda v.app(t\ v)]\!] & & & \\
& = & 1 \boxplus (z \mapsto out(z) \boxplus [\![app(t\ v)]\!]_{v \mapsto z}) & & & \\
& > & (z \mapsto out(z) \boxplus [\![t]\!][\![v]\!]_{v \mapsto z}) & & \geq & (z \mapsto [\![t]\!][\![v]\!]_{v \mapsto z}) \\
& & & & = & [\![t]\!]
\end{array}$$

Some intermediate steps used propositions from [Kah95a] including that meta-level β-reduction on the right hand side can be ignored since this cannot increase the interpretation.

6.3 Monadic Laws as a HRS

6.3.1 A Canonical System

Now the laws of the monadic style will be turned into a canonical HRS.

Definition 38 (monadic laws as HRS) *Let MFL be the following higher-order rewrite system:*

$$
\begin{array}{rcl}
\mathcal{B} & := & \{term\} \\
\mathcal{C} & := & \{unit, star\} \\
unit & : & term \rightarrow term \\
star & : & (term \rightarrow term) \rightarrow term \rightarrow term
\end{array}
$$

with the higher-order rewrite rules:

$$
\begin{array}{lrcl}
(\mathbf{su\_}) & (star\ unit\ m) & \rightarrow & m \\
(\mathbf{s\_u}) & (star\ m\ (unit\ n)) & \rightarrow & (m\ n) \\
(\mathbf{s\_s}) & (star\ m\ (star\ n\ o)) & \rightarrow & (star\ (\lambda v.(star\ m\ (n\ v)))\ o)
\end{array}
$$

Remark. In this system all the type information has been suppressed since instead the typing can also be considered as a constraint. The left-hand sides of the rules are indeed patterns because of the convention to associate to the left. The rules had to be formulated in this way in order to be of base type.

Proposition 32 (local confluence of MFL) *The HRS MFL is locally confluent.*

Proof.
The proof is done by checking higher-order critical pairs:

$$
\begin{array}{rl}
(star\ unit\ (unit\ m)) \rightarrow_{su\_} & (unit\ m) \\
\rightarrow_{s\_u} & (unit\ m) \\
(star\ unit\ (star\ m\ n)) \rightarrow_{su\_} & (star\ m\ n) \\
\rightarrow_{s\_s} & (star\ (\lambda v.(star\ unit\ (m\ v)))\ n) \\
 & (star\ (\lambda v.(m\ v))\ n) \leftarrow_{su\_} \\
 & (star\ m\ n) \leftarrow_{\eta} \\
(star\ m\ (star\ unit\ n)) \rightarrow_{su\_} & (star\ m\ n) \\
\rightarrow_{s\_s} & (star\ (\lambda v.(star\ m\ (unit\ v)))\ n) \\
 & (star\ (\lambda v.(m\ v))\ n) \leftarrow_{s\_u} \\
 & (star\ m\ n) \leftarrow_{\eta} \\
(star\ m\ (star\ n\ (unit\ o))) \rightarrow_{s\_u} & (star\ m\ (n\ o)) \\
\rightarrow_{s\_s} & (star\ (\lambda v.(star\ m\ (n\ v)))\ (unit\ o)) \\
 & ((\lambda v.(star\ m\ (n\ v)))\ o) \leftarrow_{s\_u} \\
 & (star\ m\ (n\ o)) \leftarrow_{\beta}
\end{array}
$$

$$
\begin{array}{rl}
(star\ m\ (star\ n\ (star\ o\ p))) \rightarrow_{s\_s} & (star\ (\lambda v.(star\ m\ (n\ v)))\ (star\ o\ p)) \\
& (star\ (\lambda w.(star\ (\lambda v.(star\ m\ (n\ v)))\ (o\ w)))\ p) \leftarrow_{s\_s} \\
\rightarrow_{s\_s} & (star\ m\ (star\ (\lambda v.(star\ n\ (o\ v)))\ p)) \\
& (star\ (\lambda w.(star\ m\ ((\lambda v.(star\ n\ (o\ v)))\ w)))\ p) \leftarrow_{s\_s} \\
& (star\ (\lambda w.(star\ m\ (star\ n\ (o\ w))))\ p) \leftarrow_{\beta} \\
& (star\ (\lambda w.(star\ (\lambda v.(star\ m\ (n\ v)))\ (o\ w)))\ p) \leftarrow_{s\_s}
\end{array}
$$

All these critical pairs converge hence MFL is locally confluent. The check of critical pairs can also be seen in the Appendix C.2.2 as the first trace. □

Remark. It is important to orient the associativity law s_s in this way in order to achieve local confluence. Although MFL is a left linear system it is neither a OHRS nor a WOHRS.

Theorem 33 (termination of MFL) *MFL is terminating.*

Proof.
Now we apply the method from [Kah95a]. The type interpretation is:

$$\mathfrak{b}(term) := \omega, \quad \mathfrak{b}(\iota \rightarrow \kappa) := \mathfrak{b}(\iota) \Longrightarrow \mathfrak{b}(\kappa)$$

where we only consider the type constructor $\rightarrow$ of the meta-level, $\Longrightarrow$ is the functor from [Kah95a].

The symbol interpretation is:

$$\varrho(unit)\ f := f, \quad \varrho(star)\ f\ g := (1 + out(g)) \boxplus f(g)$$

Termination can the be seen by the following calculation while exploiting the facts from Proposition 31:

$$
\begin{array}{rlcl}
[\![(star\ unit\ m)]\!] = & (1 + out([\![m]\!])) \boxplus [\![unit]\!][\![m]\!] & & \\
= & (1 + out([\![m]\!])) \boxplus [\![m]\!] & > & [\![m]\!] \\
[\![(star\ m\ (unit\ n))]\!] = & (1 + out([\![unit\ n]\!])) \boxplus [\![m]\!][\![unit\ n]\!] & & \\
= & (1 + out([\![n]\!])) \boxplus [\![m]\!][\![n]\!] & > & [\![m]\!][\![n]\!] \\
[\![(star\ m\ (star\ n\ o))]\!] = & (1 + out([\![(star\ n\ o)]\!])) \boxplus [\![m]\!][\![(star\ n\ o)]\!] & & \\
= & (1 + out((1 + out([\![o]\!])) \boxplus [\![n]\!][\![o]\!])) & & \\
& \boxplus[\![m]\!]((1 + out([\![o]\!])) \boxplus [\![n]\!][\![o]\!]) & & \\
\geq & (2 + out([\![o]\!]) + out([\![n]\!][\![o]\!])) & & \\
& \boxplus((1 + out([\![o]\!])) \boxplus [\![m]\!][\![n]\!][\![o]\!]) & & \\
= & (3 + 2 * out([\![o]\!]) + out([\![n]\!][\![o]\!])) \boxplus [\![m]\!][\![n]\!][\![o]\!] & & \\
> & (2 + 2 * out([\![o]\!]) + out([\![n]\!][\![o]\!])) \boxplus [\![m]\!][\![n]\!][\![o]\!] & & \\
= & (1 + out([\![o]\!])) \boxplus ((out([\![o]\!]) + 1 + & & \\
& out([\![n]\!][\![o]\!])) \boxplus [\![m]\!][\![n]\!][\![o]\!]) & & \\
= & [\![star]\!](z \mapsto out(z) \boxplus [(1 + out([\![n]\!][\![v]\!]))\boxplus & & \\
& [\![m]\!][\![n]\!][\![v]\!]_{v \mapsto z})[\![o]\!] & & \\
= & [\![star]\!](z \mapsto out(z) \boxplus [\![(star\ m\ (n\ v))]\!]_{v \mapsto z})[\![o]\!] & & \\
= & [\![(star\ \lambda v.(star\ m\ (n\ v))\ o)]\!] & &
\end{array}
$$

□

Corollary 34 *MFL is a canonical HRS. Furthermore it remains canonical when types and the action of the type constructor T are again taken into account.*

```
%%% definitions for typed lambda calculus
tp    : type.
term  : tp -> type.
=>    : tp -> tp -> tp.
%infix right 10 =>

abs   : (term A -> term B) -> term (A => B).
app   : term (A => B) -> term A -> term B.
```

Table 6.1: Lambda Calculus Coded with Dependent Types

```
%%% definitions for monadic style with type information
tp    : type.
term  : tp -> type.
t     : tp -> tp.

unit  : term A -> term (t A).
star  : (term A -> term (t B)) -> term (t A) -> term (t B).
```

Table 6.2: Monadic Laws Coded with Dependent Types

Proof.
MFL is canonical since it is locally confluent and terminating (i.e. also confluent). Local confluence with typing holds since the joining of critical pairs did not conflict with the type information (this can be verified by an analysis of the reductions). Finally the presence of types does not influence the proof of termination since it only restricts the applicability of rules. In the Appendix C.2.2 the second trace of critical pairs shows local confluence with the help of dependent types. □

6.3.2 Using Dependent Types

Here we argue for the importance of *dependent types* in the context of HRS. Since Elf is based on the Logical Framework LF [HHP93] it allows also dependent types (types indexed by objects). We will demonstrate their application by means of examples.

Example. Let us consider again lambda calculus and we focus on the typed variant. The typing is not reflected in a HRS. In Elf the static definitions for a typed calculus looks like in Table 6.1.

This code clearly reflects that terms are indexed by a type and the definition of the constants makes use of this information. Also for the monadic laws we can give a better encoding as in Table 6.2.

As a last demonstration of dependent types we encode the constants for higher-order primitive recursion from Gödel's system T which can be seen in

```
%%% definitions for higher order primitive recursion
tp   : type.
term : tp -> type.
=>   : tp -> tp -> tp.
%infix right 10 =>
nat  : tp.

z    : term nat.
s    : term nat -> term nat.
abs  : (term A -> term B) -> term (A => B).
app  : term (A => B) -> term A -> term B.
rec  : term A -> term (nat => A => A) -> term nat -> term A.
```

Table 6.3: Higher-Order Primitive Recursion Constants

Table 6.3.
Remark. The example in the Appendix C.2 shows how the extension of Elf is actually used for checking critical pairs also for dependent types. It is worth to notice that the examples of dependent types shown here can still be treated with the method of Kahrs to prove termination. A particular interesting case is higher-order primitive recursion as part of Gödel's system T which is presented in Appendix C.2.3.

Note. Elf was already suitable to express higher-order rewriting since the necessary unification is built into the language by a constraint solving algorithm. A similar approach can be found in [Fel92]. In Elf it is even possible to generate critical pairs by a program as can be seen in Appendix C.1. But it was not possible to check whether critical pairs converge because free variables which were useful for the unification of left-hand sides might get bound by a rewriting attempt. Thus in cooperation with Frank Pfenning we extended the SML implementation of Elf by an additional function on the SML level which allows this check of convergence of critical pairs. Traces of this extension of Elf can be found in the Appendix C.2.

Chapter 7

Concluding Remarks

As already shown by Huet categorical notions are accessible by rewriting techniques. The examples from this thesis focusing on the notion of a monad demonstrate:

- the orientation of the rules is not always obvious,
- the proof of termination requires special techniques,
- the check of local confluence has to deal with the object, not just morphism level,
- the original specification is not always the best for rewriting.

Huet in [Hue90a] concentrated his study on categorical notions related to Cartesian closed categories. Because of the correspondence to certain typed lambda calculi it would be of great interest to construct a canonical rewriting system for Cartesian closed categories. This would allow a different treatment of bound variables and have consequences on the implementation of functional languages.

We studied in this thesis notions related to monads since they are a tool to investigate different semantics of programming languages and they are applied to purely functional programming. Our canonical systems help to mechanize proofs which deal with related questions. Furthermore in the thesis a technique is demonstrated how to approach categorical notions from the rewriting and thus algebraic point of view.

The proofs have been assisted by a computer in checking critical pairs which is getting more complicated in the presence of the additional object information.

Summarizing, with our approach as developed in the thesis, we solved the initially stated problem of checking commutativity of morphism (arrow) diagrams for a selected class of categories. We are treating the problem as a rewriting problem. It would be interesting to extend the application of our methods and principles to a broader spectrum of categories. This is intended future work.

The rewriting based decision procedures have been finally implemented in the logical programming language Elf rather than in a traditional rewriting tool. Concerning the use of Elf we see the following advantages in contrast to other systems:

1. Dependent types help to represent morphisms more easily since they depend on objects (or even different categories).

2. The type reconstruction of Elf helps to minimize the necessary input but also checks the correctness of the input already with respect to correct composition of morphisms (which is not done in a traditional rewriting tool).

3. Since Elf is a programming language one can vary the strategy for rewriting in a way appropriate for the problem.

4. Proofs can be inspected since they are represented as terms in Elf having the type of the proposition to be proved via the "proposition as types principle".

5. Elf allows proofs on the meta-level, too, e.g. one can show at least the correctness of the proof procedure with respect to an axiomatized theory, here equality.

6. A higher-order rewrite system can be coded in Elf since it contains the necessary primitives.

7. Higher-order rewriting is enriched by the presence of dependent types.

Related to higher-order rewriting the thesis left some problems open which would have been useful in the study of monadic functional programming and which are interesting topics of future work:

- How can higher-order rewriting be extended to dependent types?

- Can the critical pair lemma be lifted to this setting?

- Do dependent types allow a better treatment of termination for higher-order rewrite systems?

First steps in this directions are already undertaken cf. [Vir95]. Furthermore the method by Kahrs [Kah95a] remains applicable for special dependencies.

Another topic for further research is the technique of "diagram chasing" from a graphical point of view. In our thesis work we focused on the algebraic side. In the area of automated theorem proving more and more the emphasis is moving towards proof checking rather than trying to achieve completely automatic proofs. We have a similar standpoint concerning our current and future work.

It would be desirable to have a system which assists the visual graphical technique of diagram handling. The results, methods and implementations of our thesis can be fruitfully combined with *graphically oriented systems* (modulo a necessary interface). Work in this direction is in the planning phase.

Bibliography

[AGM92] S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors. *Handbook of Logic in Computer Science*, volume 2. Oxford Science Publications, 1992.

[AHS90] J. Adámek, H. Herrlich, and G.E. Strecker. *Abstract and Concrete Categories.* Wiley-Interscience Series in Pure and Applied Mathematics. John Wiley & Sons, Inc., 1990.

[AL91] A. Asperti and G. Longo. *Categories, Types, and Structures – An introduction to category theory for the working computer scientist.* Foundations of Computing. MIT Press, 1991.

[Bac91] L. Bachmair. *Canonical Equational Proofs.* Progress in Theoretical Computer Science. Birkhäuser, 1991.

[Bar84] H.P. Barendregt. *The Lambda Calculus - Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science Publisher, revised edition, 1984.

[Bar92] H.P. Barendregt. *Handbook of Logic in Computer Science*, volume 2, chapter Lambda Calculi with Types, pages 117–309. Oxford Science Publications, 1992.

[BG92] S. Brookes and S. Geva. Computational Comonads and Intensional Semantics. In M.P. Fourman and P.T. Johnstone and A.M. Pitts, editor, *Categories in Computer Science*, number 177 in London Mathematical Society Lecture Notes, pages 1–44. Cambridge University Press, 1992.

[Bir35] G. Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.

[BO93] R.V. Book and F. Otto. *String-Rewriting Systems.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Buc87] B. Buchberger. History and Basic Features of the Critical-Pair/Completion procedure. *Journal of Symbolic Computation*, 3:3–38, 1987.

[Bur93] A. Burroni. Higher Dimensional Word Problems. *Theoretical Computer Science*, 115:43–62, 1993.

[BW85] M. Barr and C. Wells. *Toposes, Triples and Theories.* Number 278 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1985.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science.* International Series in Computer Science. Prentice Hall, 1990.

[Cro93] R.L. Crole. *Categories for Types.* Cambridge Mathematical Textbooks. Cambridge University Press, 1993.

[Cur93] P.-L. Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming.* Progress in Theoretical Computer Science. Birkhäuser, second edition, 1993.

[Der87] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1 & 2):69–116, 1987.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter Rewrite Systems, pages 243–320. Elsevier, MIT Press, 1990.

[Ebb86] H. Ebbinghaus. *Einführung in die mathematische Logik.* Wissenschaftliche Buchgesellschaft Darmstadt, 1986.

[Fel92] A. Felty. A logic programming approach to implementing higher-order rewrite term rewriting. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, volume 596 of *Lecture Notes in Computer Science*, pages 135–161. Springer-Verlag, 1992.

[Fil94] A. Filinski. Representing Monads. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457. ACM, 1994.

[Fok91] M.M. Fokkinga. Calculate categorically. Technical Report CS-R9132, CWI, Amsterdam, 1991.

[FS90] P.J. Freyd and A. Scedrov. *Categories, Allegories.* Elsevier Science Publishers, 1990.

[GG91] S.J. Garland and J.V. Guttag. *A Guide to LP, The Larch Prover.* Massachusetts Institute of Technology, 1991.

[HHP93] R. Harper, F. Honsel, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HL86] T. Hardin and A. Laville. Proof of Termination of the Rewriting System SUBST on CCL. *Theoretical Computer Science*, 46:306–312, 1986.

[HP91a] H. Herrlich and H.-E. Porst. *Category Theory at Work.* Number 18 in Research and Expositions in Mathematics. Heldermann-Verlag, Berlin, 1991.

[HP91b] G. Huet and G. Plotkin, editors. *Logical Frameworks.* Cambridge University Press, 1991.

[HS79] H. Herrlich and G.E. Strecker. *Category Theory.* Number 1 in Sigma Series in Pure Mathematics. Heldermann-Verlag, Berlin, second edition, 1979.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Number 1 in London Mathematical Society Student Texts. Cambridge University Press, 1986.

[Hue80] G. Huet. Confluent Reductions. *Journal of the Association for Computing Machinery*, 24(4):797–821, October 1980.

[Hue81] G. Huet. A complete proof of the correctness of the Knuth-Bendix completion algorithm. *Journal of Computation System Sciences*, 23:11–21, 1981.

[Hue90a] G. Huet. Cartesian closed categories and lambda-calculus. In *[Hue90b]*, pages 7–23. Addison Wesley, 1990.

[Hue90b] G. Huet. *Logical Foundations of Functional Programming.* University of Texas at Austin Programming Series. Addison Wesley, 1990.

[Kah95a] S. Kahrs. Towards a Domain Theory for Termination Proofs. In J. Hsiang, editor, *Proceedings of the 6th International Conference, Rewriting Techniques and Applications, RTA-95*, number 914 in Lecture Notes in Computer Science, pages 241–255. Springer-Verlag, 1995. more detailed in [Kah95b].

[Kah95b] S. Kahrs. Towards a Domain Theory for Termination Proofs. Technical Report ECS-LFCS-95-314, LFCS, Department of Computer Science, University of Edinburgh , 1995.

[KB70] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Klo92] J.W. Klop. *Handbook of Logic in Computer Science*, volume 2, chapter Term Rewriting Systems, pages 2–116. Oxford Science Publications, 1992.

[KZ89] D. Kapur and H. Zhang. *RRL: Rewrite Rule Laboratory User's Manual*, revised edition, May 1989.

[Laf92] Y. Lafont. Penrose diagrams and 2-dimensional rewriting. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science*, number 177 in London Mathematical Society Lecture Notes, pages 191–201. Cambridge University Press, 1992.

[Laf93] Y. Lafont. Equational reasoning with 2-dimensional diagrams. Technical report, Laboratoire de Mathématique Discrètes, Université Marseille, 1993.

[Lan75] D.S. Lankford. Canonical algebraic simplification in computational logic. Technical report, University of Texas, Austin, TX, 1975.

[Lan79] D.S. Lankford. On proving term rewriting systems are Noetherian. Technical report, Louisiana Technical University, Ruston, LA, 1979.

[Law66] F.W. Lawvere. The category of categories as a foundation of mathematics. In S. Eilenberg, D.K. Harrison, S. MacLane, and H. Rohrl, editors, *Proceedings of the Conference on Categorical Algebra*, pages 1–20. Springer-Verlag, 1966.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation. Springer-Verlag, second, extended edition, 1987.

[LP95] O. Lysne and J. Piris. A Termination Ordering for Higher Order Rewrite Systems. In J. Hsiang, editor, *Rewriting Techniques and Applications, RTA-95*, number 914 in Lecture Notes in Computer Science, pages 26–40. Springer-Verlag, 1995.

[LS86] J. Lambek and P.J. Scott. *Introduction to higher-order categorical logic.* Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1986.

[Man76] E. Manes. *Algebraic Theories.* Number 26 in Graduate Texts in Mathematics. Springer-Verlag, 1976.

[Mil91] D. Miller. A logic programming language with Lambda-Abstraction, Function variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[ML71] S. Mac Lane. *Categories for the Working Mathematician.* Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.

[MLM92] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic.* Universitext. Springer-Verlag, 1992.

[MN94] R. Mayr and T. Nipkow. Higher-Order Rewrite Systems and their Confluence. Technical Report TUM-I9433, Technical University, Munich, August 1994.

[Mog89] E. Moggi. Computational Lambda-calculus and Monads. In *Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

[New42] M.H.A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.

[Nip91] T. Nipkow. Higher-Order Critical Pairs. In *Sixth Annual Symposium on Logic in Computer Science*, pages 342–349. IEEE, July 1991.

[Nip93] T. Nipkow. Orthogonal Higher-Order Rewrite Systems are Confluent. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 306–317. Springer-Verlag, 1993.

[Pfa95a] J. Pfalzgraf. *Automated Practical Reasoning*, chapter On a general notion of a hull, pages 39–51. Texts and Monographs in Symbolic Computation. Springer-Verlag, 1995.

[Pfa95b] J. Pfalzgraf. Towards a general semantics for relational structures: A categorical approach. invited contribution to a special issue of Studia Logica in honor of Helena Rasiowa, 1995. forthcoming.

[Pfe89] F. Pfenning. Elf: a language for verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE, June 1989.

[Pfe91] F. Pfenning. Logic Programming in the LF logical Framework. In *[HP91b]*, pages 149–181. Cambridge University Press, 1991.

[Pfe94] F. Pfenning. Elf: A Meta-Language for Deductive Systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, number 814 in Lecture Notes in Artificial Intelligence, pages 811–815, 1994.

[Pie91] B.C. Pierce. *Basic Category Theory for Computer Scientists.* Foundations of Computing. MIT Press, 1991.

[Pow90a] A.J. Power. A 2-Categorical Pasting Theorem. *Journal of Algebra*, 129:439–445, 1990.

[Pow90b] A.J. Power. An n-Categorical Pasting Theorem. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Proceedings for Category Theory*, number 1488 in Lecture Notes in Mathematics, pages 326–358. Springer-Verlag, 1990.

[PW95] J. Pfalzgraf and D. Wang. *Automated Practical Reasoning.* Springer-Verlag, 1995.

[RB88] D.E. Rydeheard and R.M. Burstall. *Computational Category Theory.* International Series in Computer Science. Prentice Hall, 1988.

[vdP94] J. van de Pol. Termination proofs for higher-order rewrite systems. In J. Heering, K. Meinke, B. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic and Term Rewriting, HOA'93*, volume 816 of *Lecture Notes in Computer Science*, pages 305–325. Springer-Verlag, 1994.

[vdPS95] J. van de Pol and H. Schwichtenberg. Strict Functionals for Termination Proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications, TLCA'95*, number 902 in Lecture Notes in Computer Science, pages 350–364, 1995.

[Vir95] R. Virga. Higher-Order Superposition for Dependent Types. Technical Report CMU-CS-95-150, Carnegie Mellon University, 1995. to appear.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B. Elsevier, MIT Press, 1990.

[vO94] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, Netherlands, 1994.

[Vog78] E. Vogel. Unifikation von Morphismen (in German). Diplomarbeit, Universität Karlsruhe, 1978.

[Wad93] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and System Sciences*, pages 233–264. Springer-Verlag, 1993.

[WS82] D. Wätjen and W. Struckmann. An Algorithm for Verifying Equations of Morphisms in a Category. *Information Processing Letters*, 14(3):104–108, May 1982.

List of Symbols

Index

Appendix A

LP Traces

This appendix shows the traces of the runs of our Larch Prover programs. The used notation closely follows the text in the thesis. These traces establish the termination proofs related to polynomial interpretations and check local confluence.

A.1 Category

A.1.1 Cat1

Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/cat/term1.lplog´ on 15 May 1995
18:58:49.
LP2: execute term1
LP2.1: declare sort Mor
LP2.2: declare variables f, g, h: Mor
LP2.3: declare operators
        id: -> Mor
        __*__: Mor, Mor -> Mor
..
LP2.4: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.5: register polynomial     id      1
LP2.6: register polynomial     *       2 * x + y
LP2.7: assert
        id * f = f;
        f * id = f;
        (f * g) * h = f * (g * h)
..
Added 3 facts named user.1, ..., user.3 to the system.
The system now contains 3 rewrite rules.  The rewriting system is guaranteed to
terminate.
LP2.8: complete
The system is complete.
LP2.9: display
Rewrite rules:
user.1: id * f -> f
user.2: f * id -> f
user.3: f * g * h -> f * (g * h)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/cat/term1.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/cat/type1.lplog´ on 15 May 1995
```

```
18:58:53.
LP2: execute type1
LP2.1: declare sort Obj, Mor´, Mor
LP2.2: declare variables o1, o2, o3, o4: Obj
LP2.3: declare variables f´, g´, h´: Mor´
LP2.4: declare operators
        mor: Obj, Mor´, Obj -> Mor
        id: -> Mor´
        __*__: Mor, Mor -> Mor´
..
LP2.5: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.6: assert
        mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) = mor(o2, f´, o1);
        mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) = mor(o2, f´, o1);
        mor(o4,mor(o4,mor(o4,f´,o3)*mor(o3,g´,o2),o2)*mor(o2,h´,o1),o1) =
        mor(o4,mor(o4,f´,o3)*mor(o3,mor(o3,g´,o2)*mor(o2,h´,o1),o1),o1)
..
Added 3 facts named user.1, ..., user.3 to the system.
The system now contains 3 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.7: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.8: display
Rewrite rules:
user.1: mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) -> mor(o2, f´, o1)
user.2: mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) -> mor(o2, f´, o1)
user.3: mor(o4,
            mor(o4, mor(o4, f´, o3) * mor(o3, g´, o2), o2) * mor(o2, h´, o1),
            o1)
          ->
          mor(o4,
              mor(o4, f´, o3) * mor(o3, mor(o3, g´, o2) * mor(o2, h´, o1), o1),
              o1)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/cat/type1.lp´.
LP3: quit
```

A.1.2 Cat2

Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/cat/term2.lplog´ on 15 May 1995
18:58:51.
LP2: execute term2
LP2.1: declare sort Mor
LP2.2: declare variables f, g, h: Mor
LP2.3: declare operators
        id: -> Mor
        __*__: Mor, Mor -> Mor
..
LP2.4: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.5: register polynomial     id      1
LP2.6: register polynomial     *       x + 2 * y
LP2.7: assert
        id * f = f;
        f * id = f;
        (f * g) * h = f * (g * h)
..
Added 3 facts named user.1, ..., user.3 to the system.
The system now contains 3 rewrite rules.  The rewriting system is guaranteed to
terminate.
LP2.8: complete
The system is complete.
LP2.9: display
Rewrite rules:
user.1: id * f -> f
user.2: f * id -> f
```

```
user.3: f * (g * h) -> f * g * h
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/cat/term2.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/cat/type2.lplog´ on 15 May 1995
18:58:56.
LP2: execute type2
LP2.1: declare sort Obj, Mor´, Mor
LP2.2: declare variables o1, o2, o3, o4: Obj
LP2.3: declare variables f´, g´, h´: Mor´
LP2.4: declare operators
        mor: Obj, Mor´, Obj -> Mor
        id: -> Mor´
        __*__: Mor, Mor -> Mor´
..
LP2.5: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.6: assert
        mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) = mor(o2, f´, o1);
        mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) = mor(o2, f´, o1);
        mor(o4,mor(o4,f´,o3)*mor(o3,mor(o3,g´,o2)*mor(o2,h´,o1),o1),o1) =
        mor(o4,mor(o4,mor(o4,f´,o3)*mor(o3,g´,o2),o2)*mor(o2,h´,o1),o1)
..
Added 3 facts named user.1, ..., user.3 to the system.
The system now contains 3 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.7: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.8: display
Rewrite rules:
user.1: mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) -> mor(o2, f´, o1)
user.2: mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) -> mor(o2, f´, o1)
user.3: mor(o4,
            mor(o4, f´, o3) * mor(o3, mor(o3, g´, o2) * mor(o2, h´, o1), o1),
            o1)
          ->
          mor(o4,
              mor(o4, mor(o4, f´, o3) * mor(o3, g´, o2), o2) * mor(o2, h´, o1),
              o1)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/cat/type2.lp´.
LP3: quit
```

A.2 Functor

A.2.1 Fun1

Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/fun/term1.lplog´ on 15 May 1995
19:00:12.
LP2: execute term1
LP2.1: declare sort Mor1, Mor2
LP2.2: declare variables f1, g1, h1: Mor1
LP2.3: declare variables f2, g2, h2: Mor2
LP2.4: declare operators
        id1: -> Mor1
        id2: -> Mor2
        __+__: Mor1, Mor1 -> Mor1
        __*__: Mor2, Mor2 -> Mor2
        f: Mor1 -> Mor2
..
LP2.5: set ordering polynomial
The ordering-method is now `polynomial´.
```

```
LP2.6: register polynomial     id1     1
LP2.7: register polynomial     id2     1
LP2.8: register polynomial     +       2 * x + y + 1
LP2.9: register polynomial     *       2 * x + y + 1
LP2.10: register polynomial     f       2 * x
LP2.11: assert
        id1 + f1 = f1;
        f1 + id1 = f1;
        (f1 + g1) + h1 = f1 + (g1 + h1);
        id2 * f2 = f2;
        f2 * id2 = f2;
        (f2 * g2) * h2 = f2 * (g2 * h2);
        f(id1) = id2;
        f(g1 + h1) = f(g1) * f(h1)
..
Added 8 facts named user.1, ..., user.8 to the system.
The system now contains 8 rewrite rules.  The rewriting system is guaranteed to
terminate.
LP2.12: complete
The system is complete.
LP2.13: display
Rewrite rules:
user.1: id1 + f1 -> f1
user.2: f1 + id1 -> f1
user.3: f1 + g1 + h1 -> f1 + (g1 + h1)
user.4: id2 * f2 -> f2
user.5: f2 * id2 -> f2
user.6: f2 * g2 * h2 -> f2 * (g2 * h2)
user.7: f(id1) -> id2
user.8: f(g1 + h1) -> f(g1) * f(h1)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/fun/term1.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/fun/type1.lplog´ on 15 May 1995
19:00:17.
LP2: execute type1
LP2.1: declare sort Obj1, Mor1´, Mor1
LP2.2: declare sort Obj2, Mor2´, Mor2
LP2.3: declare variables o11, o21, o31, o41: Obj1
LP2.4: declare variables o12, o22, o32, o42: Obj2
LP2.5: declare variables f1, g1, h1: Mor1´
LP2.6: declare variables f2, g2, h2: Mor2´
LP2.7: declare operators
        mor1: Obj1, Mor1´, Obj1 -> Mor1
        mor2: Obj2, Mor2´, Obj2 -> Mor2
        id1: -> Mor1´
        id2: -> Mor2´
        __+__: Mor1, Mor1 -> Mor1´
        __*__: Mor2, Mor2 -> Mor2´
        f´: Obj1 -> Obj2
        f : Mor1 -> Mor2´
..
LP2.8: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.9: assert
        mor1(o21,mor1(o21,id1,o21)+mor1(o21,f1,o11),o11) = mor1(o21,f1,o11);
        mor1(o21,mor1(o21,f1,o11)+mor1(o11,id1,o11),o11) = mor1(o21,f1,o11);
        mor1(o41,mor1(o41,mor1(o41,f1,o31)+mor1(o31,g1,o21),o21)+
                mor1(o21,h1,o11),o11) =
        mor1(o41,mor1(o41,f1,o31)+mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11),o11);
        mor2(o22,mor2(o22,id2,o22)*mor2(o22,f2,o12),o12) = mor2(o22,f2,o12);
        mor2(o22,mor2(o22,f2,o12)*mor2(o12,id2,o12),o12) = mor2(o22,f2,o12);
        mor2(o42,mor2(o42,mor2(o42,f2,o32)*mor2(o32,g2,o22),o22)*
                mor2(o22,h2,o12),o12) =
        mor2(o42,mor2(o42,f2,o32)*mor2(o32,mor2(o32,g2,o22)*
                mor2(o22,h2,o12),o12),o12);
```

```
        mor2(f´(o11),f(mor1(o11,id1,o11)),f´(o11)) = mor2(f´(o11),id2,f´(o11));
        mor2(f´(o31),f(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),f´(o11)) =
        mor2(f´(o31),mor2(f´(o31),f(mor1(o31,g1,o21)),f´(o21)) *
                mor2(f´(o21),f(mor1(o21,h1,o11)),f´(o11)),f´(o11))
..
Added 8 facts named user.1, ..., user.8 to the system.
The system now contains 8 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.10: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.11: display
Rewrite rules:
user.1: mor1(o21, mor1(o21, id1, o21) + mor1(o21, f1, o11), o11)
          -> mor1(o21, f1, o11)
user.2: mor1(o21, mor1(o21, f1, o11) + mor1(o11, id1, o11), o11)
          -> mor1(o21, f1, o11)
user.3: mor1(o41,
             mor1(o41, mor1(o41, f1, o31) + mor1(o31, g1, o21), o21)
               + mor1(o21, h1, o11),
             o11)
          ->
          mor1(o41,
               mor1(o41, f1, o31)
                 + mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11),
               o11)
user.4: mor2(o22, mor2(o22, id2, o22) * mor2(o22, f2, o12), o12)
          -> mor2(o22, f2, o12)
user.5: mor2(o22, mor2(o22, f2, o12) * mor2(o12, id2, o12), o12)
          -> mor2(o22, f2, o12)
user.6: mor2(o42,
             mor2(o42, mor2(o42, f2, o32) * mor2(o32, g2, o22), o22)
               * mor2(o22, h2, o12),
             o12)
          ->
          mor2(o42,
               mor2(o42, f2, o32)
                 * mor2(o32, mor2(o32, g2, o22) * mor2(o22, h2, o12), o12),
               o12)
user.7: mor2(f´(o11), f(mor1(o11, id1, o11)), f´(o11))
          -> mor2(f´(o11), id2, f´(o11))
user.8: mor2(f´(o31),
             f(mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11)),
             f´(o11))
          ->
          mor2(f´(o31),
               mor2(f´(o31), f(mor1(o31, g1, o21)), f´(o21))
                 * mor2(f´(o21), f(mor1(o21, h1, o11)), f´(o11)),
               f´(o11))
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/fun/type1.lp´.
LP3: quit
```

A.2.2 Fun2

Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/fun/term2.lplog´ on 15 May 1995
19:00:14.
LP2: execute term2
LP2.1: declare sort Mor1, Mor2
LP2.2: declare variables f1, g1, h1: Mor1
LP2.3: declare variables f2, g2, h2: Mor2
LP2.4: declare operators
        id1: -> Mor1
        id2: -> Mor2
        __+__: Mor1, Mor1 -> Mor1
        __*__: Mor2, Mor2 -> Mor2
        f: Mor1 -> Mor2
```

```
..
LP2.5: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.6: register polynomial     id1     1
LP2.7: register polynomial     id2     1
LP2.8: register polynomial     +       x * y + x
LP2.9: register polynomial     *       x * y + x
LP2.10: register polynomial     f       x + 1
LP2.11: assert
        id1 + f1 = f1;
        f1 + id1 = f1;
        (f1 + g1) + h1 = f1 + (g1 + h1);
        id2 * f2 = f2;
        f2 * id2 = f2;
        (f2 * g2) * h2 = f2 * (g2 * h2);
        f(id1) = id2;
        f(g1 + h1) = f(g1) * f(h1)
..
Added 8 facts named user.1, ..., user.8 to the system.
The system now contains 8 rewrite rules.  The rewriting system is guaranteed to
terminate.
LP2.12: complete
The following equations are critical pairs between rewrite rules user.8 and
user.6.
  user.9: f(g1 + h1) * h2 = f(g1) * (f(h1) * h2)
The system now contains 1 formula and 8 rewrite rules.  The rewriting system is
guaranteed to terminate.
The system now contains 9 rewrite rules.  The rewriting system is guaranteed to
terminate.
The system is complete.
LP2.13: display
Rewrite rules:
user.1: id1 + f1 -> f1
user.2: f1 + id1 -> f1
user.3: f1 + g1 + h1 -> f1 + (g1 + h1)
user.4: id2 * f2 -> f2
user.5: f2 * id2 -> f2
user.6: f2 * g2 * h2 -> f2 * (g2 * h2)
user.7: f(id1) -> id2
user.8: f(g1) * f(h1) -> f(g1 + h1)
user.9: f(g1) * (f(h1) * h2) -> f(g1 + h1) * h2
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/fun/term2.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/fun/type2.lplog´ on 15 May 1995
19:00:24.
LP2: execute type2
LP2.1: declare sort Obj1, Mor1´, Mor1
LP2.2: declare sort Obj2, Mor2´, Mor2
LP2.3: declare variables o11, o21, o31, o41: Obj1
LP2.4: declare variables o12, o22, o32, o42: Obj2
LP2.5: declare variables f1, g1, h1: Mor1´
LP2.6: declare variables f2, g2, h2: Mor2´
LP2.7: declare operators
        mor1: Obj1, Mor1´, Obj1 -> Mor1
        mor2: Obj2, Mor2´, Obj2 -> Mor2
        id1: -> Mor1´
        id2: -> Mor2´
        __+__: Mor1, Mor1 -> Mor1´
        __*__: Mor2, Mor2 -> Mor2´
        f´: Obj1 -> Obj2
        f : Mor1 -> Mor2´
..
LP2.8: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.9: assert
        mor1(o21,mor1(o21,id1,o21)+mor1(o21,f1,o11),o11) = mor1(o21,f1,o11);
```

```
        mor1(o21,mor1(o21,f1,o11)+mor1(o11,id1,o11),o11) = mor1(o21,f1,o11);
        mor1(o41,mor1(o41,mor1(o41,f1,o31)+mor1(o31,g1,o21),o21)+
                mor1(o21,h1,o11),o11) =
        mor1(o41,mor1(o41,f1,o31)+mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11),o11);
        mor2(o22,mor2(o22,id2,o22)*mor2(o22,f2,o12),o12) = mor2(o22,f2,o12);
        mor2(o22,mor2(o22,f2,o12)*mor2(o12,id2,o12),o12) = mor2(o22,f2,o12);
        mor2(o42,mor2(o42,mor2(o42,f2,o32)*mor2(o32,g2,o22),o22)*
                mor2(o22,h2,o12),o12) =
        mor2(o42,mor2(o42,f2,o32)*mor2(o32,mor2(o32,g2,o22)*
                mor2(o22,h2,o12),o12),o12);
        mor2(f´(o11),f(mor1(o11,id1,o11)),f´(o11)) = mor2(f´(o11),id2,f´(o11));
        mor2(f´(o31),mor2(f´(o31),f(mor1(o31,g1,o21)),f´(o21)) *
                mor2(f´(o21),f(mor1(o21,h1,o11)),f´(o11)),f´(o11)) =
        mor2(f´(o31),f(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),f´(o11));
        mor2(f´(o31),mor2(f´(o31),f(mor1(o31,g1,o21)),f´(o21))*
        mor2(f´(o21),mor2(f´(o21),f(mor1(o21,h1,o11)),f´(o11))*
        mor2(f´(o11),h2,o12),o12),o12) =
        mor2(f´(o31),mor2(f´(o31),f(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),f´(o11))*mor2(f´(o11),h2,o12),o12)
..
Added 9 facts named user.1, ..., user.9 to the system.
The system now contains 9 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.10: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.11: display
Rewrite rules:
user.1: mor1(o21, mor1(o21, id1, o21) + mor1(o21, f1, o11), o11)
          -> mor1(o21, f1, o11)
user.2: mor1(o21, mor1(o21, f1, o11) + mor1(o11, id1, o11), o11)
          -> mor1(o21, f1, o11)
user.3: mor1(o41,
             mor1(o41, mor1(o41, f1, o31) + mor1(o31, g1, o21), o21)
               + mor1(o21, h1, o11),
             o11)
          ->
          mor1(o41,
               mor1(o41, f1, o31)
                 + mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11),
               o11)
user.4: mor2(o22, mor2(o22, id2, o22) * mor2(o22, f2, o12), o12)
          -> mor2(o22, f2, o12)
user.5: mor2(o22, mor2(o22, f2, o12) * mor2(o12, id2, o12), o12)
          -> mor2(o22, f2, o12)
user.6: mor2(o42,
             mor2(o42, mor2(o42, f2, o32) * mor2(o32, g2, o22), o22)
               * mor2(o22, h2, o12),
             o12)
          ->
          mor2(o42,
               mor2(o42, f2, o32)
                 * mor2(o32, mor2(o32, g2, o22) * mor2(o22, h2, o12), o12),
               o12)
user.7: mor2(f´(o11), f(mor1(o11, id1, o11)), f´(o11))
          -> mor2(f´(o11), id2, f´(o11))
user.8: mor2(f´(o31),
             mor2(f´(o31), f(mor1(o31, g1, o21)), f´(o21))
               * mor2(f´(o21), f(mor1(o21, h1, o11)), f´(o11)),
             f´(o11))
          ->
          mor2(f´(o31),
               f(mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11)),
               f´(o11))
user.9: mor2(f´(o31),
             mor2(f´(o31), f(mor1(o31, g1, o21)), f´(o21))
               * mor2(f´(o21),
                      mor2(f´(o21), f(mor1(o21, h1, o11)), f´(o11))
                        * mor2(f´(o11), h2, o12),
                      o12),
```

```
            o12)
         ->
         mor2(f´(o31),
              mor2(f´(o31),
                   f(mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11)),
                   f´(o11))
                * mor2(f´(o11), h2, o12),
              o12)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/fun/type2.lp´.
LP3: quit
```

A.3 Natural Transformation

A.3.1 Nat1

Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/nat/term1.lplog´ on 15 May 1995
19:01:55.
LP2: execute term1
LP2.1: declare sort Mor1, Mor2
LP2.2: declare variables f1, g1, h1: Mor1
LP2.3: declare variables f2, g2, h2: Mor2
LP2.4: declare operators
        id1: -> Mor1
        id2: -> Mor2
        __+__: Mor1, Mor1 -> Mor1
        __*__: Mor2, Mor2 -> Mor2
        f: Mor1 -> Mor2
        g: Mor1 -> Mor2
        n: -> Mor2
..
LP2.5: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.6: register polynomial     id1     1
LP2.7: register polynomial     id2     1
LP2.8: register polynomial     +       2 * x + y + 1
LP2.9: register polynomial     *       2 * x + y + 1
LP2.10: register polynomial     f       6 * x
LP2.11: register polynomial     g       2 * x
LP2.12: register polynomial     n       1
LP2.13: assert
        id1 + f1 = f1;
        f1 + id1 = f1;
        (f1 + g1) + h1 = f1 + (g1 + h1);
        id2 * f2 = f2;
        f2 * id2 = f2;
        (f2 * g2) * h2 = f2 * (g2 * h2);
        f(id1) = id2;
        f(g1 + h1) = f(g1) * f(h1);
        g(id1) = id2;
        g(g1 + h1) = g(g1) * g(h1);
        n * f(f1) = g(f1) * n
..
Added 11 facts named user.1, ..., user.11 to the system.
The system now contains 11 rewrite rules.  The rewriting system is guaranteed
to terminate.
LP2.14: complete
The following equations are critical pairs between rewrite rules user.11 and
user.8.
  user.12: n * (f(g1) * f(h1)) = g(g1) * (g(h1) * n)
The system now contains 1 formula and 11 rewrite rules.  The rewriting system
is guaranteed to terminate.
The following equations are critical pairs between rewrite rules user.11 and
user.6.
  user.13: g(f1) * (n * h2) = n * (f(f1) * h2)
The system now contains 2 formulas and 11 rewrite rules.  The rewriting system
is guaranteed to terminate.
```

```
Deleted formula user.12, which reduced to `true´.
The system now contains 12 rewrite rules.  The rewriting system is guaranteed
to terminate.
The system is complete.
LP2.15: display
Rewrite rules:
user.1:  id1 + f1 -> f1
user.2:  f1 + id1 -> f1
user.3:  f1 + g1 + h1 -> f1 + (g1 + h1)
user.4:  id2 * f2 -> f2
user.5:  f2 * id2 -> f2
user.6:  f2 * g2 * h2 -> f2 * (g2 * h2)
user.7:  f(id1) -> id2
user.8:  f(g1 + h1) -> f(g1) * f(h1)
user.9:  g(id1) -> id2
user.10: g(g1 + h1) -> g(g1) * g(h1)
user.11: n * f(f1) -> g(f1) * n
user.13: n * (f(f1) * h2) -> g(f1) * (n * h2)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/nat/term1.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/nat/type1.lplog´ on 15 May 1995
19:02:03.
LP2: execute type1
LP2.1: declare sort Obj1, Mor1´, Mor1
LP2.2: declare sort Obj2, Mor2´, Mor2
LP2.3: declare variables o11, o21, o31, o41: Obj1
LP2.4: declare variables o12, o22, o32, o42: Obj2
LP2.5: declare variables f1, g1, h1: Mor1´
LP2.6: declare variables f2, g2, h2: Mor2´
LP2.7: declare operators
        mor1: Obj1, Mor1´, Obj1 -> Mor1
        mor2: Obj2, Mor2´, Obj2 -> Mor2
        id1: -> Mor1´
        id2: -> Mor2´
        __+__: Mor1, Mor1 -> Mor1´
        __*__: Mor2, Mor2 -> Mor2´
        f´: Obj1 -> Obj2
        f : Mor1 -> Mor2´
        g´ : Obj1 -> Obj2
        g : Mor1 -> Mor2´
        n : -> Mor2´
..
LP2.8: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.9: assert
        mor1(o21,mor1(o21,id1,o21)+mor1(o21,f1,o11),o11) = mor1(o21,f1,o11);
        mor1(o21,mor1(o21,f1,o11)+mor1(o11,id1,o11),o11) = mor1(o21,f1,o11);
        mor1(o41,mor1(o41,mor1(o41,f1,o31)+mor1(o31,g1,o21),o21)+
                mor1(o21,h1,o11),o11) =
        mor1(o41,mor1(o41,f1,o31)+mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11),o11);
        mor2(o22,mor2(o22,id2,o22)*mor2(o22,f2,o12),o12) = mor2(o22,f2,o12);
        mor2(o22,mor2(o22,f2,o12)*mor2(o12,id2,o12),o12) = mor2(o22,f2,o12);
        mor2(o42,mor2(o42,mor2(o42,f2,o32)*mor2(o32,g2,o22),o22)*
                mor2(o22,h2,o12),o12) =
        mor2(o42,mor2(o42,f2,o32)*mor2(o32,mor2(o32,g2,o22)*
                mor2(o22,h2,o12),o12),o12);
        mor2(f´(o11),f(mor1(o11,id1,o11)),f´(o11)) = mor2(f´(o11),id2,f´(o11));
        mor2(f´(o31),f(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),f´(o11)) =
        mor2(f´(o31),mor2(f´(o31),f(mor1(o31,g1,o21)),f´(o21)) *
                mor2(f´(o21),f(mor1(o21,h1,o11)),f´(o11)),f´(o11));
        mor2(g´(o11),g(mor1(o11,id1,o11)),g´(o11)) = mor2(g´(o11),id2,g´(o11));
        mor2(g´(o31),g(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),g´(o11)) =
        mor2(g´(o31),mor2(g´(o31),g(mor1(o31,g1,o21)),g´(o21)) *
```

```
                mor2(g´(o21),g(mor1(o21,h1,o11)),g´(o11)),g´(o11));
        mor2(g´(o21),mor2(g´(o21),n,f´(o21))*
                mor2(f´(o21),f(mor1(o21,h1,o11)),f´(o11)),f´(o11)) =
        mor2(g´(o21),mor2(g´(o21),g(mor1(o21,h1,o11)),g´(o11))*
                mor2(g´(o11),n,f´(o11)),f´(o11));
        mor2(g´(o31),mor2(g´(o31),n,f´(o31))*mor2(f´(o31),mor2(f´(o31),
                f(mor1(o31,h1,o21)),f´(o21))*mor2(f´(o21),h2,o12),o12),o12) =
        mor2(g´(o31),mor2(g´(o31),g(mor1(o31,h1,o21)),g´(o21))*mor2(g´(o21),
                mor2(g´(o21),n,f´(o21))*mor2(f´(o21),h2,o12),o12),o12)
..
Added 12 facts named user.1, ..., user.12 to the system.
The system now contains 12 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.10: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.11: display
Rewrite rules:
user.1:  mor1(o21, mor1(o21, id1, o21) + mor1(o21, f1, o11), o11)
           -> mor1(o21, f1, o11)
user.2:  mor1(o21, mor1(o21, f1, o11) + mor1(o11, id1, o11), o11)
           -> mor1(o21, f1, o11)
user.3:  mor1(o41,
              mor1(o41, mor1(o41, f1, o31) + mor1(o31, g1, o21), o21)
                + mor1(o21, h1, o11),
              o11)
           ->
           mor1(o41,
                mor1(o41, f1, o31)
                  + mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11),
                o11)
user.4:  mor2(o22, mor2(o22, id2, o22) * mor2(o22, f2, o12), o12)
           -> mor2(o22, f2, o12)
user.5:  mor2(o22, mor2(o22, f2, o12) * mor2(o12, id2, o12), o12)
           -> mor2(o22, f2, o12)
user.6:  mor2(o42,
              mor2(o42, mor2(o42, f2, o32) * mor2(o32, g2, o22), o22)
                * mor2(o22, h2, o12),
              o12)
           ->
           mor2(o42,
                mor2(o42, f2, o32)
                  * mor2(o32, mor2(o32, g2, o22) * mor2(o22, h2, o12), o12),
                o12)
user.7:  mor2(f´(o11), f(mor1(o11, id1, o11)), f´(o11))
           -> mor2(f´(o11), id2, f´(o11))
user.8:  mor2(f´(o31),
              f(mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11)),
              f´(o11))
           ->
           mor2(f´(o31),
                mor2(f´(o31), f(mor1(o31, g1, o21)), f´(o21))
                  * mor2(f´(o21), f(mor1(o21, h1, o11)), f´(o11)),
                f´(o11))
user.9:  mor2(g´(o11), g(mor1(o11, id1, o11)), g´(o11))
           -> mor2(g´(o11), id2, g´(o11))
user.10: mor2(g´(o31),
              g(mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11)),
              g´(o11))
           ->
           mor2(g´(o31),
                mor2(g´(o31), g(mor1(o31, g1, o21)), g´(o21))
                  * mor2(g´(o21), g(mor1(o21, h1, o11)), g´(o11)),
                g´(o11))
user.11: mor2(g´(o21),
              mor2(g´(o21), n, f´(o21))
                * mor2(f´(o21), f(mor1(o21, h1, o11)), f´(o11)),
              f´(o11))
           ->
           mor2(g´(o21),
                mor2(g´(o21), g(mor1(o21, h1, o11)), g´(o11))
                  * mor2(g´(o11), n, f´(o11)),
```

```
                f´(o11))
user.12: mor2(g´(o31),
              mor2(g´(o31), n, f´(o31))
                * mor2(f´(o31),
                       mor2(f´(o31), f(mor1(o31, h1, o21)), f´(o21))
                         * mor2(f´(o21), h2, o12),
                       o12),
              o12)
           ->
           mor2(g´(o31),
                mor2(g´(o31), g(mor1(o31, h1, o21)), g´(o21))
                  * mor2(g´(o21),
                         mor2(g´(o21), n, f´(o21)) * mor2(f´(o21), h2, o12),
                         o12),
                o12)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/nat/type1.lp´.
LP3: quit
```

A.3.2 Nat2

Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/nat/term2.lplog´ on 15 May 1995
19:01:59.
LP2: execute term2
LP2.1: declare sort Mor1, Mor2
LP2.2: declare variables f1, g1, h1: Mor1
LP2.3: declare variables f2, g2, h2: Mor2
LP2.4: declare operators
        id1: -> Mor1
        id2: -> Mor2
        __+__: Mor1, Mor1 -> Mor1
        __*__: Mor2, Mor2 -> Mor2
        f: Mor1 -> Mor2
        g: Mor1 -> Mor2
        n: -> Mor2
..
LP2.5: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.6: register polynomial     id1     1
LP2.7: register polynomial     id2     1
LP2.8: register polynomial     +       x * y + x
LP2.9: register polynomial     *       x * y + x
LP2.10: register polynomial     f       3 * x + 1
LP2.11: register polynomial     g       x + 1
LP2.12: register polynomial     n       1
LP2.13: assert
        id1 + f1 = f1;
        f1 + id1 = f1;
        (f1 + g1) + h1 = f1 + (g1 + h1);
        id2 * f2 = f2;
        f2 * id2 = f2;
        (f2 * g2) * h2 = f2 * (g2 * h2);
        f(id1) = id2;
        f(g1) * f(h1) = f(g1 + h1);
        g(id1) = id2;
        g(g1) * g(h1) = g(g1 + h1);
        n * f(f1) = g(f1) * n
..
Added 11 facts named user.1, ..., user.11 to the system.
The system now contains 11 rewrite rules.  The rewriting system is guaranteed
to terminate.
LP2.14: complete
The following equations are critical pairs between rewrite rules user.11 and
user.6.
  user.12: g(f1) * (n * h2) = n * (f(f1) * h2)
The system now contains 1 formula and 11 rewrite rules.  The rewriting system
is guaranteed to terminate.
The following equations are critical pairs between rewrite rules user.8 and
```

```
user.6.
  user.13: f(g1 + h1) * h2 = f(g1) * (f(h1) * h2)
The system now contains 2 formulas and 11 rewrite rules.  The rewriting system
is guaranteed to terminate.
The following equations are critical pairs between rewrite rules user.10 and
user.6.
  user.14: g(g1 + h1) * h2 = g(g1) * (g(h1) * h2)
The system now contains 3 formulas and 11 rewrite rules.  The rewriting system
is guaranteed to terminate.
The following equations are critical pairs between rewrite rules user.12 and
user.8.
  user.15: g(f1 + h1) * n = g(f1) * (g(h1) * n)
The system now contains 3 formulas and 12 rewrite rules.  The rewriting system
is guaranteed to terminate.
Deleted formula user.15, which reduced to `true´.
The system now contains 14 rewrite rules.  The rewriting system is guaranteed
to terminate.
The system is complete.
LP2.15: display
Rewrite rules:
user.1:  id1 + f1 -> f1
user.2:  f1 + id1 -> f1
user.3:  f1 + g1 + h1 -> f1 + (g1 + h1)
user.4:  id2 * f2 -> f2
user.5:  f2 * id2 -> f2
user.6:  f2 * g2 * h2 -> f2 * (g2 * h2)
user.7:  f(id1) -> id2
user.8:  f(g1) * f(h1) -> f(g1 + h1)
user.9:  g(id1) -> id2
user.10: g(g1) * g(h1) -> g(g1 + h1)
user.11: n * f(f1) -> g(f1) * n
user.12: n * (f(f1) * h2) -> g(f1) * (n * h2)
user.13: f(g1) * (f(h1) * h2) -> f(g1 + h1) * h2
user.14: g(g1) * (g(h1) * h2) -> g(g1 + h1) * h2
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/nat/term2.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/nat/type2.lplog´ on 15 May 1995
19:02:16.
LP2: execute type2
LP2.1: declare sort Obj1, Mor1´, Mor1
LP2.2: declare sort Obj2, Mor2´, Mor2
LP2.3: declare variables o11, o21, o31, o41: Obj1
LP2.4: declare variables o12, o22, o32, o42: Obj2
LP2.5: declare variables f1, g1, h1: Mor1´
LP2.6: declare variables f2, g2, h2: Mor2´
LP2.7: declare operators
        mor1: Obj1, Mor1´, Obj1 -> Mor1
        mor2: Obj2, Mor2´, Obj2 -> Mor2
        id1: -> Mor1´
        id2: -> Mor2´
        __+__: Mor1, Mor1 -> Mor1´
        __*__: Mor2, Mor2 -> Mor2´
        f´: Obj1 -> Obj2
        f : Mor1 -> Mor2´
        g´ : Obj1 -> Obj2
        g : Mor1 -> Mor2´
        n : -> Mor2´
..
LP2.8: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.9: assert
        mor1(o21,mor1(o21,id1,o21)+mor1(o21,f1,o11),o11) = mor1(o21,f1,o11);
        mor1(o21,mor1(o21,f1,o11)+mor1(o11,id1,o11),o11) = mor1(o21,f1,o11);
        mor1(o41,mor1(o41,mor1(o41,f1,o31)+mor1(o31,g1,o21),o21)+
                mor1(o21,h1,o11),o11) =
        mor1(o41,mor1(o41,f1,o31)+mor1(o31,mor1(o31,g1,o21)+
```

```
                mor1(o21,h1,o11),o11),o11);
        mor2(o22,mor2(o22,id2,o22)*mor2(o22,f2,o12),o12) = mor2(o22,f2,o12);
        mor2(o22,mor2(o22,f2,o12)*mor2(o12,id2,o12),o12) = mor2(o22,f2,o12);
        mor2(o42,mor2(o42,mor2(o42,f2,o32)*mor2(o32,g2,o22),o22)*
                mor2(o22,h2,o12),o12) =
        mor2(o42,mor2(o42,f2,o32)*mor2(o32,mor2(o32,g2,o22)*
                mor2(o22,h2,o12),o12),o12);
        mor2(f´(o11),f(mor1(o11,id1,o11)),f´(o11)) = mor2(f´(o11),id2,f´(o11));
        mor2(f´(o31),mor2(f´(o31),f(mor1(o31,g1,o21)),f´(o21)) *
                mor2(f´(o21),f(mor1(o21,h1,o11)),f´(o11)),f´(o11)) =
        mor2(f´(o31),f(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),f´(o11));
        mor2(f´(o31),mor2(f´(o31),f(mor1(o31,g1,o21)),f´(o21))*
        mor2(f´(o21),mor2(f´(o21),f(mor1(o21,h1,o11)),f´(o11))*
        mor2(f´(o11),h2,o12),o12),o12) =
        mor2(f´(o31),mor2(f´(o31),f(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),f´(o11))*mor2(f´(o11),h2,o12),o12);
        mor2(g´(o11),g(mor1(o11,id1,o11)),g´(o11)) = mor2(g´(o11),id2,g´(o11));
        mor2(g´(o31),mor2(g´(o31),g(mor1(o31,g1,o21)),g´(o21)) *
                mor2(g´(o21),g(mor1(o21,h1,o11)),g´(o11)),g´(o11)) =
        mor2(g´(o31),g(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),g´(o11));
        mor2(g´(o31),mor2(g´(o31),g(mor1(o31,g1,o21)),g´(o21))*
        mor2(g´(o21),mor2(g´(o21),g(mor1(o21,h1,o11)),g´(o11))*
        mor2(g´(o11),h2,o12),o12),o12) =
        mor2(g´(o31),mor2(g´(o31),g(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),g´(o11))*mor2(g´(o11),h2,o12),o12);
        mor2(g´(o21),mor2(g´(o21),n,f´(o21))*
                mor2(f´(o21),f(mor1(o21,h1,o11)),f´(o11)),f´(o11)) =
        mor2(g´(o21),mor2(g´(o21),g(mor1(o21,h1,o11)),g´(o11))*
                mor2(g´(o11),n,f´(o11)),f´(o11));
        mor2(g´(o31),mor2(g´(o31),n,f´(o31))*mor2(f´(o31),mor2(f´(o31),
                f(mor1(o31,h1,o21)),f´(o21))*mor2(f´(o21),h2,o12),o12),o12) =
        mor2(g´(o31),mor2(g´(o31),g(mor1(o31,h1,o21)),g´(o21))*mor2(g´(o21),
                mor2(g´(o21),n,f´(o21))*mor2(f´(o21),h2,o12),o12),o12)
..
Added 14 facts named user.1, ..., user.14 to the system.
The system now contains 14 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.10: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.11: display
Rewrite rules:
user.1:  mor1(o21, mor1(o21, id1, o21) + mor1(o21, f1, o11), o11)
           -> mor1(o21, f1, o11)
user.2:  mor1(o21, mor1(o21, f1, o11) + mor1(o11, id1, o11), o11)
           -> mor1(o21, f1, o11)
user.3:  mor1(o41,
              mor1(o41, mor1(o41, f1, o31) + mor1(o31, g1, o21), o21)
                + mor1(o21, h1, o11),
              o11)
           ->
           mor1(o41,
                mor1(o41, f1, o31)
                  + mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11),
                o11)
user.4:  mor2(o22, mor2(o22, id2, o22) * mor2(o22, f2, o12), o12)
           -> mor2(o22, f2, o12)
user.5:  mor2(o22, mor2(o22, f2, o12) * mor2(o12, id2, o12), o12)
           -> mor2(o22, f2, o12)
user.6:  mor2(o42,
              mor2(o42, mor2(o42, f2, o32) * mor2(o32, g2, o22), o22)
                * mor2(o22, h2, o12),
              o12)
           ->
           mor2(o42,
                mor2(o42, f2, o32)
                  * mor2(o32, mor2(o32, g2, o22) * mor2(o22, h2, o12), o12),
                o12)
user.7:  mor2(f´(o11), f(mor1(o11, id1, o11)), f´(o11))
           -> mor2(f´(o11), id2, f´(o11))
```

```
user.8:  mor2(f´(o31),
              mor2(f´(o31), f(mor1(o31, g1, o21)), f´(o21))
                * mor2(f´(o21), f(mor1(o21, h1, o11)), f´(o11)),
              f´(o11))
           ->
           mor2(f´(o31),
                f(mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11)),
                f´(o11))
user.9:  mor2(f´(o31),
              mor2(f´(o31), f(mor1(o31, g1, o21)), f´(o21))
                * mor2(f´(o21),
                       mor2(f´(o21), f(mor1(o21, h1, o11)), f´(o11))
                         * mor2(f´(o11), h2, o12),
                       o12),
              o12)
           ->
           mor2(f´(o31),
                mor2(f´(o31),
                     f(mor1(o31,
                            mor1(o31, g1, o21) + mor1(o21, h1, o11),
                            o11)),
                     f´(o11))
                  * mor2(f´(o11), h2, o12),
                o12)
user.10: mor2(g´(o11), g(mor1(o11, id1, o11)), g´(o11))
           -> mor2(g´(o11), id2, g´(o11))
user.11: mor2(g´(o31),
              mor2(g´(o31), g(mor1(o31, g1, o21)), g´(o21))
                * mor2(g´(o21), g(mor1(o21, h1, o11)), g´(o11)),
              g´(o11))
           ->
           mor2(g´(o31),
                g(mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11)),
                g´(o11))
user.12: mor2(g´(o31),
              mor2(g´(o31), g(mor1(o31, g1, o21)), g´(o21))
                * mor2(g´(o21),
                       mor2(g´(o21), g(mor1(o21, h1, o11)), g´(o11))
                         * mor2(g´(o11), h2, o12),
                       o12),
              o12)
           ->
           mor2(g´(o31),
                mor2(g´(o31),
                     g(mor1(o31,
                            mor1(o31, g1, o21) + mor1(o21, h1, o11),
                            o11)),
                     g´(o11))
                  * mor2(g´(o11), h2, o12),
                o12)
user.13: mor2(g´(o21),
              mor2(g´(o21), n, f´(o21))
                * mor2(f´(o21), f(mor1(o21, h1, o11)), f´(o11)),
              f´(o11))
           ->
           mor2(g´(o21),
                mor2(g´(o21), g(mor1(o21, h1, o11)), g´(o11))
                  * mor2(g´(o11), n, f´(o11)),
                f´(o11))
user.14: mor2(g´(o31),
              mor2(g´(o31), n, f´(o31))
                * mor2(f´(o31),
                       mor2(f´(o31), f(mor1(o31, h1, o21)), f´(o21))
                         * mor2(f´(o21), h2, o12),
                       o12),
              o12)
           ->
           mor2(g´(o31),
                mor2(g´(o31), g(mor1(o31, h1, o21)), g´(o21))
                  * mor2(g´(o21),
                         mor2(g´(o21), n, f´(o21)) * mor2(f´(o21), h2, o12),
                         o12),
```

```
                o12)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/nat/type2.lp´.
LP3: quit
```

A.4 Adjunctions

A.4.1 Adj

Partial Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/adj/term1.lplog´ on 15 May 1995
18:57:19.
LP2: execute term1
LP2.1: declare sort Mor1, Mor2
LP2.2: declare variables f1, g1, h1: Mor1
LP2.3: declare variables f2, g2, h2: Mor2
LP2.4: declare operators
        id1: -> Mor1
        id2: -> Mor2
        __+__: Mor1, Mor1 -> Mor1
        __*__: Mor2, Mor2 -> Mor2
        l: Mor1 -> Mor2
        r: Mor2 -> Mor1
        zeta: Mor2 -> Mor1
        zetai: Mor1 -> Mor2
        eta: -> Mor1
        eps: -> Mor2
..
LP2.5: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.6: register polynomial     id1     1
LP2.7: register polynomial     id2     1
LP2.8: register polynomial     +       x * y + x
LP2.9: register polynomial     *       x * y + x
LP2.10: register polynomial     l       2 * x
LP2.11: register polynomial     zeta    x + 1
LP2.12: assert
        id1 + f1 = f1;
        f1 + id1 = f1;
        (f1 + g1) + h1 = f1 + (g1 + h1);
        id2 * f2 = f2;
        f2 * id2 = f2;
        (f2 * g2) * h2 = f2 * (g2 * h2);
        l(id1) = id2;
        l(f1) * l(g1) = l(f1 + g1);
        l(f1) * (l(g1) * h2) = l(f1 + g1) * h2;
        zeta(eps * l(f1)) = f1;
        zeta(eps) = id1;
        eps * l(zeta(f2)) = f2;
        eps * (l(zeta(f2)) * g2) = f2 * g2;
..
Added 13 facts named user.1, ..., user.13 to the system.
The system now contains 13 rewrite rules.  The rewriting system is guaranteed
to terminate.
LP2.13: display
Rewrite rules:
user.1:  id1 + f1 -> f1
user.2:  f1 + id1 -> f1
user.3:  f1 + g1 + h1 -> f1 + (g1 + h1)
user.4:  id2 * f2 -> f2
user.5:  f2 * id2 -> f2
user.6:  f2 * g2 * h2 -> f2 * (g2 * h2)
user.7:  l(id1) -> id2
user.8:  l(f1) * l(g1) -> l(f1 + g1)
user.9:  l(f1) * (l(g1) * h2) -> l(f1 + g1) * h2
user.10: zeta(eps * l(f1)) -> f1
user.11: zeta(eps) -> id1
user.12: eps * l(zeta(f2)) -> f2
```

```
user.13: eps * (l(zeta(f2)) * g2) -> f2 * g2
LP2.14: set ordering left-to-right
Warning: the rewriting system is known to terminate, but its termination cannot
be verified with the `left-to-right´ ordering.  When another rewrite rule is
added, the rewriting system will be treated as nonterminating.
The ordering-method is now `left-to-right´.
LP2.15: assert
        zeta(f2) + f1 = zeta(f2 * l(f1));
..
Added 1 fact named user.14 to the system.
The system now contains 14 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.16: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.17: display
Rewrite rules:
user.1:  id1 + f1 -> f1
user.2:  f1 + id1 -> f1
user.3:  f1 + g1 + h1 -> f1 + (g1 + h1)
user.4:  id2 * f2 -> f2
user.5:  f2 * id2 -> f2
user.6:  f2 * g2 * h2 -> f2 * (g2 * h2)
user.7:  l(id1) -> id2
user.8:  l(f1) * l(g1) -> l(f1 + g1)
user.9:  l(f1) * (l(g1) * h2) -> l(f1 + g1) * h2
user.10: zeta(eps * l(f1)) -> f1
user.11: zeta(eps) -> id1
user.12: eps * l(zeta(f2)) -> f2
user.13: eps * (l(zeta(f2)) * g2) -> f2 * g2
user.14: zeta(f2) + f1 -> zeta(f2 * l(f1))
LP2.18: assert
        r(f2) = zeta(f2 * eps);
        zetai(f1) = eps * l(f1);
        eta = zeta(id2)
..
Added 3 facts named user.15, ..., user.17 to the system.
The system now contains 17 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.19: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.20: display
Rewrite rules:
user.1:  id1 + f1 -> f1
user.2:  f1 + id1 -> f1
user.3:  f1 + g1 + h1 -> f1 + (g1 + h1)
user.4:  id2 * f2 -> f2
user.5:  f2 * id2 -> f2
user.6:  f2 * g2 * h2 -> f2 * (g2 * h2)
user.7:  l(id1) -> id2
user.8:  l(f1) * l(g1) -> l(f1 + g1)
user.9:  l(f1) * (l(g1) * h2) -> l(f1 + g1) * h2
user.10: zeta(eps * l(f1)) -> f1
user.11: zeta(eps) -> id1
user.12: eps * l(zeta(f2)) -> f2
user.13: eps * (l(zeta(f2)) * g2) -> f2 * g2
user.14: zeta(f2) + f1 -> zeta(f2 * l(f1))
user.15: r(f2) -> zeta(f2 * eps)
user.16: zetai(f1) -> eps * l(f1)
user.17: eta -> zeta(id2)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/adj/term1.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/adj/type1.lplog´ on 15 May 1995
18:57:29.
LP2: execute type1
```

```
LP2.1: declare sort Obj1, Mor1´, Mor1
LP2.2: declare sort Obj2, Mor2´, Mor2
LP2.3: declare variables o11, o21, o31, o41: Obj1
LP2.4: declare variables o12, o22, o32, o42: Obj2
LP2.5: declare variables f1, g1, h1: Mor1´
LP2.6: declare variables f2, g2, h2: Mor2´
LP2.7: declare operators
        mor1: Obj1, Mor1´, Obj1 -> Mor1
        mor2: Obj2, Mor2´, Obj2 -> Mor2
        id1: -> Mor1´
        id2: -> Mor2´
        __+__: Mor1, Mor1 -> Mor1´
        __*__: Mor2, Mor2 -> Mor2´
        l´: Obj1 -> Obj2
        l: Mor1 -> Mor2´
        r´: Obj2 -> Obj1
        r: Mor2 -> Mor1´
        zeta: Mor2 -> Mor1´
        zetai: Mor1 -> Mor2´
        eta: -> Mor1´
        eps: -> Mor2´
..
LP2.8: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.9: assert
        mor1(o21,mor1(o21,id1,o21)+mor1(o21,f1,o11),o11) = mor1(o21,f1,o11);
        mor1(o21,mor1(o21,f1,o11)+mor1(o11,id1,o11),o11) = mor1(o21,f1,o11);
        mor1(o41,mor1(o41,mor1(o41,f1,o31)+mor1(o31,g1,o21),o21)+
                mor1(o21,h1,o11),o11) =
        mor1(o41,mor1(o41,f1,o31)+mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11),o11);
        mor2(o22,mor2(o22,id2,o22)*mor2(o22,f2,o12),o12) = mor2(o22,f2,o12);
        mor2(o22,mor2(o22,f2,o12)*mor2(o12,id2,o12),o12) = mor2(o22,f2,o12);
        mor2(o42,mor2(o42,mor2(o42,f2,o32)*mor2(o32,g2,o22),o22)*
                mor2(o22,h2,o12),o12) =
        mor2(o42,mor2(o42,f2,o32)*mor2(o32,mor2(o32,g2,o22)*
                mor2(o22,h2,o12),o12),o12);
        mor2(l´(o11),l(mor1(o11,id1,o11)),l´(o11)) = mor2(l´(o11),id2,l´(o11));
        mor2(l´(o31),mor2(l´(o31),l(mor1(o31,g1,o21)),l´(o21)) *
                mor2(l´(o21),l(mor1(o21,h1,o11)),l´(o11)),l´(o11)) =
        mor2(l´(o31),l(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),l´(o11));
        mor2(l´(o31),mor2(l´(o31),l(mor1(o31,g1,o21)),l´(o21))*
        mor2(l´(o21),mor2(l´(o21),l(mor1(o21,h1,o11)),l´(o11))*
        mor2(l´(o11),h2,o12),o12),o12) =
        mor2(l´(o31),mor2(l´(o31),l(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),l´(o11))*mor2(l´(o11),h2,o12),o12);
        mor1(r´(o12),zeta(mor2(o12,mor2(o12,eps,l´(r´(o12)))*
          mor2(l´(r´(o12)),l(mor1(r´(o12),f1,o11)),l´(o11)),l´(o11))),o11) =
                mor1(r´(o12),f1,o11);
        mor1(r´(o12),zeta(mor2(o12,eps,l´(r´(o12)))),r´(o12)) =
                mor1(r´(o12),id1,r´(o12));
        mor2(o12,mor2(o12,eps,l´(r´(o12)))*mor2(l´(r´(o12)),l(mor1(r´(o12),
                zeta(mor2(o12,f2,l´(o11))),o11)),l´(o11)),l´(o11)) =
        mor2(o12,f2,l´(o11));
        mor2(o22,mor2(o22,eps,l´(r´(o22)))*mor2(l´(r´(o22)),
        mor2(l´(r´(o22)),l(mor1(r´(o22),zeta(mor2(o22,f2,l´(o11))),o11)),
                l´(o11))*mor2(l´(o11),g2,o12),o12),o12) =
        mor2(o22,mor2(o22,f2,l´(o11))*mor2(l´(o11),g2,o12),o12);
        mor1(r´(o12),mor1(r´(o12),zeta(mor2(o12,f2,l´(o21))),o21)+
                mor1(o21,f1,o11),o11) =
        mor1(r´(o12),zeta(mor2(o12,mor2(o12,f2,l´(o21))*
                mor2(l´(o21),l(mor1(o21,f1,o11)),l´(o11)),l´(o11))),o11);
        mor1(r´(o22),r(mor2(o22,f2,o12)),r´(o12)) =
        mor1(r´(o22),zeta(mor2(o22,mor2(o22,f2,o12)*
                mor2(o12,eps,l´(r´(o12))),l´(r´(o12)))),r´(o12));
        mor2(o12,zetai(mor1(r´(o12),f1,o11)),l´(o11)) =
        mor2(o12,mor2(o12,eps,l´(r´(o12)))*
                mor2(l´(r´(o12)),l(mor1(r´(o12),f1,o11)),l´(o11)),l´(o11));
        mor1(r´(l´(o11)),eta,o11) =
        mor1(r´(l´(o11)),zeta(mor2(l´(o11),id2,l´(o11))),o11)
..
```

```
Added 17 facts named user.1, ..., user.17 to the system.
The system now contains 17 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.10: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.11: display
Rewrite rules:
user.1:  mor1(o21, mor1(o21, id1, o21) + mor1(o21, f1, o11), o11)
           -> mor1(o21, f1, o11)
user.2:  mor1(o21, mor1(o21, f1, o11) + mor1(o11, id1, o11), o11)
           -> mor1(o21, f1, o11)
user.3:  mor1(o41,
              mor1(o41, mor1(o41, f1, o31) + mor1(o31, g1, o21), o21)
                + mor1(o21, h1, o11),
              o11)
           ->
           mor1(o41,
                mor1(o41, f1, o31)
                  + mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11),
                o11)
user.4:  mor2(o22, mor2(o22, id2, o22) * mor2(o22, f2, o12), o12)
           -> mor2(o22, f2, o12)
user.5:  mor2(o22, mor2(o22, f2, o12) * mor2(o12, id2, o12), o12)
           -> mor2(o22, f2, o12)
user.6:  mor2(o42,
              mor2(o42, mor2(o42, f2, o32) * mor2(o32, g2, o22), o22)
                * mor2(o22, h2, o12),
              o12)
           ->
           mor2(o42,
                mor2(o42, f2, o32)
                  * mor2(o32, mor2(o32, g2, o22) * mor2(o22, h2, o12), o12),
                o12)
user.7:  mor2(l´(o11), l(mor1(o11, id1, o11)), l´(o11))
           -> mor2(l´(o11), id2, l´(o11))
user.8:  mor2(l´(o31),
              mor2(l´(o31), l(mor1(o31, g1, o21)), l´(o21))
                * mor2(l´(o21), l(mor1(o21, h1, o11)), l´(o11)),
              l´(o11))
           ->
           mor2(l´(o31),
                l(mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11)),
                l´(o11))
user.9:  mor2(l´(o31),
              mor2(l´(o31), l(mor1(o31, g1, o21)), l´(o21))
                * mor2(l´(o21),
                       mor2(l´(o21), l(mor1(o21, h1, o11)), l´(o11))
                         * mor2(l´(o11), h2, o12),
                       o12),
              o12)
           ->
           mor2(l´(o31),
                mor2(l´(o31),
                     l(mor1(o31,
                            mor1(o31, g1, o21) + mor1(o21, h1, o11),
                            o11)),
                     l´(o11))
                  * mor2(l´(o11), h2, o12),
                o12)
user.10: mor1(r´(o12),
              zeta(mor2(o12,
                        mor2(o12, eps, l´(r´(o12)))
                          * mor2(l´(r´(o12)),
                                 l(mor1(r´(o12), f1, o11)),
                                 l´(o11)),
                        l´(o11))),
              o11)
           -> mor1(r´(o12), f1, o11)
user.11: mor1(r´(o12), zeta(mor2(o12, eps, l´(r´(o12)))), r´(o12))
           -> mor1(r´(o12), id1, r´(o12))
user.12: mor2(o12,
```

```
             mor2(o12, eps, l´(r´(o12)))
               * mor2(l´(r´(o12)),
                      l(mor1(r´(o12), zeta(mor2(o12, f2, l´(o11))), o11)),
                      l´(o11)),
             l´(o11))
          -> mor2(o12, f2, l´(o11))
user.13: mor2(o22,
             mor2(o22, eps, l´(r´(o22)))
               * mor2(l´(r´(o22)),
                      mor2(l´(r´(o22)),
                           l(mor1(r´(o22),
                                  zeta(mor2(o22, f2, l´(o11))),
                                  o11)),
                           l´(o11))
                        * mor2(l´(o11), g2, o12),
                      o12),
             o12)
          -> mor2(o22, mor2(o22, f2, l´(o11)) * mor2(l´(o11), g2, o12), o12)
user.14: mor1(r´(o12),
             mor1(r´(o12), zeta(mor2(o12, f2, l´(o21))), o21)
               + mor1(o21, f1, o11),
             o11)
          ->
          mor1(r´(o12),
               zeta(mor2(o12,
                         mor2(o12, f2, l´(o21))
                           * mor2(l´(o21), l(mor1(o21, f1, o11)), l´(o11)),
                         l´(o11))),
               o11)
user.15: mor1(r´(o22), r(mor2(o22, f2, o12)), r´(o12))
          ->
          mor1(r´(o22),
               zeta(mor2(o22,
                         mor2(o22, f2, o12) * mor2(o12, eps, l´(r´(o12))),
                         l´(r´(o12)))),
               r´(o12))
user.16: mor2(o12, zetai(mor1(r´(o12), f1, o11)), l´(o11))
          ->
          mor2(o12,
               mor2(o12, eps, l´(r´(o12)))
                 * mor2(l´(r´(o12)), l(mor1(r´(o12), f1, o11)), l´(o11)),
               l´(o11))
user.17: mor1(r´(l´(o11)), eta, o11)
          -> mor1(r´(l´(o11)), zeta(mor2(l´(o11), id2, l´(o11))), o11)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/adj/type1.lp´.
LP3: quit
```

A.4.2 Subst

Partial Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/adj/term2.lplog´ on 15 May 1995
18:57:26.
LP2: execute term2
LP2.1: declare sort Mor
LP2.2: declare variables f, g, h: Mor
LP2.3: declare operators
        id: -> Mor
        __*__: Mor, Mor -> Mor
        fst: -> Mor
        snd: -> Mor
        __+__: Mor, Mor -> Mor
        cur: Mor -> Mor
..
LP2.4: set ordering polynomial 3
The ordering-method is now `polynomial 3´.
LP2.5: register polynomial    id     2
LP2.6: register polynomial    fst    2
LP2.7: register polynomial    snd    2
```

```
LP2.8: register polynomial     *       x * y,  x * y,          2 * x + y
LP2.9: register polynomial     +       x + y,  x + y + 1
LP2.10: assert
        fst * (f + g) = f;
        snd * (f + g) = g;
        id * f = f;
        f * id = f;
        (fst * f) + (snd * f) = f;
        (f * g) * h = f * (g * h);
        (f + g) * h = (f * h) + (g * h);
        fst + snd = id
..
Added 8 facts named user.1, ..., user.8 to the system.
The system now contains 8 rewrite rules.  The rewriting system is guaranteed to
terminate.
LP2.11: complete
The system is complete.
LP2.12: display
Rewrite rules:
user.1: fst * (f + g) -> f
user.2: snd * (f + g) -> g
user.3: id * f -> f
user.4: f * id -> f
user.5: (fst * f) + (snd * f) -> f
user.6: f * g * h -> f * (g * h)
user.7: (f + g) * h -> (f * h) + (g * h)
user.8: fst + snd -> id
LP2.13: set ordering left-to-right
Warning: the rewriting system is known to terminate, but its termination cannot
be verified with the `left-to-right´ ordering.  When another rewrite rule is
added, the rewriting system will be treated as nonterminating.
The ordering-method is now `left-to-right´.
LP2.14: assert
        cur(f) * g = cur(f * ((g * fst) + snd))
..
Added 1 fact named user.9 to the system.
The system now contains 9 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.15: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.16: display
Rewrite rules:
user.1: fst * (f + g) -> f
user.2: snd * (f + g) -> g
user.3: id * f -> f
user.4: f * id -> f
user.5: (fst * f) + (snd * f) -> f
user.6: f * g * h -> f * (g * h)
user.7: (f + g) * h -> (f * h) + (g * h)
user.8: fst + snd -> id
user.9: cur(f) * g -> cur(f * ((g * fst) + snd))
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/adj/term2.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/adj/type2.lplog´ on 15 May 1995
18:57:55.
LP2: execute type2
LP2.1: declare sort Obj, Mor´, Mor
LP2.2: declare variables o1, o2, o3, o4: Obj
LP2.3: declare variables f´, g´, h´: Mor´
LP2.4: declare operators
        mor: Obj, Mor´, Obj -> Mor
        __#__: Obj, Obj -> Obj
        __>__: Obj, Obj -> Obj
        id: -> Mor´
        __*__: Mor, Mor -> Mor´
```

```
        fst: -> Mor´
        snd: -> Mor´
        __+__: Mor, Mor -> Mor´
        cur: Mor -> Mor´
..
LP2.5: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.6: assert
        mor(o3,mor(o3,fst,o3#o2)*mor(o3#o2,mor(o3,f´,o1)+
                  mor(o2,g´,o1),o1),o1) =
        mor(o3,f´,o1);
        mor(o2,mor(o2,snd,o3#o2)*mor(o3#o2,mor(o3,f´,o1)+
                mor(o2,g´,o1),o1),o1) =
        mor(o2,g´,o1);
        mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) = mor(o2, f´, o1);
        mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) = mor(o2, f´, o1);
        mor(o2#o3,mor(o2,mor(o2,fst,o2#o3)*mor(o2#o3,f´,o1),o1)+
                mor(o3,mor(o3,snd,o2#o3)*mor(o2#o3,f´,o1),o1),o1) =
        mor(o2#o3,f´,o1);
        mor(o4,mor(o4,mor(o4,f´,o3)*mor(o3,g´,o2),o2)*mor(o2,h´,o1),o1) =
        mor(o4,mor(o4,f´,o3)*mor(o3,mor(o3,g´,o2)*mor(o2,h´,o1),o1),o1);
        mor(o4#o3,mor(o4#o3,mor(o4,f´,o2)+mor(o3,g´,o2),o2)*mor(o2,h´,o1),o1) =
        mor(o4#o3,mor(o4,mor(o4,f´,o2)*mor(o2,h´,o1),o1)+
                mor(o3,mor(o3,g´,o2)*mor(o2,h´,o1),o1),o1);
        mor(o1#o2,mor(o1,fst,o1#o2)+mor(o2,snd,o1#o2),o1#o2) =
        mor(o1#o2,id,o1#o2);
        mor(o3>o4,mor(o3>o4,cur(mor(o4,f´,o2#o3)),o2)*mor(o2,g´,o1),o1) =
        mor(o3>o4,cur(mor(o4,mor(o4,f´,o2#o3)*mor(o2#o3,mor(o2,mor(o2,g´,o1)*
                mor(o1,fst,o1#o3),o1#o3)+mor(o3,snd,o1#o3),o1#o3),o1#o3)),o1)
..
Added 9 facts named user.1, ..., user.9 to the system.
The system now contains 9 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.7: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.8: display
Rewrite rules:
user.1: mor(o3,
            mor(o3, fst, o3 # o2)
              * mor(o3 # o2, mor(o3, f´, o1) + mor(o2, g´, o1), o1),
            o1)
          -> mor(o3, f´, o1)
user.2: mor(o2,
            mor(o2, snd, o3 # o2)
              * mor(o3 # o2, mor(o3, f´, o1) + mor(o2, g´, o1), o1),
            o1)
          -> mor(o2, g´, o1)
user.3: mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) -> mor(o2, f´, o1)
user.4: mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) -> mor(o2, f´, o1)
user.5: mor(o2 # o3,
            mor(o2, mor(o2, fst, o2 # o3) * mor(o2 # o3, f´, o1), o1)
              + mor(o3, mor(o3, snd, o2 # o3) * mor(o2 # o3, f´, o1), o1),
            o1)
          -> mor(o2 # o3, f´, o1)
user.6: mor(o4,
            mor(o4, mor(o4, f´, o3) * mor(o3, g´, o2), o2) * mor(o2, h´, o1),
            o1)
          ->
          mor(o4,
              mor(o4, f´, o3) * mor(o3, mor(o3, g´, o2) * mor(o2, h´, o1), o1),
              o1)
user.7: mor(o4 # o3,
            mor(o4 # o3, mor(o4, f´, o2) + mor(o3, g´, o2), o2)
              * mor(o2, h´, o1),
            o1)
          ->
          mor(o4 # o3,
              mor(o4, mor(o4, f´, o2) * mor(o2, h´, o1), o1)
                + mor(o3, mor(o3, g´, o2) * mor(o2, h´, o1), o1),
              o1)
user.8: mor(o1 # o2, mor(o1, fst, o1 # o2) + mor(o2, snd, o1 # o2), o1 # o2)
```

```
          -> mor(o1 # o2, id, o1 # o2)
user.9: mor(o3 > o4,
            mor(o3 > o4, cur(mor(o4, f´, o2 # o3)), o2) * mor(o2, g´, o1),
            o1)
          ->
          mor(o3 > o4,
              cur(mor(o4,
                      mor(o4, f´, o2 # o3)
                        * mor(o2 # o3,
                              mor(o2,
                                  mor(o2, g´, o1) * mor(o1, fst, o1 # o3),
                                  o1 # o3)
                                + mor(o3, snd, o1 # o3),
                              o1 # o3),
                      o1 # o3)),
              o1)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/adj/type2.lp´.
LP3: quit
```

A.5 Monad

A.5.1 Via Adjunction

Without types

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/mon/term1.lplog´ on 15 May 1995
19:00:33.
LP2: execute term1
LP2.1: declare sort Mor1, Mor2
LP2.2: declare variables f1, g1, h1: Mor1
LP2.3: declare variables f2, g2, h2: Mor2
LP2.4: declare operators
        id1: -> Mor1
        id2: -> Mor2
        __+__: Mor1, Mor1 -> Mor1
        __*__: Mor2, Mor2 -> Mor2
        l: Mor1 -> Mor2
        r: Mor2 -> Mor1
        zeta: Mor2 -> Mor1
        zetai: Mor1 -> Mor2
        eta: -> Mor1
        eps: -> Mor2
        t: Mor1 -> Mor1
        mu: -> Mor1
..
LP2.5: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.6: assert
        id1 + f1 = f1;
        f1 + id1 = f1;
        (f1 + g1) + h1 = f1 + (g1 + h1);
        id2 * f2 = f2;
        f2 * id2 = f2;
        (f2 * g2) * h2 = f2 * (g2 * h2);
        l(id1) = id2;
        l(f1) * l(g1) = l(f1 + g1);
        l(f1) * (l(g1) * h2) = l(f1 + g1) * h2;
        zeta(eps * l(f1)) = f1;
        zeta(eps) = id1;
        eps * l(zeta(f2)) = f2;
        eps * (l(zeta(f2)) * g2) = f2 * g2;
        zeta(f2) + f1 = zeta(f2 * l(f1));
        r(f2) = zeta(f2 * eps);
        zetai(f1) = eps * l(f1);
        eta = zeta(id2);
        t(f1) = zeta(l(f1) * eps);
        mu = zeta(eps * eps)
..
```

```
Added 19 facts named user.1, ..., user.19 to the system.
The system now contains 19 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.7: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.8: display
Rewrite rules:
user.1:  id1 + f1 -> f1
user.2:  f1 + id1 -> f1
user.3:  f1 + g1 + h1 -> f1 + (g1 + h1)
user.4:  id2 * f2 -> f2
user.5:  f2 * id2 -> f2
user.6:  f2 * g2 * h2 -> f2 * (g2 * h2)
user.7:  l(id1) -> id2
user.8:  l(f1) * l(g1) -> l(f1 + g1)
user.9:  l(f1) * (l(g1) * h2) -> l(f1 + g1) * h2
user.10: zeta(eps * l(f1)) -> f1
user.11: zeta(eps) -> id1
user.12: eps * l(zeta(f2)) -> f2
user.13: eps * (l(zeta(f2)) * g2) -> f2 * g2
user.14: zeta(f2) + f1 -> zeta(f2 * l(f1))
user.15: r(f2) -> zeta(f2 * eps)
user.16: zetai(f1) -> eps * l(f1)
user.17: eta -> zeta(id2)
user.18: t(f1) -> zeta(l(f1) * eps)
user.19: mu -> zeta(eps * eps)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/mon/term1.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/mon/type1.lplog´ on 15 May 1995
19:00:41.
LP2: execute type1
LP2.1: declare sort Obj1, Mor1´, Mor1
LP2.2: declare sort Obj2, Mor2´, Mor2
LP2.3: declare variables o11, o21, o31, o41: Obj1
LP2.4: declare variables o12, o22, o32, o42: Obj2
LP2.5: declare variables f1, g1, h1: Mor1´
LP2.6: declare variables f2, g2, h2: Mor2´
LP2.7: declare operators
        mor1: Obj1, Mor1´, Obj1 -> Mor1
        mor2: Obj2, Mor2´, Obj2 -> Mor2
        id1: -> Mor1´
        id2: -> Mor2´
        __+__: Mor1, Mor1 -> Mor1´
        __*__: Mor2, Mor2 -> Mor2´
        l´: Obj1 -> Obj2
        l: Mor1 -> Mor2´
        r´: Obj2 -> Obj1
        r: Mor2 -> Mor1´
        zeta: Mor2 -> Mor1´
        zetai: Mor1 -> Mor2´
        eta: -> Mor1´
        eps: -> Mor2´
        t: Mor1 -> Mor1´
        mu: -> Mor1´
..
LP2.8: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.9: assert
        mor1(o21,mor1(o21,id1,o21)+mor1(o21,f1,o11),o11) = mor1(o21,f1,o11);
        mor1(o21,mor1(o21,f1,o11)+mor1(o11,id1,o11),o11) = mor1(o21,f1,o11);
        mor1(o41,mor1(o41,mor1(o41,f1,o31)+mor1(o31,g1,o21),o21)+
                mor1(o21,h1,o11),o11) =
        mor1(o41,mor1(o41,f1,o31)+mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11),o11);
        mor2(o22,mor2(o22,id2,o22)*mor2(o22,f2,o12),o12) = mor2(o22,f2,o12);
```

```
        mor2(o22,mor2(o22,f2,o12)*mor2(o12,id2,o12),o12) = mor2(o22,f2,o12);
        mor2(o42,mor2(o42,mor2(o42,f2,o32)*mor2(o32,g2,o22),o22)*
                mor2(o22,h2,o12),o12) =
        mor2(o42,mor2(o42,f2,o32)*mor2(o32,mor2(o32,g2,o22)*
                mor2(o22,h2,o12),o12),o12);
        mor2(l´(o11),l(mor1(o11,id1,o11)),l´(o11)) = mor2(l´(o11),id2,l´(o11));
        mor2(l´(o31),mor2(l´(o31),l(mor1(o31,g1,o21)),l´(o21)) *
                mor2(l´(o21),l(mor1(o21,h1,o11)),l´(o11)),l´(o11)) =
        mor2(l´(o31),l(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),l´(o11));
        mor2(l´(o31),mor2(l´(o31),l(mor1(o31,g1,o21)),l´(o21))*
        mor2(l´(o21),mor2(l´(o21),l(mor1(o21,h1,o11)),l´(o11))*
        mor2(l´(o11),h2,o12),o12),o12) =
        mor2(l´(o31),mor2(l´(o31),l(mor1(o31,mor1(o31,g1,o21)+
                mor1(o21,h1,o11),o11)),l´(o11))*mor2(l´(o11),h2,o12),o12);
        mor1(r´(o12),zeta(mor2(o12,mor2(o12,eps,l´(r´(o12)))*
          mor2(l´(r´(o12)),l(mor1(r´(o12),f1,o11)),l´(o11)),l´(o11))),o11) =
                mor1(r´(o12),f1,o11);
        mor1(r´(o12),zeta(mor2(o12,eps,l´(r´(o12)))),r´(o12)) =
                mor1(r´(o12),id1,r´(o12));
        mor2(o12,mor2(o12,eps,l´(r´(o12)))*mor2(l´(r´(o12)),l(mor1(r´(o12),
                zeta(mor2(o12,f2,l´(o11))),o11)),l´(o11)),l´(o11)) =
        mor2(o12,f2,l´(o11));
        mor2(o22,mor2(o22,eps,l´(r´(o22)))*mor2(l´(r´(o22)),
        mor2(l´(r´(o22)),l(mor1(r´(o22),zeta(mor2(o22,f2,l´(o11))),o11)),
                l´(o11))*mor2(l´(o11),g2,o12),o12),o12) =
        mor2(o22,mor2(o22,f2,l´(o11))*mor2(l´(o11),g2,o12),o12);
        mor1(r´(o12),mor1(r´(o12),zeta(mor2(o12,f2,l´(o21))),o21)+
                mor1(o21,f1,o11),o11) =
        mor1(r´(o12),zeta(mor2(o12,mor2(o12,f2,l´(o21))*
                mor2(l´(o21),l(mor1(o21,f1,o11)),l´(o11)),l´(o11))),o11);
        mor1(r´(o22),r(mor2(o22,f2,o12)),r´(o12)) =
        mor1(r´(o22),zeta(mor2(o22,mor2(o22,f2,o12)*
                mor2(o12,eps,l´(r´(o12))),l´(r´(o12)))),r´(o12));
        mor2(o12,zetai(mor1(r´(o12),f1,o11)),l´(o11)) =
        mor2(o12,mor2(o12,eps,l´(r´(o12)))*
                mor2(l´(r´(o12)),l(mor1(r´(o12),f1,o11)),l´(o11)),l´(o11));
        mor1(r´(l´(o11)),eta,o11) =
        mor1(r´(l´(o11)),zeta(mor2(l´(o11),id2,l´(o11))),o11);
        mor1(r´(l´(o21)),t(mor1(o21,f1,o11)),r´(l´(o11))) =
        mor1(r´(l´(o21)),zeta(mor2(l´(o21),mor2(l´(o21),l(mor1(o21,f1,o11)),
                l´(o11))*mor2(l´(o11),eps,l´(r´(l´(o11)))),
                l´(r´(l´(o11))))),r´(l´(o11)));
        mor1(r´(l´(o11)),mu,r´(l´(r´(l´(o11))))) =
        mor1(r´(l´(o11)),zeta(mor2(l´(o11),mor2(l´(o11),eps,l´(r´(l´(o11))))*
                mor2(l´(r´(l´(o11))),eps,l´(r´(l´(r´(l´(o11)))))),
                l´(r´(l´(r´(l´(o11))))))),r´(l´(r´(l´(o11)))))
..
Added 19 facts named user.1, ..., user.19 to the system.
The system now contains 19 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.10: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.11: display
Rewrite rules:
user.1:  mor1(o21, mor1(o21, id1, o21) + mor1(o21, f1, o11), o11)
           -> mor1(o21, f1, o11)
user.2:  mor1(o21, mor1(o21, f1, o11) + mor1(o11, id1, o11), o11)
           -> mor1(o21, f1, o11)
user.3:  mor1(o41,
              mor1(o41, mor1(o41, f1, o31) + mor1(o31, g1, o21), o21)
                + mor1(o21, h1, o11),
              o11)
           ->
           mor1(o41,
                mor1(o41, f1, o31)
                  + mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11),
                o11)
user.4:  mor2(o22, mor2(o22, id2, o22) * mor2(o22, f2, o12), o12)
           -> mor2(o22, f2, o12)
user.5:  mor2(o22, mor2(o22, f2, o12) * mor2(o12, id2, o12), o12)
```

```
           -> mor2(o22, f2, o12)
user.6:  mor2(o42,
              mor2(o42, mor2(o42, f2, o32) * mor2(o32, g2, o22), o22)
                * mor2(o22, h2, o12),
              o12)
           ->
           mor2(o42,
                mor2(o42, f2, o32)
                  * mor2(o32, mor2(o32, g2, o22) * mor2(o22, h2, o12), o12),
                o12)
user.7:  mor2(l´(o11), l(mor1(o11, id1, o11)), l´(o11))
           -> mor2(l´(o11), id2, l´(o11))
user.8:  mor2(l´(o31),
              mor2(l´(o31), l(mor1(o31, g1, o21)), l´(o21))
                * mor2(l´(o21), l(mor1(o21, h1, o11)), l´(o11)),
              l´(o11))
           ->
           mor2(l´(o31),
                l(mor1(o31, mor1(o31, g1, o21) + mor1(o21, h1, o11), o11)),
                l´(o11))
user.9:  mor2(l´(o31),
              mor2(l´(o31), l(mor1(o31, g1, o21)), l´(o21))
                * mor2(l´(o21),
                       mor2(l´(o21), l(mor1(o21, h1, o11)), l´(o11))
                         * mor2(l´(o11), h2, o12),
                       o12),
              o12)
           ->
           mor2(l´(o31),
                mor2(l´(o31),
                     l(mor1(o31,
                            mor1(o31, g1, o21) + mor1(o21, h1, o11),
                            o11)),
                     l´(o11))
                  * mor2(l´(o11), h2, o12),
                o12)
user.10: mor1(r´(o12),
              zeta(mor2(o12,
                        mor2(o12, eps, l´(r´(o12)))
                          * mor2(l´(r´(o12)),
                                 l(mor1(r´(o12), f1, o11)),
                                 l´(o11)),
                        l´(o11))),
              o11)
           -> mor1(r´(o12), f1, o11)
user.11: mor1(r´(o12), zeta(mor2(o12, eps, l´(r´(o12)))), r´(o12))
           -> mor1(r´(o12), id1, r´(o12))
user.12: mor2(o12,
              mor2(o12, eps, l´(r´(o12)))
                * mor2(l´(r´(o12)),
                       l(mor1(r´(o12), zeta(mor2(o12, f2, l´(o11))), o11)),
                       l´(o11)),
              l´(o11))
           -> mor2(o12, f2, l´(o11))
user.13: mor2(o22,
              mor2(o22, eps, l´(r´(o22)))
                * mor2(l´(r´(o22)),
                       mor2(l´(r´(o22)),
                            l(mor1(r´(o22),
                                   zeta(mor2(o22, f2, l´(o11))),
                                   o11)),
                            l´(o11))
                         * mor2(l´(o11), g2, o12),
                       o12),
              o12)
           -> mor2(o22, mor2(o22, f2, l´(o11)) * mor2(l´(o11), g2, o12), o12)
user.14: mor1(r´(o12),
              mor1(r´(o12), zeta(mor2(o12, f2, l´(o21))), o21)
                + mor1(o21, f1, o11),
              o11)
           ->
           mor1(r´(o12),
```

```
                zeta(mor2(o12,
                          mor2(o12, f2, l´(o21))
                            * mor2(l´(o21), l(mor1(o21, f1, o11)), l´(o11)),
                          l´(o11))),
                o11)
user.15: mor1(r´(o22), r(mor2(o22, f2, o12)), r´(o12))
           ->
           mor1(r´(o22),
                zeta(mor2(o22,
                          mor2(o22, f2, o12) * mor2(o12, eps, l´(r´(o12))),
                          l´(r´(o12)))),
                r´(o12))
user.16: mor2(o12, zetai(mor1(r´(o12), f1, o11)), l´(o11))
           ->
           mor2(o12,
                mor2(o12, eps, l´(r´(o12)))
                  * mor2(l´(r´(o12)), l(mor1(r´(o12), f1, o11)), l´(o11)),
                l´(o11))
user.17: mor1(r´(l´(o11)), eta, o11)
           -> mor1(r´(l´(o11)), zeta(mor2(l´(o11), id2, l´(o11))), o11)
user.18: mor1(r´(l´(o21)), t(mor1(o21, f1, o11)), r´(l´(o11)))
           ->
           mor1(r´(l´(o21)),
                zeta(mor2(l´(o21),
                          mor2(l´(o21), l(mor1(o21, f1, o11)), l´(o11))
                            * mor2(l´(o11), eps, l´(r´(l´(o11)))),
                          l´(r´(l´(o11))))),
                r´(l´(o11)))
user.19: mor1(r´(l´(o11)), mu, r´(l´(r´(l´(o11)))))
           ->
           mor1(r´(l´(o11)),
                zeta(mor2(l´(o11),
                          mor2(l´(o11), eps, l´(r´(l´(o11))))
                            * mor2(l´(r´(l´(o11))),
                                   eps,
                                   l´(r´(l´(r´(l´(o11)))))),
                          l´(r´(l´(r´(l´(o11)))))),
                r´(l´(r´(l´(o11)))))
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/mon/type1.lp´.
LP3: quit
```

A.5.2 Via Kleisli Category

Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/mon/term2.lplog´ on 15 May 1995
19:00:37.
LP2: execute term2
LP2.1: declare sort Mor
LP2.2: declare variables f, g, h: Mor
LP2.3: declare operators
        id: -> Mor
        __*__: Mor, Mor -> Mor
        unit: -> Mor
        star: Mor -> Mor
        t: Mor -> Mor
        mu: -> Mor
..
LP2.4: set ordering polynomial 3
The ordering-method is now `polynomial 3´.
LP2.5: register polynomial     id      1
LP2.6: register polynomial     unit    1
LP2.7: register polynomial     star    x + 2
LP2.8: register polynomial     *       x + y,  2 * x * y, 2 * x + y
LP2.9: assert
        id * f = f;
        f * id = f;
        (f * g) * h = f * (g * h);
        star(f) * unit = f;
```

```
        star(unit) = id;
        star(f) * star(g) = star(star(f) * g)
..
Added 6 facts named user.1, ..., user.6 to the system.
The system now contains 6 rewrite rules.  The rewriting system is guaranteed to
terminate.
LP2.10: complete
The following equations are critical pairs between rewrite rules user.4 and
user.3.
  user.7: f * h = star(f) * (unit * h)
The system now contains 1 formula and 6 rewrite rules.  The rewriting system is
guaranteed to terminate.
The following equations are critical pairs between rewrite rules user.6 and
user.3.
  user.8: star(star(f) * g) * h = star(f) * (star(g) * h)
The system now contains 1 formula and 7 rewrite rules.  The rewriting system is
guaranteed to terminate.
The system now contains 8 rewrite rules.  The rewriting system is guaranteed to
terminate.
The system is complete.
LP2.11: display
Rewrite rules:
user.1: id * f -> f
user.2: f * id -> f
user.3: f * g * h -> f * (g * h)
user.4: star(f) * unit -> f
user.5: star(unit) -> id
user.6: star(f) * star(g) -> star(star(f) * g)
user.7: star(f) * (unit * h) -> f * h
user.8: star(f) * (star(g) * h) -> star(star(f) * g) * h
LP2.12: register polynomial     t       x + 4
LP2.13: register polynomial     mu      4
LP2.14: assert
        t(f) = star(unit * f);
        mu = star(id)
..
Added 2 facts named user.9, user.10 to the system.
The system now contains 10 rewrite rules.  The rewriting system is guaranteed
to terminate.
LP2.15: complete
The system is complete.
LP2.16: display
Rewrite rules:
user.1:  id * f -> f
user.2:  f * id -> f
user.3:  f * g * h -> f * (g * h)
user.4:  star(f) * unit -> f
user.5:  star(unit) -> id
user.6:  star(f) * star(g) -> star(star(f) * g)
user.7:  star(f) * (unit * h) -> f * h
user.8:  star(f) * (star(g) * h) -> star(star(f) * g) * h
user.9:  t(f) -> star(unit * f)
user.10: mu -> star(id)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/mon/term2.lp´.
LP3: quit
```

Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/mon/type2.lplog´ on 15 May 1995
19:01:16.
LP2: execute type2
LP2.1: declare sort Obj, Mor´, Mor
LP2.2: declare variables o1, o2, o3, o4: Obj
LP2.3: declare variables f´, g´, h´: Mor´
LP2.4: declare operators
        mor: Obj, Mor´, Obj -> Mor
        id: -> Mor´
        __*__: Mor, Mor -> Mor´
        unit: -> Mor´
```

```
        star: Mor -> Mor´
        t´: Obj -> Obj
        t: Mor -> Mor´
        mu: -> Mor´
..
LP2.5: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.6: assert
        mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) = mor(o2, f´, o1);
        mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) = mor(o2, f´, o1);
        mor(o4,mor(o4,mor(o4,f´,o3)*mor(o3,g´,o2),o2)*mor(o2,h´,o1),o1) =
        mor(o4,mor(o4,f´,o3)*mor(o3,mor(o3,g´,o2)*mor(o2,h´,o1),o1),o1);
        mor(t´(o2),mor(t´(o2),star(mor(t´(o2),f´,o1)),t´(o1))*
                mor(t´(o1),unit,o1),o1) =
        mor(t´(o2),f´,o1);
        mor(t´(o1),star(mor(t´(o1),unit,o1)),t´(o1)) = mor(t´(o1),id,t´(o1));
        mor(t´(o3),mor(t´(o3),star(mor(t´(o3),f´,o2)),t´(o2))*
                mor(t´(o2),star(mor(t´(o2),g´,o1)),t´(o1)),t´(o1)) =
        mor(t´(o3),star(mor(t´(o3),mor(t´(o3),star(mor(t´(o3),f´,o2)),t´(o2))*
                mor(t´(o2),g´,o1),o1)),t´(o1));
        mor(t´(o3),mor(t´(o3),star(mor(t´(o3),f´,o2)),t´(o2))*
                mor(t´(o2),mor(t´(o2),unit,o2)*mor(o2,g´,o1),o1),o1) =
        mor(t´(o3),mor(t´(o3),f´,o2)*mor(o2,g´,o1),o1);
        mor(t´(o4),mor(t´(o4),star(mor(t´(o4),f´,o3)),t´(o3))*
                mor(t´(o3),mor(t´(o3),star(mor(t´(o3),g´,o2)),t´(o2))*
                mor(t´(o2),h´,o1),o1),o1) =
        mor(t´(o4),mor(t´(o4),star(mor(t´(o4),mor(t´(o4),
                star(mor(t´(o4),f´,o3)),t´(o3))*
                mor(t´(o3),g´,o2),o2)),t´(o2))*mor(t´(o2),h´,o1),o1);
        mor(t´(o2),t(mor(o2,f´,o1)),t´(o1)) =
        mor(t´(o2),star(mor(t´(o2),mor(t´(o2),unit,o2)*
                mor(o2,f´,o1),o1)),t´(o1));
        mor(t´(o1),mu,t´(t´(o1))) =
        mor(t´(o1),star(mor(t´(o1),id,t´(o1))),t´(t´(o1)))
..
Added 10 facts named user.1, ..., user.10 to the system.
The system now contains 10 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.7: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.8: display
Rewrite rules:
user.1:  mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) -> mor(o2, f´, o1)
user.2:  mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) -> mor(o2, f´, o1)
user.3:  mor(o4,
             mor(o4, mor(o4, f´, o3) * mor(o3, g´, o2), o2) * mor(o2, h´, o1),
             o1)
           ->
           mor(o4,
               mor(o4, f´, o3)
                 * mor(o3, mor(o3, g´, o2) * mor(o2, h´, o1), o1),
               o1)
user.4:  mor(t´(o2),
             mor(t´(o2), star(mor(t´(o2), f´, o1)), t´(o1))
               * mor(t´(o1), unit, o1),
             o1)
           -> mor(t´(o2), f´, o1)
user.5:  mor(t´(o1), star(mor(t´(o1), unit, o1)), t´(o1))
           -> mor(t´(o1), id, t´(o1))
user.6:  mor(t´(o3),
             mor(t´(o3), star(mor(t´(o3), f´, o2)), t´(o2))
               * mor(t´(o2), star(mor(t´(o2), g´, o1)), t´(o1)),
             t´(o1))
           ->
           mor(t´(o3),
               star(mor(t´(o3),
                        mor(t´(o3), star(mor(t´(o3), f´, o2)), t´(o2))
                          * mor(t´(o2), g´, o1),
                        o1)),
               t´(o1))
user.7:  mor(t´(o3),
```

```
            mor(t´(o3), star(mor(t´(o3), f´, o2)), t´(o2))
              * mor(t´(o2), mor(t´(o2), unit, o2) * mor(o2, g´, o1), o1),
            o1)
          -> mor(t´(o3), mor(t´(o3), f´, o2) * mor(o2, g´, o1), o1)
user.8:  mor(t´(o4),
            mor(t´(o4), star(mor(t´(o4), f´, o3)), t´(o3))
              * mor(t´(o3),
                    mor(t´(o3), star(mor(t´(o3), g´, o2)), t´(o2))
                      * mor(t´(o2), h´, o1),
                    o1),
            o1)
          ->
          mor(t´(o4),
              mor(t´(o4),
                  star(mor(t´(o4),
                           mor(t´(o4), star(mor(t´(o4), f´, o3)), t´(o3))
                             * mor(t´(o3), g´, o2),
                           o2)),
                  t´(o2))
                * mor(t´(o2), h´, o1),
              o1)
user.9:  mor(t´(o2), t(mor(o2, f´, o1)), t´(o1))
          ->
          mor(t´(o2),
              star(mor(t´(o2), mor(t´(o2), unit, o2) * mor(o2, f´, o1), o1)),
              t´(o1))
user.10: mor(t´(o1), mu, t´(t´(o1)))
          -> mor(t´(o1), star(mor(t´(o1), id, t´(o1))), t´(t´(o1)))
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/mon/type2.lp´.
LP3: quit
```

A.6 Computational Co-monad

A.6.1 Termination

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/ccm/term.lplog´ on 15 May 1995
18:59:00.
LP2: execute term
LP2.1: declare sort Mor
LP2.2: declare variables f, g, h: Mor
LP2.3: declare operators
        id: -> Mor
        __*__: Mor, Mor -> Mor
        counit: -> Mor
        costar: Mor -> Mor
        gamma: -> Mor
        t: Mor -> Mor
        del: -> Mor
..
LP2.4: set ordering polynomial 3
The ordering-method is now `polynomial 3´.
LP2.5: register polynomial     id      1
LP2.6: register polynomial     counit  1
LP2.7: register polynomial     costar  x + 2
LP2.8: register polynomial     *       x + y, 2 * x * y, 2 * x + y
LP2.9: assert
        id * f = f;
        f * id = f;
        (f * g) * h = f * (g * h);
        counit * costar(f) = f;
        costar(counit) = id;
        costar(f) * costar(g) = costar(f * costar(g))
..
Added 6 facts named user.1, ..., user.6 to the system.
The system now contains 6 rewrite rules.  The rewriting system is guaranteed to
terminate.
LP2.10: complete
The following equations are critical pairs between rewrite rules user.4 and
```

```
user.3.
  user.7: f * h = counit * (costar(f) * h)
The system now contains 1 formula and 6 rewrite rules.  The rewriting system is
guaranteed to terminate.
The following equations are critical pairs between rewrite rules user.6 and
user.3.
  user.8: costar(f * costar(g)) * h = costar(f) * (costar(g) * h)
The system now contains 1 formula and 7 rewrite rules.  The rewriting system is
guaranteed to terminate.
The system now contains 8 rewrite rules.  The rewriting system is guaranteed to
terminate.
The system is complete.
LP2.11: register polynomial     gamma   1
LP2.12: assert
        counit * gamma = id;
        costar(f) * gamma = gamma * (f * gamma)
..
Added 2 facts named user.9, user.10 to the system.
The system now contains 10 rewrite rules.  The rewriting system is guaranteed
to terminate.
LP2.13: complete
The following equations are critical pairs between rewrite rules user.9 and
user.3.
  user.11: h = counit * (gamma * h)
The system now contains 1 formula and 10 rewrite rules.  The rewriting system
is guaranteed to terminate.
The following equations are critical pairs between rewrite rules user.10 and
user.3.
  user.12: gamma * (f * (gamma * h)) = costar(f) * (gamma * h)
The system now contains 1 formula and 11 rewrite rules.  The rewriting system
is guaranteed to terminate.
The system now contains 12 rewrite rules.  The rewriting system is guaranteed
to terminate.
The system is complete.
LP2.14: display
Rewrite rules:
user.1:  id * f -> f
user.2:  f * id -> f
user.3:  f * g * h -> f * (g * h)
user.4:  counit * costar(f) -> f
user.5:  costar(counit) -> id
user.6:  costar(f) * costar(g) -> costar(f * costar(g))
user.7:  counit * (costar(f) * h) -> f * h
user.8:  costar(f) * (costar(g) * h) -> costar(f * costar(g)) * h
user.9:  counit * gamma -> id
user.10: costar(f) * gamma -> gamma * (f * gamma)
user.11: counit * (gamma * h) -> h
user.12: costar(f) * (gamma * h) -> gamma * (f * (gamma * h))
LP2.15: register polynomial     t       x + 4
LP2.16: register polynomial     del     4
LP2.17: assert
        t(f) = costar(f * counit);
        del = costar(id)
..
Added 2 facts named user.13, user.14 to the system.
The system now contains 14 rewrite rules.  The rewriting system is guaranteed
to terminate.
LP2.18: complete
The system is complete.
LP2.19: display
Rewrite rules:
user.1:  id * f -> f
user.2:  f * id -> f
user.3:  f * g * h -> f * (g * h)
user.4:  counit * costar(f) -> f
user.5:  costar(counit) -> id
user.6:  costar(f) * costar(g) -> costar(f * costar(g))
user.7:  counit * (costar(f) * h) -> f * h
user.8:  costar(f) * (costar(g) * h) -> costar(f * costar(g)) * h
user.9:  counit * gamma -> id
user.10: costar(f) * gamma -> gamma * (f * gamma)
user.11: counit * (gamma * h) -> h
```

```
user.12: costar(f) * (gamma * h) -> gamma * (f * (gamma * h))
user.13: t(f) -> costar(f * counit)
user.14: del -> costar(id)
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/ccm/term.lp´.
LP3: quit
```

A.6.2 Typing

```
LP (the Larch Prover), Release 3.1a (95/04/27) logging to
`/home/wgehrke/Tex/article/monads/thesis/src/ccm/type.lplog´ on 15 May 1995
18:59:05.
LP2: execute type
LP2.1: declare sort Obj, Mor´, Mor
LP2.2: declare variables o1, o2, o3, o4: Obj
LP2.3: declare variables f´, g´, h´: Mor´
LP2.4: declare operators
        mor: Obj, Mor´, Obj -> Mor
        id: -> Mor´
        __*__: Mor, Mor -> Mor´
        counit: -> Mor´
        costar: Mor -> Mor´
        gamma: -> Mor´
        t´: Obj -> Obj
        t: Mor -> Mor´
        del: -> Mor´
..
LP2.5: set ordering left-to-right
The ordering-method is now `left-to-right´.
LP2.6: assert
        mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) = mor(o2, f´, o1);
        mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) = mor(o2, f´, o1);
        mor(o4,mor(o4,mor(o4,f´,o3)*mor(o3,g´,o2),o2)*mor(o2,h´,o1),o1) =
        mor(o4,mor(o4,f´,o3)*mor(o3,mor(o3,g´,o2)*mor(o2,h´,o1),o1),o1);
        mor(o2,mor(o2,counit,t´(o2))*mor(t´(o2),
                costar(mor(o2,f´,t´(o1))),t´(o1)),t´(o1)) =
        mor(o2,f´,t´(o1));
        mor(t´(o1),costar(mor(o1,counit,t´(o1))),t´(o1)) =
        mor(t´(o1),id,t´(o1));
        mor(t´(o3),mor(t´(o3),costar(mor(o3,f´,t´(o2))),t´(o2))*
                mor(t´(o2),costar(mor(o2,g´,t´(o1))),t´(o1)),t´(o1)) =
        mor(t´(o3),costar(mor(o3,mor(o3,f´,t´(o2))*mor(t´(o2),
                costar(mor(o2,g´,t´(o1))),t´(o1)),t´(o1))),t´(o1));
        mor(o3,mor(o3,counit,t´(o3))*mor(t´(o3),mor(t´(o3),
                costar(mor(o3,f´,t´(o2))),t´(o2))*mor(t´(o2),g´,o1),o1),o1) =
        mor(o3,mor(o3,f´,t´(o2))*mor(t´(o2),g´,o1),o1);
        mor(t´(o4),mor(t´(o4),costar(mor(o4,f´,t´(o3))),t´(o3))*
                mor(t´(o3),mor(t´(o3),costar(mor(o3,g´,t´(o2))),t´(o2))*
                mor(t´(o2),h´,o1),o1),o1) =
        mor(t´(o4),mor(t´(o4),costar(mor(o4,mor(o4,f´,t´(o3))*
                mor(t´(o3),costar(mor(o3,g´,t´(o2))),t´(o2)),t´(o2))),t´(o2))*
                mor(t´(o2),h´,o1),o1)
..
Added 8 facts named user.1, ..., user.8 to the system.
The system now contains 8 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.7: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.8: display
Rewrite rules:
user.1: mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) -> mor(o2, f´, o1)
user.2: mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) -> mor(o2, f´, o1)
user.3: mor(o4,
            mor(o4, mor(o4, f´, o3) * mor(o3, g´, o2), o2) * mor(o2, h´, o1),
            o1)
          ->
          mor(o4,
              mor(o4, f´, o3) * mor(o3, mor(o3, g´, o2) * mor(o2, h´, o1), o1),
              o1)
user.4: mor(o2,
```

```
            mor(o2, counit, t´(o2))
              * mor(t´(o2), costar(mor(o2, f´, t´(o1))), t´(o1)),
            t´(o1))
          -> mor(o2, f´, t´(o1))
user.5: mor(t´(o1), costar(mor(o1, counit, t´(o1))), t´(o1))
          -> mor(t´(o1), id, t´(o1))
user.6: mor(t´(o3),
            mor(t´(o3), costar(mor(o3, f´, t´(o2))), t´(o2))
              * mor(t´(o2), costar(mor(o2, g´, t´(o1))), t´(o1)),
            t´(o1))
          ->
          mor(t´(o3),
              costar(mor(o3,
                         mor(o3, f´, t´(o2))
                           * mor(t´(o2), costar(mor(o2, g´, t´(o1))), t´(o1)),
                         t´(o1))),
              t´(o1))
user.7: mor(o3,
            mor(o3, counit, t´(o3))
              * mor(t´(o3),
                    mor(t´(o3), costar(mor(o3, f´, t´(o2))), t´(o2))
                      * mor(t´(o2), g´, o1),
                    o1),
            o1)
          -> mor(o3, mor(o3, f´, t´(o2)) * mor(t´(o2), g´, o1), o1)
user.8: mor(t´(o4),
            mor(t´(o4), costar(mor(o4, f´, t´(o3))), t´(o3))
              * mor(t´(o3),
                    mor(t´(o3), costar(mor(o3, g´, t´(o2))), t´(o2))
                      * mor(t´(o2), h´, o1),
                    o1),
            o1)
          ->
          mor(t´(o4),
              mor(t´(o4),
                  costar(mor(o4,
                             mor(o4, f´, t´(o3))
                               * mor(t´(o3),
                                     costar(mor(o3, g´, t´(o2))),
                                     t´(o2)),
                             t´(o2))),
                  t´(o2))
                * mor(t´(o2), h´, o1),
              o1)
LP2.9: assert
        mor(o1,mor(o1,counit,t´(o1))*mor(t´(o1),gamma,o1),o1)=mor(o1,id,o1);
        mor(t´(o2),mor(t´(o2),costar(mor(o2,f´,t´(o1))),t´(o1))*
                mor(t´(o1),gamma,o1),o1) =
        mor(t´(o2),mor(t´(o2),gamma,o2)*mor(o2,mor(o2,f´,t´(o1))*
                mor(t´(o1),gamma,o1),o1),o1);
        mor(o2,mor(o2,counit,t´(o2))*mor(t´(o2),mor(t´(o2),gamma,o2)*
                mor(o2,f´,o1),o1),o1) =
        mor(o2,f´,o1);
        mor(t´(o3),mor(t´(o3),costar(mor(o3,f´,t´(o2))),t´(o2))*
                mor(t´(o2),mor(t´(o2),gamma,o2)*mor(o2,g´,o1),o1),o1) =
        mor(t´(o3),mor(t´(o3),gamma,o3)*mor(o3,mor(o3,f´,t´(o2))*
                mor(t´(o2),mor(t´(o2),gamma,o2)*mor(o2,g´,o1),o1),o1),o1)
..
Added 4 facts named user.9, ..., user.12 to the system.
The system now contains 12 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.10: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.11: display
Rewrite rules:
user.1:  mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) -> mor(o2, f´, o1)
user.2:  mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) -> mor(o2, f´, o1)
user.3:  mor(o4,
             mor(o4, mor(o4, f´, o3) * mor(o3, g´, o2), o2) * mor(o2, h´, o1),
             o1)
           ->
```

```
         mor(o4,
             mor(o4, f´, o3)
               * mor(o3, mor(o3, g´, o2) * mor(o2, h´, o1), o1),
             o1)
user.4:  mor(o2,
           mor(o2, counit, t´(o2))
             * mor(t´(o2), costar(mor(o2, f´, t´(o1))), t´(o1)),
           t´(o1))
         -> mor(o2, f´, t´(o1))
user.5:  mor(t´(o1), costar(mor(o1, counit, t´(o1))), t´(o1))
         -> mor(t´(o1), id, t´(o1))
user.6:  mor(t´(o3),
           mor(t´(o3), costar(mor(o3, f´, t´(o2))), t´(o2))
             * mor(t´(o2), costar(mor(o2, g´, t´(o1))), t´(o1)),
           t´(o1))
         ->
         mor(t´(o3),
             costar(mor(o3,
                        mor(o3, f´, t´(o2))
                          * mor(t´(o2), costar(mor(o2, g´, t´(o1))), t´(o1)),
                        t´(o1))),
             t´(o1))
user.7:  mor(o3,
           mor(o3, counit, t´(o3))
             * mor(t´(o3),
                   mor(t´(o3), costar(mor(o3, f´, t´(o2))), t´(o2))
                     * mor(t´(o2), g´, o1),
                   o1),
           o1)
         -> mor(o3, mor(o3, f´, t´(o2)) * mor(t´(o2), g´, o1), o1)
user.8:  mor(t´(o4),
           mor(t´(o4), costar(mor(o4, f´, t´(o3))), t´(o3))
             * mor(t´(o3),
                   mor(t´(o3), costar(mor(o3, g´, t´(o2))), t´(o2))
                     * mor(t´(o2), h´, o1),
                   o1),
           o1)
         ->
         mor(t´(o4),
             mor(t´(o4),
                 costar(mor(o4,
                            mor(o4, f´, t´(o3))
                              * mor(t´(o3),
                                    costar(mor(o3, g´, t´(o2))),
                                    t´(o2)),
                            t´(o2))),
                 t´(o2))
               * mor(t´(o2), h´, o1),
             o1)
user.9:  mor(o1, mor(o1, counit, t´(o1)) * mor(t´(o1), gamma, o1), o1)
         -> mor(o1, id, o1)
user.10: mor(t´(o2),
           mor(t´(o2), costar(mor(o2, f´, t´(o1))), t´(o1))
             * mor(t´(o1), gamma, o1),
           o1)
         ->
         mor(t´(o2),
             mor(t´(o2), gamma, o2)
               * mor(o2, mor(o2, f´, t´(o1)) * mor(t´(o1), gamma, o1), o1),
             o1)
user.11: mor(o2,
           mor(o2, counit, t´(o2))
             * mor(t´(o2), mor(t´(o2), gamma, o2) * mor(o2, f´, o1), o1),
           o1)
         -> mor(o2, f´, o1)
user.12: mor(t´(o3),
           mor(t´(o3), costar(mor(o3, f´, t´(o2))), t´(o2))
             * mor(t´(o2), mor(t´(o2), gamma, o2) * mor(o2, g´, o1), o1),
           o1)
         ->
         mor(t´(o3),
             mor(t´(o3), gamma, o3)
```

```
                  * mor(o3,
                        mor(o3, f´, t´(o2))
                          * mor(t´(o2),
                                mor(t´(o2), gamma, o2) * mor(o2, g´, o1),
                                o1),
                        o1),
                o1)
LP2.12: assert
        mor(t´(o2),t(mor(o2,f´,o1)),t´(o1)) =
        mor(t´(o2),costar(mor(o2,mor(o2,f´,o1)*
                mor(o1,counit,t´(o1)),t´(o1))),t´(o1));
        mor(t´(t´(o1)),del,t´(o1)) =
        mor(t´(t´(o1)),costar(mor(t´(o1),id,t´(o1))),t´(o1))
..
Added 2 facts named user.13, user.14 to the system.
The system now contains 14 rewrite rules.  The rewriting system is NOT
guaranteed to terminate.
LP2.13: complete
The system is not guaranteed to terminate.  If it does terminate, then it is
complete.
LP2.14: display
Rewrite rules:
user.1:  mor(o2, mor(o2, id, o2) * mor(o2, f´, o1), o1) -> mor(o2, f´, o1)
user.2:  mor(o2, mor(o2, f´, o1) * mor(o1, id, o1), o1) -> mor(o2, f´, o1)
user.3:  mor(o4,
             mor(o4, mor(o4, f´, o3) * mor(o3, g´, o2), o2) * mor(o2, h´, o1),
             o1)
           ->
           mor(o4,
               mor(o4, f´, o3)
                 * mor(o3, mor(o3, g´, o2) * mor(o2, h´, o1), o1),
               o1)
user.4:  mor(o2,
             mor(o2, counit, t´(o2))
               * mor(t´(o2), costar(mor(o2, f´, t´(o1))), t´(o1)),
             t´(o1))
           -> mor(o2, f´, t´(o1))
user.5:  mor(t´(o1), costar(mor(o1, counit, t´(o1))), t´(o1))
           -> mor(t´(o1), id, t´(o1))
user.6:  mor(t´(o3),
             mor(t´(o3), costar(mor(o3, f´, t´(o2))), t´(o2))
               * mor(t´(o2), costar(mor(o2, g´, t´(o1))), t´(o1)),
             t´(o1))
           ->
           mor(t´(o3),
               costar(mor(o3,
                          mor(o3, f´, t´(o2))
                            * mor(t´(o2), costar(mor(o2, g´, t´(o1))), t´(o1)),
                          t´(o1))),
               t´(o1))
user.7:  mor(o3,
             mor(o3, counit, t´(o3))
               * mor(t´(o3),
                     mor(t´(o3), costar(mor(o3, f´, t´(o2))), t´(o2))
                       * mor(t´(o2), g´, o1),
                     o1),
             o1)
           -> mor(o3, mor(o3, f´, t´(o2)) * mor(t´(o2), g´, o1), o1)
user.8:  mor(t´(o4),
             mor(t´(o4), costar(mor(o4, f´, t´(o3))), t´(o3))
               * mor(t´(o3),
                     mor(t´(o3), costar(mor(o3, g´, t´(o2))), t´(o2))
                       * mor(t´(o2), h´, o1),
                     o1),
             o1)
           ->
           mor(t´(o4),
               mor(t´(o4),
                   costar(mor(o4,
                              mor(o4, f´, t´(o3))
                                * mor(t´(o3),
                                      costar(mor(o3, g´, t´(o2))),
```

```
                                t´(o2)),
                        t´(o2))),
              t´(o2))
            * mor(t´(o2), h´, o1),
          o1)
user.9:  mor(o1, mor(o1, counit, t´(o1)) * mor(t´(o1), gamma, o1), o1)
           -> mor(o1, id, o1)
user.10: mor(t´(o2),
             mor(t´(o2), costar(mor(o2, f´, t´(o1))), t´(o1))
               * mor(t´(o1), gamma, o1),
             o1)
           ->
           mor(t´(o2),
               mor(t´(o2), gamma, o2)
                 * mor(o2, mor(o2, f´, t´(o1)) * mor(t´(o1), gamma, o1), o1),
               o1)
user.11: mor(o2,
             mor(o2, counit, t´(o2))
               * mor(t´(o2), mor(t´(o2), gamma, o2) * mor(o2, f´, o1), o1),
             o1)
           -> mor(o2, f´, o1)
user.12: mor(t´(o3),
             mor(t´(o3), costar(mor(o3, f´, t´(o2))), t´(o2))
               * mor(t´(o2), mor(t´(o2), gamma, o2) * mor(o2, g´, o1), o1),
             o1)
           ->
           mor(t´(o3),
               mor(t´(o3), gamma, o3)
                 * mor(o3,
                       mor(o3, f´, t´(o2))
                         * mor(t´(o2),
                               mor(t´(o2), gamma, o2) * mor(o2, g´, o1),
                               o1),
                       o1),
               o1)
user.13: mor(t´(o2), t(mor(o2, f´, o1)), t´(o1))
           ->
           mor(t´(o2),
               costar(mor(o2,
                          mor(o2, f´, o1) * mor(o1, counit, t´(o1)),
                          t´(o1))),
               t´(o1))
user.14: mor(t´(t´(o1)), del, t´(o1))
           -> mor(t´(t´(o1)), costar(mor(t´(o1), id, t´(o1))), t´(o1))
End of input from file
`/home/wgehrke/Tex/article/monads/thesis/src/ccm/type.lp´.
LP3: quit
```

Appendix B

Elf Programs

This appendix presents the final rewriting based decision procedures implemented in the logic programming language Elf. An Elf program splits into a static part for type checking and a dynamic part for proof search. In the following the static parts are used by both dynamic parts.

B.1 Category

B.1.1 Static Part

```
%%% category
obj   : type.
mor   : obj -> obj -> type.
id    : {O : obj} mor O O.
%prefix 10 id
*     : mor O2 O3 -> mor O1 O2 -> mor O1 O3.
%infix right 5 *
```

B.1.2 Dynamic Part Cat1

```
%%% basic laws for a category (cat1)
%%  rules
|>    : mor A B -> mor A B -> type.
%infix none 1 |>
idl   : id B * F |> F.
idr   : F * id A |> F.
ass   : (F * G) * H |> F * (G * H).
%%  rewriting and one step of reduction
*>    : mor A B -> mor A B -> type.
%infix none 1 *>
+>    : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
step*l: F * G +> F´ * G
        <- F +> F´.
step*r: F * G +> F * G´
        <- G +> G´.
try   : F *> F´
        <- F +> F´´
        <- F´´ *> F´.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.1.3 Dynamic Part Cat2

```
%%% basic laws for a category (cat2)
%%  rules
|>    : mor A B -> mor A B -> type.
%infix none 1 |>
idl   : id B * F |> F.
idr   : F * id A |> F.
ass   : F * (G * H) |> (F * G) * H.
%%  rewriting and one step of reduction
*>    : mor A B -> mor A B -> type.
%infix none 1 *>
+>    : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
step*l: F * G +> F´ * G
        <- F +> F´.
step*r: F * G +> F * G´
        <- G +> G´.
try   : F *> F´
        <- F +> F´´
        <- F´´ *> F´.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.2 Functor

B.2.1 Static Part

```
%%% functor
cat   : type.
obj   : cat -> type.
mor   : obj C -> obj C -> type.
id    : {O : obj C} mor O O.
%prefix 10 id
*     : mor O2 O3 -> mor O1 O2 -> mor O1 O3.
%infix right 5 *
f´    : obj C -> obj D.
%prefix 15 f´
f     : mor A B -> mor (f´ A) (f´ B).
%prefix 15 f
```

B.2.2 Dynamic Part Fun1

```
%%% basic laws for a functor (fun1)
%%  rules
|>    : mor A B -> mor A B -> type.
%infix none 1 |>
idl   : id B * F |> F.
idr   : F * id A |> F.
ass   : (F * G) * H |> F * (G * H).
fid   : f (id A) |> id (f´ A).
f*    : f (F * G) |> (f F) * (f G).
%%  rewriting and one step of reduction
*>    : mor A B -> mor A B -> type.
%infix none 1 *>
+>    : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
step*l: F * G +> F´ * G
        <- F +> F´.
step*r: F * G +> F * G´
```

```
        <- G +> G'.
stepf : f F +> f F'
        <- F +> F'.
try   : F *> F'
        <- F +> F''
        <- F'' *> F'.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.2.3 Dynamic Part Fun2

```
%%% basic laws for a functor (fun2)
%%  rules
|>    : mor A B -> mor A B -> type.
%infix none 1 |>
idl   : id B * F |> F.
idr   : F * id A |> F.
ass   : (F * G) * H |> F * (G * H).
fid   : f (id A) |> id (f' A).
f*    : (f F) * (f G) |> f (F * G).
f*ass : (f F) * (f G) * H |> f (F * G) * H.
%%  rewriting and one step of reduction
*>    : mor A B -> mor A B -> type.
%infix none 1 *>
+>    : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F'
        <- F |> F'.
step*l: F * G +> F' * G
        <- F +> F'.
step*r: F * G +> F * G'
        <- G +> G'.
stepf : f F +> f F'
        <- F +> F'.
try   : F *> F'
        <- F +> F''
        <- F'' *> F'.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.3 Natural Transformation

B.3.1 Static Part

```
%%% natural transformation
cat   : type.
obj   : cat -> type.
mor   : obj C -> obj C -> type.
id    : {O : obj C} mor O O.
%prefix 10 id
*     : mor O2 O3 -> mor O1 O2 -> mor O1 O3.
%infix right 5 *
c     : cat.
d     : cat.
f'    : obj c -> obj d.
f     : mor A B -> mor (f' A) (f' B).
g'    : obj c -> obj d.
g     : mor A B -> mor (g' A) (g' B).
n     : {A : obj c} mor (f' A) (g' A).
```

```
%prefix 15 n
```

B.3.2 Dynamic Part Nat1

```
%%% basic laws for a natural transformation (nat1)
%%  rules
|>    : mor A B -> mor A B -> type.
%infix none 1 |>
idl   : id B * F |> F.
idr   : F * id A |> F.
ass   : (F * G) * H |> F * (G * H).
fid   : (f (id A)) |> (id (f´ A)).
f*    : (f (F * G)) |> ((f F) * (f G)).
gid   : (g (id A)) |> (id (g´ A)).
g*    : (g (F * G)) |> ((g F) * (g G)).
n1    : (n B) * (f F) |> (g F) * (n A).
n2    : (n B) * (f F) * H |> (g F) * (n A) * H.
%%  rewriting and one step of reduction
*>    : mor A B -> mor A B -> type.
%infix none 1 *>
+>    : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
step*l: F * G +> F´ * G
        <- F +> F´.
step*r: F * G +> F * G´
        <- G +> G´.
stepf : (f F) +> (f F´)
        <- F +> F´.
stepg : (g G) +> (g G´)
        <- G +> G´.
try   : F *> F´
        <- F +> F´´
        <- F´´ *> F´.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.3.3 Dynamic Part Nat2

```
%%% basic laws for a natural transformation (nat2)
%%  rules
|>    : mor A B -> mor A B -> type.
%infix none 1 |>
idl   : id B * F |> F.
idr   : F * id A |> F.
ass   : (F * G) * H |> F * (G * H).
fid   : (f (id A)) |> (id (f´ A)).
f*    : (f F) * (f G) |> (f (F * G)).
f*ass : (f F) * (f G) * H |> (f (F * G)) * H.
gid   : (g (id A)) |> (id (g´ A)).
g*    : (g F) * (g G) |> (g (F * G)).
g*ass : (g F) * (g G) * H |> (g (F * G)) * H.
n1    : (n B) * (f F) |> (g F) * (n A).
n2    : (n B) * (f F) * H |> (g F) * (n A) * H.
%%  rewriting and one step of reduction
*>    : mor A B -> mor A B -> type.
%infix none 1 *>
+>    : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
step*l: F * G +> F´ * G
        <- F +> F´.
step*r: F * G +> F * G´
```

```
        <- G +> G´.
stepf : (f F) +> (f F´)
        <- F +> F´.
stepg : (g G) +> (g G´)
        <- G +> G´.
try   : F *> F´
        <- F +> F´´
        <- F´´ *> F´.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.4 Adjunction

B.4.1 Adj

Static Part

```
%%% adjunction
cat   : type.
obj   : cat -> type.
mor   : obj C -> obj C -> type.
id    : mor O O.
*     : mor O2 O3 -> mor O1 O2 -> mor O1 O3.
%infix right 10 *
c     : cat.
c´    : cat.
l´    : obj c -> obj c´.
l     : mor A B -> mor (l´ A) (l´ B).
r´    : obj c´ -> obj c.
r     : mor A B -> mor (r´ A) (r´ B).
zeta  : mor (l´ X) Y -> mor X (r´ Y).
zeta-1: mor X (r´ Y) -> mor (l´ X) Y.
eta   : mor O (r´ (l´ O)).
eps   : mor (l´ (r´ O)) O.
```

Dynamic Part

```
%%% laws for adjunction
%%  rules
|>    : mor A B -> mor A B -> type.
%infix none 1 |>
idlr  : id * F |> F.
idrr  : F * id |> F.
assr  : (F * G) * H |> F * (G * H).
lidr  : (l id) |> id.
l*r   : (l F) * (l G) |> (l (F * G)).
l*a   : (l F) * (l G) * H |> (l (F * G)) * H.
zel   : (zeta (eps * (l F))) |> F.
zei   : (zeta eps) |> id.
elz   : eps * (l (zeta F)) |> F.
elza  : eps * (l (zeta F)) * G |> F * G.
z*l   : (zeta F) * G |> (zeta (F * (l G))).
rdef  : (r F) |> (zeta (F * eps)).
z-1def: (zeta-1 F) |> eps * (l F).
etad  : eta |> (zeta id).
%%  rewriting and one step of reduction
*>  : mor A B -> mor A B -> type.
%infix none 1 *>
+>  : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
step*l: (F * G) +> (F´ * G)
        <- F +> F´.
```

```
step*r: (F * G) +> (F * G´)
        <- G +> G´.
stepl : (l F) +> (l F´)
        <- F +> F´.
stepr : (r G) +> (r G´)
        <- G +> G´.
stepz : (zeta F) +> (zeta F´)
        <- F +> F´.
stepzi: (zeta-1 F) +> (zeta-1 F´)
        <- F +> F´.
try   : F *> F´
        <- F +> F´´
        <- F´´ *> F´.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.4.2 Subst

Static Part

```
%%% substitution in CCL
obj   : type.
mor   : obj -> obj -> type.
i     : mor O O.
*     : mor O2 O3 -> mor O1 O2 -> mor O1 O3.
%infix right 10 *
#     : obj -> obj -> obj.
%infix right 5 #
=>    : obj -> obj -> obj.
%infix right 10 =>
f     : mor (A # B) A.
s     : mor (A # B) B.
+     : mor O A -> mor O B -> mor O (A # B).
%infix right 5 +
c     : mor (A # B) C -> mor A (B => C).
%prefix 20 c
```

Dynamic Part

```
%%% laws for substitution in CCL
%%  rules
|>  : mor A B -> mor A B -> type.
%infix none 1 |>
fstr  : f * (S + T) |> S.
sndr  : s * (S + T) |> T.
idlr  : i * F |> F.
idrr  : F * i |> F.
spair : (f * T) + (s * T) |> T.
assr  : (S * T) * U |> S * (T * U).
dpair : (S + T) * U |> (S * U) + (T * U).
dlam  : (c S) * T |> c (S * (T * f + s)).
fsi   : f + s |> i.
%%  rewriting and one step of reduction
*>  : mor A B -> mor A B -> type.
%infix none 1 *>
+>  : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
step*l: (F * G) +> (F´ * G)
        <- F +> F´.
step*r: (F * G) +> (F * G´)
        <- G +> G´.
step+l: (F + G) +> (F´ + G)
        <- F +> F´.
step+r: (F + G) +> (F + G´)
```

```
        <- G +> G´.
stepc : (c F) +> (c F´)
        <- F +> F´.
try   : F *> F´
        <- F +> F´´
        <- F´´ *> F´.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.5 Monad

B.5.1 Via Adjunction

Static Part

```
%%% definitions for monad with adjunction
cat   : type.
obj   : cat -> type.
mor   : obj C -> obj C -> type.
id    : mor O O.
*     : mor O2 O3 -> mor O1 O2 -> mor O1 O3.
%infix right 10 *
c     : cat.
c´    : cat.
l´    : obj c -> obj c´.
l     : mor A B -> mor (l´ A) (l´ B).
r´    : obj c´ -> obj c.
zeta  : mor (l´ X) Y -> mor X (r´ Y).
eps   : mor (l´ (r´ O)) O.
t     : mor A B -> mor (r´ (l´ A)) (r´ (l´ B)).
eta   : mor O (r´ (l´ O)).
mu    : mor (r´ (l´ (r´ (l´ O)))) (r´ (l´ O)).
ki    : mor O (r´ (l´ O)).
k*    : mor O2 (r´ (l´ O3)) -> mor O1 (r´ (l´ O2)) -> mor O1 (r´ (l´ O3)).
%infix right 10 k*
```

Dynamic Part

```
%%% rules for monad with adjunction
%%  rules
|>  : mor A B -> mor A B -> type.
%infix none 1 |>
idlr  : id * F |> F.
idrr  : F * id |> F.
assr  : (F * G) * H |> F * (G * H).
lidr  : (l id) |> id.
l*r   : (l F) * (l G) |> (l (F * G)).
l*a   : (l F) * (l G) * H |> (l (F * G)) * H.
zel   : (zeta (eps * (l F))) |> F.
zei   : (zeta eps) |> id.
elz   : eps * (l (zeta F)) |> F.
elza  : eps * (l (zeta F)) * G |> F * G.
z*l   : (zeta F) * G |> (zeta (F * (l G))).
tdef  : (t F) |> (zeta ((l F) * eps)).
etad  : eta |> (zeta id).
mud   : mu |> (zeta (eps * eps)).
kid   : ki |> eta.
k*d   : (F k* G) |> mu * (t F) * G.
%%  rewriting and one step of reduction
*>  : mor A B -> mor A B -> type.
%infix none 1 *>
+>  : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
```

```
step*l: (F * G) +> (F´ * G)
        <- F +> F´.
step*r: (F * G) +> (F * G´)
        <- G +> G´.
stepl : (l F) +> (l F´)
        <- F +> F´.
stepz : (zeta F) +> (zeta F´)
        <- F +> F´.
stept : (t F) +> (t F´)
        <- F +> F.
try   : F *> F´
        <- F +> F´´
        <- F´´ *> F´.
last  : F *> F.
%%  equality
==     : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.5.2 Via Kleisli Category

Static Part

```
%%% definitions for monad with Kleisli category
obj    : type.
mor    : obj -> obj -> type.
id     : mor O O.
*      : mor O2 O3 -> mor O1 O2 -> mor O1 O3.
%infix right 10 *
t´     : obj -> obj.
t      : mor A B -> mor (t´ A) (t´ B).
eta    : mor O (t´ O).
mu     : mor (t´ (t´ O)) (t´ O).
^      : mor A (t´ B) -> mor (t´ A) (t´ B).
%postfix 15 ^
```

Dynamic Part

```
%%% rules for monad with Kleisli category
%%  rules
|>   : mor A B -> mor A B -> type.
%infix none 1 |>
idlr  : id * F |> F.
idrr  : F * id |> F.
assr  : (F * G) * H |> F * (G * H).
eta^  : eta ^ |> id.
^eta  : F ^ * eta |> F.
^*^   : F ^ * G ^ |> (F ^ * G) ^.
^etaa : F ^ * eta * G |> F * G.
^*^a  : F ^ * G ^ * H |> (F ^ * G) ^ * H.
tdef  : (t F) |> (eta * F) ^.
mudef : mu |> id ^.
%%  rewriting and one step of reduction
*>   : mor A B -> mor A B -> type.
%infix none 1 *>
+>   : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
step*l: (F * G) +> (F´ * G)
        <- F +> F´.
step*r: (F * G) +> (F * G´)
        <- G +> G´.
stept : (t F) +> (t F´)
        <- F +> F´.
step^ : F ^ +> F´ ^
        <- F +> F´.
try   : F *> F´
        <- F +> F´´
```

```
        <- F´´ *> F´.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

B.6 Computational Co-monad

B.6.1 Static Part

```
%%% definitions for computational co-monad with Kleisli category
obj   : type.
mor   : obj -> obj -> type.
id    : mor O O.
*     : mor O2 O3 -> mor O1 O2 -> mor O1 O3.
%infix right 10 *
t´    : obj -> obj.
t     : mor A B -> mor (t´ A) (t´ B).
eps   : mor (t´ O) O.
del   : mor (t´ O) (t´ (t´ O)).
gam   : mor O (t´ O).
^     : mor (t´ A) B -> mor (t´ A) (t´ B).
%postfix 15 ^
cki   : mor (t´ O) O.
ck*   : mor (t´ O2) O3 -> mor (t´ O1) O2 -> mor (t´ O1) O3.
%infix right 10 ck*
alg   : mor A B -> mor (t´ A) B.
fun   : mor (t´ A) B -> mor A B.
```

B.6.2 Dynamic Part

```
%%% rules for computational co-monad with Kleisli category
%%  rules
|>  : mor A B -> mor A B -> type.
%infix none 1 |>
idlr  : id * F |> F.
idrr  : F * id |> F.
assr  : (F * G) * H |> F * (G * H).
eps^  : eps ^ |> id.
eps^  : eps * F ^ |> F.
^*^   : F ^ * G ^ |> (F * G ^) ^.
eps^a : eps * F ^ * G |> F * G.
^*^a  : F ^ * G ^ * H |> (F * G ^) ^ * H.
tdef  : (t F) |> (F * eps) ^.
deldef: del |> id ^.
ckid  : cki |> eps.
ck*d  : (F ck* G) |> F * (t G) * del.
epsgam: eps * gam |> id.
^gam  : F ^ * gam |> gam * F * gam.
ega   : eps * gam * F |> F.
^gama : F ^ * gam * G |> gam * F * gam * G.
algd  : (alg F) |> F * eps.
fund  : (fun F) |> F * gam.
%%  rewriting and one step of reduction
*>  : mor A B -> mor A B -> type.
%infix none 1 *>
+>  : mor A B -> mor A B -> type.
%infix none 1 +>
simple: F +> F´
        <- F |> F´.
step*l: (F * G) +> (F´ * G)
        <- F +> F´.
step*r: (F * G) +> (F * G´)
        <- G +> G´.
stept : (t F) +> (t F´)
        <- F +> F´.
```

```
step^ : F ^ +> F' ^
        <- F +> F'.
try   : F *> F'
        <- F +> F''
        <- F'' *> F'.
last  : F *> F.
%%  equality
==    : mor A B -> mor A B -> type.
%infix none 1 ==
moreq : F == G
        <- F *> H
        <- G *> H.
```

Appendix C

Related Code and Examples in Elf

In this appendix the usage of Elf is demonstrated. First we show that critical pairs can indeed be generated with the help of Elf and afterwards we give examples of the application of the extended version of the language which was joint work with Frank Pfenning. Finally we include test queries related to examples from the thesis.

C.1 Generating Critical Pairs

This section demonstrated that Elf can be used to generate higher-order critical pairs. The quantification method for the representation of the rules is due to Pfenning (cf. the examples from the Elf distribution). With the help of this method the explicitly quantified variables can be instantiated with free variables (for unification) but also with new bound variables (for determining variable positions in a term).

C.1.1 Static Part

```
%%% static part for monadic laws without type information
%%  for monadic laws
term  : type.
unit  : term -> term.
star  : (term -> term) -> term -> term.
%%  auxiliary for critical pairs
qrule : type.
all1  : (term -> qrule) -> qrule.
all2  : ((term -> term) -> qrule) -> qrule.
srule : term -> term -> qrule.
```

C.1.2 Dynamic Part

```
%%% dynamic part for monadic laws without type information
rules : qrule -> type.
su_   : rules (all1 [m] srule (star unit m) m).
s_u   : rules (all2 [m] all1 [n] srule (star m (unit n)) (m n)).
s_s   : rules (all2 [m] all2 [n] all1 [o]
         srule (star m (star n o)) (star ([v] star m (n v)) o)).
%%  critical pair part
nonvar: term -> type.
nvunit: nonvar (unit M).
nvstar: nonvar (star M N).
us    : term -> qrule -> term -> type.
```

```
inst21: us T1 (all1 Rule) T2
         <- us T1 (Rule (T : term)) T2.
inst22: us T1 (all2 Rule) T2
         <- us T1 (Rule (TT : term -> term)) T2.
unify : us L (srule L R) R.
unisub: term -> term -> qrule -> term -> type.
toppos: unisub L1´ T1 Rule2 T2
         <- nonvar L1´ <- us T1 Rule2 T2.
unitp : unisub (unit M´) (unit M) Rule2 (unit T2)
         <- unisub M´ M Rule2 T2.
starl : unisub (star M´ N´) (star M N) Rule2 (star T2 N)
         <- {t:term} unisub (M´ t) (M t) Rule2 (T2 t).
starr : unisub (star M´ N´) (star M N) Rule2 (star M T2)
         <- unisub N´ N Rule2 T2.
cp´   : qrule -> qrule -> qrule -> term -> term -> type.
inst11: cp´ (all1 Rule1´) (all1 Rule1) Rule2 T1 T2
         <- {t:term} cp´ (Rule1´ t) (Rule1 T) Rule2 T1 T2.
inst12: cp´ (all2 Rule1´) (all2 Rule1) Rule2 T1 T2
         <- {tt : term -> term}
             cp´ (Rule1´ tt) (Rule1 TT) Rule2 T1 T2.
overl : cp´ (srule L1´ R1´) (srule L1 R1) Rule2 R1 T2
         <- unisub L1´ L1 Rule2 T2.
cp    : qrule -> qrule -> term -> term -> type.
cpdef : cp Rule1 Rule2 T1 T2
         <- cp´ Rule1 Rule1 Rule2 T1 T2.
cps   : term -> term -> type.
cpsdef: cps T1 T2
         <- rules Rule1 <- rules Rule2 <- cp Rule1 Rule2 T1 T2.
%%  rewriting part
rew   : term -> term -> type.
step  : rew T T´
         <- rules Rule <- unisub T T Rule T´´ <- rew T´´ T´.
last  : rew T T.
eq    : term -> term -> type.
teq   : eq T1 T2
         <- rew T1 T <- rew T2 T.
```

C.1.3 Generated Critical Pairs

```
- top();
Using: /home/wgehrke/Tex/article/monads/thesis/src/cps/dynamic1.elf
Solving for: rules nonvar us unisub cp´ cp cps rew eq
?- cps X Y.
Solving...
Y = Y,
X = Y.
;
Y = unit T,
X = unit T.
;
Y = star ([v:term] star unit (T2 v)) T1,
X = star T2 T1.
;
Y = unit Y1,
X = unit Y1.
;
Y = Y,
X = X.
(( TT [] : term -> term,
   !tt : term -> term,
   T3 [!tt] : term,
   Y [] : term,
   X [] : term,
   TT T3 = Y,
   TT T3 = X ))
;
Y = star ([v:term] X2 v) X1,
X = star ([v:term] star unit (X2 v)) X1.
;
Y = star TT1 X1,
X = star ([v:term] star TT1 (unit v)) X1.
```

```
;
Y = star TT2 (X4 X3),
X = star ([v:term] star TT2 (X4 v)) (unit X3).
;
Y = star ([v:term] star TT3 (X6 v)) X5,
X = star ([v:term] star TT3 (X6 v)) X5.
;
Y = star TT4 (star ([v:term] star ([v:term] X6 v) (X8 v)) X7),
X = star ([v:term] star TT4 (X6 v)) (star X8 X7).
;
no more solutions
?- Z : {m} eq (star unit m) m.
Solving...
Z = [m:term] teq last (step last (toppos (inst21 unify) nvstar) su_).
yes
?- Z : {m} eq (star unit (unit m)) m.
Solving...
no
?- val it = () : unit
-
```

C.2 HRS

This section demonstrates the extended version of Elf which was joint work with Frank Pfenning. These traces show the check of convergence for higher-order critical pairs. The implementation is even able to handle dependent types in the higher-order case which is based on forthcoming work [Vir95].

C.2.1 Lambda Calculus

Static Part Untyped

```
%%% definitions for pure lambda calculus
term  : type.
abs   : (term -> term) -> term.
app   : term -> term -> term.
```

Dynamic Part Untyped

```
%%% rules for pure lambda calculus
%%  rules
|>    : term -> term -> type.
%infix none 1 |>
beta  : (app (abs M) N) |> (M N).
eta   : (abs ([v] app M v)) |> M.
%%  rewriting and one step reduction
*>    : term -> term -> type.
%infix none 1 *>
+>    : term -> term -> type.
%infix none 1 +>
simple: M +> M'
        <- M |> M'.
stepab: (abs M) +> (abs M')
        <- {v} (M v) +> (M' v).
stepal: (app M N) +> (app M' N)
        <- M +> M'.
stepar: (app M N) +> (app M N')
        <- N +> N'.
try   : M *> M'
        <- M +> M''
        <- M'' *> M'.
last  : M *> M.
%%  equality
==    : term -> term -> type.
%infix none 1 ==
termeq: M == N
        <- M *> O
```

```
        <- N *> O.
```

Higher-Order Critical Pairs

```
- crit_pairs (c "|>") (c "*>");
For normalization:
Using: /home/wgehrke/Tex/article/monads/thesis/src/lam/pbe.elf
Solving for: |> *> +> ==
Checking 2 rules for critical pairs:
 beta eta
All critical overlaps for beta < beta joinable
Critical pair for beta < eta
Term 1: app M3 N2
Term 2: app M3 N2
Joined trivially.
All critical overlaps for beta < eta joinable
Critical pair for eta < beta
Term 1: abs M4
Term 2: abs [v:term] M4 v
Joined trivially.
All critical overlaps for eta < beta joinable
All critical overlaps for eta < eta joinable
Checking subsumption
Checked 2 rules for critical pairs:
 beta eta
Remaining Constraint Errors: 0
Critical Pairs: 2
Trivially Joined: 2
Equal Normal Forms: 0
Subsumed: 0
Remaining New Rules: 0
val it = () : unit
-
```

Static Part Typed

```
%%% definitions for typed lambda calculus
tp    : type.
term  : tp -> type.
=>    : tp -> tp -> tp.
%infix right 10 =>
abs   : (term A -> term B) -> term (A => B).
app   : term (A => B) -> term A -> term B.
```

Dynamic Part Typed

```
%%% rules for typed lambda calculus
%%  rules
|>    : term T -> term T -> type.
%infix none 1 |>
beta  : (app (abs M) N) |> (M N).
eta   : (abs ([v] app M v)) |> M.
%%  rewriting and one step reduction
*>    : term T -> term T -> type.
%infix none 1 *>
+>    : term T -> term T -> type.
%infix none 1 +>
simple: M +> M´
        <- M |> M´.
stepab: (abs M) +> (abs M´)
        <- {v} (M v) +> (M´ v).
stepal: (app M N) +> (app M´ N)
        <- M +> M´.
stepar: (app M N) +> (app M N´)
        <- N +> N´.
try   : M *> M´
        <- M +> M´´
        <- M´´ *> M´.
last  : M *> M.
%%  equality
```

```
==    : term T -> term T -> type.
%infix none 1 ==
termeq: M == N
        <- M *> O
        <- N *> O.
```

Higher-Order Critical Pairs

```
- crit_pairs (c "|>") (c "*>");
For normalization:
Using: /home/wgehrke/Tex/article/monads/thesis/src/lam/tbe.elf
Solving for: |> *> +> ==
Checking 2 rules for critical pairs:
 beta eta
All critical overlaps for beta < beta joinable
Critical pair for beta < eta
Term 1: app M1 N1
Term 2: app M1 N1
Joined trivially.
All critical overlaps for beta < eta joinable
Critical pair for eta < beta
Term 1: abs M2
Term 2: abs [v:term X1] M2 v
Joined trivially.
All critical overlaps for eta < beta joinable
All critical overlaps for eta < eta joinable
Checking subsumption
Checked 2 rules for critical pairs:
 beta eta
Remaining Constraint Errors: 0
Critical Pairs: 2
Trivially Joined: 2
Equal Normal Forms: 0
Subsumed: 0
Remaining New Rules: 0
val it = () : unit
-
```

C.2.2 Monadic Laws

Static Part Untyped

```
%%% definitions for monadic style without type information
term  : type.
unit  : term -> term.
star  : (term -> term) -> term -> term.
```

Dynamic Part Untyped

```
%%% rules for monadic style without type information
%%  rules
|>    : term -> term -> type.
%infix none 1 |>
su_  : (star unit M) |> M.
s_u  : (star M (unit N)) |> (M N).
s_s  : (star M (star N X)) |> (star ([x] star M (N x)) X).
%%  rewriting and one step reduction
*>    : term -> term -> type.
%infix none 1 *>
+>    : term -> term -> type.
%infix none 1 +>
simple: M +> M´
        <- M |> M´.
stepu : (unit M) +> (unit M´)
        <- M +> M´.
stepsl: (star M N) +> (star M´ N)
        <- {x} ((M x) +> (M´ x)).
stepsr: (star M N) +> (star M N´)
        <- N +> N´.
try   : M *> M´
```

```
        <- M +> M´´
        <- M´´ *> M´.
last  : M *> M.
%%  equality
==    : term -> term -> type.
%infix none 1 ==
termeq: M == N
        <- M *> O
        <- N *> O.
```

Higher-Order Critical Pairs

```
- crit_pairs (c "|>") (c "*>");
For normalization:
Using: /home/wgehrke/Tex/article/monads/thesis/src/mfs/pms.elf
Solving for: |> *> +> ==
Checking 3 rules for critical pairs:
 su_ s_u s_s
All critical overlaps for su_ < su_ joinable
Critical pair for su_ < s_u
Term 1: unit N1
Term 2: unit N1
Joined trivially.
All critical overlaps for su_ < s_u joinable
All critical overlaps for s_u < su_ joinable
Critical pair for su_ < s_s
Term 1: star N2 X
Term 2: star ([x:term] star unit (N2 x)) X
Normalizing...done
Joined after normalization
All critical overlaps for su_ < s_s joinable
Critical pair for s_s < su_
Term 1: star ([x:term] star M (unit x)) X1
Term 2: star M X1
Normalizing...done
Joined after normalization
All critical overlaps for s_s < su_ joinable
All critical overlaps for s_u < s_u joinable
All critical overlaps for s_u < s_s joinable
Critical pair for s_s < s_u
Term 1: star ([x:term] star M1 (N1 x)) (unit N)
Term 2: star M1 (N1 N)
Normalizing...done
Joined after normalization
All critical overlaps for s_s < s_u joinable
Critical pair for s_s < s_s
Term 1: star ([x:term] star M1 (N3 x)) (star N2 X)
Term 2: star M1 (star ([x:term] star N3 (N2 x)) X)
Normalizing...done
Joined after normalization
All critical overlaps for s_s < s_s joinable
Checking subsumption
Checked 3 rules for critical pairs:
 su_ s_u s_s
Remaining Constraint Errors: 0
Critical Pairs: 5
Trivially Joined: 1
Equal Normal Forms: 4
Subsumed: 0
Remaining New Rules: 0
val it = () : unit
-
```

Static Part Typed

```
%%% definitions for monadic style with type information
tp    : type.
term  : tp -> type.
t     : tp -> tp.
unit  : term A -> term (t A).
star  : (term A -> term (t B)) -> term (t A) -> term (t B).
```

Dynamic Part Typed

```
%%% rules for monadic style with type information
%%  rules
|>    : term T -> term T -> type.
%infix none 1 |>
su_  : (star unit M) |> M.
s_u  : (star M (unit N)) |> (M N).
s_s  : (star M (star N X)) |> (star ([x] star M (N x)) X).
%%  rewriting and one step reduction
*>    : term T -> term T -> type.
%infix none 1 *>
+>    : term T -> term T -> type.
%infix none 1 +>
simple: M +> M'
        <- M |> M'.
stepu : (unit M) +> (unit M')
        <- M +> M'.
stepsl: (star M N) +> (star M' N)
        <- {x} ((M x) +> (M' x)).
stepsr: (star M N) +> (star M N')
        <- N +> N'.
try   : M *> M'
        <- M +> M''
        <- M'' *> M'.
last  : M *> M.
%%  equality
==    : term T -> term T -> type.
%infix none 1 ==
termeq: M == N
        <- M *> O
        <- N *> O.
```

Higher-Order Critical Pairs

```
- crit_pairs (c "|>") (c "*>");
For normalization:
Using: /home/wgehrke/Tex/article/monads/thesis/src/mfs/tms.elf
Solving for: |> *> +> ==
Checking 3 rules for critical pairs:
 su_ s_u s_s
All critical overlaps for su_ < su_ joinable
Critical pair for su_ < s_u
Term 1: unit N1
Term 2: unit N1
Joined trivially.
All critical overlaps for su_ < s_u joinable
All critical overlaps for s_u < su_ joinable
Critical pair for su_ < s_s
Term 1: star N2 X1
Term 2: star ([x:term X3] star unit (N2 x)) X1
Normalizing...done
Joined after normalization
All critical overlaps for su_ < s_s joinable
Critical pair for s_s < su_
Term 1: star ([x:term X31] star M (unit x)) X2
Term 2: star M X2
Normalizing...done
Joined after normalization
All critical overlaps for s_s < su_ joinable
All critical overlaps for s_u < s_u joinable
All critical overlaps for s_u < s_s joinable
Critical pair for s_s < s_u
Term 1: star ([x:term X31] star M1 (N1 x)) (unit N)
Term 2: star M1 (N1 N)
Normalizing...done
Joined after normalization
All critical overlaps for s_s < s_u joinable
Critical pair for s_s < s_s
Term 1: star ([x:term X31] star M1 (N3 x)) (star N2 X)
Term 2: star M1 (star ([x:term X32] star N3 (N2 x)) X)
Normalizing...done
```

```
Joined after normalization
All critical overlaps for s_s < s_s joinable
Checking subsumption
Checked 3 rules for critical pairs:
 su_ s_u s_s
Remaining Constraint Errors: 0
Critical Pairs: 5
Trivially Joined: 1
Equal Normal Forms: 4
Subsumed: 0
Remaining New Rules: 0
val it = () : unit
-
```

C.2.3 Higher-Order Primitive Recursion

Static Part

```
%%% definitions for higher order primitive recursion
tp   : type.
term : tp -> type.
=>   : tp -> tp -> tp.
%infix right 10 =>
nat  : tp.
z    : term nat.
s    : term nat -> term nat.
abs  : (term A -> term B) -> term (A => B).
app  : term (A => B) -> term A -> term B.
rec  : term A -> term (nat => A => A) -> term nat -> term A.
```

Dynamic Part

```
%%% rules for higher order primitive recursion
%%  rules
|>     : term T -> term T -> type.
%infix none 1 |>
beta  : (app (abs L) M) |> (L M).
eta   : (abs ([x] app L x)) |> L.
base  : (rec L M z) |> L.
hpr   : (rec L M (s N)) |> (app (app M N) (rec L M N)).
%%  rewriting and one step reduction
*>     : term T -> term T -> type.
%infix none 1 *>
+>     : term T -> term T -> type.
%infix none 1 +>
simple: M +> M´
        <- M |> M´.
steps : (s N) +> (s N´)
        <- N +> N´.
stepab: (abs M) +> (abs M´)
        <- {v} (M v) +> (M´ v).
stepal: (app M N) +> (app M´ N)
        <- M +> M´.
stepar: (app M N) +> (app M N´)
        <- N +> N´.
stepr1: (rec L M N) +> (rec L´ M N)
        <- L +> L´.
stepr2: (rec L M N) +> (rec L M´ N)
        <- M +> M´.
stepr3: (rec L M N) +> (rec L M N´)
        <- N +> N´.
try   : M *> M´
        <- M +> M´´
        <- M´´ *> M´.
last  : F *> F.
==    : term T -> term T -> type.
%infix none 1 ==
trmeq: M == N
        <- M *> O
        <- N *> O.
```

Higher-Order Critical Pairs

```
- crit_pairs (c "|>") (c "*>");
For normalization:
Using: /home/wgehrke/Tex/article/monads/thesis/src/hpr/hpr.elf
Solving for: |> *> +> ==
Checking 4 rules for critical pairs:
 beta eta base hpr
All critical overlaps for beta < beta joinable
Critical pair for beta < eta
Term 1: app L M1
Term 2: app L M1
Joined trivially.
All critical overlaps for beta < eta joinable
Critical pair for eta < beta
Term 1: abs L1
Term 2: abs [x:term X1] L1 x
Joined trivially.
All critical overlaps for eta < beta joinable
All critical overlaps for beta < base joinable
All critical overlaps for base < beta joinable
All critical overlaps for beta < hpr joinable
All critical overlaps for hpr < beta joinable
All critical overlaps for eta < eta joinable
All critical overlaps for eta < base joinable
All critical overlaps for base < eta joinable
All critical overlaps for eta < hpr joinable
All critical overlaps for hpr < eta joinable
All critical overlaps for base < base joinable
All critical overlaps for base < hpr joinable
All critical overlaps for hpr < base joinable
All critical overlaps for hpr < hpr joinable
Checking subsumption
Checked 4 rules for critical pairs:
 beta eta base hpr
Remaining Constraint Errors: 0
Critical Pairs: 2
Trivially Joined: 2
Equal Normal Forms: 0
Subsumed: 0
Remaining New Rules: 0
val it = () : unit
-
```

C.3 Sample Query Runs

In this section we present queries related to problems stated in the thesis. A query in Elf is entered similar to Prolog by "?- ". Elf does not only compute an answer substitution like Prolog but can additionally compute a term representing the proof found by the search.

C.3.1 Monad via Adjunction

```
- top();
Using: /home/wgehrke/Tex/article/monads/thesis/src/mon/adjmon.elf
Solving for: |> *> +> ==
?- Q1 : {F} (ki k* F) == F.
Solving...
Q1 =
   [F:mor X1 (r´ (l´ X))]
      moreq last
         (try (try (try (try (try (try (
try (try (try (try (try (try (try last (simple zel)) (stepz (simple idlr)))
                          (stepz (simple elza)))
                     (stepz (step*r (step*l (stepl (simple etad))))))
                (stepz (step*r (step*l (stepl (simple kid))))))
         (stepz (step*r (simple assr))))
    (stepz (step*r (simple elz))))
```

```
                                       (stepz
                                           (step*r (step*r (stepl (simple z*l))))))
                                  (stepz
                                      (step*r
                                          (step*r (stepl (step*l (simple tdef)))))))
                             (stepz (simple assr)))
                        (simple z*l))
                   (step*l (simple mud)))
              (simple k*d)).
yes
?- Q2 : {F} (F k* ki) == F.
Solving...
Q2 =
   [F:mor X1 (r´ (l´ X))]
      moreq last
         (try (try (try (try (try (try (
try (try (try (try (try (try (try last (simple zel))
                             (stepz (step*r (simple idrr))))
                        (stepz (step*r (step*r (simple elz)))))
                   (stepz (step*r (step*r (step*r (stepl (simple etad))))))
              (stepz (step*r (step*r (step*r (stepl (simple kid)))))))
         (stepz (step*r (simple assr))))
    (stepz (step*r (simple elz))))
                                       (stepz
                                           (step*r (step*r (stepl (simple z*l))))))
                                  (stepz
                                      (step*r
                                          (step*r (stepl (step*l (simple tdef)))))))
                             (stepz (simple assr)))
                        (simple z*l))
                   (step*l (simple mud)))
              (simple k*d)).
yes
?- Q3 : {F}{G}{H} ((F k* G) k* H) == (F k* (G k* H)).
Solving...
[Major collection... 73% used (3218472/4397596), 1860 msec]
[Increasing heap to 11060k]
Q3 =
   [F:mor X1 (r´ (l´ X))] [G:mor (X2 F) (r´ (l´ X1))]
      [H:mor (X3 F G) (r´ (l´ (X2 F)))]
      moreq
         (try (try (try (try (try (try (
try (try (try (try (try (try (try (try (
try (try (try last (stepz (step*r (step*r (step*r (simple assr))))))
         (stepz (step*r (step*r (step*r (simple elz))))))
    (stepz (step*r (step*r (step*r (step*r (stepl (simple z*l))))))))
                                       (
stepz (step*r (step*r (step*r (step*r (stepl (step*l (simple tdef)))))))))
                                  (stepz (step*r (step*r (simple assr)))))
                             (stepz (step*r (step*r (simple elz)))))
                        (stepz (step*r (step*r (step*r (stepl (simple z*l)))))))
                   (stepz
                       (step*r (step*r (step*r (stepl (step*l (simple mud))))))))
              (stepz (step*r (step*r (step*r (stepl (simple k*d)))))))
         (stepz (step*r (simple assr))))
    (stepz (step*r (simple elz))))
                                       (stepz
                                           (step*r (step*r (stepl (simple z*l))))))
                                  (stepz
                                      (step*r
                                          (step*r (stepl (step*l (simple tdef)))))))
                             (stepz (simple assr)))
                        (simple z*l))
                   (step*l (simple mud)))
              (simple k*d))
         (try (try (try (try (try (try (
try (try (try (try (try (try (try (try (
try (try (try (try (try last (stepz (step*r (simple assr))))
                   (stepz (step*r (simple assr))))
              (stepz (step*r (simple elza))))
         (stepz (step*r (step*r (step*l (stepl (simple z*l)))))))
    (stepz (step*r (step*r (step*l (stepl (step*l (simple tdef))))))))
```

```
                                       (stepz (simple assr)))
                                  (stepz (simple assr)))
                             (stepz (simple elza)))
                        (stepz (step*r (step*l (stepl (simple z*l))))))
                   (stepz (step*r (step*l (stepl (step*l (simple mud)))))))
              (stepz (step*r (step*l (stepl (simple k*d))))))
         (stepz (step*r (simple assr))))
    (stepz (step*r (simple elz))))
                                       (stepz
                                           (step*r (step*r (stepl (simple z*l))))))
                                  (stepz
                                      (step*r
                                          (step*r (stepl (step*l (simple tdef)))))))
                             (stepz (simple assr)))
                        (simple z*l))
                   (step*l (simple mud)))
              (simple k*d)).
yes
?- val it = () : unit
-
```

C.3.2 Monad via Kleisli Category

```
- top();
Using: /home/wgehrke/Tex/article/monads/thesis/src/mon/mon.elf
Solving for: |> *> +> ==
?- Q0 : {F} (t (t mu)) * (t (t (t (t (t (t F)))))) * (t eta) *> (H F).
Solving...
H = [F:mor X1 X] (eta * eta * (eta * (eta * (eta * F) ^) ^) ^ ^) ^,
Q0 =
   [F:mor X1 X]
      try (try (try (try (try (try (try
(try (try (try (try (try (try (try (try
(try (try (try (try (try (try (try (try last (step^ (step*r (simple ^eta))))
                                   (step^ (simple assr)))
                              (step^ (simple ^etaa)))
                         (simple ^*^))
                    (step*r (simple tdef)))
               (step*l
                   (step^
                       (step*r
                           (step^
                               (step*r
                                   (step^
                                       (step^
                                           (step*r
                                               (step^ (step*r (simple tdef))))))))))))
          (step*l
              (step^
                  (step*r
                      (step^ (step*r (step^ (step^ (step*r (simple tdef))))))))))
     (step*l (step^ (step*r (step^ (step*r (step^ (simple tdef))))))))
                                        (step*l
                                            (step^
                                                (step*r
                                                    (step^
                                                        (step*r
                                                            (step^ (simple idlr))))))))
                                   (step*l
                                       (step^
                                           (step*r
                                               (step^
                                                   (step*r
                                                       (step^ (simple ^etaa))))))))
                              (step*l
                                  (step^ (step*r (step^ (step*r (simple ^*^)))))))
                         (step*l
                             (step^
                                 (step*r (step^ (step*r (step*r (simple tdef))))))))
                    (step*l (step^ (step*r (step^ (simple assr))))))
               (step*l (step^ (step*r (step^ (simple ^etaa))))))
```

```
          (step*l (step^ (step*r (simple ^*^)))))
     (step*l (step^ (step*r (step*r (simple tdef))))))
                                         (step*l (step^ (simple assr))))
                                    (step*l (step^ (simple ^etaa))))
                               (simple ^*^a))
                          (step*r (step*l (simple tdef))))
                     (step*l (step^ (step*r (step^ (step*r (simple mudef)))))))
                (step*l (step^ (step*r (simple tdef)))))
          (step*l (simple tdef)).
yes
?- {F} (t (t mu)) * (t (t (t (t (t (t F)))))) * (t eta) ==
(t (t (t (t (t F))))) * (t (t mu)) * (t eta).
   Solving...
solved
yes
?- {F} (t (t mu)) * (t (t (t (t (t (t F)))))) * (t eta) ==
(t (t mu)) * (t eta) * (t (t (t (t (t F))))).
   Solving...
solved
yes
?- val it = () : unit
-
```

C.3.3 Computational Co-monad

```
- top();
Using: /home/wgehrke/Tex/article/monads/thesis/src/ccm/ccm.elf
Solving for: |> *> +> ==
?- Q0 : {F}{G} F * G == G * F.
Solving...
no
?- Q1 : {F} (cki ck* F) == F.
Solving...
Q1 =
   [F:mor (t´ X1) X]
      moreq last
         (try (try (try (try (try (try (try (try last (simple idrr))
                                            (step*r (simple eps^)))
                                       (step*r (step*r (simple deldef))))
                                  (simple assr))
                             (simple eps^a))
                        (step*r (step*l (simple tdef))))
                   (step*l (simple ckid)))
              (simple ck*d)).
yes
?- Q2 : {F} (F ck* cki) == F.
Solving...
Q2 =
   [F:mor (t´ X1) X]
      moreq last
         (try (try (try (try (try (try (try (try (try (try last (simple idrr))
                                                       (step*r (simple %eps^%)))
                                                  (step*r (step^ (simple idrr))))
                                             (step*r
                                                 (step^ (step*r (simple eps^)))))
                                        (step*r (step^ (simple assr))))
                                   (step*r (simple ^*^)))
                              (step*r (step*r (simple deldef))))
                         (step*r (step*l (step^ (step*l (simple ckid))))))
                    (step*r (step*l (simple tdef))))
              (simple ck*d)).
yes
?- Q3 : {F}{G}{H} ((F ck* G) ck* H) == (F ck* (G ck* H)).
Solving...
[Major collection... 69% used (2948848/4262988), 1790 msec]
[Increasing heap to 9616k]
Q3 =
   [F:mor (t´ X1) X] [G:mor (t´ (X2 F)) X1] [H:mor (t´ (X3 F G)) (X2 F)]
      moreq
         (try (try (try (try (try (try (
try (try (try (try (try (try (try (
```

```
try (try (try last (step*r (step^ (step*r (simple idrr)))))
         (step*r (step^ (step*r (step*r (simple eps^))))))
    (step*r (step^ (step*r (simple assr)))))
                                       (step*r (step^ (simple assr))))
                                  (step*r (simple ^*^)))
                             (step*r (step*r (simple deldef))))
                        (step*r
                            (step*l
                                (step^ (step*r (step*l (step^ (simple idrr)))))))
                   (step*r
                       (step*l
                           (step^
                               (step*r (step*l (step^ (step*r (simple eps^)))))))))
              (step*r (step*l (step^ (step*r (step*l (step^ (simple assr)))))))
         (step*r (step*l (step^ (step*r (simple ^*^a))))))
    (step*r (step*l (step^ (step*r (step*r (step*l (simple deldef)))))))
                                       (step*r
                                           (step*l
                                               (step^
                                                   (step*r
                                                       (step*l (simple tdef)))))))
                                  (step*r
                                      (step*l (step^ (step*r (simple assr))))))
                             (step*r (step*l (step^ (simple assr)))))
                        (step*r (step*l (step^ (step*l (simple ck*d))))))
                   (step*r (step*l (simple tdef))))
              (simple ck*d))
         (try (try (try (try (try (try (
try (try (try (try (try (try (try (try (
try (try (try (try (try last (step*r (step^ (step*r (step^ (simple idrr))))))
                   (step*r (step^ (step*r (step^ (step*r (simple eps^)))))))
              (step*r (step^ (step*r (step^ (simple assr))))))
         (step*r (step^ (step*r (simple ^*^)))))
    (step*r (step^ (simple assr))))
                                       (step*r (simple ^*^)))
                                  (step*r (step*r (simple deldef))))
                             (step*r (simple ^*^a)))
                        (step*r (step*r (step*l (simple tdef)))))
                   (step*r (step*l (step^ (simple idrr)))))
              (step*r (step*l (step^ (step*r (simple eps^))))))
         (step*r (step*l (step^ (simple assr)))))
    (step*r (simple ^*^a)))
                                       (step*r (step*r (step*l (simple deldef)))))
                                  (step*r (step*l (simple tdef))))
                             (step*r (simple assr)))
                        (simple assr))
                   (step*l (simple ck*d)))
              (simple ck*d)).
yes
?- (alg id) == cki.
Solving...
solved
yes
?- {F}{G} (alg (F * G)) == (alg F) ck* (alg G).
Solving...
solved
yes
?- (fun cki) == id.
Solving...
solved
yes
?- {F}{G} (fun (F ck* G)) == (fun F) * (fun G).
Solving...
solved
yes
?- {F} (fun (alg F)) == F.
Solving...
solved
yes
?- {F} (fun (alg (fun F))) == (fun F).
Solving...
solved
```

```
yes
?- val it = () : unit
-
```