Rewriting Techniques applied to Basic Category Theory

Wolfgang Gehrke[1]

Research Institute for Symbolic Computation
Johannes Kepler University
A – 4040 Linz, AUSTRIA
Wolfgang.Gehrke@risc.uni-linz.ac.at

July 17, 1994

[1]sponsored by the Austrian Science Foundation (FWF), ESPRIT BRP 6471 MEDLAR II

Abstract

In this report canonical rewriting systems for the most basic categorical notions are presented. These are categories themselves, functors, and natural transformations. The appendix explains the methods used to prove termination and confluence of the discussed systems. Finally traces of tests with the Larch Prover and a program in Elf illustrating a more faithful implementation are included.

Introduction

Category theory can be seen from very different points of view: as an abstract *theory of functions*, as an abstract *theory of deductions*, as an abstract *theory of structures and structure preserving mappings*. The importance of this field is stressed by the fact that it became relevant for the foundations of mathematics in giving a new ground to formalize mathematics (see [AHS90, HS79, HP91, ML71, MLM92]).

Checking commutativity of diagrams in category theory and other mathematical disciplines like homological algebra or algebraic topology is a task which arises permanently. This fact was the motivation to start our work with the aim to establish a system for automated commutativity-check of diagrams being well aware that we can meet hard decidability problems. Consequently we will concentrate on certain tractable problems. Diagrams in this context are a pictorial encoding of equations (in [FS90] one can even find a special graphical language). Thus the check for commutativity amounts to the decision whether the composition of a given sequence of morphisms (corresponding to a path in the diagram) equals another one. Since term rewriting is a powerful method for equational reasoning it is very natural to exploit it in this context.

The techniques from term rewriting (cf. [DJ90, Klo92] for an overview) are applied to the most simple categorical notions such that canonical systems can be achieved for the equational specifications of these notions. All the techniques used are explained more detailed in the appendix. All the canonical systems were checked with the "Larch Prover" ([GG91]) where the traces are included in the appendix, too.

Finally the difference between ordinary untyped rewriting and typed rewriting is pointed out which is important in the context of categories. This gives the starting point for a more faithful implementation where first steps have been done in the logic programming language Elf (see [Pfe89, HHP93, Pfe90]). A sample implementation can be found as the last part of the appendix.

The approach presented here is not the first attempt to provide methods for mechanizing category theory. First applications of rewriting techniques in category theory can be found in [Hue90a]. Another attempt towards implementing category theory is [RB88] but with a different goal which had more turning categorical concepts into algorithms in mind.

Chapter 1

Basic categorical notions

Basic categorical notions like category, functor, and natural transformation are discussed. Canonical systems are presented and proved where termination is shown with polynomial interpretations and confluence is shown by checking critical pairs. These methods used to show termination and confluence can be found in greater detail in the appendix. Finally the issue of the necessity to work with types representing morphisms is pointed out.

1.1 Categories

Remark 1.1 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 1.1 (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 codomain* $cod(m)$ *which pictorially is expressed as* $m : dom(m) \rightarrow cod(m)$.
- *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 $f : A \rightarrow B$*, and the composability of all the morphisms):*

$$f \circ_C id_A = f \tag{1.1}$$

$$id_B \circ_C f = f \tag{1.2}$$

$$(f \circ_C g) \circ_C h = f \circ_C (g \circ_C h) \tag{1.3}$$

Remark 1.2 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 $A \xrightarrow{f} B$. We will also make use of another way to represent composition which is done by the symbol ";" such that $f; g := g \circ f$. In "$f; g$" the order of application if more intuitive (at least from a computer scientist point of view). Both forms are in use. Furthermore that notation makes it easier to turn paths of diagrams into terms since one can simply follow the path in the same order and does not have to go "backwards".

Example 1.1 In the category of sets objects are sets and morphisms are set theoretic functions. The composition is the usual composition of functions. Identities are the identity functions.

Example 1.2 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 by itself.

Example 1.3 Partially ordered sets form a category where objects are partially ordered sets and morphisms are monotone mappings. The composition is the composition of mappings. Identities are the identity functions.

Discussion 1.1 In the following considerations the fundamental problem will always be to decide the equality of two morphisms in a given category. 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 codomain of the morphisms. The reader is assumed to be familiar with the basic notions of rewriting which are additionally summarized in the appendix.

Lemma 1.1 (CAT1) *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 follows from the Knuth-Bendix procedure.
ii) Termination follows from the following polynomial interpretation Pi_{cat1} over $(\mathbb{N}^+, <)$:

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

$\square$

Remark 1.3 Instead of grouping parenthesis to the right they also can be grouped to the left:

Lemma 1.2 (CAT2) *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 follows from the Knuth-Bendix procedure.
ii) Termination follows from the following polynomial interpretation Pi_{cat2} over $(\mathbb{N}^+, <)$:

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

□

Remark 1.4 One of the most popular references for category theory is [ML71] which provides lots of examples in greater detail than it is possible here.

1.2 Functors

Definition 1.1 (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 composability of morphisms):

$$F(id_A) = id_{F(A)} \tag{1.4}$$
$$F(f \circ_C g) = F(f) \circ_D F(g) \tag{1.5}$$

Remark 1.5 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 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:

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

Example 1.4 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* where the same category C is involved twice.

Example 1.5 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 1.6 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 1.3 (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 follows from the Knuth-Bendix procedure.
ii) Termination follows from the following polynomial interpretation Pi_{fun1} over $(\mathbb{N}^+, <)$:

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

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

□

Remark 1.6 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 1.4 (FUN2) *There is the following canonical system for a functor $F : C \to C'$:*

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

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

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

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

□

Remark 1.7 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.

1.3 Natural transformations

Definition 1.2 (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 $f : A \to B$):

$$\tau_B \circ_D F(f) \quad = \quad G(f) \circ_D \tau_A \tag{1.6}$$

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

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

Figure 1.1: Natural Transformation

Example 1.7 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 $(\nu \circ \tau)_A := \nu_A \circ \tau_A$ for all objects of A. Identities are the identity natural transformations $ID(A) := id_{F(A)}$ for the functor F.

Example 1.8 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 ring R and $Im_n(R), Un(R) : CommutativeRing \to Group$.

Example 1.9 A finite-dimensional vector space is naturally equivalent to its double dual. This can be seen from the natural equivalence κ for the functor $Id : FinVec \to FinVec$ and $(Dual \circ Dual) : FinVec \to FinVec$ where $FinVec$ is the category of finite-dimensional vector spaces.

Remark 1.8 The concept of a natural transformation can be nicely visualized by a diagram cf. figure 1.1.

Lemma 1.5 (NAT1) *There is the following canonical system for a natural transformation $\tau : F \to G$ where $F, G : C \to C'$:*

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

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

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

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

□

Remark 1.9 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 1.6 (NAT2) *There is the following canonical system for a natural transformation $\tau : F \to G$ where $F, G : C \to C'$:*

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

Proof.
i) Local confluence follows from the Knuth-Bendix procedure.

ii) Termination follows from the following polynomial interpretation Pi_{nat2} over $(\mathbb{N}^+, <)$:

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

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

□

Remark 1.10 The same system works for endofunctors. Furthermore the last two equations reversed by swapping the interpretations of F and G.

1.4 Concluding remarks

The actual check of all the polynomial interpretations presented here together with a test of all critical pairs has been done in the "Larch Prover" (see [GG91]). The traces can be checked in the appendix.

Nevertheless there is a subtle point here. Morphisms have been treated in a purely algebraic way without taking care of their types - i.e. domain and codomain. For example the only algebraic constant id represents an entire universe of constants id_A. But it was not necessary in these simple cases to consider the types.

In the cases of category, functor and natural transformation it is enough to check the compatibility of morphisms once before doing any kind of rewriting. Then all possible rewrites in the algebraic case coincide with the rewrites which are allowed in the typed case. A short comparison between untyped and typed terms is shown in table 1.1.

Real difficulties start when more complicated categorical constructions have to be considered. A *terminal object* in a category is an easy example. Such a terminal object 1 of a category C has the property that all morphisms $f : A \to 1$ coincide with a single distinguished morphism nil_A where an equational characterization would be: $f_{A\to 1} = nil_A$. But such an equation makes only sense in the typed case. In the untyped case a single variable on the left hand side is not allowed.

untyped term	typed term
$f * id$	$f_{A\to B} \circ id_A$
$id * f$	$id_B \circ f_{A\to B}$
$(f * g) * h$	$(f_{A3\to A4} \circ g_{A2\to A3}) \circ h_{A1\to A2}$

Table 1.1: typed versus untyped terms

typed term	Elf representation
$f_{A\to B} \circ id_A$	(id A) ; (f : mor A B)
$id_B \circ f_{A\to B}$	(f : mor A B) ; (id B)
$(f_{A3\to A4} \circ g_{A2\to A3}) \circ h_{A1\to A2}$	(h : mor A1 A2) ; ((g : mor A2 A3) ; (f : mor A3 A4))

Table 1.2: typed terms in Elf

Morphisms and their equality have to be encoded more faithfully than it could be done with the Larch Prover. As a comparison only the basic notion of a category was implemented in the logic programming language Elf. This program is included as the last part of the appendix.

In a certain way Elf can be seen as a kind of Prolog enriched by types and lambda binding (but without negation as finite failure). The choice of Elf has several advantages:

1. dependent types can be used to represent morphisms,
2. types also encode propositions via the "proposition as types principle",
3. terms encode entire proofs for further inspection,
4. several strategies for rewriting can be implemented easily.

In the trace of the Elf program at first the axiomatic characterization of the equality of morphisms is shown which is expressed by the predicate "=". This is only a static declaration which cannot be used for proof search since the rule "*symm*" makes a search fall into a loop. Secondly the dynamic predicate "==" is introduced which can be used for proof search. Thirdly soundness of "==" with respect to the axiomatic definition of "=" is shown as a kind of meta-theorem inside Elf. In fact the predicate "*sd*" gives a complete transformation of terms for "==" into terms for "=". One only has to check by visual inspection that all the possible constructors of "==" terms have a translation.

Finally some test queries are presented. These start with "?-". From the trace it can be seen that the Elf-interpreter is an image of sml (Standard ML of New Jersey) where some functions provide the interface to the interpreter. Usually first a small piece of ML code is loaded which takes care of loading the Elf program. The arguments of "initload" are a list of files for static definitions followed by a list of files with dynamic definitions.

A short comparison between the typed terms and their representation in Elf is given in table 1.2

Conclusion

Category theory can benefit from the application of methods used in term rewriting to solve problems which arise in the context of checking commutativity of diagrams.

For the simplest notions it is enough to treat the specification in a simple algebraic way without paying attention to the types involved. Here the Larch Prover is used to assist the construction of canonical systems.

In general it is not possible to forget the types of a morphism - its source and target. These can be nicely expressed in the logic programming language Elf which is also suitable for the implementation of various rewriting strategies.

Beyond the implementation of rewriting Elf can be used to generate critical pairs (but this is not shown in the appendix). To check the convergence every critical pair found by backtracking has to be transformed into a corresponding query. Unfortunately in the present stage it is not possible to formulate the entire Knuth-Bendix completion procedure in Elf. It would be highly desirable to have more support for equational reasoning build into the the interpreter for Elf.

Acknowledgments

My thanks to Jochen Pfalzgraf as my advisor for constantly encouraging me in this topic. Furthermore thanks to Hoon Hong for giving the first insights into methods from term rewriting. Finally, I especially like to thank Frank Pfenning for detailed comments on his programming language Elf.

Bibliography

[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.

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

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

[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.

[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*. MIT, 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.

[HP91] H. Herrlich and H.-E. Porst. *Category Theory at Work*. Number 18 in Research and Expositions in Mathematics. Heldermann-Verlag, Berlin, 1991.

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

[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.

[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.

[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.

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

[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.

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

[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.

[Pfe90] F. Pfenning. Logic Programming in the LF logical framework. In *[Hue90b]*, pages 149–181. Addison Wesley, 1990.

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

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

Appendix A

Review of facts from reduction relations

Remark A.1 To compare with the literature [Klo92] is the key reference especially at the beginning.

Notion A.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,

$\leftrightarrow$ *its symmetric closure,*

$\rightarrow^+$ *its transitive closure,*

$\rightarrow^*$ *its transitive reflexive closure,*

$\leftrightarrow^*$ *its symmetric, transitive, reflexive closure.*

Remark A.2 $(s,t) \in\, \rightarrow$ is also written as $s \rightarrow t$ and t is called a *one-step* $\rightarrow$ *reduct* of s. If $s \rightarrow^* t$ then t is called a *reduct* of s. Furthermore $s \in S$ is a *normal form* if there is no t with $s \rightarrow t$.

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

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

$\rightarrow_1 \cup \rightarrow_2$ *the union of $\rightarrow_1, \rightarrow_2$ and*

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

Definition A.1 (Church-Rosser, (local) Confluence, Termination) *A reduction relation $\rightarrow$ is called*

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

$$\begin{tikzcd}
s \arrow[rr, leftrightarrow, "*"] \arrow[dr, "*"'] & & s' \arrow[dl, "*"] \\
& u &
\end{tikzcd}
\qquad
\begin{tikzcd}
& s \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t \arrow[dr, "*"'] & & t' \arrow[dl, "*"] \\
& u &
\end{tikzcd}
\qquad
\begin{tikzcd}
& s \arrow[dl] \arrow[dr] & \\
t \arrow[dr, "*"'] & & t' \arrow[dl, "*"] \\
& u &
\end{tikzcd}$$

Figure A.1: Church-Rosser property, Confluence, Local Confluence

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* $s_1 \rightarrow s_2 \rightarrow \ldots$

Remark A.3 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. figure A.1. If two elements $s, t \in S$ have a common reduct $u \in S$, i.e. $s \rightarrow^* u \leftarrow^* t$ they are said to *converge* ($s \downarrow t$).

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

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

Example A.1 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 figure A.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).

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

$$\begin{tikzcd}
a & b \arrow[l] \arrow[r, shift left] & c \arrow[l, shift left] \arrow[r] & d
\end{tikzcd}$$

Figure A.2: Local confluence $\neq$ confluence

Proof. (idea cf. [New42])
$\Rightarrow$) obvious
$\Leftarrow$) Noetherian induction applied to $\rightarrow$ $\square$

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

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 $\check{s}$ $\square$

Remark A.4 Terminating and confluent reduction relations are also called *complete* or *canonical*.

Appendix B

Review of facts from term rewriting

Remark B.1 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 B.1 (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 B.2 0-ary function symbols are called *constants*. Sometimes the existence of at least one constant is assumed cf. [DJ90, page 249].

Notion B.2 (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 B.3 Terms which do not contain any variable are called *ground terms*.

Notion B.3 (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 B.4 $s =_E t$ means that there is an equation $\{s, t\} \in E$ (or $\{t, s\} \in E$) and $s \to_R t$ 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 $l \to_R r$ 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 B.4 (Substitution, Instantiation, Context) *Let $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ be a signature. Then*

$\sigma : \mathcal{V} \to \mathcal{T}$ *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,*

$C[t]$ *denotes the context C of the term $t \in \mathcal{T}$.*

Remark B.5 Substitutions acting on one variable $x \in \mathcal{V}$ only are also denoted as $[x := t]$. The *composition of two substitutions* σ and θ is denoted by concatenation $\sigma\theta$. A *renaming* substitution is a bijection $\rho : \mathcal{V} \to \mathcal{V} \subset \mathcal{T}$ which is a special kind of substitution.

Definition B.1 (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:*

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

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

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

Two terms s and d 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 $(\Sigma, E) \vdash s = t$).

Theorem B.1 (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 B.6 Birkhoff's theorem (see [Bir35]) states the equivalence between syntactic and semantic proving. $\mathcal{M}od(E) \models s \sim t$ means that the semantical equivalence $s \sim t$ is valid in all models of E. Although it is not possible to handle all models in a computer it is possible to do simple syntactic proofs.

Problem 1 *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 B.7 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 B.2 (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 $l \rightarrow_R r$ with $C[l^\sigma] \equiv s \longrightarrow_R t \equiv C[r^\sigma]$.*

Method 1 *Transform an equational specification into a term rewriting system*
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 B.8 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 B.9 Now a method for proving termination is presented. But it is undecidable in general whether a given term rewriting system terminates.

Definition B.3 (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 $x \prec x'$ implies $f_\phi(\cdots x \cdots) \prec f_\phi(\cdots x' \cdots)$.*

Proposition B.1 (Termination criterion) *A term rewriting system (Σ, R) where:*

1. *there is a well-founded set $(\mathcal{W}, \prec)$ and*
2. *there is a terminating function $\phi : \mathcal{T} \rightarrow \mathcal{W}$ 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 in the variables of that rule*

is terminating.

Remark B.10 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 2 *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 the set of terms $\mathcal{T}$ to the well-founded set $(\mathbb{N}^+, <)$. 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 B.1 Let $R = \{(x;y);z \to x;(y;z)\}$ over the signature with only one function symbol ";" having arity 2. Choose as the well-founded set $(\mathbb{N}^+, <)$. 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:

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

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

Remark B.11 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 B.12 Before the important notion of a critical pair can be introduced another technical notion is required: the most general unifier.

Definition B.4 (Unifier and most general unifier) A unifier *of two terms s and t is a substitution $\upsilon = \rho\sigma$ where:*

1. ρ *is a renaming substitution such that* $Var(s^\rho) \cap Var(t^\rho) = \emptyset$ *and*

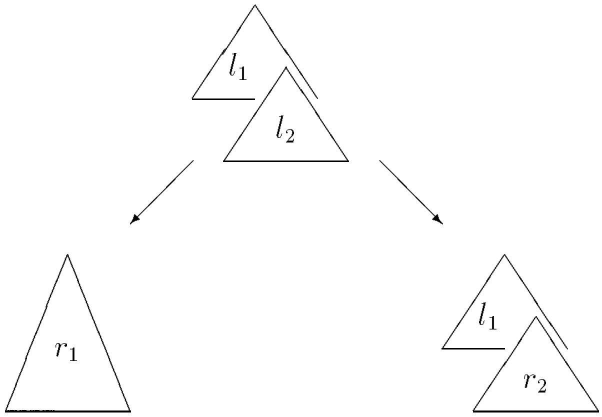

Figure B.1: Critical overlap

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 v where:*

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

Remark B.13 A most general unifier v 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 B.5 (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 $\sigma = mgu(st, l_2)$*

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

Remark B.14 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].

Proposition B.2 (Critical Pair Lemma) *A term rewriting system is locally confluent iff all critical pairs converge.*

Remark B.15 A *critical overlap* is shown in figure B.1 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 B.2 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:

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

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 what can be seen from:

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

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

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

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].

Appendix C

Traces from LP sessions

C.1 Category

C.1.1 System Cat1

```
Larch Prover (27 November 1991) logging on 21 June 1994 18:37:36 to
`/usr/fsys/jade/rz1c/wgehrke/rew/category/lp/cat1.lplog´.
LP2: execute cat1
LP2.1: declare sort M          % morphisms
LP2.2: declare variables f, g, h: M
LP2.3: declare operators
        id: -> M        % identity
        *: M,M -> M     % composition
..
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                  % ususal category
        f * id == f
        id * f == f
        (f * g) * h == f * (g * h)
..
Added 3 equations named user.1, ..., user.3 to the system.
The system now contains 3 rewrite rules.
LP2.8: complete
The system is complete.
LP2.9: display
Rewrite rules:
user.1: f * id -> f
user.2: id * f -> f
```

```
user.3: (f * g) * h -> f * (g * h)
End of input from file `/usr/fsys/jade/rz1c/wgehrke/rew/category/lp/cat1.lp´.
LP3: quit
```

C.1.2 System Cat2

```
Larch Prover (27 November 1991) logging on 21 June 1994 18:37:39 to
`/usr/fsys/jade/rz1c/wgehrke/rew/category/lp/cat2.lplog´.
LP2: execute cat2
LP2.1: declare sort M          % morphisms
LP2.2: declare variables f, g, h: M
LP2.3: declare operators
        id: -> M       % identity
        *: M,M -> M    % composition
..
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                  %  ususal category
        f * id == f
        id * f == f
        (f * g) * h == f * (g * h)
..
Added 3 equations named user.1, ..., user.3 to the system.
The system now contains 3 rewrite rules.
LP2.8: complete
The system is complete.
LP2.9: display
Rewrite rules:
user.1: f * id -> f
user.2: id * f -> f
user.3: f * (g * h) -> (f * g) * h
End of input from file `/usr/fsys/jade/rz1c/wgehrke/rew/category/lp/cat2.lp´.
LP3: quit
```

C.2 Functor

C.2.1 System Fun1

```
Larch Prover (27 November 1991) logging on 21 June 1994 19:23:50 to
`/usr/fsys/jade/rz1c/wgehrke/rew/functor/arbi/lp/fun1.lplog´.
```

```
LP2: execute fun1
LP2.1: declare sort M, M´      % morphisms
LP2.2: declare variables f, g, h: M
LP2.3: declare variables f´, g´, h´: M´
LP2.4: declare operators
        id: -> M        % identity in C
        id´: -> M´      % identity in C´
        t: M -> M´      % functor
        +: M,M -> M     % composition in C
        *: M´,M´ -> M´  % composition in C´
..
LP2.5: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.6: register polynomial     id      1
LP2.7: register polynomial     id´     1
LP2.8: register polynomial     t       2 * x
LP2.9: register polynomial     +       2 * x + y + 1
LP2.10: register polynomial     *       2 * x + y + 1
LP2.11: assert
        f + id == f     % category C
        id + f == f
        (f + g) + h == f + (g + h)
        f´ * id´ == f´  % category C´
        id´ * f´ == f´
        (f´ * g´) * h´ == f´ * (g´ * h´)
        t(id) == id´    % arbitrary functor
        t(f + g) == t(f) * t(g)
..
Added 8 equations named user.1, ..., user.8 to the system.
The system now contains 8 rewrite rules.
LP2.12: complete
The system is complete.
LP2.13: display
Rewrite rules:
user.1: f + id -> f
user.2: id + f -> f
user.3: (f + g) + h -> f + (g + h)
user.4: f´ * id´ -> f´
user.5: id´ * f´ -> f´
user.6: (f´ * g´) * h´ -> f´ * (g´ * h´)
user.7: t(id) -> id´
user.8: t(f + g) -> t(f) * t(g)
End of input from file
```

```
`/usr/fsys/jade/rz1c/wgehrke/rew/functor/arbi/lp/fun1.lp´.
LP3: quit
```

C.2.2 System Fun2

```
Larch Prover (27 November 1991) logging on 21 June 1994 19:23:54 to
`/usr/fsys/jade/rz1c/wgehrke/rew/functor/arbi/lp/fun2.lplog´.
LP2: execute fun2
LP2.1: declare sort M, M´      % morphisms
LP2.2: declare variables f, g, h: M
LP2.3: declare variables f´, g´, h´: M´
LP2.4: declare operators
        id: -> M        % identity in C
        id´: -> M´      % identity in C´
        t: M -> M´      % functor
        +: M,M -> M     % composition in C
        *: M´,M´ -> M´  % composition in C´
..
LP2.5: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.6: register polynomial     id      1
LP2.7: register polynomial     id´     1
LP2.8: register polynomial     t       x + 1
LP2.9: register polynomial     +       x * y + x
LP2.10: register polynomial     *       x * y + x
LP2.11: assert
        f + id == f     % category C
        id + f == f
        (f + g) + h == f + (g + h)
        f´ * id´ == f´  % category C´
        id´ * f´ == f´
        (f´ * g´) * h´ == f´ * (g´ * h´)
        t(id) == id´    % arbitrary functor
        t(f) * t(g) == t(f + g)
        t(f) * (t(g) * h´) == t(f + g) * h´
..
Added 9 equations named user.1, ..., user.9 to the system.
The system now contains 9 rewrite rules.
LP2.12: complete
The system is complete.
LP2.13: display
Rewrite rules:
user.1: f + id -> f
```

```
user.2: id + f -> f
user.3: (f + g) + h -> f + (g + h)
user.4: f´ * id´ -> f´
user.5: id´ * f´ -> f´
user.6: (f´ * g´) * h´ -> f´ * (g´ * h´)
user.7: t(id) -> id´
user.8: t(f) * t(g) -> t(f + g)
user.9: t(f) * (t(g) * h´) -> t(f + g) * h´
End of input from file
`/usr/fsys/jade/rz1c/wgehrke/rew/functor/arbi/lp/fun2.lp´.
LP3: quit
```

C.3 Natural Transformation

C.3.1 System Nat1

```
Larch Prover (27 November 1991) logging on 21 June 1994 20:29:05 to
`/usr/fsys/jade/rz1c/wgehrke/rew/nat_trafo/arbi/lp/nat1.lplog´.
LP2: execute nat1
LP2.1: declare sort M, M´      % morphisms
LP2.2: declare variables f, g, h: M
LP2.3: declare variables f´, g´, h´: M´
LP2.4: declare operators
        id: -> M        % identity in C
        id´: -> M´      % identity in C´
        n: -> M´        % natural transformation
        s,t: M -> M´    % functors
        +: M,M -> M     % composition in C
        *: M´,M´ -> M´  % composition in C´
..
LP2.5: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.6: register polynomial     id      1
LP2.7: register polynomial     id´     1
LP2.8: register polynomial     n       1
LP2.9: register polynomial     s       6 * x
LP2.10: register polynomial     t       2 * x
LP2.11: register polynomial     +       2 * x + y + 1
LP2.12: register polynomial     *       2 * x + y + 1
LP2.13: assert
        f + id == f     % category C
        id + f == f
```

```
        (f + g) + h == f + (g + h)
        f´ * id´ == f´  % category C´
        id´ * f´ == f´
        (f´ * g´) * h´ == f´ * (g´ * h´)
        s(id) == id´    %  arbitrary functors and natural transformation
        s(f + g) == s(f) * s(g)
        t(id) == id´
        t(f + g) == t(f) * t(g)
        n * s(f) == t(f) * n
        n * (s(f) * h´) == t(f) * (n * h´)
..
Added 12 equations named user.1, ..., user.12 to the system.
The system now contains 12 rewrite rules.
LP2.14: complete
The system is complete.
LP2.15: display
Rewrite rules:
user.1:  f + id -> f
user.2:  id + f -> f
user.3:  (f + g) + h -> f + (g + h)
user.4:  f´ * id´ -> f´
user.5:  id´ * f´ -> f´
user.6:  (f´ * g´) * h´ -> f´ * (g´ * h´)
user.7:  s(id) -> id´
user.8:  s(f + g) -> s(f) * s(g)
user.9:  t(id) -> id´
user.10: t(f + g) -> t(f) * t(g)
user.11: n * s(f) -> t(f) * n
user.12: n * (s(f) * h´) -> t(f) * (n * h´)
End of input from file
`/usr/fsys/jade/rz1c/wgehrke/rew/nat_trafo/arbi/lp/nat1.lp´.
LP3: quit
```

C.3.2 System Nat2

```
Larch Prover (27 November 1991) logging on 21 June 1994 20:29:11 to
`/usr/fsys/jade/rz1c/wgehrke/rew/nat_trafo/arbi/lp/nat2.lplog´.
LP2: execute nat2
LP2.1: declare sort M, M´      % morphisms
LP2.2: declare variables f, g, h: M
LP2.3: declare variables f´, g´, h´: M´
LP2.4: declare operators
        id: -> M        % identity in C
```

```
        id´: -> M´      % identity in C´
        n: -> M´        % natural transformation
        s,t: M -> M´    % functors
        +: M,M -> M     % composition in C
        *: M´,M´ -> M´  % composition in C´
..
LP2.5: set ordering polynomial
The ordering-method is now `polynomial´.
LP2.6: register polynomial     id      1
LP2.7: register polynomial     id´     1
LP2.8: register polynomial     n       1
LP2.9: register polynomial     s       3 * x + 1
LP2.10: register polynomial     t       x + 1
LP2.11: register polynomial     +       x * y + x
LP2.12: register polynomial     *       x * y + x
LP2.13:
LP2.14: assert
        f + id == f     % category C
        id + f == f
        (f + g) + h == f + (g + h)
        id´ * f´ == f´  % category C´
        f´ * id´ == f´
        (f´ * g´) * h´ == f´ * (g´ * h´)
        s(id) == id´    % arbitrary functors and natural transformation
        s(f) * s(g) == s(f + g)
        s(f) * (s(g) * h´) == s(f + g) * h´
        t(id) == id´
        t(f) * t(g) == t(f + g)
        t(f) * (t(g) * h´) == t(f + g) * h´
        n * s(f) == t(f) * n
        n * (s(f) * h´) == t(f) * (n * h´)
..
Added 14 equations named user.1, ..., user.14 to the system.
The system now contains 14 rewrite rules.
LP2.15: complete
The system is complete.
LP2.16: display
Rewrite rules:
user.1:  f + id -> f
user.2:  id + f -> f
user.3:  (f + g) + h -> f + (g + h)
user.4:  id´ * f´ -> f´
user.5:  f´ * id´ -> f´
```

```
user.6:  (f´ * g´) * h´ -> f´ * (g´ * h´)
user.7:  s(id) -> id´
user.8:  s(f) * s(g) -> s(f + g)
user.9:  s(f) * (s(g) * h´) -> s(f + g) * h´
user.10: t(id) -> id´
user.11: t(f) * t(g) -> t(f + g)
user.12: t(f) * (t(g) * h´) -> t(f + g) * h´
user.13: n * s(f) -> t(f) * n
user.14: n * (s(f) * h´) -> t(f) * (n * h´)
End of input from file
`/usr/fsys/jade/rz1c/wgehrke/rew/nat_trafo/arbi/lp/nat2.lp´.
LP3: quit
```

Appendix D

Example in Elf

D.1 Static part of the program

```
%%% basic category theory

obj  : type. %name obj O

mor  : obj -> obj -> type. %name mor M

id   : {A:obj} mor A A.

;    : mor A B -> mor B C -> mor A C. %infix right 10 ;

=    : mor A B -> mor A B -> type. %infix none 8 = %name = Q

refl : F = F.
symm : F = G -> G = F.
tran : F = G -> G = H -> F = H. %infix left 8 tran

idl  : id A ; F = F.
idr  : F ; id A = F.
ass  : (F ; G) ; H = F ; (G ; H).

cong : F = G -> F´ = G´ -> (F ; F´) = (G ; G´).
```

D.2 Dynamic part of the program

D.2.1 Proof search for equality of morphisms

```
%%% a simple prover
```

```
==    : mor A B -> mor A B -> type.
%name == EQ
%infix none 8 ==

refl´ : F == F.

idl1  : id A ; F == G <- F == G.
idl2  : F == id A ; G <- F == G.
idr1  : F ; id A == G <- F == G.
idr2  : F == G ; id A <- F == G.

ass1  : (F1 ; F2) ; F3 == G
 <- F1 ; (F2 ; F3) == G.
ass2  : F == (G1 ; G2) ; G3
 <- F == G1 ; (G2 ; G3).

cong´ : F1 ; F2 == G1 ; G2
 <- F1 == G1
 <- F2 == G2.
```

D.2.2 Correctness of the dynamic "==" versus the static "="

```
%%% soundness of the prover

sd       : F == G -> F = G -> type.

sd_refl´ : sd (refl´) (refl).

sd_idl1  : sd (idl1 EQ1) (idl tran Q1)
    <- sd EQ1 Q1.
sd_idl2  : sd (idl2 EQ2) (Q2 tran (symm idl))
    <- sd EQ2 Q2.
sd_idr1  : sd (idr1 EQ1) (idr tran Q1)
    <- sd EQ1 Q1.
sd_idr2  : sd (idr2 EQ2) (Q2 tran (symm idr))
    <- sd EQ2 Q2.

sd_ass1  : sd (ass1 EQ1) (ass tran Q1 : (F1 ; F2) ; F3 = G)
    <- sd EQ1 (Q1 : F1 ; (F2 ; F3) = G).
sd_ass2  : sd (ass2 EQ2) (Q2 tran (symm ass))
    <- sd EQ2 (Q2 : F = G1 ; (G2 ; G3)).
```

```
sd_cong´ : sd (cong´ EQ2 EQ1) (cong Q1 Q2)
    <- sd EQ1 (Q1 : F1 = G1)
    <- sd EQ2 (Q2 : F2 = G2).
```

D.3 Loading and some queries

```
% cat load.sml
(* loading procedure in ML code *)

fun load () = initload ["category.elf"] ["prover.elf" , "sound.elf"];

% elf-tools
Standard ML of New Jersey, Version 0.93, February 15, 1993
Elf, Version 0.4, July 1, 1993, saved on Mon Jul 19 20:43:54 MET DST 1993
val it = () : unit
- use "load.sml";
[opening load.sml]
val load = fn : unit -> unit
val it = () : unit
- load();
[reading file category.elf]
obj : type.
%%name obj  O
mor : obj -> obj -> type.
%%name mor  M
id : {A:obj} mor A A.
; : {A:obj} {B:obj} {C:obj} mor A B -> mor B C -> mor A C.
%%infix right 10 ;
= : {A:obj} {B:obj} mor A B -> mor A B -> type.
%%infix none 8 =
refl : {O:obj} {O1:obj} {F:mor O O1} = O O1 F F.
symm : {O:obj} {O1:obj} {F:mor O O1} {G:mor O O1} = O O1 F G -> = O O1 G F.
tran :
   {O:obj} {O1:obj} {F:mor O O1} {G:mor O O1} {H:mor O O1}
      = O O1 F G -> = O O1 G H -> = O O1 F H.
%%infix left 8 tran
idl : {A:obj} {O:obj} {F:mor A O} = A O (; A A O (id A) F) F.
idr : {O:obj} {A:obj} {F:mor O A} = O A (; O A A F (id A)) F.
ass :
   {O:obj} {O1:obj} {O2:obj} {O3:obj} {F:mor O O3} {G:mor O3 O2} {H:mor O2 O1}
      = O O1 (; O O2 O1 (; O O3 O2 F G) H) (; O O3 O1 F (; O3 O2 O1 G H)).
```

```
cong :
   {O:obj} {O1:obj} {F:mor O O1} {G:mor O O1} {O2:obj} {F´:mor O1 O2}
      {G´:mor O1 O2}
      = O O1 F G -> = O1 O2 F´ G´ -> = O O2 (; O O1 O2 F F´) (; O O1 O2 G G´).
[closed file category.elf]
[reading file prover.elf]
== : {A:obj} {B:obj} mor A B -> mor A B -> type.
%%name ==  EQ
%%infix none 8 ==
refl´ : {O:obj} {O1:obj} {F:mor O O1} == O O1 F F.
idl1 :
   {A:obj} {O:obj} {F:mor A O} {G:mor A O}
      == A O F G -> == A O (; A A O (id A) F) G.
idl2 :
   {A:obj} {O:obj} {F:mor A O} {G:mor A O}
      == A O F G -> == A O F (; A A O (id A) G).
idr1 :
   {O:obj} {A:obj} {F:mor O A} {G:mor O A}
      == O A F G -> == O A (; O A A F (id A)) G.
idr2 :
   {O:obj} {A:obj} {F:mor O A} {G:mor O A}
      == O A F G -> == O A F (; O A A G (id A)).
ass1 :
   {O:obj} {O1:obj} {O2:obj} {F1:mor O O2} {O3:obj} {F2:mor O2 O3}
      {F3:mor O3 O1} {G:mor O O1}
      == O O1 (; O O2 O1 F1 (; O2 O3 O1 F2 F3)) G
         -> == O O1 (; O O3 O1 (; O O2 O3 F1 F2) F3) G.
ass2 :
   {O:obj} {O1:obj} {F:mor O O1} {O2:obj} {G1:mor O O2} {O3:obj} {G2:mor O2 O3}
      {G3:mor O3 O1}
      == O O1 F (; O O2 O1 G1 (; O2 O3 O1 G2 G3))
         -> == O O1 F (; O O3 O1 (; O O2 O3 G1 G2) G3).
cong´ :
   {O:obj} {O1:obj} {F2:mor O O1} {G2:mor O O1} {O2:obj} {F1:mor O2 O}
      {G1:mor O2 O}
      == O O1 F2 G2 -> == O2 O F1 G1
         -> == O2 O1 (; O2 O O1 F1 F2) (; O2 O O1 G1 G2).
[closed file prover.elf]
[reading file sound.elf]
sd :
   {O:obj} {O1:obj} {F:mor O O1} {G:mor O O1} == O O1 F G -> = O O1 F G -> type.
sd_refl´ :
   {O:obj} {O1:obj} {M:mor O O1} sd O O1 M M (refl´ O O1 M) (refl O O1 M).
```

```
sd_idl1 :
   {O:obj} {O1:obj} {M:mor O O1} {M1:mor O O1} {EQ1:== O O1 M M1}
      {Q1:= O O1 M M1}
      sd O O1 M M1 EQ1 Q1
         -> sd O O1 (; O O O1 (id O) M) M1 (idl1 O O1 M M1 EQ1)
               (tran O O1 (; O O O1 (id O) M) M M1 (idl O O1 M) Q1).
sd_idl2 :
   {O:obj} {O1:obj} {M:mor O O1} {M1:mor O O1} {EQ2:== O O1 M M1}
      {Q2:= O O1 M M1}
      sd O O1 M M1 EQ2 Q2
         -> sd O O1 M (; O O O1 (id O) M1) (idl2 O O1 M M1 EQ2)
               (tran O O1 M M1 (; O O O1 (id O) M1) Q2
                   (symm O O1 (; O O O1 (id O) M1) M1 (idl O O1 M1))).
sd_idr1 :
   {O:obj} {O1:obj} {M:mor O O1} {M1:mor O O1} {EQ1:== O O1 M M1}
      {Q1:= O O1 M M1}
      sd O O1 M M1 EQ1 Q1
         -> sd O O1 (; O O1 O1 M (id O1)) M1 (idr1 O O1 M M1 EQ1)
               (tran O O1 (; O O1 O1 M (id O1)) M M1 (idr O O1 M) Q1).
sd_idr2 :
   {O:obj} {O1:obj} {M:mor O O1} {M1:mor O O1} {EQ2:== O O1 M M1}
      {Q2:= O O1 M M1}
      sd O O1 M M1 EQ2 Q2
         -> sd O O1 M (; O O1 O1 M1 (id O1)) (idr2 O O1 M M1 EQ2)
               (tran O O1 M M1 (; O O1 O1 M1 (id O1)) Q2
                   (symm O O1 (; O O1 O1 M1 (id O1)) M1 (idr O O1 M1))).
sd_ass1 :
   {O:obj} {O1:obj} {O2:obj} {F1:mor O O2} {O3:obj} {F2:mor O2 O3}
      {F3:mor O3 O1} {G:mor O O1}
      {EQ1:== O O1 (; O O2 O1 F1 (; O2 O3 O1 F2 F3)) G}
      {Q1:= O O1 (; O O2 O1 F1 (; O2 O3 O1 F2 F3)) G}
      sd O O1 (; O O2 O1 F1 (; O2 O3 O1 F2 F3)) G EQ1 Q1
         -> sd O O1 (; O O3 O1 (; O O2 O3 F1 F2) F3) G
               (ass1 O O1 O2 F1 O3 F2 F3 G EQ1)
               (tran O O1 (; O O3 O1 (; O O2 O3 F1 F2) F3)
                   (; O O2 O1 F1 (; O2 O3 O1 F2 F3)) G (ass O O1 O3 O2 F1 F2 F3)
                   Q1).
sd_ass2 :
   {O:obj} {O1:obj} {F:mor O O1} {O2:obj} {G1:mor O O2} {O3:obj} {G2:mor O2 O3}
      {G3:mor O3 O1} {EQ2:== O O1 F (; O O2 O1 G1 (; O2 O3 O1 G2 G3))}
      {Q2:= O O1 F (; O O2 O1 G1 (; O2 O3 O1 G2 G3))}
      sd O O1 F (; O O2 O1 G1 (; O2 O3 O1 G2 G3)) EQ2 Q2
         -> sd O O1 F (; O O3 O1 (; O O2 O3 G1 G2) G3)
```

```
               (ass2 O O1 F O2 G1 O3 G2 G3 EQ2)
               (tran O O1 F (; O O2 O1 G1 (; O2 O3 O1 G2 G3))
                   (; O O3 O1 (; O O2 O3 G1 G2) G3) Q2
                   (symm O O1 (; O O3 O1 (; O O2 O3 G1 G2) G3)
                       (; O O2 O1 G1 (; O2 O3 O1 G2 G3))
                       (ass O O1 O3 O2 G1 G2 G3))).
sd_cong´ :
   {O:obj} {O1:obj} {F2:mor O O1} {G2:mor O O1} {EQ2:== O O1 F2 G2}
      {Q2:= O O1 F2 G2} {O2:obj} {F1:mor O2 O} {G1:mor O2 O} {EQ1:== O2 O F1 G1}
      {Q1:= O2 O F1 G1}
      sd O O1 F2 G2 EQ2 Q2 -> sd O2 O F1 G1 EQ1 Q1
         -> sd O2 O1 (; O2 O O1 F1 F2) (; O2 O O1 G1 G2)
               (cong´ O O1 F2 G2 O2 F1 G1 EQ2 EQ1)
               (cong O2 O F1 G1 O1 F2 G2 Q1 Q2).
[closed file sound.elf]
category.elf --- 4 --- static
prover.elf --- 5 --- dynamic
sound.elf --- 6 --- dynamic
val it = () : unit
- top();
Using: prover.elf sound.elf
Solving for: == sd
?- Q1 : {F}{G}{H} ((F ; G) ; (H ; id A) == F ; G ; H).
Solving...

A = A1,
Q1 =
   [F:mor O1 O] [G:mor O (O2 F)] [H:mor (O2 F) A1]
      ass1 (cong´ (cong´ (idr1 refl´) refl´) refl´).
;
no more solutions
?- Q2 : {F : mor A B}{G : mor B C}{H : mor C D}
     ((F ; G) ; (H ; id D) == F ; G ; H).
   Solving...

D = D1,
C = C1,
B = B1,
A = A1,
Q2 =
   [F:mor A1 B1] [G:mor B1 C1] [H:mor C1 D1]
      ass1 (cong´ (cong´ (idr1 refl´) refl´) refl´).
;
```

```
no more solutions
?- Q3 : {F}{G}{H} ((F ; G) ; (id A ; H) == F ; G ; H).
Solving...

A = A1,
Q3 =
   [F:mor O1 O] [G:mor O A1] [H:mor A1 (O2 F G)]
      ass1 (cong´ (cong´ (idl1 refl´) refl´) refl´).
;
no more solutions
?- Q4 : {F}{G} (F ; G == G ; F).
Solving...
no
?- Q5 : sd (ass1 (cong´ (cong´ (idr1 refl´) refl´) refl´)) Q.
Solving...

Q = ass tran cong refl (cong refl (idr tran refl)),
Q5 = sd_ass1 (sd_cong´ (sd_cong´ (sd_idr1 sd_refl´) sd_refl´) sd_refl´).
;
no more solutions
?- (ass tran cong refl (cong refl (idr tran refl))) : Q6.
Solving...

Q6 = (M ; M1) ; M2 ; id O = M ; M1 ; M2.

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