Formal Software Development: From Foundations to Tools

Exposé der schriftlichen Habilitationsleistung

Christoph Lüth

FB 3 — Informatik und Mathematik
Universität Bremen

Abstract

This exposé gives an overview of the author's contributions to the area of formal software development. These range from foundational issues dealing with abstract models of computation to practical engineering issues concerned with tool integration and user interface design. We can distinguish three lines of work:

Firstly, there is foundational work, centred around categorical models of rewriting. A new semantics for rewriting is developed, which abstracts over the concrete term structure while still being able to express key concepts such as variable, layer and substitution. It is based on the concept of a monad, which is well-known in category theory to model algebraic theories. We generalise this treatment to term rewriting systems, infinitary terms, term graphs, and other forms of rewriting. The semantics finds applications in functional programming, where monads are used to model computational features such as state, exceptions and I/O, and modularity proofs, where the layer structure becomes central.

Secondly, we are concerned with formal proof and development, where we understand 'formal' as in formal logic (i.e. modelled inside a theorem prover.) The main contribution here are techniques for systematic generalisation of developments and proofs, called abstraction for reuse. We develop abstraction procedures for any logical framework, and implement them in Isabelle, combining proof term transformation known from type theory with tactical theorem proving. We further show how to model transformational development in-the-small in the TAS system by modelling window inferencing in Isabelle, and model development in-the-large by theory morphisms implemented via proof term transformation.

Thirdly, we turn to tool development. We discuss suitable system architectures to implement formal methods tools, or integrate existing tools into a common framework. We consider user interfaces, introducing the design of a graphical user interface for the transformational development system TAS based on principles of direct manipulation, and showing how to combine the graphical user interface with a textual one based on the popular Emacs editor.

Contents

Chapter 1

Introduction

The development of correct software is one of the core problems of computer science, and over the years numerous approaches have been developed. In this exposé, we are concerned with approaches which apply mathematical methods and theories to the development of software, collectively known as *formal methods*.

Formal methods have been around for a while. Early beginnings can be traced back to Turing [156] and von Neumann [57] in the dawn of computing,[1] and the seminal work by Hoare [69] and Dijkstra [41] is by now over thirty years in the past. So why has the formal derivation and verification of programs not become common industrial practice today? Has the long history of formal methods not shown this up as a pipe dream? Of course not — the history of formal methods has in fact been a long evolution always edging closer to the goal of being able to completely formalise the software development process rather than a single revolutionary breakthrough. It was a bold vision thirty years ago; twenty years ago, the first pioneering systems such as CIP [19] appeared; ten years ago theorem provers such as PVS [116], Isabelle [112], HOL [59] or Coq [22] became mature, stable and powerful tools, five years ago model-checking became industrially viable, and today, mass-marketed microprocessors are verified formally, and semi-formal methods such as UML have become state-of-the-art, showing that indeed there is an awareness and need for formal specification and development.

Thus, today the mathematics for formally developing sequential algorithms is well-known, automatic theorem proving has become powerful enough to tackle nearly all 'trivial' or 'obvious' proofs, and software development is slowly inching towards industrial usability. What is still needed to help it there?

One of the key challenges of formal software development is *scalability*. Realistic software tends to be huge, and this is a challenge because mathematics does not prepare us for this. Traditionally, mathematics is concerned

[1] Jones [79] gives a comprehensive account of the early history of formal methods.

with proving single theorems, not with structured reasoning; thus, while the reasoning in-the-small is quite formal, the structure of a mathematics text, and in particular the interrelation between different texts (such as results from one being used in the other), are very much left implicit in the text, and not formalised explicitly; in fact, the scalability problem arises for efforts to formalise reasonable parts of mathematics just as it does for software. Apart from that, rather more down-to-earth engineering problems (what is a good system architecture for tools to support formal development, how can we combine these tools, what should a good user interface look like, etc.) need to be tackled in a systematic way. If we want formal program development to become an engineering effort, surely the tools should themselves be well engineered!

The work as summarised in this exposé is concerned with these areas of formal software development. With the state of the art quite advanced, as outlined above, there is no single breakthrough to be expected, but rather slow advances in all areas. Hence, the main body of this exposé comprises three chapters, concerned with *categorical models of rewriting*, *formal proof and development*, and *tool development*.

Categorical Models of Rewriting

We develop a model for rewriting (or reduction) which generalises previously known approaches to rewriting, and has applications from automated theorem proving over denotational semantics to functional programming. This new model is based on the concept of a *monad* as known from category theory. Monads are well-known to model algebraic theories (i.e. equational logic and universal algebra), so we generalise this to a treatment of directed equations.

The crucial properties of our semantics are *compositionality* and *modularity*. The first means that it interacts well with structuring operations: the semantics of a composed system can be given in terms of the semantics of the components, and need not be recalculated from scratch. The second means that reasoning about the combined system can be broken down into reasoning about the components. Both together allow us to deal with large systems: while composing them, compositionality allows us to handle the complexity, and modularity makes reasoning about large systems feasible.

Formal Proof and Software Development

Theorem proving has come in leaps and bounds over the last fifteen years. Today, higher-order theorem provers are powerful enough to cover nearly all areas of mathematics. (Even though only a very tiny amount of mathematics has been formalised yet — formalising a published proof, even one published in a textbook, is still a large step, and will more often than not uncover small

inaccuracies or even errors — this has now become a problem of quantity, rather than quality.) We use the theorem prover Isabelle as the basis for our work, because it is powerful, generic, and allows easy but safe extension.

In formal development, reuse is a necessity, because one rarely gets a specification right the first time, and formal proof is far too hard work to start from scratch every time. In fact, changing the specification is the rule rather than the exception, and so the ability to reuse as much of the proof work is necessary. And there is an even greater potential gain: if we had a systematic way to abstract a given development to make it more general, we could systematically reuse developments on a large scale, building up a library of readily usable, formally proven rules or patterns. The potential usefulness of is demonstrated by the recent emergence of (non-formal) development patterns [52] in the OO community.

We develop *inter alia* an abstraction procedure for Isabelle, which allows to generalise proofs from a given context, and reuse them elsewhere. This is based on the technique of proof term transformation, and can also be used to implement notions of theory morphisms leading to transformations in-the-large. We further show how to encode a calculus for transformational proof, namely *window inferencing*, into Isabelle, which gives a more transformation style of reasoning and development.

Tool Development

The last chapter is concerned with the engineering of tools for formal development. The main focus is tool integration and user interfaces. For tool integration, we have had good experiences with loosely coupled tools communicating in a light-weight protocol such as XML. This strikes a good balance between being easy to implement on the one hand, and on the other hand the well-formedness conditions of the XML documents guaranteeing some consistency, in particular when we use a type-preserving embedding of XML into a typed language such as Haskell [159].

Interface design has long since been neglected; for a long time, formal method tools had a 'one-user interface' (the one user being their implementor), and still command-line interfaces with an often arcane syntax dominate the scene. This need not be the case; we show how to design and implement graphical user interfaces for formal methods tools, and introduce a project to implement a generic interface, which combines the best of graphical and textual user interfaces, and can be connected to any prover by a well-specified interaction protocol.

How to read this Document

This document summarises the author's work over the last years. Due to its expository nature, we neither give proofs nor go into the specifics of an

implementation, but we give the appropriate references where these details may be found. We also do not necessarily follow the historical development of the work, but rather give a comprehensive overview. Further, for reasons of space we only briefly relate our work to others, but we endeavour to mark our specific contribution.

At the end of each chapter, a brief section by the title of 'Bibliographical Remarks' details the development of the author's contributions, when and where these were published, collaborations, research projects and so on. In particular, we remark on the specific contribution of the author in case of joint papers and research projects, if possible.

The author's works are referenced as in [L-3], and a full list of publications can be found on page 80ff. In the appendix, we provide copies of the relevant papers which appeared either in journal form or in refereed conferences. We do not provide copies of papers which have been superceded by later developments, e.g. journal articles.

Chapter 2

Categorical Models of Rewriting

This chapter deals with foundational issues, centered around our work on categorical models of rewriting. The notion of *rewriting* or *reduction* is a flexible notion of computation with a long history dating back to the beginnig of the 20^{th} century, when first rewriting systems were used to study the word problem in group theory. In the thirties, the emerging theory of algorithms developed the theory of rewriting further, with systems such as the λ-calculus, combinatory logic and Turing machines. In recent times, the combination of abstract algebra with rewriting known as term rewriting has become an important field of theoretical computer science, with applications to programming and automated theorem proving.

Despite this wide applicability, or probably precisely because it, there has been little meta-theory of rewriting. For example, the Knuth-Bendix completion of term rewriting system, and the construction of a Gröbner basis are clearly equivalent, but we lack a mathematical framework in which to express this. Even in the more narrowly defined field of term rewriting, there is no such unifying meta-theory; results are typically established and proven at the syntactic level, which means that for different forms of term rewriting results have to be reproven and they cannot benefit as much from each other as potentially possible. This holds in particular for the results we are most interested in, namely *modularity* results.

Our approach to this problem is to use *category theory* as the unifying framework, as it has proven a useful meta-theory in other branches of mathematics and computer science. Specifically, the categorical concept of a *monad* captures the notion of inductively generated data at a sufficiently abstract level, and will take center stage in our work. In the rest of this chapter, we will explore the theory of categorical rewriting, following the historic development: starting with a categorical treatment of term rewriting in Sect. 2.1, generalising gradually to other forms of rewriting, such as

infinitary term rewriting (Sect. 2.2), and finally arriving at a construction where the rewriting is independent of the data being rewritten in Sect. 2.3.

All of these semantics are based on the categorical concept of a monad. The key advantage of this semantics are *compositionality* and *modularity*, as we will explore in Sect. 2.4. We will show how monads can be combined, allowing us to combine the semantics of constituting parts (compositionality), and to reason about properties of the whole by reasoning about properties of the constituting parts (modularity). This will have applications in denotational semantics and functional programming as well.

2.1 Categorical Term Rewriting

Term rewriting systems (TRSs) combine the flexibility and generality of universal algebra with the computational notion of reduction. They originally arose in the study of the λ-calculus and combinatory logic, and more recently, have played a prominent rôle in the design and implementation of logical and functional programming languages, the theory of abstract data types, and automated theorem proving. They have also found applications in recursion theory and algebra, as they are a Turing-complete model of computation which can be used to provide decision procedures.

The concreteness of TRSs has lead to a tendency to concentrate on the syntactical details of specific problems to the detriment of a wider understanding of the subject. Abstract models of TRSs have been developed to address this problem with the simplest being *abstract reduction systems* based upon relations. While useful for certain results, they nevertheless lack sufficient structure to model key concepts such as substitution, context, redex, or layer structure.

Our approach starts from the observation that term rewriting systems can be regarded as generalisations of algebraic theories. Syntactically, algebraic theories declare term constructors and equations, while TRSs declare term constructors and rewrite rules, thus both are presented via signatures. Semantically, algebraic theories are modelled by *sets* of terms while TRSs are modelled by *pre-orders* of terms. Categorically, algebraic theories are modelled as *monads* on the category of sets and so it is natural to model a term rewriting system by a monad over a more structured base category.

2.1.1 A Category Theory Prelude

Since a comprehensive introduction into category theory is out of the scope of this text, we will assume a passing knowledge of the basic concepts such as categories, functors, natural transformations, limits and colimits, and adjunctions, which can be found e.g. in Chapter I–V of [97]. The following overview emphasises examples and motivation, giving basic definitions and

results while eliding the technical proofs which can be found in the references. Instead, we shall concentrate on where the categorical perspective gives new insights into rewriting.

However, we need some constructions which go beyond these basic concepts, namely finitariness, rank, local presentability, and Kan extensions. so we give brief definitions here; more details can be found in [97] or [5]. Readers may want to skip this section and refer back to here once they encounter the actual concepts in the text.

Finally, we put order structures on our hom-sets in the following, and thus deal with *enriched categories* [85]; we de-emphasise this level here, since all the constructions go through as usual, except where noted, if one takes care that morphisms preserve the order structure.

Finitary and Ranked Monads

Let κ be a regular cardinal. A diagram $\mathcal{D}$ is *κ-filtered* iff every subcategory with less than κ objects and morphisms has a compatible cocone in $\mathcal{D}$. A functor is κ-accessible iff it preserves κ-filtered colimits; we also say it has *rank* κ. A monad has rank κ if its action has. We call an $\aleph_0$-filtered diagram *filtered*, and a $\aleph_0$-accessible functor (monad) *finitary*. The category of finitary functors from $\mathcal{A}$ to $\mathcal{B}$ is denoted by $[\mathcal{A}, \mathcal{B}]_f$.

Given two monads $\mathsf{T} = \langle T, \eta, \mu \rangle$ and $\mathsf{S} = \langle S, \zeta, \xi \rangle$ on $\mathcal{C}$, a *monad morphism* is a natural transformation $\alpha : T \Rightarrow S$ between the actions commuting with unit and multiplication. The finitary monads on a category $\mathcal{C}$ and monad morphisms between them form a category $\mathbf{Mon}_f(\mathcal{C})$; similarly, monads with rank form a category $\mathbf{Mon}(\mathcal{C})$. Note that all monads do not form a category, for the obvious size problems.

Locally Presentable and Accessible Categories

An object X of a category $\mathcal{A}$ is said to be *κ-presentable* iff the hom-functor $\mathcal{A}(X, \_)$ preserves κ-filtered colimits. A category is *locally κ-presentable* (abbreviated as lκp) if it is cocomplete and has a set N_κ of κ-presentable objects such that every object is a κ-filtered colimit of objects from N_κ. The discrete category on N_κ is denoted $\mathcal{N}_\kappa$. The full subcategory of κ-presentable objects is denoted $\mathcal{A}_\kappa$. The inclusion functors are denoted $J_\kappa : \mathcal{N}_\kappa \longrightarrow \mathcal{A}_\kappa$ and $I_\kappa : \mathcal{A}_\kappa \longrightarrow \mathcal{A}$.

For the special case of $\kappa = \aleph_0$, we speak of *finitely presentable* objects, *locally finitely presentable* categories (abbreviated lfp), and allow the subscript $\aleph_0$ to be dropped. The subcategory of finitely presentable objects is denoted as $\mathcal{A}_f$.

Finite presentability is the categorical notion for finiteness. For example, for $\mathcal{A} = \mathbf{Set}$, the finitely presentable sets are precisely finite sets and the set N can be taken to be the natural numbers which we denote $\mathbb{N}$. Local finite

presentability is the categorical notion of being generated from a finite set of elements and finitely many equations; for example, every set is the filtered colimit of the diagram of all its finite subsets ordered by inclusion. A finitary functor, then, is one who preserves this property of being generated in the sense that its action on all objects is given by the action on the generating objects. For example, a functor $F : \mathbf{Set} \longrightarrow \mathbf{Set}$ is finitary if its action on infinite set X is isomorphic to the the colimit of its images on finite subsets X_0 of X, ordered under inclusion: $F(X) = \cup_{X_0 \subseteq X} F(X_0)$.

Kan Extensions

Given a functor $I : \mathcal{A} \longrightarrow \mathcal{B}$ and a category $\mathcal{C}$, precomposition with I defines a functor $\_ \circ I : [\mathcal{B}, \mathcal{C}] \longrightarrow [\mathcal{A}, \mathcal{C}]$. The problem of left and right Kan extensions is the problem of finding left and right adjoints to $\_ \circ I$. More concretely, given a functor $F : \mathcal{A} \longrightarrow \mathcal{C}$, the left and right Kan extensions satisfy the natural isomorphisms

$$[\mathcal{B}, \mathcal{C}](Lan_I F, H) \cong [\mathcal{A}, \mathcal{C}](F, H \circ I) \qquad [\mathcal{B}, \mathcal{C}](H, Ran_I F) \cong [\mathcal{A}, \mathcal{C}](H \circ I, F).$$

Thus, one can view the left and right Kan extension of F along I as the canonical extensions of F to $\mathcal{B}$. Kan extensions can be given pointwise using colimits and limits, or more elegantly using ends and coends (see [97] for details). Since we work over the complete and cocomplete categories **Set** and **Pre**, all our Kan extensions exist.

In fact, given a lfp category $\mathcal{A}$, a functor $F : \mathcal{A} \longrightarrow \mathcal{B}$ is finitary precisely if it is (isomorphic to) the left Kan extension of its restriction to $\mathcal{A}_f$ along the inclusion $A_f \longrightarrow \mathcal{A}$. Thus, we have an isomorphism of monoidal categories $[\mathcal{A}, \mathcal{A}]_f \cong [A_f, \mathcal{A}]$, with the composition of functors on the left corresponding to the product $G \diamond F = Lan_I G \cdot F$. In other words, we have a composition with finitariness built in.

2.1.2 Signatures, Algebraic Theories and Monads

Since our approach generalises the categorical treatment of universal algebra, where algebraic theories are equivalent to monads on the category of sets, we will give a self-contained presentation of this equivalence here (see also [100] or [97, Chapter VI]).

Definition 2.1 (Signature). *A (single-sorted)* signature *consists of a function* $\Sigma : \mathbb{N} \longrightarrow \boldsymbol{Set}$.

As a running example, we use the following signature for addition:

Example 2.2 (Addition). The signature $\Sigma_{Add} : \mathbb{N} \longrightarrow \mathbf{Set}$ for the theory of addition is defined as $\Sigma_{Add}(0) = \{0\}, \Sigma_{Add}(1) = \{S\}, \Sigma_{Add}(2) = \{+\}$ and $\Sigma_{Add}(n) = \emptyset$ for all other $n \in \mathbb{N}$. Thus Σ_{Add} declares one operation 0 of arity

0 (a constant), one unary operation S and one binary operation $+$ (written in infix notation).

Definition 2.3 (Term Algebra). *Given a signature Σ and a set of variables X, the* term algebra $T_\Sigma(X)$ *is defined inductively:*

$$\frac{x \in X}{\text{'}x \in T_\Sigma(X)} \qquad \frac{f \in \Sigma_n \quad t_1, \ldots t_n \in T_\Sigma(X)}{f\,(t_1, \ldots, t_n) \in T_\Sigma(X)}$$

Quotes are used to distinguish a variable $x \in X$ from the term $\text{'}x \in T_\Sigma(X)$. This will become important when analysing the layer structure on terms formed by the disjoint union of signatures.

The key idea of the categorical semantics is that the free term algebra construction, which for every set X gives us the set of terms $T_\Sigma(X)$, extends to a *functor* $T_\Sigma : \mathbf{Set} \longrightarrow \mathbf{Set}$ over the category **Set** of sets. Further, every variable $x \in X$ gives a term $\text{'}x \in T_\Sigma(X)$, so the variables induce a function $\eta_X : X \longrightarrow T_\Sigma(X)$. Lastly, substitution takes terms built over terms and flattens them, as described by a function $\mu_X : T_\Sigma(T_\Sigma(X)) \longrightarrow T_\Sigma(X)$. These three pieces of data define a monad:

Definition 2.4 (Monad). *A* monad $\mathsf{T} = \langle T, \eta, \mu \rangle$ *on a category $\mathcal{C}$ is given by an endofunctor $T : \mathcal{C} \longrightarrow \mathcal{C}$, called the* action*, and two natural transformations, $\eta : 1 \Rightarrow T$, called the* unit*, and $\mu : TT \Rightarrow T$, called the* multiplication *of the monad, satisfying the* monad laws*: $\mu \cdot T\eta = 1 = \mu \cdot \eta_T$, and $\mu \cdot T\mu = \mu \cdot \mu_T$.*

Lemma 2.5. *For every signature Σ, $\mathsf{T}_\Sigma = \langle T_\Sigma, \eta, \mu \rangle$ is a monad.*

The data from Def. 2.4 are actually enough to characterise term construction. The fact that the term algebra $T_\Sigma(X)$ is inductively defined is expressed categorically by the fact that $T_\Sigma(X)$ is the initial algebra of the functor $X + F_\Sigma$ which sends Y to $X + F_\Sigma Y$, where F_Σ is the polynomial endofunctor

$$F_\Sigma(Y) = \coprod_{n \in \mathbb{N}, f \in \Sigma(n)} Y^n = \coprod_{n \in \mathbb{N}} Y^n \times \Sigma(n) \tag{2.1}$$

The structure map $X + F_\Sigma T_\Sigma(X) \longrightarrow T_\Sigma(X)$ is equivalent to two maps $X \longrightarrow T_\Sigma(X)$ and $F_\Sigma T_\Sigma(X) \longrightarrow T_\Sigma(X)$ stating that all variables are contained in $T_\Sigma(X)$, and all operations can be interpreted in $T_\Sigma(X)$. Initiality of this algebra says that $T_\Sigma(X)$ is the smallest such set. For our running example, we have

$$F_{\Sigma_{Add}}(Y) = Y^0 + Y^1 + Y^2 = 1 + Y + Y^2$$

We can explicitly describe F_Σ as calculating the terms of depth one over a set of variables; for example,

$$F_{\Sigma_{Add}}(\{x, y\}) = \{0, Sx, Sy, x + x, x + y, y + x, y + y\}. \tag{2.2}$$

In summary, monads provide an abstract calculus for algebraic theories where *variables, substitution* and *term algebra* (represented by the unit, multiplication and action of the monad) are taken as primitive concepts.

2.1.3 Term Rewriting and Abstract Reduction Systems

This section reviews the basic concepts of term rewriting systems and abstract reduction systems. Further details may be found in [13, 89].

Definition 2.6 (Rewrite Rules and TRSs). *Given a signature Σ, a Σ-rewrite rule is of the form $Y \vdash l \to r$ where Y is a set of variables and $l, r \in T_\Sigma(Y)$. A* term rewriting system *$\mathcal{R} = \langle \Sigma, R \rangle$ consists of a signature Σ and a set R of Σ-rewrite rules.*

Usually one requires that l is not a variable and that the free variables of r are contained in the free variables of l, but semantically these restrictions are not necessary and hence omitted here. In the following examples, we denote the arity of an operation by subscripts; so $f \in \Sigma(n)$ is denoted by f_n. We also use infix notation for binary operators:

Example 2.7 (Combinatory Logic). The TRS $CL = \langle \Sigma_{CL}, R_{CL} \rangle$ is defined by

$$\begin{array}{rcl} \Sigma_{CL} & = & \{S_0, K_0, \cdot_2\}, \\ R_{CL} & = & \{\, \{x, y\} \vdash (K \cdot x) \cdot y \to x, \\ & & \quad \{x, y, z\} \vdash ((S \cdot x) \cdot y) \cdot z \to \,\} \end{array}$$

Example 2.8 (Addition). The TRS $Add = \langle \Sigma_{Add}, R_{Add} \rangle$ implements addition on the natural numbers, and is defined by

$$\begin{array}{rcl} \Sigma_{Add} & = & \{0_0, S_1, +_2\}, \\ R_{Add} & = & \{\, \{x\} \vdash 0 + x \to x, \\ & & \quad \{x, y\} \vdash Sx + y \to S(x + y)\} \end{array}$$

A term rewriting system $\mathcal{R}$ induces a *one-step reduction relation* $\to_{\mathcal{R}}$ on the terms. Formally, this is obtained by closing the rewrite rules under contexts and substitutions — for the exact definitions, see [89]. The *many-step reduction relation* $\twoheadrightarrow_{\mathcal{R}}$ is the reflexive-transitive closure of the one-step reduction relation. For example, the addition TRS supports the following rewrites:

$$\begin{array}{ccccc} S(0+0)+0 & \twoheadrightarrow & S((0+0)+0) & & \\ \downarrow & & \downarrow & & \\ S0+0 & \twoheadrightarrow & S(0+0) & \twoheadrightarrow & S0 \end{array}$$

Note that although the term $S(0 + 0) + 0$ can be reduced to two different terms, this choice is vacuous as both reduce to $S(0 + 0)$. Further, there are

no infinite reduction sequences starting from the term $S(0+0)+0$. These examples illustrate the two key properties of TRSs:

Definition 2.9 (Confluence and SN). *A term rewriting system* $\mathcal{R} = \langle \Sigma, R \rangle$ *is* confluent *iff* $\forall t, s_1, s_2.\, t \twoheadrightarrow_R s_1 \wedge t \twoheadrightarrow_R s_2\ \exists u.\, s_1 \twoheadrightarrow_R u \wedge s_2 \twoheadrightarrow_R u$. *It is* strongly normalising *(SN) iff there is no infinite sequence* $t_1 \rightarrow_R t_2 \rightarrow_R t_3 \rightarrow_R \ldots$.

Example 2.7 is confluent, but not strongly normalising while Example 2.8 is confluent and strongly normalising. This may not be apparent at first sight, and indeed in general these properties are undecidable. Consequently a variety of proof tools have been developed which are helpful in certain cases [68, 90, 114], but these methods are rather technical, often involving copious amounts of induction on the structure of terms, and tend to impose syntactic preconditions on term rewriting systems, such as left-linearity or orthogonality.

In order to remedy this situation, we turn to the *semantics* of term rewriting systems hoping to obtain a new ontology for rewriting. The simplest models of TRSs are the *abstract reduction systems* based upon relations; the advantages of this approach are its simplicity and versatility.

Definition 2.10 (Abstract Reduction Systems). *An abstract reduction system (ARS) is a pair* $\mathcal{A} = \langle A, \rightarrow_A \rangle$ *consisting of a carrier set* A *and a binary relation* $\rightarrow_A \subseteq A \times A$, *called the reduction or rewrite relation.*

For every term rewriting system $\mathcal{R} = \langle \Sigma, R \rangle$, the one-step reduction relation defines an ARS. Properties like confluence and strong normalisation can be (and usually are) defined on the level of ARSs. We can then prove lemmas and theorems about these properties at the level of relations, such as Newman's Lemma (if an ARS is locally confluent and strongly normalising then it is confluent), or the Hindley-Rosen Lemma (if two ARS are confluent and commute, then their union is confluent). These lemmas are formulated and proven at the level of relations. Nowhere do we need the fact that the carrier set consists of terms and subsequently, we cannot prove results by induction on the term structure. And while these lemmas are very useful, we cannot express or prove more sophisticated results. Hence, ARSs are mainly used as an organisational tool, while the difficult results are proven directly at the syntactic level. The very simplicity which is the advantage of relational models is also its main limitation. What is needed is a semantics at an intermediate level of abstraction between the actual syntax and the relational model.

2.1.4 Categorical Models of Term Rewriting Systems

The key observation underlying our semantics for TRSs is that TRSs can be viewed as generalised algebraic theories. This generalisation occurs at both

a syntactic and a semantic level. Syntactically, algebraic theories and TRSs are both presented via signatures — the former declares term constructors while TRSs declare term constructors and rewrite rules. Semantically, algebraic theories are modelled by sets with extra structure while TRSs are modelled by pre-orders with extra structure.

Given the categorical modelling of algebraic theories by monads on **Set** as presented in Sect. 2.1.2, it is natural to model a term rewriting system by a monad over a more structured base category. In this section we formalise these comments, first with a concrete construction and then with a more abstract, and in many ways more illuminating construction.

A Monadic Semantics for TRSs

We will now generalise the monadic semantics from algebraic theories to TRSs. Our aim is to model a TRS $\mathcal{R} = \langle \Sigma, R \rangle$ by a monad $\mathsf{T}_{\mathcal{R}}$.

First, we have to decide on the base category of the monad. Since the main focus of rewriting is to prove properties of the many-step reduction relation, we choose to use the category **Pre** of pre-orders and monotone functions. Other possibilities are the category **Cat** of all small categories and functors, if we were interested in labelled many-step reduction (i.e. distinguishing between different rewrites between the same two terms), or the category **Rel** of relations and monotone maps if one wanted to model the one-step reduction relation.

Next, we consider the action of the monad $\mathsf{T}_{\mathcal{R}}$. Recall that if Σ is an algebraic theory, then T_Σ maps a set of variables to the free algebra over it. In general, the action is an endofunctor, and variables and theory must belong to the same category which in our case is **Pre**. Thus, the action $T_{\mathcal{R}}$ should map a *pre-order* of variables P to the pre-order of terms and rewrites built over P. The order structure on the variables can be thought of as reduction assumptions: if $a \to b$, then whatever we substitute for a has to reduce to what we substitute for b.

Thus, we have two main differences from the traditional semantics: a TRSs is modelled by a particular kind of *function* rather than a relation; and we endow variables with an order structure. A third categorically inspired idea is that there is no fixed collection of variables, but that we have a term reduction algebra defined for *every* pre-order of variables.

Definition 2.11 (Term Reduction Algebra). *Given a term rewriting system $\mathcal{R} = \langle \Sigma, R \rangle$ and a pre-order X, the* term reduction algebra $T_{\mathcal{R}}X$ *is the smallest pre-order $\to$ on the terms $T_\Sigma(X)$ satisfying the following rules:*

$$[\textsc{Var}]\ \frac{x \to y \text{ in } X}{\text{'}x \longrightarrow \text{'}y} \qquad [\textsc{Cong}]\ \frac{f \in \Sigma_n, t_1 \longrightarrow s_1, \ldots, t_n \longrightarrow s_n}{f(t_1, \ldots, t_n) \longrightarrow f(s_1, \ldots, s_n)}$$

$$[\textsc{Inst}]\ \frac{Y \vdash l \to r \in R,\ \sigma : Y \to T_{\mathcal{R}}X}{l\sigma \longrightarrow r\sigma}$$

As with algebraic theories, the term reduction algebra gives rise to a monad. The essential observation is that the action, unit and substitution preserve the reduction order; note how the clause [VAR] guarantees that the unit is a monotone function.

Lemma 2.12. *Let $\mathcal{R}$ be a TRS. The map $X \mapsto T_{\mathcal{R}}(X)$ extends to a monad $\mathcal{T}_{\mathcal{R}} = \langle T_{\mathcal{R}}, \eta, \mu \rangle$ on **Pre**.*

Lemma 2.12 provides us with a categorical semantics for TRSs. This semantics is powerful enough to prove non-trivial modularity results, but in a number of ways, it is still unsatisfactory and can be improved on. Firstly, the semantics is still term-generated, so proofs about it may still use induction on the term structure; secondly, the semantics has been constructed based on a number of choices, like that of the base category, and if we change any of these, we have to start the construction from scratch again; and thirdly, the construction does not generalise easily to other forms of rewriting. These deficiencies are answered by the theory of enriched monads.

TRSs and Enriched Monads

As we have seen, algebraic theories are modelled by monads over **Set** while TRSs are modelled by monads over **Pre**. These results are instances of a general theory developed by Kelly and Power [85, 87] which shows how monads over categories other than **Set** give rise to a generalised notion of algebraic theories. For any such category $\mathcal{A}$, this theory provides us with suitably generalised notions of *signature*, *term*, and *equations* (Def. 2.13, 2.16 and 2.17 below). Technically, we *enrich* the monads [85], but because of the overview nature of this paper, we de-emphasise this here; we also make several simplifications to keep technicalities at a minimum. A gentle but more exact introduction into this theory can be found in [126].

In general, we replace every occurrence of **Set** in our definitions by an arbitrary lfp category $\mathcal{A}$ (as introduced in Sect. 2.1.1 above). Recall from Def. 2.1 that a signature is a function $\Sigma : \mathbb{N} \longrightarrow \mathbf{Set}$; here, we replace a *set of operations* by an *$\mathcal{A}$-object of operations*. The key point is that the natural numbers represent the isomorphism classes of finite sets (i.e. objects of the base category). Hence, we replace $\mathbb{N}$ by the object $\mathcal{N}$ of $\mathcal{A}$ representing sets of *finitely presentable* objects of $\mathcal{A}$:

Definition 2.13 (Generalised Signatures). *If $\mathcal{A}$ is a category, then a $\mathcal{A}$-signature is a map $\Sigma : \mathcal{N} \longrightarrow \mathcal{A}$. The category of $\mathcal{A}$-signatures is written **Sig**$(\mathcal{A})$ and is the functor category **Sig**$(\mathcal{A}) = [\mathcal{N}, \mathcal{A}]$*

A pre-order is finitely presentable iff its carrier set is finite. Hence if we present a TRS as a generalised signature Σ, the signature will map a finite pre-order P to a pre-order $\Sigma(P)$ of operations of that arity. The carrier set of $\Sigma(P)$ can be thought of as term constructors and the order relation

as rewrite rules. This is another instance where the categorical approach differs from the traditional one where rewrite rules are more often viewed as oriented equations rather than constructors in their own right.

We write $\mathbf{1}, \mathbf{2}, \ldots, \mathbf{n}$ for the finite discrete pre-orders, the elements of which we denote as $x_1, x_2, \ldots, x_n$.

Example 2.14 (Addition cont'd). In the generalised signature $\mathcal{R}_{Add}$ for the addition TRS, the term constructors retain their arities, while the two rewrite constructors have arity $\mathbf{1}$ and $\mathbf{2}$, respectively. Hence, we have

$$\begin{array}{rcll} \mathcal{R}_{Add}(\emptyset) & = & \{0\} & \\ \mathcal{R}_{Add}(\mathbf{1}) & = & \{S, s_1 \xrightarrow{r_1} t_2\} & \\ \mathcal{R}_{Add}(\mathbf{2}) & = & \{+, s_2 \xrightarrow{r_2} t_2\} & \\ \mathcal{R}_{Add}(X) & = & \{\} & \text{for all other } X \end{array}$$

Since the arity of the rewrite rule $0+x_1 \longrightarrow x_1$ is the discrete 1-element pre-order, we have represented it by an arrow in $\mathcal{R}_{Add}(\mathbf{1})$. But of course arrows need sources and targets, and hence the carrier set of $\mathcal{R}_{Add}(\mathbf{1})$ has three elements and thus declares three term constructors. To model a TRS as a generalised signature we extend the signature of the TRS with a term constructor for the source and target of every rewrite rule r_i (which above are denoted by s_i and t_i). We shall later use equations to enforce that e.g. the target of rule r_2 equals $S(x_1 + x_2)$. Importantly, arities can be non-discrete. Syntactically, this means we allow *generalised rewrite rules*, given as $Y \vdash l \to r$ where Y is a *pre-order* and $l, r \in T_\Sigma(Y)$.

The next step is to generalise the term algebra construction from Def. 2.3. Equation (2.1) generalises with F_Σ the *left Kan extension* (see Sect. 2.1.1 above) along the inclusion $J : \mathcal{N} \longrightarrow A_f$, i.e. $F_\Sigma = Lan_J \Sigma$. The associated term algebra $T_\Sigma : \mathcal{A} \longrightarrow \mathcal{A}$ is then the free monad over the endofunctor F_Σ, which can be constructed in a number of ways [84]:

Lemma 2.15. *Let F be a finitary endofunctor over an lfp category $\mathcal{A}$. Then the free monad H_F on F satisfies the following:*

1. *For every X in $\mathcal{A}$, $H_F(X)$ is the carrier of the initial $X + F$-algebra.*
2. *The forgetful functor $U : F\text{-}\mathbf{alg} \longrightarrow \mathcal{A}$ from the category of F-algebras to $\mathcal{A}$ has a left adjoint L, and $H_F \cong UL$.*
3. *H_F is the colimit of the sequence*

$$S_0 = 1 \quad S_{n+1} = 1 + F \diamond S_n \quad H_F = \operatorname*{colim}_{i<\omega} S_i \tag{2.3}$$

and the composition of $F, E : \mathcal{A}_\kappa \longrightarrow \mathcal{A}$ is given as $F \diamond E \stackrel{def}{=} (Lan_I F \circ E)$.

The sequence S_i in (2.3) in Lemma 2.15 is called the *free algebra sequence* and can be seen as a uniform calculation of the initial $X + F-$ algebra.

Definition 2.16 (Generalised Term Algebra). *For a $\mathcal{A}$-signature $\Sigma : \mathcal{N} \longrightarrow \mathcal{A}$, the term algebra $T_\Sigma(X)$ over an object X of $\mathcal{A}$ is the free algebra over the free polynomial generated by the signature:*

$$T_\Sigma(X) = H_{F_\Sigma}(X)$$

We can calculate $T_\Sigma(X)$ more explicitly, using the formula for left Kan extension and Lemma 2.15, as the limit of the following sequence:

$$\begin{aligned} T_\Sigma(X) &\stackrel{def}{=} \operatorname*{colim}_{n<\omega} T_n(X) \\ T_0(X) &\stackrel{def}{=} X \\ T_{n+1}(X) &\stackrel{def}{=} X + \sum_{c \in \mathcal{N}} \mathcal{A}(c, T_n(X)) \times \Sigma(c) \end{aligned}$$

The semantics given by Def. 2.16 is *free* in the sense that there this an adjunction between the category of signatures, and the category of finitary monads over $\mathcal{A}$ [87]:

$$\mathbf{Sig}_{\mathcal{A}} \underset{\mathsf{U}}{\overset{T_-}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathbf{Mon}_f(\mathcal{A}) \tag{2.4}$$

Resuming our running example, we now need to replace the term constructors representing the source and target of rewrites with the intended terms. Intuitively, we would like to write

$$s_1(x) = 0 + x, \quad t_1(x) = x, \quad s_2(x, y) = Sx + y \quad \text{and} \quad t_2(x, y) = S(x + y).$$

In order to be able write down equations like this, we turn towards the categorical treatment of equations.

Given a signature Σ, a Σ-equation is of the form $Y \vdash l = r$ where Y is a set of variables and $l, r \in T_\Sigma(Y)$. An *algebraic theory* $\langle \Sigma, E \rangle$ consists of a signature Σ and a set E of Σ-equations.

The term algebra construction generalises from signatures to algebraic theories by mapping a set X to the term algebra quotiented by the equivalence relation $\sim$ generated from the equations. That is, $T_{\langle \Sigma, E \rangle}(X) = T_\Sigma(X)/\!\sim$. To move to generalised algebraic theories, note that an algebraic theory $\langle \Sigma, E \rangle$ specifies a set of equations, each of which has a left hand side and a right hand side. Thus, the appropriate generalisation is to represent an algebraic theory as a pair of maps:

Definition 2.17 (Generalised Algebraic Theory). *A $\mathcal{A}$-*algebraic theory *$\mathcal{A} = \langle \Sigma, E \rangle$ is given by an $\mathcal{A}$-signature Σ, and a set of equations, which*

are given by a function $E : \mathcal{N} \longrightarrow \mathcal{A}$ *together with a family of pairs of maps*

$$\left\{ E(c) \underset{\tau_c}{\overset{\sigma_c}{\rightrightarrows}} T_\Sigma(c) \right\}_{c \in \mathcal{N}}$$

The idea is that E specifies the shape of the equations while the maps σ and τ can be thought of as exhibiting a subobject of $T_\Sigma(c) \times T_\Sigma(c)$, i.e. pairs of terms, just like usual equations.

In the addition example there are two equations between terms of arity $\mathbf{1}$ and two equations between terms of arity $\mathbf{2}$, hence $E(\mathbf{1}) = \{e_1, e_2\}$, $E(\mathbf{2}) = \{e_3, e_4\}$. The maps σ_1 and τ_1 pick out a pair of elements of $T_\Sigma(1)$, and similarly σ_2 and τ_2 of $T_\Sigma(2)$. Therefore we define:

$$\begin{array}{rclrcl} \sigma_1(e_1) & = & s_1(x_1), & \sigma_1(e_2) & = & t_1(x_1), \\ \tau_1(e_1) & = & 0, & \tau_1(e_2) & = & x_1, \\ \sigma_2(e_3) & = & s_2(x_1, x_2), & \sigma_2(e_4) & = & t_2(x_1, x_2), \\ \tau_2(e_3) & = & Sx_1 + x_2, & \tau_2(e_4) & = & S(x_1 + x_2). \end{array}$$

This definition forces the source and targets of the rewrites to be the terms we want. Just as for usual algebraic theories, the semantics of such a generalised algebraic theory is constructed as the quotient of the term algebra by the equations. Categorically, quotients are defined by *coequalisers* and hence the monad $T_\mathcal{A}$ is given as the *coequaliser* of the transposition of the two maps σ and τ in the category of finitary monads over $\mathcal{A}$ (or in other words the smallest monad which makes these maps equal):

$$\mathsf{T}_E \underset{r'}{\overset{l'}{\rightrightarrows}} \mathsf{T}_\Sigma. \tag{2.5}$$

Under adjunction (2.4), such pairs of maps are in bijection to families of maps as in Def. 2.17. As a side note, *every* finitary monad can be given as the coequaliser of two such free monads; we say every monad can be *represented* by generalised operations and equations [87].

We finish this section with a crucial observation about our semantics, namely that it is constructed *categorically*, and hence we can reason about it categorically. This contrasts, for example, with the 2-categorical semantics of a TRS where the theory of a TRS is constructed syntactically. It is also more amenable to generalisations, as we shall see in the following sections.

2.1.5 Other Forms of Term Rewriting

The most simple way of generalising the construction is to allow equations not only between a source or target and a term, but also between any two terms. This models *equational rewriting*, where we rewrite equivalence classes of terms with respect to a set of equations. However, more interesting generalisations can be developed if we start varying the base category.

Many-Sortedness and Order-Sortedness

The term rewriting systems from Def. 2.6 are homogeneous (i.e. there is only one sort). In order to model *many-sorted term rewriting systems* given a set S of sorts, we pick the base category $\mathbf{Pre}^S$ of maps from S to $\mathbf{Pre}$ as our base.

Instantiating Def. 2.13, we get a functor $\Sigma : \mathcal{N} \longrightarrow \mathbf{Pre}^S$ as a signature. The objects $\mathcal{N}$ presenting finitely presentable objects are S-tuples of natural numbers[1][126]. Under the bijection $[\mathcal{N}, \mathbf{Pre}^S] \cong [\mathcal{N} \times S, \mathbf{Pre}]$ a signature takes the form of a map $\Sigma' : \mathcal{N} \times S \longrightarrow \mathbf{Pre}$ which gives us a pre-order of operations for each tuple $\langle s_1 \cdots s_n, t \rangle$ of source sorts $s_1 \cdots s_n$ (with $s_i \in S$) and target sort $t \in S$; this is the form known from the literature (e.g. [47]). More details of this construction, including how to choose the correct enrichment, can be found in [126], which also shows how a certain form of *order-sorted* signatures can be modelled by allowing S a pre-order of sorts rather than a discrete set.

Higher-Order Rewriting and Variable Binding

Modelling higher-order systems as such is pretty straightforward; after all, Example 2.7 is a higher-order system. However, typically higher-order systems include *variable-binding* (e.g. the λ-calculus).

To model variable binding, we can combine our categorical model of rewriting with Fiore, Plotkin and Turi's [50] model of variable binding in so-called *pre-sheaves*, i.e. the category $F \stackrel{def}{=} [\mathbf{F}, \mathbf{Set}]$, where $\mathbf{F}$ is the category of finite cardinals and functions between them, or equivalently the free cocartesian category on one object. To see how this works, consider the special case of the λ-calculus. Let $\Lambda_\alpha(n)$ be the set of λ-terms quotiented by α-equivalence with free variables represented by indices $1, \ldots, n$. λ-binding takes a term from $\Lambda_\alpha(n+1)$ and binds one variable, giving a term from $\Lambda_\alpha(n)$. We can then describe the set of $\Lambda_\alpha(n)$ as the smallest set closed under variables, λ-binding and application by the following isomorphism

$$\Lambda_\alpha(n) \cong n + \Lambda_\alpha(n+1) + \Lambda_\alpha(n) \times \Lambda_\alpha(n). \tag{2.6}$$

Now define $\delta : F \longrightarrow F$ as the pointwise application of the function $(-)+1$ in $\mathbf{F}$ (corresponding to λ-binding above), then the functor Λ_α from (2.6) is given as the fixpoint

$$\Lambda_\alpha \stackrel{def}{=} \mu M.1 + \delta M + M \times M,$$

and we have seen above how fixpoints are given by a signature; here, we have the variables, an operation of arity δ, and an operation of arity 2 (see e.g. [122] for a more exact definition of signature in this context).

[1] Only finitely of which are allowed to be non-zero; this is vacuously true of S is finite.

To model rewriting, we replace the target **Set** with **Pre**. In the example of λ-calculus, this lets us write down β-reduction as a rewrite rule. Hamana [63] has developed this combination of our model of categorical rewriting with variable binding in more detail.

To treat typing as well, we can use the approach of the previous section, at least for the simply typed λ-calculus. From a set S of base sorts, we generate the set of T_S types as the smallest set closed under a function space constructor and finite products. Then, our base category is the presheaf $[\mathbf{F}, \mathbf{Pre}^{T_S}]$ (see [63] for details).

2.2 Infinitary Term Rewriting

Another possible generalisation is to *dualise* the construction, a well-known principle in category theory. This is no idle fancy, but motivated by the success of using *coalgebraic* methods to model such things as transition systems, process algebras and other potentially infinite systems. As coalgebras are dual to algebras — and in particular the final coalgebra is the dual of the initial algebra — the question arose wether a dualisation of our construction gives us a model of *infinitary term rewriting*, i.e. rewriting infinite terms.

It turns out that our construction can be dualised in more than one way [L-6]. As mentioned above, a signature Σ gives rise to a free endofunctor F_Σ, and the initial algebra of terms gives the free monad T_Σ. One dualisation is therefore to consider not initial algebras, but rather *final coalgebras*, while the other dualisation considers not the free functor and monad over a signature, but rather the *cofree functor and comonad* over the signature. Dualising the initial $X + F_\Sigma$-algebra of terms built over variables X to the final $X + F_\Sigma$-coalgebra gives not just finite terms, but the finite and infinite terms built over X. On the other hand, the final $X \times G_\Sigma$-coalgebra (or initial $X \times G_\Sigma$-algebra), where G_Σ is the *cofree* endofunctor generated from Σ, defines a comonad (and no terms, but something entirely different, as we will see below).

	Monads	Comonads
Initial Algebras	$\mu Y.\, X + FY$	$\mu Y.\, X \times FY$
Final Coalgebras	$\nu Y.\, X + FY$	$\nu Y.\, X \times FY$

Table 2.1: Algebras and Coalgebras for an endofunctor

Table 2.1 summarises the possible dualisations. The upper left-hand entry is the traditional initial algebra semantics, the lower left-hand entry is the coalgebraic approach which we will explore in Sect. 2.2.1, the lower right-hand entry will be explored in Sect. 2.2.2, and the upper right-hand entry has remained unchartered territory.

2.2.1 Coalgebraic Approaches: Algebraic and Rational Terms

The first result in comparing the initial algebra $\mu Y.X + FY$ to the final coalgebra $\nu Y.X+FY$ is due to Barr [17], who showed that the final coalgebra is the Cauchy-completion of the initial algebra. In other words, if F arises from a finitary signature, then $\nu Y.X + FY$ is the set of terms of finite *and infinite* depth. Barr's result was generalised from the base category of **Set** to lfp categories by Adámek [3].

The first version of the following result was shown by Moss [106]; the present form was shown indepently by Aczel, Adamék and Velebil [2], and de Marchi [38],[L-6].

Lemma 2.18 (Final Coalgebras form a Monad). *Let $\mathcal{A}$ be an lfp category such that the unique map $! : 0 \longrightarrow 1$ is a strong monomorphism. Let F be a polynomial functor such that F preserves monos and ω^{op}-chains, and that there is a map $p : 1 \longrightarrow F0$.*

Then the map T_F^∞ assigning X to the carrier of the final coalgebra $\nu Y.X+FY$ can be extended to a monad of rank $\aleph_1$.

Crucially, the monad T_F^∞ has a higher rank than the monad T_F which assigns X to the initial coalgebra. This can be seen easily — consider F as arising from the signature with just one unary operation, then T_F^∞ is the monad which assigns X to the set of infinite streams of elements from X. If X is infinite, an infinite stream can contain an infinite number of elements from X, so it cannot arise from a finitary monad.

Lemma 2.18 requires some technical conditions, which may be awkward to verify. In subsequent work, these conditions have been replaced by more tractable conditions, which describe the terms by means of recursive equations. We can show that all terms arising as solutions of *rational* and *algebraic* equations form a monad. Algebraic equations for a signature Σ equations of the form

$$\phi_i(x_{i,1}, \ldots, x_{i,n_i}) = t_i(x_{i,1}, \ldots, x_{i,n_i})$$

for $i = 1, \ldots, m$ where the unknowns $\phi_1, \ldots, \phi_m$ form a signature Ω and the terms $t_1, \ldots, t_m$ are all finite terms in $T_{\Omega\cup\Sigma}(X)$ with the top symbol from Σ. As an example, consider the single equation $\phi(x) = \mathtt{A}(x, \phi(\mathtt{B}(x)))$. (Rational equations are similar, but the unknowns are not allowed to have parameters.)

Categorically, algebraic equations can be regarded as a natural transformation

$$\Omega \longrightarrow (1 + F_\Sigma T_{\Sigma\cup\Omega})\cdot J \tag{2.7}$$

where J is the inclusion of the finitely presentable objects into the base category, and F_Σ the free functor on the signature Σ (2.1). This sort of scheme can be captured abstractly by the closely related notions of F-coalgebraic

[L-5] and F-guarded [L-4] monad; the latter means that the monad comes equipped with an endofunctor F and a natural transformation $\alpha : F \longrightarrow T$ (where F is the abstract counterpart of the concrete F_Σ in (2.7)).

In general, [L-4] also gives criteria when pointwise colimits (such as the Cauchy completion) form a monad. Using this, we can also show that term graphs form a monad, by showing that term graphs can be considered as a coalgebra: a term graph with nodes S can also be given as a coalgebra

$$S \longrightarrow X + F_\Sigma(S)$$

which maps every node S to a variable if this node corresponds to a variable, or a term $\omega(s_1, \ldots, s_n)$ (where ω is an operation of arity n and $s_1, \ldots, s_n$ are successor nodes), if this node represents an operation ω applied to arguments represented by the successor nodes.

Summing up, we can show that various and interesting subclasses of infinite terms form monads, hence the underlying notion of monads as abstract models of computation captures all of these. Note that all of these monads are infinitary, i.e. of rank $\aleph_1$.

What has not been developed in depth yet is the modularity theory for infinite terms. It is known that in contrast to the results for term rewriting, strong normalisation is modular and confluence is not modular for term graph rewriting; a categorical account of these phenomena would be interesting.

2.2.2 The Weird And Wonderful World Of Comonads

Whereas the consideration of the final coalgebra of an endofunctor leads to a wealth of useful and well-known concepts, the comonad and its final coalgebra lead us into stranger territory.

First, we have to be careful what to dualise. Obviously, the base category should stay lfp (as all well-known base categories such as **Set**, **Pre**, **Cat** etc. are lfp and not co-lfp). Recall that given a finitary signature $\Sigma : \mathcal{N} \longrightarrow \mathcal{A}$, we constructed the corresponding endofunctor $F_\Sigma : \mathcal{A}_f \longrightarrow \mathcal{A}$ as the left Kan extension, followed by the left Kan extension to get a finitary monad, and the free algebra sequence. Now, in the dual case given a signature $\Sigma : \mathcal{N} \longrightarrow \mathcal{A}$ we first take the *right Kan* extension to get a functor $G_\Sigma : A_f \longrightarrow \mathcal{A}$ from the f.p. objects, and then the left Kan extension to get a finitary endofunctor with the free algebra sequence as before. This mix of left and right Kan extensions makes the construction technically challenging.

In the comonadic case, signatures behave differently so it makes sense to actually talk about cosignatures, even if the definition is the same. For example, for signatures we usually have $\Sigma(c) = \emptyset$ for many arities c, i.e. the signature has no operations of that arity. For cosignatures, if there is one or more arities c with $\Sigma(x) = \emptyset$, then $G_\Sigma(X) = \emptyset$ for all X, i.e. there are no terms! (This is because the right Kan extension is calculated as a

product.) It turns out that the default case is $\Sigma(c) = 1$, i.e. the unit of the monoidal structure. The simplest non-trivial cosignature we can consider is therefore $\Sigma_P(2) = 2$ and $\Sigma_P(c) = 1$ for all $c \neq 2$ (working over **Set**); then $G_{\Sigma_P}(X) = [\mathbf{Set}(X, 2), 2] = [[X, 2], 2]$.

To see what this means more concretely, let us consider models for these cosignatures. A comonadic model of a cosignature Σ is a G_Σ-coalgebra (with some size restrictions), or in more elementary terms an object X (of certain size) together with, for each arity c a function $\mathcal{A}(X, c) \longrightarrow \mathcal{A}(X, \Sigma(c))$. Then, a model for Σ_P is a finite set X and a function $\mathbf{Set}(X, 2) \longrightarrow \mathbf{Set}(X, 2)$ (for all other arities, we have $\mathbf{Set}(X, c) = 1$, and exactly one function into it). Since maps $f : X \longrightarrow 2$ are just predicates over X, this is akin to a predicate transformer, and if we move to the base category $\omega\mathbf{CPO}$ of ω-complete partial orders and continuous maps we get the usual definition of predicate transformer as a model.

Building the cofree comonad (or equivalently constructing the final coalgebra) over the endofunctor G_Σ is again not quite straightforward, because of the increase in rank; in general, the cofree comonad has a higher rank than G_Σ, and it is not given by the straightforward colimit of a chain of terms of increasing depth, but as the quotient of a generating set in the categories of G_Σ-coalgebras. For certain cases, this construction is known; for example, for the case of Σ_P above, the cofree comonad maps sets X to the final $X + G_{\Sigma_P}$-coalgebra, which are bipartite, rooted, acyclic graphs (quotiented by bisimulation; see [L-27] for more details).

We can then define a notion of coequational presentation as the equaliser of two cofree comonads in the category of accessible comonads, which gives a notion of equation restricting the coalgebras of a comonad. For example, we can specify that in the bipartite, rooted acyclic graphs mentioned before no state has more then n outgoing branches.

All of these are rather abstract characterisations, and they do not lend themselves to the clean syntactic treatment of their initial counterpart. Related work [4, 12] characterises coequational presentations as monomorphisms into the final coalgebra, which is equivalent to our description as an equaliser, or a modal logic [94].

Briefly summing up this sections, it turns out that monads also capture the notion of *coalgebras*, and hence can also be used to model potentially infinite objects such as infinite terms or term graphs, whereas comonads are a completely different kettle of fish.

2.3 A General Model of Rewriting

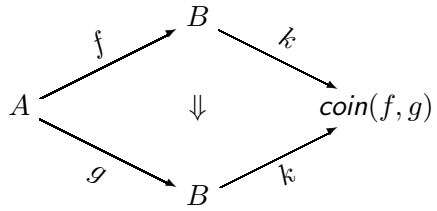
We have started to model unsorted first-order term rewriting with monads, and extended this to cover various other structures: many-sorted, higher-order, equational, rational and algebraic terms, and term graphs. And yet

there are more structures which may want to rewrite: to solve the ideal membership problem, we may want to rewrite polynomials over a given ring (to obtain the Gröbner basis) [31], or we may want to rewrite strings to solve word problems in group theory [49]. In each of these cases, we have to instantiate the general framework from Sect. 2.1 separately. Can we not get a general framework to model rewriting independently of the data being rewritten?

The answer to this goes back to Gordon Plotkin, who once remarked that 'rewriting is just equational logic without symmetry'. At first this is obvious — if we present equational logic by a set of rules and delete the symmetry rule we get term rewriting — but the real insight is that this holds for *all* forms of rewriting. Hence, we can develop a general model of rewriting if we take the categorical model of equational logic — a coequaliser — and remove the symmetry by inserting a 2-cell to obtain a *coinserter*.

A coinserter [86] is a particular instance of a weighted colimit [85]. It can be defined in any preorder-enriched category:

Definition 2.19 (Coinserter). *Let $\mathcal{A}$ be a preorder-enriched category with a pair of arrows $f, g : A \longrightarrow B$. Their coinserter is given by an object $\mathsf{coin}(f,g)$ and a morphism $k : B \longrightarrow \mathsf{coin}(f,g)$, such that $kf \Rightarrow kg$.*



Furthermore, for any other object P and morphism $p : B \longrightarrow P$ such that $pf \Rightarrow pg$, there exists a unique morphism $!_p : \mathsf{coin}(f,g) \longrightarrow P$ such that $!_p k = p$; and this assignment is monotone, i.e. if there is another $p' : B \longrightarrow P$ such that $p' \Rightarrow p$ and $p'f \Rightarrow p'g$, then $!_{p'} \Rightarrow !_p$.

In the particular case of monads over a base category $\mathcal{A}$, we have morphisms between monads, and 2-cells between the morphisms (for example, by pointwise ordering if the base category is ordered).

The data to be rewritten are modelled as monads, as these model the key notions of substitution, layer, and variables. Thus, a *rewrite presentation* is given given by two finitary monads T, E on **Pre** and a pair of arrows $l, r : \mathsf{E} \longrightarrow \mathsf{T}$ in $\mathbf{Mon}_f(\mathbf{Pre})$, and their coinserter $\mathsf{coin}(l,r)$ is the representing monad for the corresponding rewrite system [L-2]. This means that l and r pick pairs of terms from T, and if we spell out Def. 2.19 here, this means that l always reduces to r (this is the existence of the 2-cell k), and that the coinserter has the least such order (i.e. only as much rewrites as necessary; this is the universal property of the coinserter). Note that we do not require the monads E, T to be free, since T contains the data to be rewritten, which for example in the higher-order case is not given by a free monad.

However, this generality comes at a price — for example, there are no obvious modularity results to prove anymore, since modularity results vary considerably across the different forms of rewriting (see the remarks above about modularity of confluence and strong normalisation in term graph rewriting and term rewriting respectively). As we will see later, it can also be fruitful to go the other way — to be *less* general in order to capture situations where more can be said.

2.4 Modularity and Compositionality

We have developed a semantics for various flavours of rewriting, but is it of any use? It is of a certain elegance, fitting everything nicely into one semantic framework, but can we prove something substantial with it? It turns out we can, and it turns out there are further applications in such fields as functional programming.

The two main advantages of the semantics are that it promotes *modularity* and *compositionality*, i.e. it allows top-down reasoning about large systems, breaking down assumptions about the whole into assumptions about its components and promoting results about the components to results about the whole, and it allows us to compose without having to recalculate the semantics from scratch.

For this, it is essential that our semantics is free, i.e. that the mapping from the category of rewriting systems to the representing monads is a left adjoint. This enables compositionality: many structuring operations, such as disjoint union, shared union or parameter instantiations, can be modelled by colimits, and left adjoints preserve colimits, hence instead of combining the rewriting systems, we can combine their semantics. For example, given two term rewriting systems $\mathcal{R}_1$, $\mathcal{R}_2$, we have

$$\mathsf{T}_{\mathcal{R}_1+\mathcal{R}_2} \cong \mathsf{T}_{R_1} + \mathsf{T}_{R_2} \tag{2.8}$$

where the coproduct on the left is the disjoint union of the term rewriting systems, and the coproduct on the right is the disjoint union of the two monads. We will first turn to the construction of this coproduct (and colimits of monads in general), considering first the general and then simplified cases.

2.4.1 Coproducts of Monads

We first look at coproducts: given two monads T_1 and T_2, what should their coproduct look like? As an example, consider the coproduct of two monads T_Ω and T_Σ given by two signatures Σ and Ω. The terms in $T_{\Sigma+\Omega}(X)$ (which by (2.8) is equivalent to coproduct) have an inherent notion of layer: a term in $T_{\Sigma+\Omega}$ decomposes into a term from T_Σ (or T_Ω), and strictly smaller subterms whose head symbols are from Ω (or Σ). This suggests that we can

build the action of the coproduct $T_{\Sigma+\Omega}(X)$ by successively applying the two actions (T_Σ and T_Ω):

$$\begin{aligned} T_\Sigma + T_\Omega(X) = \; & X + T_\Sigma(X) + T_\Omega(X) + \\ & T_\Sigma T_\Sigma(X) + T_\Sigma T_\Omega(X) + T_\Omega T_\Sigma(X) + T_\Omega T_\Omega(X) + \\ & T_\Sigma T_\Omega T_\Sigma(X) + T_\Omega T_\Sigma T_\Omega(X) + \ldots \end{aligned} \tag{2.9}$$

Crucially, theories are built over variables, and the instantiation of variables builds layered terms. The quotes of Def. 2.3 can now be seen as encoding layer information within the syntax. For example, if $\Sigma = \{\mathtt{F}, \mathtt{G}\}$ then the term `G'G'x` is an element of $T_\Sigma(T_\Sigma(X))$ and hence has two Σ-layers. This is different from the term `GG'x` which is an element of $T_\Sigma(X)$ and hence has only one Σ-layer.

Equation (2.9) is actually too simple. In particular there are different elements of the sum which represent the same element of the coproduct monad, and we therefore need to quotient the sum. To see this, note that variables from X are contained in each of the summands in the right hand side. Similarly, operations from the same signature are layered above each other and should be composed using the multiplication from that monad. Therefore, the coproduct is a quotient of the sum in equation (2.9).

Kelly [84, Sect. 27] has shown the construction of colimits of ranked monads, from which we can deduce coproducts of monads as a special case. Roughly, the construction of the coproduct for such monads proceeds in two steps: we first construct the coproduct of pointed functors, and then the coproduct of two monads.

A *pointed functor* is an endofunctor $S : \mathcal{A} \longrightarrow \mathcal{A}$ with a natural transformation $\sigma : 1 \longrightarrow S$. Given two pointed functors $\langle T, \eta_T \rangle$ and $\langle R, \eta_R \rangle$, their coproduct is given by the functor $Q : \mathcal{A} \longrightarrow \mathcal{A}$ which maps every object X in $\mathcal{A}$ to the colimit in (2.10).

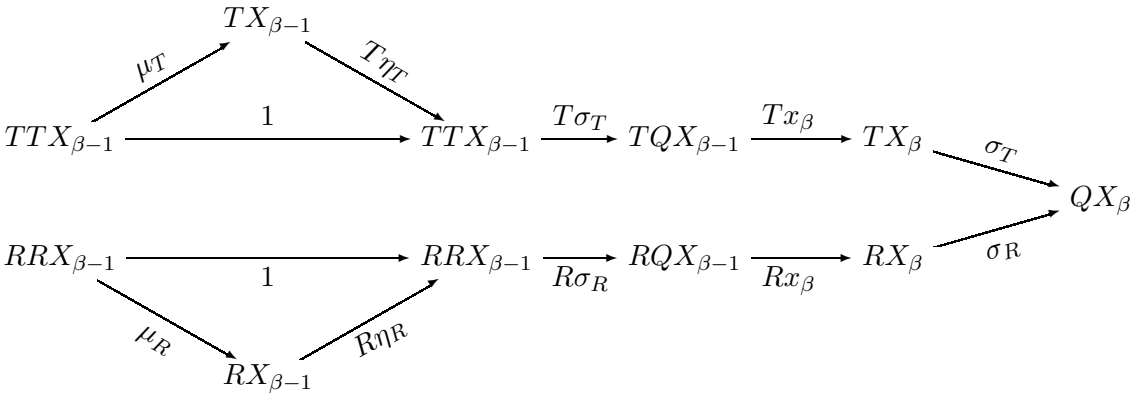
$$\begin{array}{ccc} X & \xrightarrow{\eta_T} & TX \\ {\scriptstyle \eta_R}\downarrow & & \downarrow{\scriptstyle \sigma_T} \\ RX & \xrightarrow[\sigma_R]{} & QX \end{array} \tag{2.10}$$

The coproduct of monads is constructed pointwise as well: the coproduct monad maps each object X to the colimit of a specific diagram.

Theorem 2.20. *Given two finitary monads $\mathsf{T} = \langle T, \eta_T, \mu_T \rangle$ and $\mathsf{R} = \langle R, \eta_R, \mu_R \rangle$, the coproduct monad $\mathsf{T} + \mathsf{R}$ maps every object X to the colimit of sequence X_β defined as follows:*

$$T + R(X) = colim_{\beta<\omega}\, X_\beta$$
$$X_0 = X \qquad X_1 = QX \qquad X_{\beta+1} = colim(D_\beta)$$

where Q, σ_T, σ_R is the coproduct of the pointed functors $\langle T.\eta_T \rangle$ and $\langle R, \eta_R \rangle$, and D_β is the diagram (2.11) with the colimiting morphism $x_\beta : D_\beta \longrightarrow X_{\beta+1}$. Given the shape of the diagram, x_β is a single morphism $x_\beta : QX_\beta \longrightarrow X_{\beta+1}$ making all arrows in the diagram commute.



Note that the two triangles on the left of (2.11) are *not* the unit laws of the two monads T, R, otherwise the diagram would be trivial.

Despite being the general answer to the construction of the coproduct of two monads, the usefulness of Theorem 2.20 is limited in practice since the shape, size and contents of the diagram make it difficult to reason with directly. Hence, we consider alternative constructions which trade less generality for greater simplicity.

2.4.2 Ideal Monads

An *ideal monad* (called *non-collapsing* in [L-14]) is one where, in rewriting terms, there are no rewrites where either the left-hand or right-hand side is a variable and the other one is not. Categorically, this means that TX can be decomposed into the variables X and non-variable terms T_0X, i.e. $TX = X + T_0X$ for some T_0. More succinctly, this decomposition can be written as an equation on functors, i.e. $T = Id + T_0$ (where Id is the identity functor):

Definition 2.21 (Ideal Monads). *A monad $\mathsf{T} = \langle T, \eta, \mu \rangle$ is ideal iff $T = Id + T_0$, with the unit the inclusion $in_1 : Id \Rightarrow T$ and the other inclusion written $\alpha_T : T_0 \Rightarrow T$. In addition, there is a natural transformation $\mu_0 : T_0T \Rightarrow T_0$ such that $\alpha \cdot \mu_0 = \mu \cdot \alpha_T$.*

The fundamental observation behind the construction of the coproduct $R + S$ of two ideal monads $R = Id + R_0$ and $S = Id + S_0$ is that $R + S$ should essentially consist of alternating sequences of R_0 and S_0.

Theorem 2.22 (Coproduct of Ideal Monads). *The action of the coproduct of ideal monads $Id + R_0$ and $Id + S_0$ is the functor $T = Id + (T_1 + T_2)$, with*

$$T_1 \cong R_0(Id + T_2) \qquad\qquad T_2 \cong S_0(Id + T_1)$$

The proof can be found in [L-14]. That paper also gives two other constructions, namely a quotiented layer structure where we first alternate the coproduct Q of the two pointed functors, and then quotient (rather than quotient as we build the chain, as in Theorem 2.20), and a non-alternating layers structure which is similar to that of Theorem 2.22, but does not enforce alternation of the layers, instead quotienting afterwards.

2.4.3 Abstract Modularity

A property P is *modular* (for the coproduct) if the coproduct $\mathsf{T} + \mathsf{R}$ satisfies P iff both T and R do. Modularity allows reasoning about the coproduct to be broken down into reasoning about the constituting systems, and on the other hand it allows desirable properties to be propagated from the smaller systems to the whole.

Thus, modularity results are of great practical relevance. It is well-known that confluence is modular for term rewriting systems [155], and that strong normalisation is only under a variety of restrictions [114].

To show modularity in our setting, we first have to connect properties about (term) rewriting systems with the corresponding monads. Typically, such a property P is expressed as a predicate over the base category **Pre**, and a term rewriting system $\Theta = \langle \Sigma, \mathcal{R} \rangle$ satisfies this property if $T_R(X)$ does (for an arbitrary, countably infinite set X of variables). For a monad T, we can say it satisfies this property if its action preserves it; so a monad is confluent iff TX is confluent whenever X is. Of course, we have to show separately that this coincides with the usual definition. We call such property *monadic*; [L-8] shows that confluence and SN are monadic in this sense.

Once that is done, we can show modularity of the corresponding property. In [L-22], modularity of confluence for all so-called regular monads, which are monads arising from term rewriting systems was shown, but this used a rather different diagram from Theorem 2.20, and in [L-14], modularity of strong normalisation for ideal monads was shown, but again it was all on a case-by-case basis. [L-8] shows a general result that for ideal monads, any monadic property P is modular. In our view, this strong result justifies singling out ideal monads as a particular class of monads.

2.4.4 Monads in Denotational Semantics

In denotational semantics, Moggi's *computational monads* [105] have been used to model computational effects such as stateful computations, exceptions, I/O etc. The idea is that a base category provides a basic computational model, and then a monad adds the desired feature.

The problem here is how do we combine two different computational features? If there is a *distributive law* [18] between the two monads, then we can combine them as in [80, 88] in what amounts to the *compatible monad*

construction from [18, Section 9.2]. Another way to combine computational monads is by *monad transformers*, which are pointed endofunctors on the category of monads over the base; however, monad transformers are used as an organisational tool, not as a semantic framework, and for example we have to determine all possible different combinations of our monads *a priori*, which is not compositional.

We can use the coproduct to combine computational monads [L-14,L-13]. This combination is completely compositional — we can combine any two monads — but it is no panacea, since all it does is layer the actions over each other. So if we add the monad giving us stateful computations with the monad modelling non-determinism, we get a monad where stateful computations alternate with non-determinism, but the different layers do not interact at all. This means that for example the states in two different layers of stateful computation do not interact, i.e. the state gets forgotten in between, which may not be what we want. However, we can show that combination with the exception monad always yields the desired result [L-14], as this monad has a very simple structure, and that more general if there is a strong distributive law between the two monads, their coproduct is equivalent to their compatible monad, thus our construction generalises the ones in [80, 88].

Since then, other authors have taken up this work. In particular, Plotkin, Power and Hyland have given the *tensor* [74, 75], which corresponds to a construction where each operations of one monad is forced to commute with all operations from the other monad. Their construction is in terms of Lawvere theories, but Lawvere theories and monads are well-known to be equivalent [121]. The construction, however, is not modular as it takes the disjoint union of all operations and adds in new equations. It would be interesting to rephrase this construction in terms of monads, probably with a *colimit* in the category of monads, as it seems clear that this construction is what is needed for example when combining state with other effects, as it forces the state to commute over any other action, and so preserves the state across the other effects.

2.4.5 Monads in Functional Programming

Computational monads have been very successful in functional programming, where they helped to solve the long-standing problem of integrating inherently imperative (i.e. state-based) features such as I/O into a pure functional language without comprising purity.

The solution taken for the language Haskell [118] was to encapsulate all these effects in a parameterised datatype `Monad` (a so-called constructor class), and only use the operations of the monad to combine computational effects. The problem that we cannot combine monads was solved pragmatically by combining all computational effects one can possibly want (I/O,

state, exceptions and concurrency, to name a few) into one particular monad called `IO`.

The `Monad` datatype in Haskell cannot be a faithful representation of the mathematical concept, as there is no way to enforce the monad equations in Haskell; their validity is supposed to be checked by the programmer, but *in realiter* even the ubiquitous `IO` monad does not satisfy the monad laws.

However, we can model the coproduct construction in Haskell. Just as there is a parameterised datatype `Monad`, we can define a parameterised datatype `Plus` which given two monads implements their coproduct. This datatype comes together with two injections, and the universal property:

```
inl    :: Monad t1=> t1 a-> Plus t1 t2 a
inr    :: Monad t2=> t2 a-> Plus t1 t2 a
coprod ::(Monad t1, Monad t2, Monad s)=>
         (forall a.t1 a-> s a)->
         (forall a.t2 a-> s a)-> Plus t1 t2 a-> s a
```

The datatype can then be used without needing to know how it is implemented. For the implementation, we employed another special case of monads to make the construction of the coproduct tractable:

Definition 2.23 (Layered Monads). *A* layered monad *is a monad* $\mathsf{T} = \langle T, \eta, \mu \rangle$ *such that there is a natural transformation* $\eta_X^{-1} : TX \Rightarrow 1 + X$, *which is a partial left inverse for* η_X, *i.e. for all* X, $\eta_X^{-1} \cdot \eta_X = in_1$ *(where* $in_1 : X \longrightarrow X + 1$ *is the inclusion).*

The inverse of the unit makes it decidable whether a term is a variable or not, and hence allows the construction of the coproduct [L-13]. Intuitively, we build up the coproduct as a sequence of alternating layers (layers in the term structure correspond to computations here); if we add two layers from the same monad, we can use that monad's computation, and if we add a variable we do not need to add anything because of the unit law.

For the situations the coproduct can model (non-interacting combination), this works very smoothly, except it maybe needs some syntactic sugar to make it more palatable.

2.5 Concluding Remarks

This chapter gave an overview of the work on categorical models of rewriting. The key concept is the notion of a *monad*, which as we have shown captures a wide array of useful computational models, from term rewriting systems over infinite objects to computational effects.

The advantages of monads is that they can be combined using colimits, allowing compositionality and modular reasoning. This allows us to build systems in a compositional manner, combining the basic building blocks in

a modular way without concern for their inner structure. It also allows modular reasoning, by breaking down properties of the whole system into properties of the components.

Monads can be combined by using colimits in general. We have concentrated on the *coproduct* of monads, which have used to prove modularity results in term rewriting, generalising known results and proving new ones, and found new uses of the coproduct in denotational semantics and functional programming. Other colimits, in particular the shared union (push-out), will be the subject of future research. However, it transpired that in all of these efforts it makes sense to restrict the monads considered (to such as layered monads, ideal monads etc.), as in general a monad is such an abstract concept that not much can be said. This is not a flaw, it shows that we are working at the right level of abstraction.

2.5.1 Bibliographical Remarks

Category theory has been used to provide a semantics for term rewriting systems at an intermediate level of abstraction between the actual syntax and the relational model. Research originally focused on structures such as *2-categories* [131, 135], *Sesqui-categories* [149] and *ordered categories* [77, 78] as models of TRSs. However, despite some one-off results [53, 131], these approaches did not make a lasting impact on term rewriting. Part of the problem was that these semantics are for the most part term-generated, and we often ended up translating the problem back into the syntax.

The papers [L-22,L-20] were the first to prove and extend state-of-the art term rewriting results, due to the fact that a semantics had been found which was not term generated but crucially could model the key concepts of variables, layers and substitution. The joint paper [L-20] lead to a long-standing collaboration with Neil Ghani (University of Leicester). Together with Fer-Jan de Vries (then ETL Japan), we have organised the workshop on categorical rewriting at the Rewriting Techniques and Applications RTA'98 conference 1998 in Tsukuba, Japan to disseminate these results further.

In fact, the work was taken up by the 'Leicester school' headed by Dr Ghani. Starting with the question of dualising the approach [L-27,L-6], this work lead to the investigation of infinitary rewriting [L-26,L-5,L-4] and two EPSRC grants to investigate categorical rewriting and its applications. Other work in Leicester, without direct involvement of the author, include the categorical modelling of computational group theory though Kan extensions [54, 55, 109]. Joint work further includes the applications in functional programming [L-13].

After the rather diverse work over the last five years, the papers [L-2] and [L-8] then developed a more abstract formulation, incorporating all the various formalisms in one coherent framework with an aim of obtaining results at that level.

Chapter 3

Formal Proof and Software Development

This chapter deals with *formal* proof and software development. To start, we make precise what we mean by formal proof: Formal does not merely mean that it is written down in mathematical language, but that it is actually written down in formal logic so it can be checked for correctness. Thus, a *formal proof (or development)* is one which can be *automatically checked* for correctness by a machine.

Formal software development in this sense has three distinct advantages:

- Firstly, it greatly increases correctness. Note there is still enough room for mistakes — specifications can be wrong, proofs may rely on side conditions or the implementation of the theorem prover may be wrong or inconsistent. But formal proof make all these dependencies *explicit* — we know precisely all side conditions which need to hold, and we know also which parts of the prover we must trust.[1]

- Secondly, it opens the room for *proof support*. Once we write down specifications formally, we can let the machine do the trivial and simple parts of the development, leaving the user with the complicated design decisions.

- Thirdly, it allows for *abstraction* and *reuse*. Once the development is a formal object, we can develop techniques to reuse a development in a different setting, or to make it generally more applicable.

While the first two advantages are widely acknowledged and utilised, the third is still largely untapped potential; our advances in this area are the major contribution here.

[1] This leads into questions of prover architecture which we will later consider.

The definition above is a rather loose, and covers a lot of different logics, approaches and systems. In our work, we have made specific choices in these matters, which we want to explicate — if not justify — in the following:

Firstly, we believe that *higher-order logic* is a *sine qua non* for realistic proof and development. Nearly all realistic applications of formal software development or verification use higher-order principles, sometimes in restricted form; for example ACL2 [83] or Inka [10] restrict themselves to induction. This makes perfect sense; most users certainly do not want to see the encoding of the natural numbers in higher-order logic, they just want the Peano axioms. Which brings us to the second principle:

Secondly, we believe that *logical embeddings* make perfect sense in the context of software development: on the one hand, one wants to use domain-specific languages, i.e. specification (or even programming) languages which are tailored to the situation at hand; on the other hand, we want to reuse as much of the technical infrastructure (proof support etc.) as possible. The solution is to encode domain specific-languages into a *meta-logic* where the actual formal proof work is done. Of course this needs a powerful enough meta-logic, explaining our preference for higher-order logic above.

Thirdly, we found the generic theorem prover Isabelle to be an excellent tool for our purpose. It is a logical framework in itself, but has a well-supported encoding of classical higher-order logic (HOL) which can in turn serve as a meta-logic for the encoding of other specification languages, in particular with its encoding of the logic of computable functions (LCF) [108], which gives us an executable sublanguage and makes HOL suitable as the basis of a wide-spectrum language. It has powerful proof support which is competitive with what most automatic provers can offer, and its flexible syntax machine make it easy to encode one's favourte logic or specfication formalism in a readable way.

In the rest of this chapter, we will explain the logical foundations of a generic system for formal development. We will argue that formal software development comprises development in-the-small, which corresponds to formal proof, and development in-the-large, which corresponds to the manipulation of specifications. We will show the logical foundations of transformational program development, and see how this is implemented in Isabelle in our TAS system. We will also investigate how to make theorems more general, and how to use this to abstract develoments, e.g. to derive new transformation rules. We will finally consider an ongoing refactoring of the TAS system. We will not consider issues of system architecture, interface design and implementation as this will be the focus of the next chapter.

3.1 Transformational Program Development

In general, a formal *specification* of a program or system is an abstraction on the actual system, designed to make its properties clear and reasoning about them feasible. In a formal *development*, the specification formalism comes equipped with at least one transitive correctness-preserving refinement relation $S \sqsubseteq S'$ which expresses that a specification S' is a correct specialisation of another specification S. Refinement is used to verify that a proposed system design preserves the properties of its specification, i.e. the correctness of the implementation. However, as the program may be substantially different from the original specification, it is preferable to repeatedly prove refinement for minor changes one at a time until one has arrived at the final program. A *stepwise refinement process* can be understood as a sequence of specifications

$$SP_1, SP_2, \ldots, SP_n, \tag{3.1}$$

in which each successive stage of the program development process is a correct refinement of its preceding stage, e.g. $SP_i \sqsubseteq SP_{i+1}$ holds for $i = 1, \ldots, n-1$. The transitivity of the refinement relation guarantees that SP_n is a refinement of SP_1. Usually, SP_1 is an initial (abstract) specification and SP_n is a program (or executable specification), but various cases in between may be considered; for example, refining a requirement specification to a design specification or refining an executable specification to an equivalent but more efficient one. This incremental approach has a long and distinguished history. It has been present implicitly in Dijkstra's and Gries' early works [42, 61], and been advocated and studied for different notions of refinement in the literature [70, 29, 39, 133]. However, refinement relations provide little guidance for program development. Stepwise refinement is essentially an "invent and verify" process to prove the correctness of development steps a posteriori; stepwise refinement is not an effective method for program development.

Transformational development offers a solution to this problem. Instead of inventing refinement steps, each step arises from applying a *transformation rule*, which has been proven correct previously. Thus, transformation rules may serve to guide the development and to reduce the amount of proof work, and the sequence (3.1) becomes a sequence of applications of transformations $T_1, \ldots, T_n$:

$$SP_1 \overset{T_1}{\rightsquigarrow} SP_2 \overset{T_2}{\rightsquigarrow} SP_3 \overset{T_3}{\rightsquigarrow} \ldots \overset{T_{n-1}}{\rightsquigarrow} SP_n \tag{3.2}$$

and the correctness of the transformation rules guarantees that $SP_i \sqsubseteq SP_{i+1}$. This approach was first explicitly advocated for equivalence transformations [32], and later on for development by transformation of predicate transformers [15], or in general restriction of the class of models [133, 134].

Many systems have been built to support this notion of correct development. Early systems such as CIP [19], KIDS [141] and its successors Planware [25] and Specware [145, 142], KIV [123, 124] or PROSPECTRA [71] have been constructed from scratch, with a built-in notion of correctness, a fixed notion of refinement, and a given library of transformation rules.

However, transformation systems can profitably be encoded in general purpose theorem provers. The theorem prover helps organise the overall development and provides proof support for discharge of applicability conditions. If the theorem prover itself is correct, and every transformation rule has been proved correct inside the theorem prover, correctness of the overall development is guaranteed. This approach has particularly been investigated for the Refinement Calculus (RC) [15]; examples are the Refinement Calculator [95, 35], the Program Refinement Tool [36], and the work of Staples [147] or Hemer et al. [67]. One drawback of these systems is that they use are built around a particular refinement methodology and formal method, which may not be the one we may want to use in a particular setting.

3.1.1 The TAS System.

TAS is a generic transformation system built on top of the Isabelle theorem prover. It comes together with a graphical user interface (GUI), but we will explain the design and construction of that in the next chapter. Although TAS is built on Isabelle, the basic principles of modelling transformational proof and development hold for any logical framework. This treatment of transformation based on rewriting program schemas and a second-order matching algorithm was first proposed by Huet and Lang [72].

Transformational Development in a Logical Framework

A logical framework [64, 120] is a meta-level inference system which can be used to specify other, object-level, deductive systems. Well-known examples of implementations of logical frameworks are Elf [119], λProlog [103], and Isabelle [117]. We use Isabelle, the meta-logic of which is intuitionistic higher-order logic extended with Hindley-Milner polymorphism and type classes. A central idea in logical framework encodings is to represent object logic variables by meta-logic variables, which are placeholders for meta-logic terms, and which can be instantiated by the unification of the logical framework. Following Isabelle's nomenclature, such variables will be called *meta-variables* in the following.

We represent transformation rules by a theorem of the form

$$A \Longrightarrow I \sqsubseteq O \tag{3.3}$$

where A is the *applicability condition*, I is the *input pattern* and O the *output pattern*. Theorem (3.3) is called the *logical core theorem* of the rule. To apply such a transformation to a term t, we match (or unify) the input pattern I with a subterm of t, say t_0, such that $t = C[t_0]$ where $C[\,]$ is the *context*. Let σ be a substitution which appropriately instantiates I, i.e. $I\sigma = t_0$. Then $I\sigma$ may be replaced by $O\sigma$ at the position of this subterm, i.e. the current specification $t = C[t_0] = C[I\sigma]$ is transformed to $C[O\sigma]$. Meta-variables which occur in the output pattern O but not in the input pattern I will not be instantiated by this match; they are called *parameters*, and their instantiation is left to the user. The instantiated applicability condition $A\sigma$ becomes a *proof obligation* which ensures the correctness of the transformational development. When $A\sigma$ holds, we know that $I\sigma \sqsubseteq O\sigma$. We now need to show that we can conclude $SP_1 = C[I\sigma] \sqsubseteq C[O\sigma] = SP_2$, and further that such a sequence of transformation steps

$$SP_1 \sqsubseteq SP_2 \sqsubseteq SP_3 \sqsubseteq \ldots \sqsubseteq SP_m \tag{3.4}$$

allows us to conclude that $SP_1 \sqsubseteq SP_m$. Since the system should be generic over the refinement relation used, which properties are needed for the refinement relation $\sqsubseteq$?

- Firstly, in order to be able to apply the transformation rule to a subterm, we need monotonicity (or in general, the refinement to be a congruence), i.e. $I\sigma \sqsubseteq O\sigma \Longrightarrow C[I\sigma] \sqsubseteq C[O\sigma]$.
- Secondly, transitivity is needed to deduce the refinement $S_1 \sqsubseteq S_m$ from the single refinements $SP_i \sqsubseteq SP_{i+1}$ in (3.4).
- Thirdly, reflexivity is not strictly necessary, but its presence makes reasoning much easier, and moreover we have not yet encountered practical examples of non-reflexive refinement relations.

The first requirement is actually both too strong and too weak. Too strong on the one hand, because a given refinement relation may not be monotone with respect to all given operators, but only some. That means we can only apply it in certain contexts, but when it occurs in these contexts, we are fine. On the other hand, it is too weak, because refinement inside a given context may add additional assumptions. For example, when refining the positive part P of a conditional **if** B **then** P **else** Q we can assume that the condition B holds. This is crucial: in the course of formal development, we introduce conditional expression precisely in order to handle such case distinctions.

Basics of Window Inferencing

Logically, these notions are captured by *window inferencing* [127], structured calculational proof [43, 14, 15] or transformational hierarchical reasoning

[62]. We will subsume these as window inferencing in the following, since they are quite similar. Window inferencing is a calculus for formal proof, just like natural deduction or the sequent calculus; however, it is more compact and built to mimic the equivalence-based transformational reasoning found in algebra and calculus. Here is a short example proof of $(A \wedge B \Longrightarrow C) \Longrightarrow (B \wedge A \Longrightarrow C)$.

We start with $A \wedge B \Longrightarrow C$. In the first step, we open a *subwindow* on the sub-expression $B \wedge A$, denoted by the markers. We then transform the sub-window and obtain the desired result for the whole expression:

$$\begin{array}{lll} & \llcorner A \wedge B \lrcorner \Longrightarrow C & (3.5) \\ \Rightarrow & \quad \{\text{focus on } A \wedge B\} & \\ & \quad \bullet \quad A \wedge B & \\ & \quad = \quad \{\wedge \text{ is commutative}\} & \\ & \quad \quad B \wedge A & \\ & \cdot \quad \ulcorner B \wedge A \urcorner \Longrightarrow C & \end{array}$$

The proof uses the fact that we can replace equivalent subexpressions. This is formalised by *window rules*. In this case the rule has the form

$$\frac{\Gamma \vdash A = B}{\Gamma \vdash E[A] \Longrightarrow E[B]} \tag{3.6}$$

where the second-order variable E stands for the unchanged *context*, while the subterm A (the *focus* of the transformation) is replaced by the transformation.

Implementing Window Inferencing in Isabelle

The logical core of TAS is a generic window inference package, which translates proof in the window inference style above into Isabelle's natural deduction, i.e. it allows transformational reasoning within Isabelle.

Just as equality is at the heart of algebra, at the heart of window inference there is a family of binary preorders (reflexive and transitive relations) $\{\sqsubseteq_i\}_{i \in I}$. These preorders are called *refinement relations*. Practically relevant examples of refinement relations in formal system development are impliedness $S \Leftarrow P$, process refinement $S \sqsubseteq P$ (the process P is more defined and more deterministic than the process S), set inclusion, or arithmetic orderings for numerical approximations [157].

The refinement relations have to satisfy a number of properties, given as a number of theorems. Firstly, we require reflexivity and transitivity for all $i \in I$:

$$a \sqsubseteq_i a \qquad [\text{Refl}_i]$$

$$a \sqsubseteq_i b \wedge b \sqsubseteq_i c \Longrightarrow a \sqsubseteq_i c \qquad [\text{Trans}_i]$$

The refinement relations can be ordered. We say $\sqsubseteq_i$ is *weaker* than $\sqsubseteq_j$ if $\sqsubseteq_i$ is a subset of $\sqsubseteq_j$, i.e. if $a \sqsubseteq_i b$ implies $a \sqsubseteq_j b$:

$$a \sqsubseteq_i b \Longrightarrow a \sqsubseteq_j b \qquad [\mathrm{Weak}_{i,j}]$$

The ordering is optional; in a given instantiation, the refinement relations may not be related at all. However, because of reflexivity, equality is weaker than any other relation, i.e. for all $i \in I$, the following is a derived theorem:[2]

$$a = b \Longrightarrow a \sqsubseteq_i b \tag{3.7}$$

The main device of window inferencing are the window rules shown in the previous section:

$$(A \Longrightarrow a \sqsubseteq_i b) \Longrightarrow F\,a \sqsubseteq_j F\,b \qquad [\mathrm{Mono}^F_{i,j}]$$

Here, F can either be a meta-variable, or a constant-head expression, i.e. a term of the form $\lambda y_1 \ldots y_m.c x_1 \ldots x_n$ with c a constant. Note how there are different refinement relations in the premise and conclusion of the rule. Using a family of rules instead of one monotonicity rule has two advantages: firstly, it allows us to handle, on a case by case basis, instantiations where the refinement relations are not congruences, and secondly, by allowing an additional assumption A in the monotonicity rules, we get *contextual assumptions* assumptions when refining inside a context. To finish off the picture, a transformation rule is given by a *logical core theorem* of the form (3.3) above, i.e. a theorem the conclusion of which is a refinement relation.

As an example of contextual assumptions and window rules, consider the expression $x + (\textbf{if } y = 0 \textbf{ then } u + y \textbf{ else } v + y)$. If we want to simplify $u + y$, then we can do so under the assumption that $y = 0$ (hence in this context $u + y = y$), because of the window rule given by the theorem

$$(B \Longrightarrow x = y) \Longrightarrow (\textbf{if } B \textbf{ then } x \textbf{ else } z = \textbf{if } B \textbf{ then } y \textbf{ else } z) \qquad [\mathrm{Mono}^{\mathtt{If}}_{=}]$$

Note that if we had just used the congruence rule for equality $x = y \Longrightarrow f\,x = f\,y$ we would have lost the contextual assumption $x = 0$ in the refinement of $u + y$.

The window inference package implements the basic window inferencing operations as Isabelle tactics, such as opening and closing subwindows, applying transformations, searching for applicable transformations, and starting and concluding developments. In general, our implementation follows

[2] In order to keep our transformation system independent of the object logic being used, we do not include any equality per default, as different object logics may have different equalities.

Staples' approach [146]. When we start a proof, we may not know the goal, i.e. we start with a specification and may not know what it is going to be transformed into, so we use a *meta-variable* as a placeholder which eventually gets instantiated with the goal. Further, transitivity rules are used to translate the forward chaining of transformation steps into backwards proofs on top of Isabelle's goal package, and the reflexivity rules are used to close subwindows or conclude developments. Our implementation moreover adds point&prove functionality (click on a subterm to open a window there), and search and browsing functionalities (e.g. search for applicable transformations, or display current development). We always use the most specific rule when opening a subwindow (cf. the example above). The search functions use an indexing scheme for better performance, and users can configure the search to their needs.

3.1.2 Instances of TAS

TAS has two main instances which we will consider in the following, namely one for higher-order logic itself based on model restriction, and one for process refinement.

Higher-Order Logic

As refinement relation, we will use model-inclusion — when refining a specification of some function f, the set of possible interpretations for f is reduced. The logical equivalent of this kind of refinement is the implication, which leads to the following definition:

$$\sqsubseteq : \mathit{Bool} \times \mathit{Bool} \longrightarrow \mathit{Bool} \qquad P \sqsubseteq Q \stackrel{def}{=} Q \longrightarrow P$$

Based on this definition, we easily prove the theorems `ref_trans` and `ref_refl` (transitivity and reflexivity of $\sqsubseteq$). We can also prove that $\sqsubseteq$ is monotone for all boolean operators, e.g.

$$s \sqsubseteq t \Longrightarrow s \wedge u \sqsubseteq t \wedge u \qquad \texttt{ref\_conj1}$$

Most importantly, we can show that

$$(B \Longrightarrow s \sqsubseteq t) \Longrightarrow \textbf{if } B \textbf{ then } s \textbf{ else } u \sqsubseteq \textbf{if } B \textbf{ then } t \textbf{ else } u \qquad \texttt{ref\_if}$$
$$(\neg B \Longrightarrow u \sqsubseteq v) \Longrightarrow \textbf{if } B \textbf{ then } s \textbf{ else } u \sqsubseteq \textbf{if } B \textbf{ then } s \textbf{ else } v \qquad \texttt{ref\_then}$$

which provides the contextual assumptions mentioned above.

On the face of it, this instance looks too basic to be of much use, but we note that the underlying notion of refinement is the same as used in e.g. the refinement calculus, and systems derived from it. In particular, we can now formulate a substantial part of the theory of algorithm design [142, 139, 143] in our framework. In this work, schemata for algorithms are formalised as

design transformations. A design transformation embodies a rather complex design decision, typically one which transforms a specification of a given form into an algorithm. Thus, a design transformation when encoded as a logical core theorem (3.3) has a *specification* given by pre- and postcondition as input pattern, and a *program* given as a recursive function as output pattern. Examples of design transformations include branch and bound, global search [143], or divide and conquer [140]. The latter implements a program $f : X \longrightarrow Y$ by splitting X into two parts, the termination part of f, which can be directly embedded into the codomain Y of f, and the rest, where the values are divided into smaller parts, processed recursively, and reassembled. The core theorem for divide and conquer based on model-inclusion refinement and well-founded recursion reads:[3]

$$\begin{array}{lll} A \longrightarrow & (\mathbf{Pre}(x) \longrightarrow \mathit{Post}(x, f(x)) & \\ & \sqsubseteq & \\ & \mathbf{Pre}(x) \longrightarrow f = & \texttt{let fun } F(x) = \texttt{ if } \mathit{isPrim}(x) \texttt{ then } \mathit{Dir}(x) \\ & & \texttt{else } \mathit{Com}(\langle G, F\rangle(\mathit{Decom}(x))) \\ & & \texttt{in } F \texttt{ end measure } <) \end{array} \tag{3.8}$$

where A are the (fairly lengthy) applicability conditions. As explained above, the user has to instantiate the parameters of the transformation when applying it. Here, the parameters are

- the termination criterion $\mathit{isPrim} : X \longrightarrow \mathit{Bool}$;
- the embedding of terminal values $\mathit{Dir} : X \longrightarrow Y$;
- the decomposition function of input values $\mathit{Decom} : X \longrightarrow Z \times X$;
- a function $G : Z \longrightarrow U$ for those values which are not calculated by recursive calls of F;
- the composition function $\mathit{Com} : U \times Y \longrightarrow Y$ that joins the subsolutions given by G and recursive calls of F;
- and the measure $<$ assuring termination.

Process Refinement

The instantiation of TAS for process refinement is based on the formalisation of CSP [129] in Isabelle/HOL by Tej and Wolff [152]. CSP is a process calculus with three different ways of process refinement, based on traces, failure sets and failure-divergence sets [129]. Trace refinement is most commonly used in security, whereas failure-divergence refinement is the one corresponding to implementation; it expresses that one process is less deterministic and less often diverging than the other.

[3] $\langle f, g\rangle$ is the pairing of functions defined as $\langle f, g\rangle(x, y) \stackrel{def}{=} (f(x), f(y))$.

There is no known theory of "process design", in analogy to algorithm design for sequential algorithm; instead, we have rich theory of simple refinements given by the algebraic laws of CSP. Tej's thesis [151] contains a couple of useful transformations going into the directory of design transformations for CSP, such as the following

$$\begin{array}{l} \forall m, a. m \neq e \wedge a \neq e \wedge m \neq f \wedge a \neq f \\ \quad \Longrightarrow \ \mu X. e \rightarrow f \rightarrow X \\ \qquad \sqsubseteq \\ \qquad (\mu X. e \rightarrow m \rightarrow a \rightarrow X \ |[\ \{m, a\}\ ]|\ \mu X. m \rightarrow f \rightarrow a \rightarrow X) \setminus \{m, a\} \end{array}$$

which splits a single process into two processes communicating over an internal channel; one can consider this to be a buffer introduction. The thesis has longer case studies of transformational development in HOL-CSP.

Besides transformational development, the TAS instance introduces another application, namely abstraction. Instead of refining a specification into an executable process using correctness-preserving refinement, we can transform an executable process into a more abstract specification which we can then modelcheck, using for example the model-checker FDR [51]. This allows the detection of deadlocks and livelocks in concurrent systems; the same approach, but without using Isabelle and instead relying on pencil-and-paper proofs, has been employed in [33, 34]. In order to formalise it in our setting, we need a formalisation of the theory of deadlock-preserving (or livelock-preserving) refinement, based e.g. on Lazić' data independence [96]. We can then connect FDR as an external decision procedure (a so-called *oracle*) to Isabelle, and obtain an integrated system where we can transform arbitrary processes into a form where they can be model-checked.

3.1.3 Conclusions

We have shown the two main instances of TAS. There have been other, less developed or prototypical instances, such as one for Staple's set-theory based refinement calculus [147, 148] in [L-16], or the specification language Z [L-19], based on HOL-Z [92], the encoding of Z [144] into Isabelle/HOL. We found that we could in principle capture all of the refinement-based approaches found in the literature, although of course encoding them into Isabelle first is hard work; hence we concentrated on the two main instances above.

TAS was successful as an academic prototype. It has a novel user interface and a generic architecture which can be instantiated to different formalisms. However, its impact beyond the academic world was somewhat limited.

In essence, we learned three lessons from building TAS:

- Transformational development is very appealing, and a calculus like window inferencing is far more suitable to formal software development

then e.g. natural deduction. However, window inferencing is a calculus to prove theorems, not to transform specifications. Thus, it is suitable for development in-the-small, but not for development in-the-large.

- An easy-to-use, well-designed graphical user interface adds very much to the usability of the system. However, one must be careful not to tie the user interface too closely to the system; we will reconsider this point in Sect. 4.3.

- Finally, TAS could display developments interactively, with the possibility to navigate through subdevelopments, or export them as Isabelle proof scripts. However, the development was represented internally, so users had no specification text to work on; and moreover, although users could annotate developments with comments, there was no integrated view of the development.

The last point is a serious drawback. After all, the specification is the main artefact the user is working on, and hence should always be in the centre of both the user's attention and the system's presentation.

We do not claim that these points are in any way original; specification languages like FOCUS [30], and CASL [24] also know the distinction between in-the-small and in-the-large. Also, for example the Isar user interface for Isabelle also supports the generation of typeset documentation from a proof (although in Isar's case, it is the theorem prover which generates the document).

In Sect. 3.3, we will report on ongoing work to rectify these shortcomings. However, this also incorporates work on abstracting developments for later reuse, so we will elaborate on that first.

3.2 Abstraction and Reuse

From the three advantages given at the beginning of this chapter, the third has until now not been supported by many systems. Some systems, such as KIV [16, 123], Maya [11] or Specware [145] have good support for reuse, that is if we have a development (or verification), and change the original specification, a new proof is constructed, reusing as much of the original proof as possible.

However, no system supports the systematic *generalisation* of developments. This seems surprising, because generalisation is actually the way in which mathematics develops: one starts with a concrete example, and then generalises it by deciding for each property whether it is actually relevant or not. For example, we start with natural and rational numbers, and end up with the theory of groups, rings and fields.

Since we identify transformation rules with their core theorem (3.3), the foundations of abstracting transformational developments are the abstraction of proofs and theorems. We can formulate these principles for any logical framework, and we have implemented them in our favourite one, namely Isabelle. We will give a survey of this work in the following, and consider its introduction into TAS in Sect. 3.3.

3.2.1 General Principles

The proposed generalisation process will transform a proof π of a theorem ϕ in a stepwise manner into a proof of a schematic theorem which may be instantiated in any other setting, i.e. a derived inference rule of the logic. The process consists of three phases as follows.

Making Proof Assumptions Explicit

In tactical theorem provers such as Isabelle, the use of auxiliary theorems in a proof may be hidden to the user, due to the automated proof techniques. These contextual dependencies of a theorem can be made explicit by inspecting its proof term. In a natural deduction proof, auxiliary theorems can be introduced as leaf nodes in open branches of the proof tree.

Given an open branch with a leaf node theorem in the proof, we can close the branch by the implication introduction rule, thus transforming the conclusion of the proof. By closing all open branches in this manner, every auxiliary theorem used in the proof becomes visible in the root formula of the proof. To illustrate this process, consider the proof π of theorem ϕ. At the leaf node of an open branch π_i in the proof we find a theorem, say $\psi_i(x^i_1, \ldots, x^i_{k_i})$. We close the branch π_i by applying $\Longrightarrow$-introduction at the root of the proof, which leads to a proof of a formula $\forall x^i_1, \ldots, x^i_{k_i}\ \psi_i(x^i_1, \ldots, x^i_{k_i}) \Longrightarrow \phi$, where ψ_i has been transformed into a closed formula ψ'_i by quantifying over free variables, to respect variable scoping. The transformation of a branch is illustrated in Fig. 3.1. This process is repeated for every branch in π with a relevant theorem in its leaf node. If we need to make j theorems explicit, we thereby derive a proof π' of the formula $(\psi'_1 \wedge \ldots \wedge \psi'_j) \Longrightarrow \phi$.

Abstraction Function Symbols

The next phase of the transformation process consists of replacing function symbols by variables. When all implicit assumptions concerning a function symbol F have been made explicit, as in the transformed theorem above, all relevant information about this function symbol is contained within the new theorem. The function symbol has become an *eigenvariable* because the proof of the theorem is independent of the context with regard to this function symbol. Such function symbols can be replaced by variables throughout

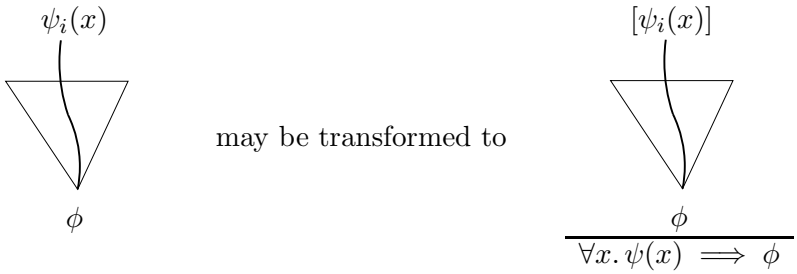


Figure 3.1: The transformation and closure of a branch in the proof, binding the free variable x of the leaf node formula.

the proof. Let $\phi[x/t]$ and $\pi[x/t]$ denote substitution, replacing t by x in a formula ϕ or proof π, renaming bound variables as needed to avoid variable capture.

If the function symbol F is of type τ and a is a meta-variable of this type, the theorem $(\psi'_1 \wedge \ldots \wedge \psi'_i) \implies \phi$ may be further transformed into

$$(\psi'_1[a/F] \wedge \ldots \wedge \psi'_i[a/F]) \implies \phi[a/F], \tag{3.9}$$

by transforming the proof π' into a new proof $\pi'[a/F]$.

Abstracting Types

When all function symbols depending on a given type have been replaced by term variables, the name of the type is arbitrary. In fact, we can now replace such type constants by free type variables. The higher-order resolution mechanism of the theorem prover will then instantiate type variables as well as term variables when we attempt to apply the derived inference rule to a proof goal.

In order to replace function symbols by variables, all relevant information about these symbols, such as defining axioms, must be made explicit. In order to replace a type constant by a type variable, function symbols of this type must have been replaced by variables. Hence, each phase of the transformation assumes that the necessary steps of the previous phases have already occurred. Note that a necessary precondition for the second abstraction step is that the logical framework allows for higher-order variables, and for the third step that the logical framework allows for type variables (which Isabelle does, with certain restrictions — see below).

It is in principle possible to abstract over all theorems, function symbols, and types occurring in a proof. However, the resulting theorem would be hard to use. For applicability, it is essential to strike a balance between

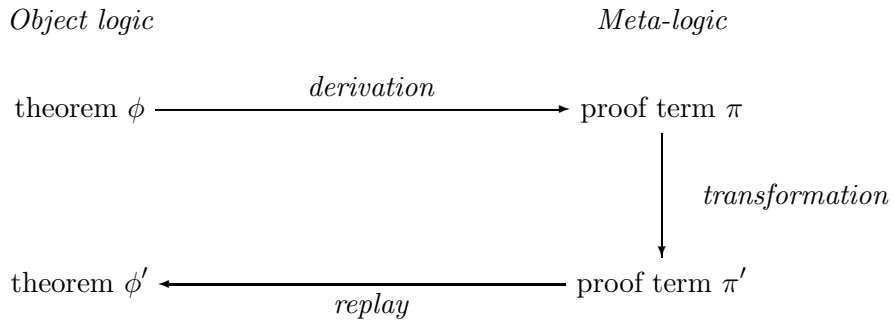


Figure 3.2: Generalising theorems by proof term transformation.

abstracting too much and too little. Thus, abstraction can never be a fully automatic process.

3.2.2 Abstraction in Isabelle

The proof terms of Isabelle are defined in a typed λ-calculus corresponding to Isabelle's meta-logic under the Curry-Howard isomorphism. Syntactically, the meta-logic proof terms can be presented as

$$p ::= h \mid c_{[\tau_n/\alpha_n]} \mid \boldsymbol{\lambda} h : \phi.p \mid \lambda x :: \tau.p \mid p \cdot p \mid p\, t \tag{3.10}$$

where h, c, x, t, ϕ, α, and τ denote proof variables, proof constants, term variables, terms of arbitrary type, propositions, type variables, and types, respectively. The language defined by (3.10) allows abstraction over term and proof variables, and application of terms and proofs to proofs, corresponding to the introduction and elimination of $\bigwedge$ and $\Longrightarrow$. Proof terms live in an environment which maps proof variables to terms representing propositions and term variables to their type. Proof constants correspond to axioms or already proven theorems. More details, including the definition of provability, can be found in [20].

The explicit representation of object logic terms, types, and proof rules in the logical framework allows any object-logic proof to be transformed by manipulating its meta-logic proof term (see Fig. 3.2). The manipulated meta-level proof terms are replayed by a proof checker. Reduction of the proof term yields a new object level inference rule, and the logical framework ensures the correctness of the derived rule. This way, we have a conservative implementation which does not compromise the integrity of Isabelle's logical kernel.

We have structured the abstraction process into the *basic abstraction steps*, corresponding to the three steps above, and *abstraction procedures* we can compose from these.

Basic Abstraction

The abstraction process iterates the following three basic abstraction steps, which are implemented as functions in Isabelle (the technical details can be found in [L-9]):

(i) *Make an assumption explicit:* A theorem ψ named *thm* used in the proof π will appear as a proof constant $thm : \psi$. The implication introduction rule (corresponding to abstraction in the λ-calculus) allows us to transform the proof π into the proof $\boldsymbol{\lambda} h : \psi.\pi[h/thm]$ of the theorem $\psi \implies \phi$, lifting ψ to a premise of the proof.

The implementation is not quite straightforward, as we need to quantify over all free variables in ψ, and need to adjust the deBruijn-indices used by Isabelle to handle bound variables in proof terms.

(ii) *Abstract a function symbol:* When function symbol has become an eigenvariable because the proof of the theorem is independent of the context with regard to this function symbol f, and can be replaced by a variable x throughout the proof. We obtain a proof $\pi[x/f]$ of the theorem $\phi[x/f]$.

The implementation of this is straightforward: we traverse the proof term, replacing the symbol f with a fresh meta-variable.

(iii) *Abstract a type constant:* Similarly, when all function symbols depending on a given type have been replaced by term variables, the name of the type is arbitrary, and we can replace such type constants by free type variables in a similar manner.

When we use a theorem in a proof, both schematic and type variables are instantiated. If we make the theorem an applicability condition we need to quantify over both the schematic and type variables. However, abstraction over type variables is not possible in the Hindley-Milner type system of Isabelle's meta-logic, where type variables are always implicitly quantified at the outermost level. Instead, distinct assumptions must be provided for each type instance. For example, a proof of the theorem

$$map\ (f \cdot g)\ x = map\ f\ (map\ g\ x), \tag{3.11}$$

contains three different type instances of the definition of map for non-empty lists $map\ f\ (Cons\ x\ y) = Cons\ (f\ x)\ (map\ f\ y)$. Similarly, when abstracting over polymorphic operations, we need distinct variables for each type instance of the operation symbol; hence, in Theorem (3.11), we need to abstract over each of the three type instances separately, resulting in three different function variables.

3.2.3 Abstraction Procedures

One rarely wants to abstract over a specific theorem. Moreover, the previous basic abstractions steps depend on each other; without first abstracting the preconditions on which an operation symbol occurs, abstraction over this operation will fail. Thus, users will not want to use the basic abstraction steps directly. For a more high-level approach, *abstraction procedures* are defined, which combine series of elementary abstraction steps.

One such procedure is *abstraction over theories*. A theory in Isabelle can be thought of as a signature defining type constructors and operations, and a collection of theorems. Theories are organised hierarchically, so all theorems established in ancestor theories remain valid. The abstraction procedure *ABS_THEORY* abstracts a theorem which belongs to a theory T_1 into an ancestor theory T_2. It collects all theorems, operations, and types from the proof term which do not occur in T_2, and applies elementary tactics recursively to abstract over each, starting with theorems and continuing with function symbols and types. Finally, the derived proof term is replayed in the ancestor theory, thus establishing the validity of the abstracted theorem in the theory T_2.

Another abstraction procedure is *abstraction over a given type*, *ABS_TYPE*. In order to abstract over a type τ, we identify in the proof term all operations $f_1, \ldots, f_n$ in the signature of which τ occurs, and all theorems $\phi_1, \ldots, \phi_m$ which contain any of these operations. We abstract over all occurrences of these theorems and operations, and finally over the type.

The approach to abstraction as implemented here combines proof term manipulation as known from type theory [99, 98] with the flexibility and power of a logical framework like Isabelle. It can be applied to any theorem directly, allowing multiple reuse of the abstracted theorem in different settings, as we shall examine in the next section.

3.2.4 Uses of Abstraction

Most directly, we can reapply theorems in different settings; for example, theorems about lists such as

$$\texttt{append\_Nil2} \equiv x \mathbin{@} [\,] = x$$

can be abstracted and reapplied for natural numbers. However, in order to reuse theorems on a larger scale (Isabelle has more then 200 theorems about lists, reusing each individually requires more effort than work saved) a more systematic approach is required.

Change of Data Type Representation

Change of data type representation occurs when we *implement* one data type with another. For example, we may implement the natural numbers by a *binary representation*:

datatype bNat =	*datatype Pos* =
Zero	*One*
| *PBin Pos*	| *Bit Pos bool*

This binary representation has a different induction scheme than the usual representation in terms of zero and successor, and a different recursion scheme (see [98, 99]). However, we can prove that the usual induction as theorems in this theory (rather than stating them axiomatically), and we can further prover the usual recursive equations defining operations like addition. We can then abstract theorems about natural numbers up to the induction scheme, and the recursive definitions, and reapply them in this context. We have used this mechanism to move all theorems about natural numbers and addition from Isabelle's standard presentation to the binary representation [L-9].

Signature and Theory Morphisms

In the previous example, we established a systematic mapping from natural numbers to the binary representation. This approach is captured by the notion of signature and theory morphisms.

In Isabelle, a *signature* $\Sigma = \langle T, \Omega \rangle$ is given by a set T of *type constructors*, and a set Ω of *operations*. Each type constructor $c \in T$ has an *arity* $ar(c) \in \mathbb{N}$. Assuming a fixed and infinite set of type variables $\mathcal{X}$, the set T^* of *types* generated from T is the smallest set containing all the type variables, and closed under application of the type constructors. Each operation $\omega \in \Omega$ has a *type* $\tau(\omega) \in T^*$. Given a signature Σ and a set X of variables together with a type assignment $\tau : X \longrightarrow T^*$, the set $T_\Sigma(X)$ of *terms* is defined in the standard way. Each term $t \in T_\Sigma(X)$ has a type τ, denoted $t : \tau$.

A *theory* $\mathcal{T}h = \langle \Sigma, \mathcal{A}x \rangle$ is given by a signature Σ, and a set $\mathcal{A}x$ of formulae called *axioms*. Typical axioms include constant and data type definitions. By closing the set $\mathcal{A}x$ under derivability, we obtain the *theorems* of $\mathcal{T}h$, written as $Thm(\mathcal{T}h)$. Theories and signatures are organised hierarchically; we build new theories by extending existing ones.

Signature morphisms are maps between signatures preserving operations and arities. They allow us to move terms between signatures. Formally, given two signatures $\Sigma_1 = \langle T_1, \Omega_1 \rangle$ and $\Sigma_2 = \langle T_2, \Omega_2 \rangle$ a *signature morphism* $\sigma : \Sigma_1 \longrightarrow \Sigma_2$ consists of two maps $\sigma_T : T_1 \longrightarrow T_2$ and $\sigma_\Omega : \Omega_1 \longrightarrow \Omega_2$, such that

(i) for all type constructors $t \in T_1$, $ar(\sigma_T(t)) = ar(t)$, and

(ii) for all operations $\omega \in \Omega_1$, $\overline{\sigma}_T(\tau(\omega)) = \tau(\sigma_\Omega(\omega))$,

where $\overline{\sigma}_T : T_1^* \longrightarrow T_2^*$ is the extension of the map between the type constructors to a map between the set of types built from these type constructors.

Theory morphisms are maps between theories. Given two theories $\mathcal{T}h_1 = \langle \Sigma_1, \mathcal{A}x_1 \rangle$ and $\mathcal{T}h_2 = \langle \Sigma_2, \mathcal{A}x_2 \rangle$, a *theory morphism* σ consists of a signature morphism $\sigma_\Sigma : \Sigma_1 \longrightarrow \Sigma_2$, and a map $\sigma_A : \mathcal{A}x_1 \longrightarrow Thm(\mathcal{T}h_2)$ which maps every axiom of $\mathcal{T}h_1$ to a *theorem* of $\mathcal{T}h_2$.

A theory morphism gives us a canonical way of moving theorems from $\mathcal{T}h_1$ to $\mathcal{T}h_2$: first abstract all theorems from $\mathcal{T}h_1$ occuring in the proof of the theorem, then replace type constructors τ with $\sigma_T(\tau)$, and all operation symbols ω with $\sigma_\Omega(\omega)$, and replay the proof in $\mathcal{T}h_2$. Conditions (i) and (ii) ensure that the translated proof term is well-typed.

Analysing Theorem Dependencies

In the example of natural numbers and their binary representation, it was clear that we had to translate the induction scheme, and the defining equations of the operations into the binary representation, and translate them there. In general, if we can identify such a small set of theorems from which all (or many of) the others in the theory are derived, we call them an *axiomatic base*. The axiomatic base divides the theorems into two parts, those from which the axiomatic base derives, and those deriving from the base.

By analysing the proof objects, we also find out dependencies between theorems (i.e. one theorem's proof requires another theorem). These dependencies impose an order on the translation: if one theorem Ψ depends on Φ, then we have to translate Φ first, then Ψ. This dependency between theorems defines the *dependency graph*, which has theorems as nodes and dependencies as vertices. To establish an axiomatic base, we build the transitive closure under dependency, and then check whether the axiomatic base forms a set of articulation points in the graph, i.e. whether their removal makes the graph disconnected.

Of course, finding the axiomatic base requires some intuition. For the natural numbers, the well-known Peano axioms are the starting point. For other datatypes, we may equally try the relevant induction scheme, plus the defining equations for the operations.

This section has shown how we can make theorems more abstract, and how we can reapply those theorems in a different setting. The implementation technique was proof term transformation. Using this technique, we also implemented signature and theory morphisms in Isabelle. These will also be useful as a tool to structure developments in the next section.

3.3 TAS Revisited

In this section, we will describe a refactoring of the TAS system, alleviating the shortcomings identified in Sect. 3.1.3, and integrating the facilities for abstraction and reuse as presented in the previous section.

3.3.1 Development In-The-Large

In Sect. 3.1.1, we have introduced the TAS system, and we have seen how window inferencing models transformational reasoning on the level of theorems. TAS relied on Isabelle's theories to structure the development and specification, but it had no support to transform theories. Here, the implementation of signature and theory morphisms by proof term transformation from the last section can help, as they allow manipulation of theories.

In this approach, a transformation rule is modelled as *parameterised theory*, i.e. a pair of theories $\Theta = \langle P, B \rangle$ such that $P \subseteq B$. We call P the *parameter* and B the *body* of the rule. In order to apply the rule, we *match* the parameter P with an *instantiating theory* I by constructing a theory morphism $\sigma : P \longrightarrow I$ as in Diagram (3.12). The resulting theory

$$\begin{array}{ccc} P & \xrightarrow{\quad\sigma\quad} & I \\ \cap\downarrow & & \cap\downarrow \\ B & \xrightarrow{\quad\sigma'\quad} & R \end{array} \tag{3.12}$$

R will be I extended with the part of B that does not come from P. All theorems in B will be translated into this extension, reusing the proofs from the generic setting of the transformation rule in the concrete context of I. In order to construct the matching theory morphism σ, we have to provide theorems for each axiom of P. In this way, the translated axioms of P become applicability conditions. This notion of parameterised specifications and their instantiation is also found in algebraic specification languages such as CASL [24]; Diagram (3.12) can be more succinctly defined as the push-out of σ along the inclusion of P in B.

3.3.2 Document Presentation

As opposed to TAS' implicit presentation of developments, we propose a *document-centred approach* to software development (aka. literate specification). As a first step towards this, we have integrated our approach into Isar [113, 111], Isabelle's high-level proof language and user interface. Isar's proof language allows the user to write proofs in a structured, formal, and human-readable format; it defines a notation for theories in Isabelle, and for proofs within these theories. Isar's user interface is a system for integrated and interactive proof and document preparation for Isabelle. We are

utilising the second aspect here: using Isar's extensibility, we have introduced new commands for abstraction, transformation rules and so on. For example, abstraction of theorems are realised with Isar's attributes. The following line abstracts the theorem `append2_Nil2` from the type of lists, using the abstraction procedure *ABS_TYPE* mentioned in Sect. 3.2.3:

theorems *TailRec* = *append2_Nil*[*ABS_TYPE "list"*]

However, Isar's document flow is too restrictive in the long run. Because every document has to be fully run through Isar we can only use as much of LaTeX as Isar can parse, and mixing document text with specification is very delicate. A better document flow would be to have one comprehensive document, from which we can on the one hand extract a formal proof script, and which on the other hand can be typeset. The technical and methodological foundations for this literate specification technique can come from the work in the MMiSS project [L-12,L-23]. Since this has not been implemented in TAS yet, we will come back to this when discussing future work in Sect. 5.2.

3.3.3 Example: Divide-And-Conquer Revisited

In order to represent a transformation rule $R = \langle P, B \rangle$ in Isar, we declare two theories, one called *R_parameter* which contains the parameter P, and one called R which extends *R_parameter* and contains the body. Both *R_parameter* and R are regular Isabelle theories. However, in order to guarantee that application of the transformation rule results in a conservative extension, we check that the transformation rule is well-formed, i.e. the body does not contain axioms. The formulation of the divide-and-conquer transformation from p. 42 in this approach is shown in Fig. 3.3.

The advantage of this formulation over the previous one is a clear separation of concerns. In the previous approach, we modelled the parameter signature by type and functions variables in the core theorem (3.8). This has the advantage that it allows Isabelle's higher-order resolution to help with the instantiation, but on the other hand formulae like (3.8) with six free function and four free type variables do become awkward to handle. Moreover, it mixes in-the-large structuring (signatures and theories) and in-the-small structuring by reflecting the former back into the latter. This is elegant, but as there is no way back it means we cannot use any of Isabelle's theory-level declarations, so e.g. a transformation cannot introduce recursive definitions or datatype definitions. Note how in in (3.8) we had to provide an object-logic representation of the recursive function definition (the **let fun**...**in**...**measure** construct), whereas in Fig. 3.3 we can use Isabelle's standard **recdef**.

To apply a transformation rule, we need to construct a theory morphism from the parameter part to the instantiating theory. This is done with the new Isar command **apply_trafo**, which lets the user specify mappings for

theory *DaC_parameter* = *Main* :
typedecl D_f
typedecl R_f
...
consts
  measure :: $D_f \Rightarrow$ *nat*
  primitive :: $D_f \Rightarrow$ *bool*
  dir-solve :: $D_f \Rightarrow R_f$
  g :: $D_g \Rightarrow R_g$
  compose :: $(R_g(R_f R_f)) \Rightarrow R_f$
  ...
axioms
  A-AUX : "$\forall x.\ g_{pre}\ x \longrightarrow g_{post}\ x\ (g\ x)$"
  A-DS : "$\forall x.\ (f_{pre}\ x\ \wedge\ \mathit{primitive}\ x) \longrightarrow f_{post}\ x\ (\mathit{dir\text{-}solve}\ x)$"
  ...
end

theory *DaC* = *DaC_parameter*:
consts f :: $D_f \Rightarrow R_f$
recdef f "**measure**($\lambda x.$ *measure* x)"
  "$f\ x =$ (*if* (*primitive* x) *then* (*dir-solve* x)
    else (*compose* $\circ$ (*prod* $g\ f\ f$) $\circ$ *decompose*) x) "
theorem *DaC*: "$\forall x.\ f_{pre}\ x \longrightarrow f_{post}\ x\ (f\ x)$"
...
end

Figure 3.3: A representation of divide-and-conquer in Isar.

types, operations and the axioms which make up a theory morphism. We only need to provide as much of the mapping as needed to make it unique; the command constructs the rest of theory morphism and checks that the provided mappings satisfy the necessary invariants and hence form a theory morphism. We can additionally rename the resulting operations or theorems from the body. The application of the divide-and-conquer rule, as declared in Fig. 3.3, is shown in Fig. 3.4 with the resulting function renamed to *quicksort*. As the main result of this instantiation we obtain the recursive definition of *quicksort* along with a correctness theorem, namely a proof of $\forall l.$ *qsort-post* l (*quicksort* l).

```
theory QuickSort = Main :
constdefs
qsort-pre :: nat list ⇒ bool                                   -- specification
"qsort-pre ≡def λx. True"                                      -- of QuickSort
qsort-post :: nat list ⇒ nat list ⇒ bool                       -- by pre- and
"qsort-post ≡def λli lo. permutation(li, lo) ∧ sorted lo"      -- postcondition
...
qsort-aux :: nat ⇒ nat
"qsort-aux ≡def id"
qsort-dsolve :: nat list ⇒ nat list
"qsort-dsolve ≡def id"
qsort-measure :: nat list ⇒ nat
"qsort-measure ≡def length"
qsort-compose :: (nat nat list nat list) ⇒ nat list
"qsort-compose ≡def λ(x, l, r). l @ [x] @ r"
...
theorem a-ds : "∀l. (qsort-pre l ∧ qsort-prim l) ⟶
                     qsort-post l (qsort-dsolve l)"
...
apply_trafo( ("DaC_parameter", "DaC"),                         -- traforule
             [("D_f" ↦ "nat list"),
              ("D_g" ↦ "nat"), ... ],                          -- type map
             [("compose" ↦ "qsort-compose"), ... ],            -- operation map
             [("A-DS" ↦ "a-ds"), ... ],                        -- theorem map
             [("f" ↦ "quicksort"), ... ] )                     -- list of renames
...
end
```

Figure 3.4: Application of divide-and-conquer in Isar.

3.3.4 Deriving New Transformation Rules

One problem with transformation systems in general is how to come up with transformation rules. The solution proposed here, and our long-term goal, is to derive new rules by generalising existing developments.

In general, a transformation rule Θ can be constructed from an ordinary unparameterised theory T by identifying a subtheory of T as the rule parameter P, i.e. $\Theta = \langle P, T \rangle$. The parameter should contain type declarations and operations, and the body should contain the main proofs of the original theory; exactly how to split T is a design decision. Some theorems of T can be stated as axioms of P, turning them into applicability conditions. This way, various assumptions from the body may be collected in the parameter. Free term or type variables in the body can be replaced by operations or type declarations from the parameter.

We will demonstrate this approach by deriving a (well-known) transformation rule from a simple refinement proof. Consider a function which adds up a list of natural numbers. A simple recursive definition is

$$\textbf{recdef}\ \mathit{sum}\ x \stackrel{\mathit{def}}{=} \mathit{if}\ x = [\,]\ \mathit{then}\ 0\ \mathit{else}\ \mathit{hd}\ x + \mathit{sum}(\mathit{tl}\ x).$$

An more efficient tail-recursive definition can be given as follows

$$\textbf{recdef}\ \mathit{sum2}(x, y) \stackrel{\mathit{def}}{=} \mathit{if}\ x = [\,]\ \mathit{then}\ y\ \mathit{else}\ \mathit{sum2}(\mathit{tl}\ x, y + \mathit{hd}\ x),$$

and we can prove that the two are equivalent by showing that for all lists x,

$$\mathit{sum}(x) = \mathit{sum2}(x, 0), \tag{3.13}$$

In Isabelle, the proof proceeds by induction on x using three lemmas, the most involved of which is $\forall x\, a\, l\, .\, \mathit{sum2}(l, x + a) = x + \mathit{sum2}(l, a)$.

With our abstraction procedure *ABS_TYPE* from above, we can generalise this theorem away from the type *nat*, and replace it with a type variable α; 0 and + are replaced by schematic variables *zero* and p (typed as $\mathit{zero} :: \alpha$ and $p :: \alpha \Rightarrow \alpha \Rightarrow \alpha$), and *sum* and *sum2* by schematic variables *sum* and *sum2*, respectively. The procedure results in the following abstracted theorem *TailRec*:

$$\begin{array}{l}
(\forall x.\ \mathit{sum}\ x = \mathit{if}\ x = [\,]\ \mathit{then}\ \mathit{zero}\ \mathit{else}\ p\ (\mathit{hd}\ x)(\mathit{sum}\ (\mathit{tl}\ x)) \\
\ \wedge\ \forall x\ y.\ \mathit{sum2}(x, y) = \mathit{if}\ x = [\,]\ \mathit{then}\ y\ \mathit{else}\ \mathit{sum2}\ (\mathit{tl}\ x, p\ y\ (\mathit{hd}\ x)) \\
\ \wedge\ \forall u.\ p\ \mathit{zero}\ u = u\ \ \wedge\ \ \forall u.\ p\ u\ \mathit{zero} = u \\
\ \wedge\ \forall u\, v\, c.\ p\ u\ (p\ v\ c) = p\ (p\ u\ v)\ c) \\
\qquad \longrightarrow\ \mathit{sum}\ x = \mathit{sum2}(x, \mathit{zero})
\end{array}$$

In *TailRec*, we find the definitions of the two functions as the first two premises. In particular, the second line is a *schematic definition* of the tail-recursive function, which will be instantiated when the transformation

theory *TalRec_parameter* = *Main* :
typedecl *a*
consts
zero :: *a*
p :: *a* ⇒ *a* ⇒ *a*
sum :: *a list* ⇒ *a list*
axioms
ax1 : "*sum x* = *if x* = [ ] *then zero else p* (*hd x*)(*sum* (*tl x*))"
ax2 : "*p zero u* = *u*"
ax3 : "*p u zero* = *u*"
ax4 : "*p u* (*p v c*) = *p* (*p u v*) *c*)"
end

theory *TailRec* = *TailRec_parameter*:
primrec
sum2-nil : "*sum2*([ ], *y*) = *y*"
sum2-cons : "*sum2*(*x*#*xs*, *y*) = *sum2*(*xs*, *p y x*))"
theorem *TailRec* : "*sum x* = *sum2*(*x*, *zero*)"
...

Figure 3.5: The transformation rule derived from theorem *TailRec*.

rule is applied. The remaining three premises reflect those properties of the natural numbers that were needed for the proof of the original theorem, namely that $(0, +)$ forms a monoid. All of this was derived automatically by the abstraction procedure.

We can now make *TailRec* a transformation rule by making the parameter theory explicit (see Fig. 3.3.4), and reapply it as shown in the previous section; for example, the signature morphism

$$\sigma = \{\ sum \mapsto concat, sum2 \mapsto sum2, zero \mapsto \varepsilon, p \mapsto \mathbf{Cat}\ \}$$

gives us a tail-recursive formulation of the function *concat*, which concatenates lists of strings.

Note that the abstracted theorem *TailRec* is useful in its own right; with the substitution

$$[\, \lambda l\,.\,(foldr\ f\ l\ e)/sum, \lambda(l, e)\,.\,(foldl\ f\ e\ l)/sum2\,]$$

we get the following generally useful theorem

$$(\forall u\,.\,f\ e\ u = u \wedge \forall u\,.\,f\ u\ e = u \wedge \forall u\ v\ c\,.\,f\ u\ (f\ v\ c) = f\ (f\ u\ v)\ c) \\ \longrightarrow\ foldr\ f\ x\ e = foldl\ f\ e\ x$$

which states that folding from the left, and folding from the right are equivalent if the arguments of the fold form a monoid.

A more extensive case study has been undertaken in [L-1], where we have demonstrated how to derive transformation rules from refinement proofs. The main example was a transformation rule derived by generalising the well-known data refinement from stacks as lists to stacks as arrays with a pointer.

This concludes our overview of the ongoing redesign of TAS. We have seen how theory morphisms form the missing like to support development in-the-large, taken first steps to a document-centred approach, and integrated the work on abstraction to allow users to derive new transformation rules. The document-centred approach requires reengineering the user interface as well, as we will see in the next chapter.

3.4 Conclusions

This chapter has presented the foundations of formal program development. We have made three specific choices here: we use *higher-order logic*, because it is simple, powerful and allows for generalisation and abstraction; we work in *logical frameworks*, because one should be able to switch object logics while keeping the prover infrastructure, and because it allows us to use domain-specific languages tailored to a specific purpose; and we use *Isabelle* as our theorem prover, because it is a powerful, generic, easily extendible higher-order prover.

Formal software development in Isabelle requires support for development in-the-small and development in-the-large. The former corresponds to the formal proof of theorems, except that we do not necessarily know the goal of the proof; here, *meta-variables* can serve as placeholders to be filled in during the development. In order to model transformational development, we implemented a window inferencing package in Isabelle. The package is generic over the refinement or logic used, and we have seen instantiations for refinement along model restriction (i.e. inverse implication) in higher-order logic, and for failure-divergence refinement in the process calculus CSP. These two rather different examples show the usefulness of a generic approach. The window inferencing package, together with a graphical user interface to be presented in the next chapter, forms the TAS system.

One major untapped potential of formal software development is the systematic generalisation of developments in order to make them applicable in a wider context, which we call *abstraction for reuse*. In order to support the abstraction of whole developments, we have developed abstractions for theorems, abstracting over assumptions, operations and types. We have implemented these methods in Isabelle by transforming proof terms.The advantages of this approach are that we can add it as a conservative extension to Isabelle, without changing its logical kernel and thus jeopardising its con-

sistency. This technique was also useful to implement signature and theory morphisms in Isabelle.

The lack of support for abstraction, a lack of support for development in-the-large and a rather implicit representation of the specification were identified as the three shortcomings of TAS, and have lead to a refactoring of the system which moreover integrates the abstraction procedures. The implementation of theory morphisms provides the missing link for structuring development of theories; transformation rules can now be modelled as parameterised theories, with application of transformation rules by a push-out construction.

In order to rectify the third deficiency and support a document-centred approach to formal software development, we integrated our approach into Isabelle's command language and user interface, Isar. However, the resulting document flow is still too restricted in the long run, as all documents have to pass through Isabelle; we are planning to amend this with the techniques for document structuring from the MMiSS project.

3.4.1 Bibliographical Remarks

TAS was based on the system YATS [91], which implemented transformation rules by proven theorems but crucially had no support for window inferencing rules.

The TAS system underwent two refactorisations. The initial version of TAS was a direct descendant of the YATS system [91]. Instead of using window inferencing, it used Isabelle directly to discharge proof obligations. This system was developed during the UniForM project, and presented on the TAPSOFT'97 [L-21] and ETAPS'99 [L-17] conference.

The first refactoring made TAS into a generic window inferencing system, and was presented at TPHOLs'00 [L-16] and ETAPS'00 [L-15] conferences.

In order to address the issue of reuse and abstraction, the project AWE[4] was started in 2000, supported by the German research council DFG. The methods of proof reuse ([L-9]), implementation of transformation rules as parameterised theories the [L-24] and the abstraction case study [L-1] all resulted from the first phase of this project. A second phase of this project, in which we intend to finish the refactorisation of TAS by moving towards fully towards the document-centred approach, is scheduled to start this year.

[4] *Abstraktion und Wiederverwendung formaler Entwicklungen* (Abstraction and Reuse of Formal Developments)

Chapter 4

Tool Development

This chapter will deal with the technical side of tool development. We will discuss issues of design, system architecture and implementation arising when implementing tools for formal methods, summarising the experience with the development and design of various tools the author has taken part in, such as the generic transformation system TAS [L-16,L-7], the UniForM workbench [L-18,L-19], the GUI libraries sml_tk [L-35] and HTk, the MMiSS repository [L-12], the CASL consistency checker CCC [L-10], or the generic prover interface PG Kit [L-25]. We will present some of these systems as case studies to illustrate the discussion.

We have mostly used functional programming languages in the implementation of these tools, so we will argue why functional languages are a good choice for tool development; the key arguments are higher productivity, strict typing and better structuring facilities.

We will then consider *tool integration*. This becomes relevant when one wants to reuse tools written by other research groups or vendors. In the previous chapter, we have argued for using domain-specific languages by encoding them into a general meta-logic; this means that we also want to reuse tools for these methods (e.g. a model checker), so tool integration becomes an issue.

The design of user interfaces for formal tools has been a rather neglected field for a long time. Typically, formal method tools come with command-line interfaces aimed at the expert (i.e. the implementor of the tool). This has been successful in the past as long as only experts would use these tools, but as formal methods are in the process of becoming more and more mainstream in computer science, more widely accessible interfaces are required. We will first discuss the design of graphical user interfaces for provers, and then the next generation of prover interfaces (the PG Kit), which allows a synthesis of textual and graphical user interfaces.

4.1 Advantages of Functional Programming

Most of our implementation work involves functional programming languages, specifically Standard ML (SML, [104]) or Haskell [118]. The reasons for this are threefold:

- By programming at a more abstract leve, functional languages are well-suited to rapid prototyping. This is particularly useful in the research context, where a working prototype or proof-of-concept is more important than an industrial-strength production-ready tool.
- Functional languages are particularly suited for the kind of symbolic computation that characterises formal development and proof.[1] For extending Isabelle (e.g. the window inference package for TAS), SML is the logical choice.
- SML and Haskell have *strict typing*, allowing a more stringent system design which can increase confidence in the correctness of the tool. Further, modern functional languages have *sophisticated structuring mechanisms*, such as SML's functors or Haskell's type classes, which allow flexbile and exact ways of structuring the implementation.

Empirical evidence suggests that productivity with functional languages is about three times as high than with usual (imperative or object-oriented) programming languages. However, in day-to-day industrial software production functional languages are still the exception rather than the rule. Reasons for this apparent contradiction include that coding is in fact only a small portion of software development, that functional languages do not enjoy the same support as for example Java in terms of libraries and tools, that functional languages are almost exclusively taught at universities (so functional programmers are comparatively expensive), and finally that management is technologically agnostic and inherently conservative ('Nobody ever got fired for using C++'). Nevertheless, there are industrial success stories involving functional programming, such as Ericsson's Erlang [6].

4.1.1 LCF Architectur: The CCC

One can use the typing discipline of a strictly typed functional language to increase confidence in the tool. One of the first tools to make use of this was the LCF prover. In LCF, theorems were implemented as an abstract datatype, and all operations on this datatype corresponded to correct logical inferences. Thus, new theorems could only be produced by applying

[1] Indeed, Standard ML started out as the command language for the LCF prover [58]; ML stands for *meta language*, i.e. the command language as opposed to the object language.

correct inference rules to existing theorems. This design, the so-called *LCF architecture*, is still used with provers derived from the early LCF system, such as HOL [59] or Isabelle [113]. The LCF architecture can also be good design if building a new tool from scratch.

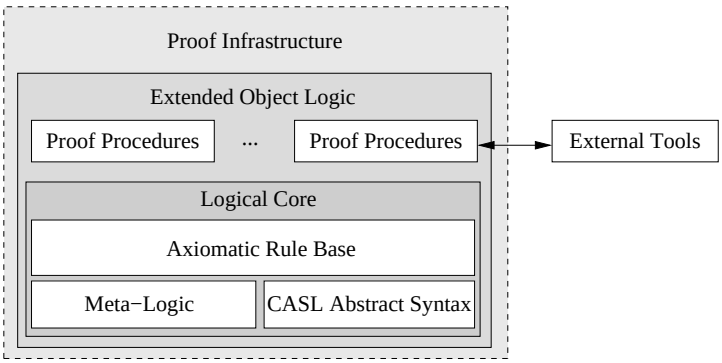


Figure 4.1: Simplified view of the CCC system architecture.

The CASL Consistency Checker (CCC, [L-10]) has an LCF-style architecture. CCC is a faithful implementation of a calculus for consistency proofs of CASL specifications [128]. The calculus comprises more than fourty rules, organised in four different subcalculi. When implemening a system to conduct proofs in this calculus, we needed to ensure the following invariants: firstly, that rules are only combined in a safe way; secondly, that no further rules can be added axiomatically; and thirdly, that proof procedures needed to discharge side conditions such as static typing of specifications can only be used for this side conditions, not arbitrary propositions. These objectives were achieved by a three-tier architecture (Fig. 4.1 shows a simplified view).

The core of the system is the meta-logic, a weak fragment of conjunctive logic which allows us to write down *rules* with a number of premises and one conclusion, to compose these rules and instantiate variables therein. Premises and conclusion are given by *propositions* (corresponding to statements to be shown, such as *this specification is consistent*). Rules and propositions are encoded by datatypes `prop` and `rule`. The meta-logic, together the axiomatically assumed rules and the CASL abstract syntax form the *logical core*, the innermost layer of the system. The second layer implements proof procedures. By putting this in a separate layer from the logical core, we ensure that no decision procedure, however erroneously implemented, can prove arbitrary propositions; it may only wrongly prove the particular proposition it was implemented for. The outermost layer contains proof infrastructure, e.g. derived rules, a database to hold proven propositions, and facilities for goal-directed proof and automatic proof procedures.

4.1.2 Typing in an Untyped World

Typing and structuring can also be useful when we integrate a foreign library. For example, sml_tk and HTk are encapsulations of the graphical user interface toolkit and command language Tcl/Tk [115] in SML and Haskell, respectively. Tk is the interface toolkit, implementing of a number of interface elements (called *widgets*). Tk has a C API, or it can be programmed in the tool command language Tcl, an untyped scripting language. Both together form the *wish*, the window shell.

In Tcl/Tk, we first create widgets, and then we *configure* them. For example, we can create a button, and label it with some text and a colour. Tk knows about thirty widgets and at least twenty different configurations, and not every configuration can be used on every widget. Because Tcl is completely untyped, using an illegal or malformed configuration results in a runtime error, so it is advantageous to ensure by typing at compile time that this can not happen.

In HTk, we can do so by using Haskell's type classes. Each configuration option corresponds to a class, with the class functions setting or querying the option. Each widget is represented by an abstract datatype, which is an instance of the class if and only if that option is allowed on that widget.

For example, text configuration is given by this class:

```
class (GUIObject w) => HasText w String where
      text    :: HasText w => String -> Config w
      getText :: HasText w => w -> IO String
```

`Config w` is a synonym for `w-> IO w`, the action which sets the configuration. All widgets which instance `HasText` can be configured with a text. For example the type `Button` is an instance of `HasText`, as buttons can be labelled with a text, but the type `ScrollBar` representing scroll bars is not, as scroll bars cannt be labelled.

Not only can we use typing to make programming the interface less error-prone, we can also use an implementation of concurrent events as composable first-class values in Haskell [130] to allow a more abstract, modular and composable modelling of the behaviour of the user interface. This is a significant abstraction over Tcl/Tk, where the behaviour is implemented using call-back functions.

4.1.3 Structuring with Functors and Classes

In SML, modules are called *structures*, interfaces are called *signatures*, and *functors* are parameterised modules. Functors are quite a powerful structuring concept, unmatched by most other programming languages; the closest are templates in C++ (and in Java, as of lately) or generic packages in Ada, but functors are more versatile and type-safe, as anything can be the argument of a functor, including datatypes and other structures.

The first good example of a functor is the window inferencing package `Trafos` used for the TAS system. As mentioned in Sect. 3.1.1, the `Trafos` package is parametric over a family of reflexive, transitive and monotone relations. In order to describe such a family of of relations, we need to give the *theorems* stating reflexivity, transitivity and monotonicity (and the conversions between the different relations given by the weakening). This is described by the following SML signature:

```
signature TRAFOTHY_SIG =
  sig val refl  : thm list
      val trans : thm list
      val mono  : thm list
      val refl  : thm list
  end
```

Note that we do not need to give the actual names of the relations, as these can be found out by investigating the theorems. The `Trafos` package is now implemented as a functor

```
functor Trafos(structure TrafoThy : TRAFOTHY_SIG) : TRAFOS_SIG =  ...
```

where `TRAFOS_SIG` describes the export interface of the `Trafos` package, implementing essentially three functions to open and close subdevelopments, and to apply transformations given by their core theorem (`thm`), where `path` specifies the path in the subgoal term to the subdevelopment to be opened:

```
signature TRAFOS_SIG =
  sig val open_tac   : path -> int -> tactic
      val close_tac  : int -> tactic
      val apply_tac  : thm -> int -> tactic
```

By instantiating the functor `Trafos` with different theorems, we get different instances of the generic window inferencing. This use of functors is good practice. For example, in Isabelle functors are used to implement automatic proof procedures in a generic way across different object logics (e.g. a rewriting engine, a tableaux prover or an induction package).

Functors can also be used to describe program construction. In this view, a functor is a parameterised module, but it rather constructs a new structure from the arguments. This point of view is particularly useful when considering refinement, as in Extended ML [132, 81] or refinement in CASL [107]. However, one criticism levelled at SML functors is that they lead to a program structure which is hard to understand; indeed, the revised version of the Standard ML definition (SML'97, [104]) restricted the use of sharing constraints to allay this criticism.

4.2 Tool Integration

Tool integration has many aspects. The ECMA reference model [46] distinguishes framework integration, data integration, control integration, presentation integration and process integration as different areas of integration which cover such different aspects as a common data exchange format, similar appearance and user interaction, or integration into a common development or business process. Here, we want to cover the first three: *framework integration*, which means basic interoperability, *control integration*, which means the ability to combine the functionalities, and *data integration*, which is the ability to share and exchange data between tools.

4.2.1 Framework Integration

Framework integration covers a wide area, but here, we are concerned with the integration on a technical level: how do we get our tools to run together and talk to each other?

There are many ways to do this. We can make use of a component framework such as CORBA, COM+ or JavaBeans, but these are fairly heavyweight if we just want to connect a modelchecker with a theorem prover, as we have to wrap up each tool into a component.

Tight Coupling

By tight coupling, we mean that we combine the tools into a single program and process by linking them into one executable. The advantages of tight coupling are the compactness of the resulting tool. It is also the most efficient way of combining tools, as there is nearly no communication overhead (only marshalling if we convert data between different programming languages). The disadvantages are technical complications (e.g. name clashes, different calling conventions, cross-language interoperability) and a lack of modularity in the development, as we have to track every minor change in the interface of the tool. A further drawback of the lack of modularity is the resulting instability (e.g. when one application diverges the whole tool does).

For these reasons, tight coupling should only be used with stable, mature tools which are called frequently, or where a lot of data is exchanged across calls. It is mostly useful with libraries or toolkits. For example, the MMiSS repository [L-12] uses the Berkeley DB database [138], which consists of a library with a C interface which can be linked into executable. For Haskell, interfacing such libraries is a trivial effort, thanks to a standardised foreign function interface [37].

Loose Coupling

A more light-weight method is to use *loose coupling* and run each tool as a separate process, communicating over channels such as pipes or sockets. The communication protocol is not given by the framework, but depends on the tools involved. For a simple point-to-point connection, an ad-hoc protocol may be used.

For example, both sml_tk and HTk use loose coupling to run Tcl/Tk's main proces (the window shell *wish*) over a pipe. The communication is in Tcl, which is easy to produce and read from SML and Haskell. Here, the loose coupling has definitely been an advantage, as Tcl/Tk has gone through a number of major revisions during the lifetime of sml_tk and HTk, with wide ranging changes in the C API (it can also be linked into the client program), yet the Tcl interface has remained stable.

The Prosper toolkit [40] also uses loose coupling to integrate formal method tools into a common framework. However, special precaution has been taken to route interrupts around the system by developing a purpose-built middleware, the Prosper Integration Interface (PII). The inability to send asynchronous messages (such as interrupts) over pipes or sockets is one of drawbacks of loose coupling, together with performance problems arising from the communication overhead. However, the advantages outweigh these problems in most cases: added stability, modularity in development, and compositionality — we can run different tools on different machines for better performance, or to avoid platform problems.

4.2.2 Data and Control Integration

Data integration is concerned with the exchange of data between tools, and control integration is about combining the functionalities of single tools into a meaningful whole. When we use a component framework, the format of the data, and the interfaces of the tools (the available functions and their signature), will be described in a particular interface language like IDL for CORBA, or even in a programming language, like Java interfaces for Java Beans.

Without a component framework, we need another way to define exchange formats, and a good choice here is XML [153]. Nearly all progamming languages have tools to support XML parsing and generation (for the functional languages we are interestd in, there is FXP for SML [110] or HaXml for Haskell [159]). By defining the format of messages in an XML document type definition (DTD), XML schema, or a Relax NG schema [125] (our preferred option), a precise definition of the data format is given, and it is easy to do sanity checks on both sides. The PG Kit framework (see Sect. 4.4 below) shows an example of such a setup.

In fact, under the slogans of *web services* and *service-oriented architec-*

ture (SOA), loosely coupled components talking in XML (in particular, in a standardised XML meta-format like SOAP or XML-RPC) have recently become *de rigeur*.

The MMiSS repository [L-12] also uses XML for data exchange. Documents are exchanged in an XML format called MMiSS-XML, and externally converted to and from a LaTeX dialect. The use of XML here has been beneficial, as it allowed easy integration of the XML-based ActiveMath tool suite [101]. ActiveMath uses the XML format OMDOC internally, making communication with MMiSS-XML was just a matter of defining the right XSLT stylesheet.[2] This is an advantage over the old UniForM workbench [82], where the data to be exchanged was modelled using Haskell datatypes, which limited portability and made it cumbersome to integrate other tools.

4.3 Interface Design

When it comes to interface design, graphical user interface (GUIs) have come to be seen a necessity *eo ipso*. This is not necessarily the case; what matters is good interface design, text-based or graphical. General design guidelines for user interfaces can be found in the European Norm EN ISO 9241-10 [48]. Graphical user interfaces, if they are designed badly, can actually decrease usability, and it has been argued that graphical user interfaces for theorem provers are a bad idea [102]; the gist of that argument is that they invite users to play around without purpose or planning proofs ahead, leading to an overall loss of quality in the proof work. (The same argument has been put forward against the use of debuggers in programming.) On the other hand, there are strong arguments for graphical user interfaces: they allow mathematics to be rendered in the traditional typeset form, lightening the user's cognitive load; having to remember an unfamiliar command-line syntax while at the same time struggling with the mathematics may distract users from their main aim, namely proving. There is not enough taxonomic data for a conclusive answer yet, but initial studies suggest that a well-designed GUI can increase productivity [76, 1].

Design Principles

Graphical or not, what we need is some good interface design principles. Designing a graphical user interface should be more than 'bolting a bit of Tcl/Tk onto a text-command-driven theorem prover in an afternoon's work' [26]. Arguably, it is harder to design a good graphical user interface, as it is easy to overwhelm the user with unnessary information, buttons and menues.

[2] Roughly spoken, an XSLT stylesheet defines rewrite rules which transform documents from one XML format to another.

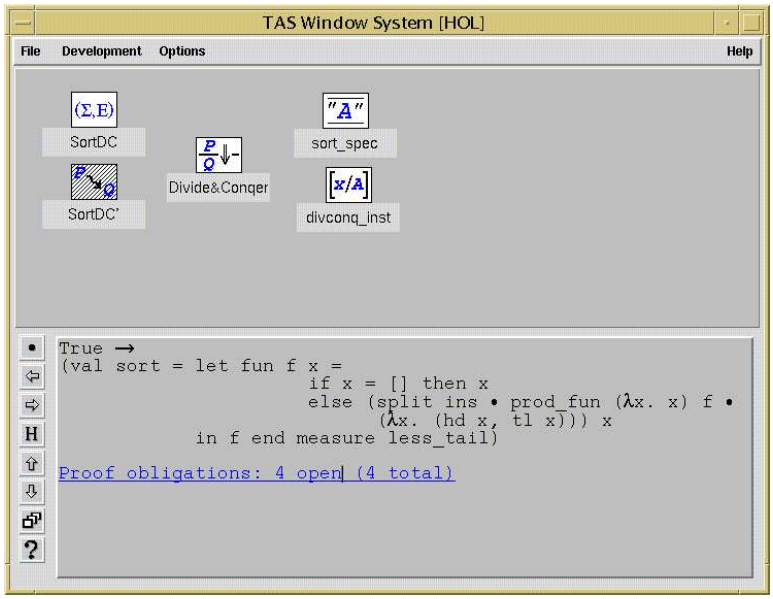


Figure 4.2: The TAS user interface: the upper half of the window shows the *notepad*, with icons representing theories, transformations and specifications; the lower half of the are is the construction area, where a development is currently in progress.

An example of a well-designed interface is Jape [27, 28], which was designed particularly to be *quiet*, i.e. convey exactly as much information as needed.

Another widely acknowldged principle of good user interface design is *direct manipulation* [137, 154, 44], a term attributed to Shneiderman [136]. It is characterized by *continuous representation* of the objects and actions of interest with a *meaningful visual metaphor* and *incremental, reversible, syntax-free operations* with *rapid feedback* on all actions.

The design of the graphical user interface for TAS [L-7] follows this principle. It is based on the visualisation metaphor of a *notepad*, which is in turn motivated by the way we do everyday mathematics and calculations: one typically uses a piece of paper or a blackboard to write down intermediate results, calculations or lemmas, but overall in an unstructured way, adding a column of numbers in one part of the pad, while doing a multiplication in the lower corner and a difficult diagram-chase in the middle.

Thus, we have an area of the screen, the notepad, where we can manipulate objects visualised by *icons*. Objects are *typed*, with the type determining the icon and the possible operations. Operations are either effected by a pop-up menue, or by drag&drop, in which case the types of all objects involved are taken into consideration. (E.g. dropping a transformation onto an ongoing development applies that rule.) Objects can also be opened in

the *construction area*, where their inner structure can be manipulated (in particular, where ongoing proofs and developments are conducted). Fig. 4.2 shows a screenshot of TAS' user interface.

This user interface is *functional*: objects do not have an internal state, and gestures correspond to applying operations, producing new objects. Thus, we gradually build up terms (in the notepad metaphor above, we do not overwrite existing results, altough we may hide previous results from the user). The sequence of operations leading to the construction of a particular object is called its *construction history*. The functional character of the interface makes undoing operations trivial. It allows systematic replay, and even a limited form of reuse, by cutting out parts of the construction history and replaying them on other objects.

TAS has a point&prove-functionality, i.e. one can select subterms with the mouse, and apply an operation (such as opening a window, or applying a rewrite rule) on this subterm. This was implemented using *annotations*, i.e. user-invisible markup on the proof terms, which required a number of technical modifications and extensions to Isabelle's pretty-printer, which are not easily portable across Isabelle versions; for this reason, TAS has not been ported to Isabelle's later versions.

The Generic Graphical User Interface

TAS is in fact one instance of the generic graphical user interface `GenGUI`, which is implemented in SML using sml_tk. The definition of objects, types and operations is collected in an application structure characterised by a signature `APPL_SIG`. `GenGUI` is implemented as a *functor*, which given such a description, returns an implementation of the GUI comprising notepad, construction area, history display etc. A generic sessions management is also provided.

`GenGUI` has two main instances, the window inferencing system TAS, and IsaWin, a graphical user interface to Isabelle itself; the difference between IsaWin and TAS is that IsaWin is based on Isabelle's native calculus, natural deduction with meta-variables, whereas TAS is of course based on window inferencing.

Conclusions

While TAS and IsaWin are usually met with initial approval, in particular from theorem proving neophytes, it has some drawbacks. From the user's point of view, it has a rudimentary management of target proof scripts, and integrates foreign proof scripts only reluctantly. From the developer's point of view, customising or adapting it to other proof assistants requires Standard ML programming, and a good understanding of the data structures. It is not possible to build a new instance gradually, and it is hard to connect

to provers not implemented in Standard ML. From an architecture view, the tight coupling between prover and interface makes the interface difficult to reuse, and it also makes the interface less robust (see the discussion in Sect. 4.2.1 above). For these reasons, not many different adaptations of `GenGUI` exist, and in comparison with other interfaces, `GenGUI` has not fully delivered on its promise of genericity.

Also, the basic metaphor of the notepad bears some critical reflection: when writing a mathematics proof, we may use a notepad or blackboard to sketch calculations or diagrams, but in the end we do not want a proof scribbled all over a notepad, we instead want to produce a linear, well type-set, easy to read document which which we can print out and show to our peers.[3] Similarly, when writing a specification or verification, we want something to show for our trouble. Arguably, this is the most important aspect of formal specification: it establishes a verified basis for communication between all participants in the development process. This is the *document-centered* approach introduced above in Sect. 3.3.2. It does not mean we have to get rid of the notepad completely, it is still useful for side-calculations and to organise the display, but the main focus of the interface should be the documents representing the development we are producing.

In summary, TAS and IsaWin have been a successful design study. The feasibility of the approach has been shown, and its strengths and weaknesses highlighted. Now it is time for the next step.

4.4 The Next Generation

In order to refactor our design, it makes sense to look at other interfaces and see if we can combine their advantages with ours, hopefully sharing the implementation work in the process. An interface which has been very sucessful with respect to genericity is *Proof General* [7, 9]. It has instances for three major provers, and is document-centered in so far as it is built on the Emacs text editor, and in short excels in the areas where `GenGUI` is lacking. However, we can combine both designs into the next generation of prover interfaces, making `GenGUI` to a truly generic system and overcoming the deficiencies of the first implementation while keeping its strengths.

4.4.1 Proof General

Proof General is a generic interface for interactive provers built on the Emacs text editor. It has proved rather successful in recent years, and is popular with users of several theorem proving systems. Its success is due to its genericity, allowing particularly easy adaption to a variety of provers (such as

[3] An old insight: 'Denn was man schwarz auf weiß besitzt, kann man getrost nach Hause tragen.' [56]

Isabelle, Coq, LEGO, and many more), and its design strategy, which targets experts as well as novice users. Its central feature is an advanced version of script management [23], closely integrated with the file handling of the proof assistant. This provides a good work model for dealing with large-scale proof developments, by treating them similarly to large-scale programming developments. Proof General also provides support for high-level operations such as proof-by-pointing, although these are less emphasised.

Although successful, there are drawbacks to the present Proof General. From the users' point of view, it requires learning Emacs and putting up with its idiosyncratic and at times unintuitive UI. From the developers' point of view, it is rather too closely tied with the Emacs Lisp API which is restricted, somewhat unreliable, often changing, and differs between different flavours of Emacs. Another engineering disadvantage of the present Proof General arose from its construction following a product-line architecture, by successively generalising a generic basis to handle more provers. This strategy meant that little or no specific adjustment of the provers was required, but it resulted in an overcomplicated instantiation mechanism.

4.4.2 The PG Kit Project

To address the limits of the existing Proof General model, and particularly of the Emacs Lisp implementation, the Proof General Kit (PG Kit) has been conceived [8].The central idea is to use the experience of connecting to diverse provers to prescribe a uniform protocol for interaction. Instead of tediously adapting Proof General to each prover, Proof General calls the shots, by mandating a uniform *protocol for interactive proof*, dubbed *PGIP*, which each prover must support. Although initially designed for a textual interface, we soon found that the PGIP protocol could easily be extended to cover graphical interaction as well.

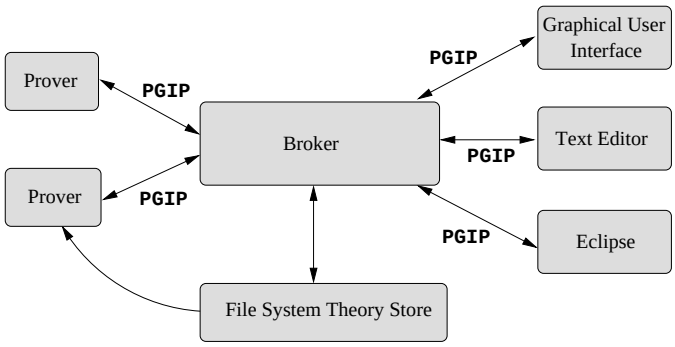


Figure 4.3: PG Kit Framework architecture

PG Kit is a *software framework*, a way of connecting together interact-

ing components customized to the domain of interactive theorem proving. The framework has three main component types: interactive *prover* engines, front-end *display* components, and a central *broker* component which orchestrates proofs-in-progress. The architecture is pictured in Fig. 4.3.

The components communicate using messages in the PGIP protocol. The general control flow is that a user's action causes a command to be sent from the display to the broker, the broker sends commands to the prover, which sends responses back to the broker which relays them to the displays. The format of the messages is XML, defined by a Relax NG schema. Messages are sent over channels, which are sockets or Unix pipes.

The PGIP protocol specifies both the interaction between provers and the broker, and between the broker and displays. We distinguish between several *kinds* of messages, such as:

- *Display commands* are sent from the display to the broker, and correspond to user interaction, such as start a prover, load a file or edit text.
- *Prover commands* are sent to the prover, and may affect the internal (proof-relevant) state of the prover.
- *Output messages* are sent from the prover or broker, and contain output directed to the user. A display model gives hints where and how the messages are displayed: in a status line, a window of their own, or a modal dialog box.
- *Configuration messages*, used for setting up components.

Other message kinds include system inspection and control commands, and meta data sent from the prover, for example dependency information between loaded files and proven theorems.

On the prover side, the broker has an abstract view of the internal state of the prover. The prover occupies one of four fundamental states, namely the *top level* state where nothing is open yet, the *file-open* state where a file is currently being processed, the *theory-open* state where a theory is being built, or the *proof-open* state where a proof is currently in progress. The reason for distinguishing the states is that different commands are available in each state, and that the prover's undo behaviour in each state can be different. This model is based on abstracting the common behaviour of many interactive proof systems, but it is not intended to capture precisely the way every proof system works. Rather it acts as a clearly specified 'virtual layer' that must be emulated in each prover to cooperate properly with the broker. With the help of the Isabelle development team, a PGIP-enabled version of Isabelle has been implemented.

On the display side, we have refined the simple linear script management model [23] to handle non-linear dependencies as well. The broker has to

translate this non-linear model into a sequence of linear operations for the prover. We assume that the source text is split into several spans of text, each of which can have one of five different states: unparsed, parsed, being processed, processed, or outdated. The transitions between these states corresponds to sending messages to the prover. A span of text always starts out as unparsed, then becomes parsed, then if the command is sent to the prover it becomes being processed and if the evaluation finishes successfully, it becomes processed. This the edit-parse-proof cycle [L-32].

The basic principle for representing proof scripts in PGIP is to use the prover's native language and *mark up* the content with PGIP *prover commands* which expose some structure of the proof script which is needed for the interface. This also means that the broker has to leave parsing to the prover (or a separate tool specific to the prover in question).

Thus, the central artefact under construction are textual *proof scripts*, as per our document-centered paradigm. But we can also accomodate graphical user interaction into this, we only need to translate user gestures into operations producing text. To this end, PGIP allows the prover to define types and operations, where types define icons and possible operations, and operations are triggered by gestures. This is in the same vein as `GenGUI`, but here operations are translated into actual proof text, which is sent back to the prover and executed. Thus, for Isabelle dropping a theorem named th_1 onto a theorem named th_2 results in the text th_1 `RS` th_2 to be produced and sent to Isabelle, whereas dropping a theorem named th onto the current proof results in `by (rule` *th*`)` to be inserted in the proof text. Because the text produced by operations is specific to the prover in question, each prover has to configure a graphic display; it does so with the configuration messages mentioned above.

By reimplementing IsaWin as the PGWin display in the PG Kit framework [L-25], we have an interface which combines textual and graphical interaction. It allows the user to enter text both via the keyboard or via mouse gestures. Presently, finishing touches are put on the implementation of this display. Two other displays are currently implemented, one based on Emacs and one as a plug-in for the Eclipse IDE [L-33], enabling both Emacs and Eclipse as displays for PG Kit.

4.5 Conclusions

This chapter has dealt with the engineering side of tool development, with an emphasis on tool integration and the design of user interfaces.

Tool integration has many aspects. We covered platform integration, where we compared tight and loose coupling, i.e. linking together into one executable vs. processes running separately, connected via pipes or sockets. The former, although faster, requires more effort both in development and

maintenance, and in our experience is only appropriate with libraries or toolkits with which a very tight interaction is required, and which are stable (not subject to changes in the interface). In general, loose coupling is our preferred option, as it is more versatile, easier to set up and more stable to run, because there is no single point of failure. The overhead of passing data over channels is neglectible for most cases.

Loose coupling becomes particularly viable when combined with XML as the data description language. XML is easy to parse and produce, and parsers are available for nearly all programming languages. In particular, for Haskell there is the typed embedding HaXml [159], which produces a set of Haskell datatype declarations from an XML document type definition (DTD), along with functions to parse and produce them. The typing makes it impossible for the program to produce or accept ill-formed XML, so there is a seamless integration of the XML type discipline with that of the programming language. This has proven very valuable in the implementation of XML-based middleware (such as the PG Kit broker) which are not straightforward to test.

Finally, interfaces need to be well-designed in order to be productive. Just adding buttons, menues and fancy icons may leave the user more confused than a well-designed command language. However, in general graphical user interfaces adhering to well-kent design principles such as direct manipulation will be better received, as they allow users to read and write mathematics in well-known, typeset form, and thus to concentrate on the content of the proof, not the syntax of the prover.

The grapical user interface for TAS, and the graphical user interface for Isabelle called IsaWin, were based on these principles. It visualised transformations, theories and other objects of interest by icons, and translated gestures such as drag&drop into development steps. This is intuitive to use, but had some technical drawbacks, as the interface was too tightly connected to the actual prover. This shows another benificial aspect of loose coupling: to be able to run tools as separate processes, their interaction must be clearly specified. In TAS (and IsaWin), developments were produced by gestures and held in internal datastructures rather then being explicitly represented as text, which meant they were hard to extract and import. Another drawback was that TAS was rather too closesly tied to the specific version of Isabelle used, which made it hard to update with later versions and thus unable to benefit from the continuing development of Isabelle.

In order to overcome the deficiencies of the graphical user interface while keeping its strenghts, we joined forces with the Proof General development team and created PG Kit, the next generation of prover interfaces. PG Kit stipulates a protocol called PGIP between prover and interface, which was designed as a generalisation of the way Proof General can be configured to work with over ten different provers. We have extended the PGIP protocol to cover non-textual interaction as well, and developed PG Kit as a framework

for interactive proof and development. This framework knows three types of components, namely the actual provers, a central broker, and displays which handle user interaction. Currently, there are three displays, one based on the popular Emacs editor, one based on the TAS and IsaWin design, and one as a plug-in for the Eclipse IDE.

4.5.1 Bibliographical Remarks

The work on tool integration as described here started with the UniForM project [93]. Einar Karlsen designed the UniForM workbench [82], a tool integration framework in Haskell, but unfortunately at that point the design of XML was just finished and tool support for it was just emerging. Together with other project partners, various tools were integrated into the UniForM workbench [L-18,L-19].

After the UniForM project, the author together with George Russel adapted the UniForM workbench to become the central repository and tool integration framework for the MMiSS project. Central to this effort was a new event model by George Russel [130], and the use of XML for data integration [L-12].

The work on interface design started with TAS (see Chapter 3). The graphical user interface was always very much part of TAS and IsaWin, and designed and developed together with Burkhart Wolff [L-7]. The design of sml_tk was based on the design of GoferTk [158], and a first version of HTk was part of Karlsen's thesis [82].

The PG Kit project was originally conceived by David Aspinall, but soon developed into a joint project. The extension to graphical interaction was based on the author's contribution. The current development is a distributed effort, with the broker and graphical display being developed mainly in Bremen, and the Eclipse and Emacs displays in Edinburgh.

The User Interfaces for Theorem Provers (UITP) workshop series, which had been a podium for discussing issues of user interface design and technology, and disemminating new results, unfortunately went into hibernation after its meeting in 1998. Together with Aspinall, the author revived it for the UITP'03 meeting co-located with TPHOLs'03 in Rome (2003). The meeting was a success, with eleven talks, about thirty participants and proceedings published in an ENTCS volume [L-31]. The next meeting UITP'05 is scheduled as a satellite workshop for ETAPS'05 in Edinburgh (2005).

Chapter 5

Conclusions

The previous chapters have given an exposition of the author's contributions in the areas of formal software development. We will briefly summarise the work again, and give an outlook to future research.

5.1 Concluding Summary

The work presented here has three main strands: categorical models of rewriting, formal proof and software development, and tool development.

Categorical Models of Rewriting. We propose a new model for rewriting based on the concept of a *monad*, known from category theory. It is the natural generalisation of the modelling of universal algebra. It can model various guises of rewriting, such as (normal) first-order rewriting, higher-order rewriting with variable bindings, term graphs and infinitary terms.

The key properties of this semantics are *compositionality* and *modularity*, which allow us to reason about large systems. We have shown various applications of this model, such as modularity and compositionality results and combinations of monads in functional programming.

Formal Proof and Software Development. We have shown how to model transformational software development in Isabelle, by using window inferencing, a calculus for transformational reasoning, for development in-the-small, and by adding a notion of theory morphisms to Isabelle for development in-the-large. But the main focus and contribution here is *abstraction*, the systematic generalisation of proofs and theorems, which allows us to reuse given developments in a wider setting.

The work on the systematic generalisation of theorems and proofs is based on proof term transformation. We have given a general procedure which works in any logical frameworks supporting proof terms, and implemented our work in the generic theorem prover Isabelle. Using this work,

we can further extend Isabelle by signature and theory morphisms. These extensions to Isabelle are light-weight extensions which do not modify Isabelle's logical kernel, and hence do not jeopardise its logical integrity.

Tool Development. In general, we found that for implementing tool support for formal methods, functional programming languages should be the tool of choice, as they allow rapid prototyping by programming at an abstract level, are good at symbolic manipulation, and can provide reliability by strict typing.

When it comes to tool integration, the approach we found useful in most cases was loosely coupled components talking over sockets or pipes in XML. This is particularly useful in connection with typed functional languages, as tools such as HaXml allow seamless integration of the typing given by XML document type definitions or schemas with the typing of the programming language. Thus, once we look past the inevitable hype surrounding it, XML is a useful tool for data integration.

We also used this architecture in the implementation of user interfaces. After discussing the advantages and drawbacks of an earlier implementation of a graphical user interface for our transformation system TAS, we have shown how to combine the design of this interface with Proof General, a generic interface based on the Emacs text editor. The combinations gives an interface which supports both text-based and graphical user interaction. This is the PG Kit project, the next generation of prover interfaces.

5.2 Outlook

In this final section, we give an outlook on what might constitute further work and research. In particular, we want to sketch how the different strands of research detailed in this exposé can converge.

Categorical Rewriting. In the are of categorical rewriting, research into abstract modularity should be continued, with results about non-ideal monads and combinations other than the disjoint union the next questions. Another obvious question to be resolved here is to find a suitable formulation of completion procedures (such as Knuth-Bendix completion [90]) at the abstract level.

Further, we have considered coproducts of monads as structuring operations. The extension to other structuring operations is the most relevant question here. Two immediately spring to mind: the tensor product [74, 75], and the non-disjoint union. For the first, we need to find a suitable formulation at the level of monads. The second can of course be modelled as a push-out, but just as with the monads the trick here is to restrict oneself to

cases which are still tractable yet widely applicable, so for example consider constructor-sharing systems.

Applications of Monads. With the extension to these structuring operations, applications to formal methods and functional programming abound. For example, due to the lack of a practical methodology to combine monads, Haskell lumps together all possible computational feature (such as file I/O, stateful computations, exceptions, concurrency and non-determinism) in the `IO` monad. With appropriate structuring operations, we could deconstruct the `IO` monad into its constituting monads. This would make reasoning about `IO` feasible; presently, most approaches to reasoning about Haskell do not consider `IO` actions [65, 45]. However, this has applications beyond functional programming: we can do very much the same for a simple imperative programming language like C. The aim is to reason about imperative programs in a modular way, i.e. considering each computational feature separately, and then reason about their interaction. (For example, reading from a file does not interfere at all with reading from a reference.) Even more speculative, monads have been used to model security issues like non-interference [66]; can we use the combination of monads to model compositional security?

Literate Specification. For formal development, we have proposed the document-centred approach where the document itself is the central artefact. The document can either be run through the prover, and formally verified, or it can be typeset into a proper documentation. Isabelle already implements an aspect of this with Isar, in that the document can be run though Isabelle, which generates LaTeX code which can be typeset (see Fig. 5.1). However, this approach is not optimal, as having Isabelle generate LaTeX is error-prone, restricts to particular LaTeX commands known to Isabelle, and is obviously specific to LaTeX — we cannot use another typesetting or word processing program to format our specification.

A better document flown is shown in Fig. 5.1 on the right: from a comprehensive document, we would extract a proof script, which can be run through Isabelle, and on the other hand, we would extract a document to be typeset in LaTeX, or any other document preparation systems (in particular, WYSIWYG systems which do seem enjoy more acceptance then LaTeX outside academia). Of course, we have to be sure that the part which we run through the prover is still the one appearing in the documentation, so the way to split the document should be simple and tractable.

An important aspect in Fig. 5.1 is that the theorem prover can feed back text into the document. During development, for example, when the user applies a transformation rule, the prover will calculate a new specification and communicate that back to the interface, which will insert it here in the

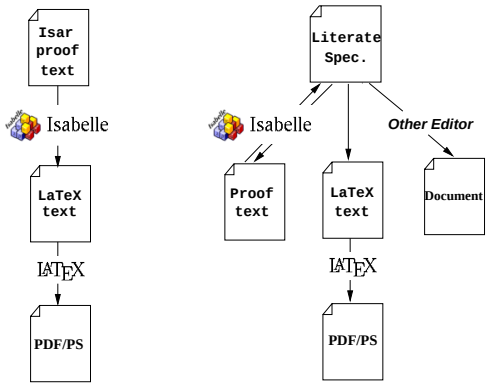


Figure 5.1: Document flow in Isar (left); document flow for literate specification (right).

document. When checking the document, the prover only needs to make sure it is actually correct. The PGIP protocol already supports this kind of prover feedback.

Abstraction. Abstraction and reuse easily fit into the literate specification paradigm, as the prover can suggest changes to the document via the prover feedback. This opens the way for *evolutionary formal program development*, where we combine formal specification with the evolutionary software development methodologies such as the spiral model. Abstraction is the big unused potential of formal program development, as it allows a community of users to gradually build up a library of trusted, used and guaranteed correct development tools.

Applications. One problem we encountered during the AWE project was that there are not many substantial case studies in formal program development available. For most encodings of specification formalisms and logics into a theorem prover, only smaller examples are considered. Larger case studies are either not publicly available, or run on old versions of the particular prover which is not supported or available anymore. There are two ways to solve this: either we import case studies from those provers where they exist (for example, the VSE system [73] has a rich fundus of developments), or we build new case studies in application areas where this is interesting in its own right, such as the area of robotics, where formal methods until now were not able to make an impact. We have already started work in this direction with the *SafeRobotics* project, which uses the Bremen autonomous wheelchair *Rolland* as the target platform. This is also a test case for the abstraction process — will be able to get useful transformation rules in the robotics domain by abstracting from our case studies?

Appendix A

Publications

Journal Articles

[L-1] Einar Broch Johnsen and Christoph Lüth: Abstracting Refinements for Transformation. *Nordic Journal of Computing*, 10: 316– 336, 2003.

[L-2] Neil Ghani and Christoph Lüth. Rewriting via Coinserters. *Nordic Journal of Computing*, 10: 290– 312, 2003.

[L-3] Christoph Lüth. Haskell in Space — An Interactive Game as a Functional Programming Exercise. *Educational Pearl*, *Journal of Functional Programming* **14**(6): 1077– 1085, November 2003.

[L-4] Neil Ghani, Christoph Lüth and Federico de Marchi: Monads of Coalgebras: Rational Terms and Term Graphs. To appear in *Mathematical Structures in Computer Science.*

[L-5] Federico de Marchi, Neil Ghani and Christoph Lüth. Solving Algebraic Equations using Coalgebra. *Theoretical Informatics and Applications*, 37: 301– 314, 2003.

[L-6] Neil Ghani, Christoph Lüth, Federico de Marchi and John Power. Dualizing initial algebras. *Mathematical Structures in Computer Science*, **13**(2):349– 370, 2003.

[L-7] Christoph Lüth and Burkhart Wolff. Functional design and implementation of graphical user interfaces for theorem provers. *Journal of Functional Programming*, **9**(2):167– 189, March 1999.

Refereed Conferences

[L-8] Micheal Abbott, Neil Ghani and Christoph Lüth: Abstract Modularity. Accepted for *Rewriting Techniques and Applications, RTA'05*. To appear in Lecture Notes in Computer Science.

[L-9] Einar Broch Johnsen and Christoph Lüth. Theorem Reuse by Proof Term Transformation. In K. Slind, A. Bunker and G. Gopalakrishnan (eds), *International Conference on Theorem Proving in Higher-Order Logic TPHOLs 2004*, Lecture Notes in Computer Science 3223, p. 152– 167. Springer, 2004.

[L-10] Christoph Lüth, Markus Roggenbach and Lutz Schröder. CCC — The CASL Consistency Checker. In José Fiadeiro (ed.), *Recent Trends in Algebraic Development Techniques WADT 2004*, Lecture Notes in Computer Science 3423, p. 94–105. Springer, 2004.

[L-11] Lutz Schröder, Till Mossakowski and Christoph Lüth. Type class polymorphism in an institutional framework. In José Fiadeiro (ed.), *Recent Trends in Algebraic Development Techniques WADT 2004*, Lecture Notes in Computer Science 3423, p. 234–248. Springer, 2004.

[L-12] Bernd Krieg-Brückner, Dieter Hutter, Arne Lindow, Christoph Lüth, Achim Mahnke, Erica Melis, Philipp Meier, Arnd Poetzsch-Heffter, Markus Roggenbach, George Russell, Jan-Georg Smaus and Martin Wirsing. MultiMedia Instruction in Safe and Secure Systems. *Recent Trends in Algebraic Development Techniques, 16th International Workshop WADT 2002*, Frauenchiemsee, Germany, Sep 2002. Lecture Notes in Computer Science 2755, p. 82 – 117. Springer, 2002.

[L-13] Christoph Lüth and Neil Ghani. Composing Monads Using Coproducts. In *Proc. Seventh ACM SIGPLAN International Conference on Functional Programming ICFP'02*, p. 133– 144. ACM Press, 2002.

[L-14] Christoph Lüth and Neil Ghani. Monads and modularity. In A. Armando (ed.), *Frontiers of Combining Systems FroCos 2002, 4th International Workshop*, Lecture Notes in Artificial Intelligence 2309, p. 18–32. Springer, 2002.

[L-15] Christoph Lüth and Burkhart Wolff. More about TAS and IsaWin: Tools for formal program development. In T. Maibaum (ed.), *Fundamental Approaches to Software Engineering FASE 2000. Joint European Conferences on Theory and Practice of*

Software ETAPS 2000, Lecture Notes in Computer Science 1783, p. 367– 370. Springer, 2000.

[L-16] Christoph Lüth and Burkhart Wolff. TAS — a generic window inference system. In J. Harrison and M. Aagaard (ed.), *Theorem Proving in Higher Order Logics: 13th International Conference TPHOLs 2000*, Lecture Notes in Computer Science 1869, p. 406–423. Springer, 2000.

[L-17] Christoph Lüth, Haykal Tej, Kolyang, and Bernd Krieg-Brückner. TAS and IsaWin: Tools for transformational program development and theorem proving. In J.-P. Finance (ed.), *Fundamental Approaches to Software Engineering FASE'99. Joint European Conferences on Theory and Practice of Software ETAPS'99*, Lecture Notes in Computer Science 1577, p. 239– 243. Springer, 1999.

[L-18] Christoph Lüth, Einar W. Karlsen, Kolyang, Stefan Westmeier, and Burkhart Wolff. Tool integration in the UniForM workbench. In *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science, p. 160–173. Springer-Verlag Wien New York, 1999.

[L-19] Christoph Lüth, Einar W. Karlsen, Kolyang, Stefan Westmeier, and Burkhart Wolff. Hol-Z in the UniForM-workbench – a case study in tool integration for Z. In J. P. Bowen, A. Fett, and M. G. Hinchey (ed.),*ZUM'98*: The Z Formal Specification Notation, 11th International Conference of Z Users, Lecture Notes in Computer Science 1493, p. 116–134. Springer, 1998.

[L-20] Christoph Lüth and Neil Ghani. Monads and modular term rewriting. In E. Moggi and G. Rosolini (ed.), *Category Theory and Computer Science CTCS 97*, Lecture Notes in Computer Science 1290, p. 69– 86. Springer, 1997.

[L-21] Kolyang, Christoph Lüth, Thomas Meier, and Burkhart Wolff. TAS and IsaWin: Generic interfaces for transformational program development and theorem proving. In M. Bidoit and M. Dauchet (ed.), *TAPSOFT '97: Theory and Practice of Software Development*, Lecture Notes in Computer Science 1214, p. 855– 859. Springer, 1997.

[L-22] Christoph Lüth. Compositional term rewriting: An algebraic proof of Toyama's theorem. In H. Ganzinger (ed.), *Rewriting Techniques and Applications RTA'96*, Lecture Notes in Computer Science 1103, p. 261– 275. Springer, 1996.

Refereed Workshops

[L-23] Bernd Krieg-Brückner, Arne Lindow, Christoph Lüth, Achim Mahnke and George Russell. Semantic Interrelation of Documents via an Ontology. In G. Engels and S. Seehusen (eds.), *DeLFI 2004, Tagungsband der 2. Deutschen e-Learning Fachtagung Informatik*. Lecture Notes in Informatics P-52, p. 271–282. Springer, 2004.

[L-24] Einar Broch Johnsen, Christoph Lüth and Maksym Bortin. An Approach to Transformational Development in Logical Frameworks. In *Proc. International Symposium on Logic-based Program Synthesis and Transformation, LOPSTR 2004*, Verona, Italy, August 2004.

[L-25] David Aspinall and Christoph Lüth. Proof General meets IsaWin: Combining Text-Based And Graphical User Interfaces. In David Aspinall and Christoph Lüth (ed.), *User Interfaces for Theorem Provers, International Workshop UITP'03*, Rome, Italy, September 2003. Electronic Notes in Theoretical Computer Science 103, p. 3–26.

[L-26] Neil Ghani, Christoph Lüth, and Federico de Marchi. Coalgebraic monads. In L. Moss (ed.), *Coalgebraic Methods in Computer Science CMCS'02*, Electronic Notes in Theoretical Computer Science 65.1, 2002.

[L-27] Neil Ghani, Christoph Lüth, Federico de Marchi, and John Power. Algebras, coalgebras, monads and comonads. In U. Montanari (ed.), *Coalgebraic Methods in Computer Science CMCS'01*, Electronic Notes in Theoretical Computer Science 44.1, 2001.

[L-28] Christoph Lüth. Transformational program development in the UniForM workbench. In M. Haveraan and O. Owe (ed.), *Selected papers from the 8th Nordic Workshop on Programming Theory*. Oslo University, 1996.

[L-29] Kolyang, Christoph Lüth, Thomas Meier, and Burkhart Wolff. Generating graphical user-interfaces in a functional setting. In N. Merriam (ed.), *User Interfaces for Theorem Provers UITP '96*, Technical Report, p. 59– 66. University of York, 1996.

[L-30] Kolyang, Christoph Lüth, Thomas Meier, and Burkhart Wolff. Generic interfaces for transformation systems and interactive theorem provers. In R. Berghammer, B. Buth, and J. Peleska (ed.), *International Workshop on Tool Support for Validation and Verification*, BISS Monograph 1. Shaker Verlag, 1998.

Edited

[L-31] David Aspinall and Christoph Lüth (eds.), *Proc. User Interfaces for Theorem Provers, International Workshop UITP'03*, Rome 2003. *Electronic Notes in Theoretical Computer Sciences* 103.

Miscellenea

[L-32] David Aspinall, Christoph Lüth and Daniel Winterstein. Parsing, Editing, Proving: The PGIP Display Protocol. Submitted to *User Interfaces for Theorem Provers, International Workshop UITP'05*, Edinburgh, April 2005.

[L-33] Daniel Winterstein, David Aspinall, and Christoph Lüth. Proof General/Eclipse: A Generic Interface for Interactive Proof. Submitted to *User Interfaces for Theorem Provers, International Workshop UITP'05*, Edinburgh, April 2005.

[L-34] Neil Ghani, Christoph Lüth and Stefan Kahrs. Rewriting the conditions in conditional rewriting. Technical Report 2000/20, Dept. Mathematics and Computer Science, University of Leicester, 2000.

[L-35] Christoph Lüth, Stefan Westmeier and Burkhart Wolff. sml_tk: Functional programming for graphical user interfaces. Technischer Bericht 8/96, FB 3, Universität Bremen, 1996.

Bibliography

[1] J. C. Aczel, P. Fung, R. Bornat, M. Oliver, T. OShea, and B. Sufrin. Using computers to learn logic: undergraduates experiences. In G. Cumming, T. Okamoto, and L. Gomez, editors, *Advanced Research in Computers and Communications in Education: Porceedings of the 7th International Conference on Computers in Education*, Amsterdam, 1999. IOS Press.

[2] P. Aczel, J. Adámek, and J. Velebil. A coalgebraic view of infinite trees and iteration. In U. Montanari, editor, *CMCS 2001, Coalgebraic Methods in Computer Science*, volume 44 of *Electronic Notes in Theoretical Computer Science*, pages 1– 26, 2001.

[3] J. Adámek. On final coalgebras of continuous functors. *Theoretical Computer Science*, To appear.

[4] J. Adámek and H.-E. Porst. On varieties and covarieties in a category. *Mathematical Structures in Computer Science*, 13(2):201–232, 2003.

[5] J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories*. Number 189 in London Mathematical Society Lecture Note Series. Cambridge University Press, 1994.

[6] J. Armstrong. The development of Erlang. In *Proc. ACMSIGPLAN International Conference on Functional Programming*, pages 196– 203. ACM Press, 1997.

[7] D. Aspinall. Proof General: A generic tool for proof development. In Graf and Schwartzbach [60], pages 38–42.

[8] D. Aspinall. Proof General Kit. White paper, 2002. Available from `http://proofgeneral.inf.ed.ac.uk/kit/`.

[9] D. Aspinall, H. Goguen, T. Kleymann, and D. Sequeira. Proof General, 2003. System documentation.

[10] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. inka 5.0 — a logic voyager. In *Proc. Conference on Automated Deduction CADE-*

16, volume 1632 of *Lecture Notes in Computer Science*, pages 207–211. Springer, 1999.

[11] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The development graph manager MAYA. In H. Kirchner and C. Ringeissen, editors, *Proc. 9th Int. Conf. Algebraic Methodology and Software Technology (AMAST'02)*, volume 2422 of *Lecture Notes in Computer Science*, pages 495– 501. Springer, 2002.

[12] S. Awodey and J. Hughes. The coalgebraic dual of Birkhoffs variety theorem, 2000. Preprint available at `http://www.andrew.cmu.edu/user/awodey/`.

[13] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[14] R. Back, J. Grundy, and J. von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9:467–483, 1997.

[15] R.-J. Back and J. von Wright. *Refinement Calculus: a Systematic Introduction*. Springer, 1998.

[16] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 363–366. Springer, 2000.

[17] M. Barr. Terminal algebras in well-founded set theory. *Theoretical Computer Science*, 114:299– 315, 1993.

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

[19] F. L. Bauer et al. *The Munich Project CIP. The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer, 1985.

[20] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'00)*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52. Springer, 2000.

[21] Y. Bertot, editor. *User Interfaces for Theorem Provers UITP'97*. INRIA Sophia Antipolis. Electronic proceedings at `http://www.inria.fr/croap/events/ uitp97-papers.html`, 1997.

[22] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Construction*. Texts in Theoretical Computer Science. Springer, 2004.

[23] Y. Bertot and L. Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(7):161–194, Feb. 1998.

[24] M. Bidoit and P. D. Mosses. CASL *User Manual*, volume 2900 of *LNCS*. Springer, 2004.

[25] L. Blaine, L. Gilham, J. Liu, D. Smith, and S. Westfold. Planware — domain-specific synthesis of high-performance schedulers. In *Proceedings of the Thirteenth Automated Software Engineering Conference*, pages 270– 280. IEEE Computer Society Press, 1998.

[26] R. Bornat and B. Sufrin. Using gestures to disambiguate unification. In *User Interfaces for Theorem Provers UITP'98*, 1998.

[27] R. Bornat and B. Sufrin. Animating formal proof at the surface: the Jape proof calculator. *The Computer Journal*, 42(3):177– 192, 1999.

[28] R. Bornat and B. Sufrin. A minimal graphical user interface for the Jape proof calculator. *Formal Aspects of Computing*, 11(3):244– 271, 1999.

[29] M. Broy. Compositional refinement of interactive systems. *Journal of the ACM*, 44(6):850–891, Nov. 1997.

[30] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Springer Verlag, 2001.

[31] B. Buchberger and F. Winkler. Gröbner bases and applications: 33 years of gröbner bases. In *Proc. London Math. Soc.*, volume 251. Cambridge University Press, 1998.

[32] R. M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, Jan. 1977.

[33] B. Buth, M. Kouvaras, J. Peleska, and H. Shi. Deadlock analysis for a fault-tolerant system. In M. Johnson, editor, *Algebraic Methodology and Software Technology. Proceedings of the AMAST'97*, number 1349 in Lecture Notes in Computer Science, pages 60– 75. Springer Verlag, Dec. 1997.

[34] B. Buth, J. Peleska, and H. Shi. Combining methods for the livelock analysis of a fault-tolerant system. In A. M. Haeberer, editor, *Algebraic Methodology and Software Technology. Proceedings of the AMAST'98*,

number 1548 in Lecture Notes in Computer Science, pages 124–139. Springer Verlag, 1998.

[35] M. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The refinement calculator: Proof support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, Discrete Mathematics & Theoretical Computer Science, pages 40–61, Wellington, New Zealand, July 1997. Springer.

[36] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10:97–124, 1998.

[37] The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report. Available at `http://www.haskell.org/definition/`, 2003.

[38] F. De Marchi. *Monads in Coalgebra*. PhD thesis, University of Leicester, 2003.

[39] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY, 1998.

[40] L. A. Dennis et al. The PROSPER toolkit. In Graf and Schwartzbach [60].

[41] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453– 457, 1975.

[42] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[43] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1990.

[44] A. Dix, J. Finley, G. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice-Hall, 1998.

[45] P. Djyber, Q. Hayian, and M. Takeyama. Verifiying Haskell programs by combining testing and proving. In *Proc. 3rd International Conference on Quality Software*, pages 272– 279. IEEE Computer Society Press, Nov 2003.

[46] Reference model for frameworks of software engineering environments. Technical Report ECMA TR/55, European Computer Manufacturers Association, 1990.

[47] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[48] Europäische Norm EN ISO 9241-10: Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten, Teil 10: Grundsätze der Dialoggestaltung, Sept. 1995. Deutsche Fassung.

[49] D. Epstein. *Word processing in groups*. Jones and Bartlett, 1992.

[50] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In G. Longo, editor, *Proc. 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 193–202. IEEE Computer Society Press, 1999.

[51] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*. Formal Systems (Europe) Ltd, Oct. 1997.

[52] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[53] N. Ghani. $\beta\eta$-equality for coproducts. In *Second Conference on Typed Lambda Calculus and its Applications*, Edinburgh, Apr. 1995.

[54] N. Ghani and A. Heyworth. Computing over k-modules. In J. Harland, editor, *Electronic Notes in Theoretical Computer Science*, volume 61. Elsevier Science Publishers, 2002.

[55] N. Ghani and A. Heyworth. A rewriting alternative to reidermeister schrier. In R. Nieuwenhuis, editor, *Procedings of Rewriting Techniques and Applications, RTA 2003*, number 2706 in Lecture Notes in Computer Science, pages 452–466. Springer Verlag, 2003.

[56] J. W. Goethe. Faust: der Tragödie erster Teil, 1808.

[57] H. H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. 1947. Part II, Vol. 1 of Report prepared for U.S. Army Ord. Dept. Reprinted in [150].

[58] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[59] M. J. C. Gordon and T. M. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logics*. Cambridge University Press, 1993.

[60] S. Graf and M. Schwartzbach, editors. *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1785. Springer, 2000.

[61] D. Gries. *A Science of Programming*. Springer Verlag, 1981.

[62] J. Grundy. Transformational hierarchical reasoning. *Computer Journal*, 39:291–302, 1996.

[63] M. Hamana. Term rewriting with variable binding: an initial algebra approach. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 148–159. ACM Press, 2003.

[64] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993. Preliminary version in LICS'87.

[65] W. Harrison and R. Kieburtz. The logic of demand in Haskell. *Journal for Functional Programming*, to appear.

[66] W. Harrison, M. Tullsen, and J. Hook. Domain separation by construction. *Electronic Notes in Theoretical Computer Science*, 90(1), 2003.

[67] D. Hemer, I. Hayes, and P. Strooper. Refinement calculus for logic programming in Isabelle/HOL. In R. J. Boulton and P. B. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'01)*, volume 2152 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2001.

[68] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Number 1 in London Mathematical Society Student Texts. Cambridge University Press, 1986.

[69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576– 585, Oct. 1969.

[70] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[71] B. Hoffmann and B. Krieg-Brückner, editors. *PROSPECTRA: Program Development by Specification and Transformation*, volume 690 of *Lecture Notes in Computer Science*. Springer, 1993.

[72] G. P. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11(1):31–55, Dec. 1978.

[73] D. Hutter, B. Langenstein, C. Sengler, J. H. Siekmann, W. Stephan, and A. Wolpers. Verification Support Environment (VSE). *Journal of High Integrity*. Forthcoming.

[74] M. Hyland, G. Plotkin, and J. Power. Combining computational effects: Commutativity and sum. In *TCS 2002, 2nd IFIP International Conference on Computer Science*, Montreal, 2002.

[75] M. Hyland, G. Plotkin, and J. Power. Combining effects: Sum and tensor. Submitted, 2003.

[76] M. Jackson. A pilot study of an automated theorem prover. In Bertot [21].

[77] C. B. Jay. Modelling reductions in confluent categories. In *Proceedings of the Durham Symposium on Applications of Categories in Computer Science*, 1990.

[78] C. B. Jay and N. Ghani. The virtues of η-expansion. *Journal for Functional Programming*, 5(2):135– 154, Apr. 1995.

[79] C. B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26– 49, 2003.

[80] M. Jones and L. Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University, Dept. Comp. Sci, Dec 1993.

[81] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[82] E. W. Karlsen. *Tool Integration in a Functional Programming Language*. PhD thesis, Universität Bremen, 1998.

[83] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[84] G. M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves and so on. *Bulletins of the Australian Mathematical Society*, 22:1– 83, 1980.

[85] G. M. Kelly. *Basic Concepts of Enriched Category Theory*, volume 64 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1982.

[86] G. M. Kelly. Elementary observations on 2-categorical limits. *Bulletins of the Australian Mathematical Society*, 39:301–317, 1989.

[87] G. M. Kelly and A. J. Power. Adjunctions whose counits are coequalizers, and presentations of finitary monads. *Journal for Pure and Applied Algebra*, 89:163– 179, 1993.

[88] D. King and P. Wadler. Combining monads. In J. Launchbury and P. Samson, editors, *Glasgow Workshop on Functional Programming*, Workshops in Computing Series, Ayr, July 1992. Springer Verlag.

[89] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2 (Background: Computational Structures), pages 1–116. Oxford University Press, 1992.

[90] D. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Universal Algebras*, pages 263–297. Pergamon Press, 1970.

[91] Kolyang, T. Santen, and B. Wolff. Correct and user-friendly implementations of transformation systems. In M. C. Gaudel and J. Woodcock, editors, *Formal Methods Europe FME'96*, volume 1051 of *Lecture Notes in Computer Science*, pages 629– 648. Springer, 1996.

[92] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle. In J. von. Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, pages 283 – 298. Springer Verlag, 1996.

[93] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, and A. Baer. The UniForM workbench, a universal development environment for formal methods. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods. Proceedings, Vol. II*, number 1709 in Lecture Notes in Computer Science, pages 1186–1205. Springer, 1999.

[94] A. Kurz. *Logics for Coalgebra and Applications to Computer Science.* Dissertation, Ludwig-Maximilans-Universtität München, 2000.

[95] T. Långbacka, R. Rukšėnas, and J. von Wright. TkWinHOL: A tool for window interference in HOL. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 245–260, Aspen Grove, Utah, USA, Sept. 1995. Springer.

[96] R. Lazić. *A semantic study of data-indepedence with applications to the mechanical verification of concurrent systems.* PhD thesis, Oxford University, 1997.

[97] S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, 1971.

[98] N. Magaud. Changing data representation within the Coq system. In *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume 2758 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2003.

[99] N. Magaud and Y. Bertot. Changing data structures in type theory: A study of natural numbers. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000*, volume 2277 of *Lecture Notes in Computer Science*, pages 181– 196. Springer, 2002.

[100] E. G. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer Verlag, 1976.

[101] E. Melis, E. Andrès, J. Büderbender, A. Frischauf, G. Goguadze, P. Libbrecht, M. Pollet, and C. Ullrich. ActiveMath: a generic and adaptive web-based learning environment. *Artificial Intellligence in Education*, 12(4), 2001.

[102] N. Merriam and M. Harrison. What is wrong with GUIs for theorem provers? In Bertot [21].

[103] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[104] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML — Revised*. The MIT Press, 1997.

[105] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, Computer Society Press, June 1989.

[106] L. Moss. Coalgebraic logic. *Annals of Pure and Applied Logic*, 96:277–317, 1999.

[107] T. Mossakowski, D. Sannella, and A. Tarlecki. A simple refinement language for CASL. In *Recent Trends in Algebraic Development Techniques WADT 2004*, volume 3423 of *Lecture Notes in Computer Science*, pages 162–185. Springer, 2004.

[108] O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal for Functional Programming*, 9:191–223, 1999.

[109] R. B. Neil Ghani, Anne Heyworth and C. Wensley. Computing with double cosets. *Journal of Symbolic Computation*, 2004. Submitted.

[110] A. Neumann and A. Berlea. fxp — the functional XML parser. Home page at `http://atseidl2.informatik.tu-muenchen.de/~berlea/Fxp/`.

[111] T. Nipkow. Structured proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lecture Notes in Computer Science*, pages 279–302. Springer, 2003.

[112] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[113] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[114] E. Ohlebusch. On the modularity of termination of term rewriting systems. *Theoretical Computer Science*, 136:333– 360, 1994.

[115] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[116] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, jun 1992.

[117] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[118] S. Peyton Jones, editor. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[119] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[120] F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1063–1147. Elsevier Science Publishers, 2001.

[121] J. Power. Enriched Lawvere theories. *Theories and Applications of Categories*, 6:83–93, 2000.

[122] J. Power. A unified category theoretic approach to variable binding. In *MERLIN '03: Proceedings of the 2003 workshop on Mechanized reasoning about languages with variable binding*. ACM Press, 2003.

[123] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques*. Kluwer Academic Publishers, Dordrecht, 1998.

[124] W. Reif and K. Stenzel. Reuse of proofs in software verification. In R. K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 284–293, Berlin, Germany, Dec. 1993. Springer.

[125] RELAX NG XML schema language, 2003. Home page at `http://www.relaxng.org/`.

[126] E. Robinson. Variations on algebra: monadicity and generalisations of equational theories. Technical Report 6/94, Sussex Computer Science Technical Report, 1994.

[127] P. J. Robinson and J. Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal for Logic and Computation*, 14(1):43–52, 1993.

[128] M. Roggenbach and L. Schröder. Towards trustworthy specifications I: Consistency checks. In *Recent Trends in Algebraic Development Techniques (WADT 201)*, volume 2267 of *LNCS*, pages 305–327. Springer, 2002.

[129] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[130] G. Russell. Events in haskell and how to implement them. In *International Conference on Functional Programming ICFP'01*. ACM Press, 2001.

[131] D. E. Rydeheard and J. G. Stell. Foundations of equational deduction: A categorical treatment of equational proofs and unification algorithms. In *Category Theory and Computer Science*, number 283 in Lecture Notes in Computer Science, pages 114– 139. Springer Verlag, 1987.

[132] D. Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, Workshops in Computing, pages 99–130. Springer, 1991.

[133] D. Sannella. Algebraic specification and program development by stepwise refinement. In *Proc. 9th Intl. Workshop on Logic-based Program*

Synthesis and Transformation (LOPSTR'99), volume 1817 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2000.

[134] D. T. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementation revisited. *Acta Informatica*, 25:233–281, 1988.

[135] R. A. G. Seely. Modelling computations: A 2-categorical framework. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 65–71, 1987.

[136] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour and Information Technology*, 1(3):237– 256, 1982.

[137] B. Shneiderman. *Designing the User Interface*. Addison-Wesley, 3rd edition, 1998.

[138] Sleepycat Software. Berkeley DB. `http://www.sleepycat.com/`.

[139] D. Smith. Constructing specification morphisms. *Journal of Symbolic Computation*, 15:571– 606, 1993.

[140] D. R. Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5(1):37–58, Feb. 1985.

[141] D. R. Smith. KIDS: a semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, Sept. 1990.

[142] D. R. Smith. Mechanizing the development of software. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, Proceedings of the Marktoberdorf International Summer School, NATO ASI Series, pages 251–292. IOS Press, Amsterdam, 1999.

[143] D. R. Smith and M. R. Lowry. Algorithm theories and design tactics. *Science of Computer Programming*, 14:305– 321, 1990.

[144] M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992. 2nd edition.

[145] Y. V. Srinivas and R. Jullig. Specware: Formal support for composing software. In *Proc. Conf. Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*. Springer, 1995. Also appeared as Kestrel Institute Technical Report KES.U.94.5.

[146] M. Staples. Window inference in Isabelle. In *Proc. Isabelle Users Workshop*. University of Cambridge Computer Laboratory, 1995.

[147] M. Staples. *A Mechanised Theory of Refinement*. PhD thesis, University of Cambridge, 1998.

[148] M. Staples. Representing WP semantics in Isabelle/ZF. In G. Dowek, C. Paulin, and Y. Bertot, editors, *TPHOLs: The 12th International Conference on Theorem Proving in Higher-Order Logics*, number 1690 in lncs. springer, 1999.

[149] J. G. Stell. Modelling term rewriting systems by Sesqui-categories. Technical Report TR94-02, Keele Unversity, Jan. 1994.

[150] A. H. Taub. *John von Neumann: Collected Works*, volume V: Design of Computers, Theory of Automata and Numerical Analysis. Pergamon Press, 1963.

[151] H. Tej. *HOL-CSP: Mechanised Formal Development of Concurrent Processes*. Dissertation, FB 3 — Mathematik und Informatik, Universität Bremen, 2002.

[152] H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe FME '97*, number 1313 in Lecture Notes in Computer Science, pages 318–337. Springer Verlag, 1997.

[153] The W3C Consortium. Extensible markup language (XML). Technical report, W3C Recommendation, 2004.

[154] H. Thimbleby. *User Interface Design*. ACM Press Frontier Series. Addison-Wesley, 1990.

[155] Y. Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *Journal of the ACM*, 34(1):128–143, 1987.

[156] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67– 69, Cambridge, June 1949. University Mathematical Laboratory. Quoted from [79].

[157] J. von Wright. Extending window inference. In *Proc. TPHOLs '98*, number 1497 in Lecture Notes in Computer Science, pages 17–32. Springer Verlag, 1998.

[158] T. Vullinghs, D. Tuijnman, and W. Schulte. Lightweight GUIs for Functional Programming. In *7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 341–356, 1995.

[159] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming ICFP'99*, pages 148– 159. ACM Press, 1999.