Nordic Journal of Computing 13(2006), 2–21

Structured Formal Development in Isabelle

MAKSYM BORTIN *
Universität Bremen, Department of Mathematics and Computer Science
P.O. Box 330 440, D-28334 Bremen, Germany
Email: maxim@informatik.uni-bremen.de

EINAR BROCH JOHNSEN
University of Oslo, Department of Informatics
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
Email: einarj@ifi.uio.no

CHRISTOPH LÜTH
Deutsches Forschungszentrum für Künstliche Intelligenz, Lab Bremen
D-28359 Bremen, Germany
Email: Christoph.Lueth@dfki.de

Abstract. General purpose theorem provers provide advanced facilities for proving properties about specifications, and may therefore be a valuable tool in formal program development. However, these provers generally lack many of the useful structuring mechanisms found in functional programming or specification languages. This paper presents a constructive approach to adding theory morphisms and parametrisation to theorem provers, while preserving the proof support and consistency of the prover. The approach is implemented in Isabelle and illustrated by examples of an algorithm design rule and of the modular development of computational effects for imperative language features based on monads.

ACM CCS Categories and Subject Descriptors: F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic; I.2.2 [Artificial Intelligence]: Automatic Programming.

Key words: Program development, transformation, theorem provers, monads.

1. Introduction

Formal program development requires considerable proof effort, as well as careful planning and design. In order to simplify the development process, development tools should both provide high-level strategies for proving properties and help structure developments. General purpose theorem provers such as PVS [Owre *et al.* 1995], Coq [Bertot and Castéran 2004], and Isabelle [Nipkow *et al.* 2002] offer powerful proof support, and include libraries formalising a considerable amount of mathematics. This is particularly useful for developing safety-critical systems where a full formalisation is needed,

*Research supported by DFG under grants LU 707/2-1 and 2-2.

Received January 30, 2006; revised May 27, 2006; accepted June 14, 2006.

e.g., for continuous functions in hybrid systems or geometric algorithms in robot control systems. In these cases, fully automatic model-checking or light-weight methods such as ESC/Java are not sufficient.

However, these general purpose theorem provers mostly lack structuring mechanisms to express *development in-the-large*, as provided by specification languages such as CASL, OBJ, or Z, and supported by algorithm design systems such as Specware [Srinivas and Jullig 1995]. Two important structuring mechanisms are theory morphisms and parametrised specifications. In order to improve the applicability of theorem provers for program development, it is desirable to integrate these structuring techniques in theorem provers.

This paper shows how theorem provers may be extended with theory morphisms and parametrised theories by means of proof term transformation. The extension preserves the consistency of the logical development. Proof term transformations allow the systematic manipulation of the proofs associated with logical theories, see Johnsen and Lüth [2003, 2004]. This technique is now extended to support structuring mechanisms for formal program development. In contrast to related techniques available in PVS [Owre and Shankar 2001], our approach is *constructive*: When moving a theorem along a theory morphism, a proof of the theorem is automatically constructed in the target theory. The approach applies to theorem provers and logical frameworks with proof terms, is implemented for Isabelle, and illustrated by an algorithmic design tactic and by the development of monads, which allow the modular specification of computational effects such as state update and exceptions in higher-order logic.

The paper is structured as follows: Section 2 introduces logical frameworks and proof term transformations, and gives a formal account of the considered structuring mechanisms. Section 3 discusses details specific to the implementation in Isabelle. Section 4 and Section 5 apply parametrised theories to the development of algorithm design rules and monads. Section 6 discusses related work and Section 7 concludes the paper.

2. Structured Reasoning in Logical Frameworks

A *logical framework* may be understood as a meta-level inference system in which other deductive systems (object logics) can be specified [Pfenning 2001]. This section presents our approach in the general context of logical frameworks in which the object logic is organised in hierarchical theories and proof terms record the proofs of derived theorems. Section 3 considers the specific aspects of the implementation of this approach in Isabelle.

2.1 Signatures and Theories

A *logical framework* defines a meta-logic, i.e., a weak logic in which object logics may be encoded. Development in the logical framework is structured using signatures and theories. 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* $ar(c) \in \mathbb{N}$. Let $\mathcal{X}$ be a fixed, infinite set of type variables. The set T^* of *types* generated from T is the smallest set including the type variables $\mathcal{X}$, and closed under application of the type constructors. Two types $s, t \in T^*$ are α-equivalent, written $s =_\alpha t$, if they are equal up to a consistent renaming of type variables. Each operation $\omega \in \Omega$ has a *type* $\tau(\omega) \in T^*$. Given a signature Σ, a set X of variables, and a type assignment $\tau : X \to T^*$, the set of type-correct *terms* is denoted $T_\Sigma(X)$. Each term $t \in T_\Sigma(X)$ has a type; $t : \tau$ denotes that the term t has the type τ.

The meta-logic of a logical framework typically defines a type $prop \in T$ of *propositions*, a function space type constructor $\to \in T$, with $ar(\to) = 2$, and operations for meta-level equality $\equiv$, implication $\Longrightarrow$ and quantification $\bigwedge$.

An object logic is encoded in the logical framework by defining an abstract syntax for the type constructors and operations of the object logic, and then encoding object-logic derivability as meta-level implication. A set of axioms reflects the inference rules of the object logic. Specification and formal development is done in the object logic. Examples of object logics are classical higher-order logic (HOL), set theory (ZF), and type theories.

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*. In addition to the derivability rules of the object logic, typical axioms include constant and datatype definitions. The set of *theorems* of $\mathcal{T}h$, denoted $Thm(\mathcal{T}h)$, is obtained by closing $\mathcal{A}x$ under derivability. Theories and signatures may be organised hierarchically. Given two signatures Σ_1 and Σ_2, the *union* $\Sigma_1 \cup \Sigma_2$ is defined by the unions of type constructors and of operators, respectively. Similarly, the union $\mathcal{T}h_1 \cup \mathcal{T}h_2$ of two theories is defined by the unions of the signatures and axioms of the two theories. New theories may be built by extending existing theories, i.e.,

$$\mathcal{T}h = \mathcal{T}h_1 \cup \ldots \cup \mathcal{T}h_n \cup \langle \langle T, \Omega \rangle, \mathcal{A}x \rangle.$$

The theories $\mathcal{T}h_1, \ldots, \mathcal{T}h_n$ are called the *ancestors* of $\mathcal{T}h$. If $\mathcal{T}h_i$ is an ancestor of $\mathcal{T}h$, then $Thm(\mathcal{T}h_i) \subseteq Thm(\mathcal{T}h)$; i.e., all theorems of $\mathcal{T}h_i$ are derivable in $\mathcal{T}h$. The theory $\mathcal{T}h$ is a *conservative extension* of its ancestors if $\forall \phi \in Thm(\mathcal{T}h) \cdot \phi \in T_{\Sigma_i}(X) \Longrightarrow \phi \in Thm(\mathcal{T}h_i)$; i.e., if ϕ is syntactically expressible in $\mathcal{T}h_i$, then ϕ is a theorem of $\mathcal{T}h_i$. Thus, $\mathcal{T}h$ does not introduce new properties for old operations or types. Conservative extensions define new types and operations without introducing inconsistencies.

2.2 Signature and Theory Morphisms

Signature and theory morphisms provide a systematic means to relate different theories. *Signature morphisms* are maps between signatures which preserve operations and arities, and allow terms to be moved 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 $\sigma : \Sigma_1 \to \Sigma_2$ consists of two maps $\sigma_T : T_1 \to T_2$ and $\sigma_\Omega : \Omega_1 \to \Omega_2$, such that (1) for all type constructors $t \in T_1$, $ar(\sigma_T(t)) = ar(t)$, and (2) for all operations $\omega \in \Omega_1$, $\overline{\sigma}_T(\tau(\omega)) =_\alpha \tau(\sigma_\Omega(\omega))$. Here, $\overline{\sigma}_T : T_1^* \to T_2^*$ is the extension of the map between type constructors to a map between the

types built from these type constructors. The use of α-equivalence rather then equality between types accommodates for parametric polymorphism.

Theory morphisms are maps between theories, and allow theorems to move 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 τ consists of a signature morphism $\tau_\Sigma : \Sigma_1 \to \Sigma_2$, and a map $\tau_A : \mathcal{A}x_1 \to \mathit{Thm}(\mathcal{T}h_2)$ which maps every axiom of $\mathcal{T}h_1$ to a *theorem* of $\mathcal{T}h_2$. A signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ can be extended to a map $\overline{\sigma} : T_{\Sigma_1}(X) \to T_{\Sigma_2}(X)$ between terms, which replaces all operations ω with $\sigma_\Omega(\omega)$. A theory morphism $\tau : \mathcal{T}h_1 \to \mathcal{T}h_2$ can be extended to a map $\overline{\tau} : \mathit{Thm}(\mathcal{T}h_1) \to \mathit{Thm}(\mathcal{T}h_2)$ between theorems, which translates the terms by $\overline{\tau_\Sigma}$ and replaces all axioms ϕ in the proof by $\tau_A(\phi)$.

2.3 Parametrised Theories

The introduction of theory morphisms allows the modeling of parametrised theories and their instantiation, which may be used to structure developments. A *parametrised theory* is a pair of theories $\Theta = \langle P, B \rangle$ such that $P \subseteq B$. We call P the *parameter* and B the *body*. In order to instantiate the theory, we *match* the parameter P with an *instantiating theory* I by constructing a theory morphism $\sigma : P \to I$ (see Diagram 2.1). Axioms, including constant and type definitions, from B are translated to construct an extension R of I. All theorems in B are converted to theorems in R, by translating the proofs from the generic setting of B to the specific context of I. In order to construct the matching theory morphism σ, only those theorems to which the axioms $\mathcal{A}x_P$ of P are mapped may need interactive proof. Formally, an instantiation of Θ with a target theory $I = \langle \Sigma_I, \mathcal{A}x_I \rangle$ is

$$\begin{array}{ccc} P & \xrightarrow{\quad\sigma\quad} & I \\ \big\downarrow & & \big\downarrow \\ B & \xrightarrow{\quad\sigma'\quad} & R \end{array} \tag{2.1}$$

given by a theory morphism $\sigma : P \to I$. The resulting theory

$$R = \langle \Sigma_I \cup (\Sigma_B \setminus \Sigma_P),\ \mathcal{A}x_I \cup (\mathcal{A}x_B \setminus \mathcal{A}x_P) \rangle$$

is obtained from I by transforming the rest of B. The dashed morphisms in Diagram 2.1 are obtained as follows: I is included in R by construction, and σ' maps type constructors, operations, and axioms from P to their image under σ, and injects them into R otherwise. The parametrised theory is *well-formed* if B is a conservative extension of P. In this case, R will also be a conservative extension of I for any instantiating theory I, and the instantiation will never introduce inconsistencies.

2.4 Proof Term Transformations

A constructive approach to moving theorems along theory morphisms is to transform proof terms. A proof term represents the inference steps of a proof

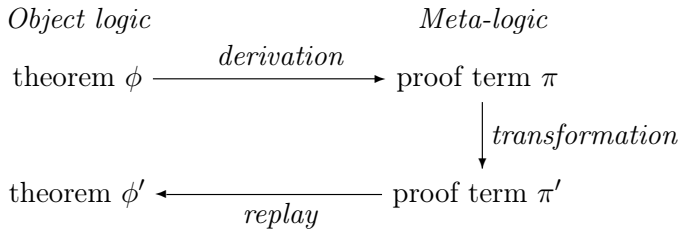


Figure 2.1: Implementing theorem manipulation by proof term transformation.

in a prover-independent way. In principle, the proof term can be checked by an independent proof checker. The proof terms of a logical framework may be defined in a typed λ-calculus corresponding to the meta-logic under the Curry-Howard isomorphism. A λ-abstraction represents an instantiated proof rule, its formal parameters are the premises of the proof rule, and β-reduction represents an inference. Note that the proof terms represent inferences in the meta-logic, in contrast to object-logic proof terms in, e.g., Coq. This has two advantages: (1) operations on meta-logic proof terms apply to any object logic, and (2) proof term manipulation is easier for a weak meta-logic than for an expressive object logic.

An object-logic proof is transformed by manipulating its meta-logic proof term and the resulting proof term is replayed to derive a new theorem (see Figure 2.1). Consequently the logical framework ensures the correctness of the derived theorem, as its derivation uses only correct meta-level inferences.

3. Implementation in Isabelle and Integration with Isar

The constructive approach to structuring techniques presented in Section 2 has been implemented in the Isabelle theorem prover. The meta-level inference system of Isabelle is intuitionistic higher-order logic extended with Hindley-Milner polymorphism and type classes. The technical realization of structured development techniques for Isabelle is based on a technique for proof term transformation, extending previous work by Johnsen and Lüth [2003, 2004]. The implementation of the proof term transformation technique uses Isabelle's proof terms, as introduced by Berghofer and Nipkow [2000]. Isabelle uses the high-level proof language Isar, introduced by Wenzel [2001], which allows proofs and theories to be written in a structured and understandable format. In order to facilitate the use of our structured development techniques, Isar has been extended to cover the necessary commands.

3.1 Signature and Theory Morphisms

Isabelle does not support theory morphisms (except for the implicit inclusions resulting from extending theories); the work presented in this paper

extends Isabelle with a mechanism to express, handle, and use signature and theory morphisms. Signature and theory morphisms are implemented as abstract datatypes. For signature morphisms, the invariants of the abstract datatype ensure that a translated term typechecks if the original term did. For theory morphisms, the invariants ensure that source theory theorems are provable in the target theory after translation.

Let $\mathcal{T}h = \langle \Sigma, \mathcal{A}x \rangle$ and $\mathcal{T}h' = \langle \Sigma', \mathcal{A}x' \rangle$ be two theories, $\tau : \mathcal{T}h \to \mathcal{T}h'$ a theory morphism, and ϕ a theorem of $\mathcal{T}h$ with a proof term π. By replacing the operation symbols and types in π according to the underlying signature morphism τ_Σ, and the axioms from $\mathcal{T}h$ in π according to the theory morphism τ, we construct a proof term which can be replayed in $\mathcal{T}h'$ to obtain a theorem $\overline{\tau}(\phi)$. Note that theorems need to be translated in order of their dependency: if theorem ϕ depends on theorem ψ, ψ must be translated before ϕ.

In the corresponding Isar extension, the user only needs to provide enough information about the types and operations to make the map valid and unambiguous. The theorem map is constructed *automatically* by inspecting the theorem database to find a valid theorem for each translated axiom from the domain of the theory morphism. In order to illustrate theory morphisms in the Isar language, let `Nat` be a theory of natural numbers and consider the following theory of orders:

```
theory Order imports Main begin
typedecl elem
consts
  ord :: "elem ⇒ elem ⇒ bool"
axioms
  trans:   "ord x y ⟹ ord y z ⟹ ord x z"
  antisym: "ord x y ⟹ ord y x ⟹ x= y"
```

A theory morphism into `Nat` is constructed by the Isar command

```
thymorph s: Order ⟶ Nat
  maps [("Order.elem" ↦ "nat")] [("Order.ord" ↦ "less")]
```

Isabelle tries to deduce how to map the operations. In this case there may be many binary operations on `Nat`, so we explicitly select the `less` than relation. (The Isar keyword **sigmorph** similarly constructs a signature morphism.) Theorems proving that `less` is transitive and antisymmetric are required, in this case they are found automatically. Theorems can now be translated along the theory morphism:

```
translate_thm "Order.total" as total along s
```

The translation uses an underlying SML tactic based on proof term transformation to automatically construct a proof for the translated theorem `total`.

3.2 Parametrised Theories

In practice it is not necessary to syntactically distinguish the body and the parameter of a parametrised theory. In the hierarchical organisation of Isabelle theories, any ancestor `R` of a theory `T` may be the parameter. To

instantiate `R` with a theory `S`, construct a theory morphism from `R` to `S`, and the instantiation of Diagram 2.1 is derived. As an example, consider the following extension of `Order` to sorted lists (the empty list is denoted `[]` and the list constructor `#`):

```
theory Sorted imports Order begin
consts
  sorted :: "elem list ⇒ bool"
primrec
  "sorted [] = True"
  "sorted (x#xs) = ((∀y ∈ set xs. ord x y) ∧ sorted xs)"
```

In order to construct an instance of sorted lists, we give a theory morphism `s` mapping the `elem` type to a target type. The instantiation of `Sorted`, using `s`, is then given by the keyword **t_instantiate**:

```
theory SortLess imports Main begin
thymorph s: Order ⟶ SortLess
  maps [("Order.elem" ↦ "nat")] [("Order.ord" ↦ "less")]
t_instantiate Sorted mapping s
```

All definitions from the body are translated by the morphism and inserted into the theory `SortLess`, including auxiliary definitions (user-defined syntactic sugar, datatype definitions, etc.). Further, a theory morphism `s'` from `Sorted` to `SortLess` is constructed automatically (corresponding to σ' in Diagram 2.1). All theorems from the body can be translated along `s'`.

The implementation does not distinguish between well-formed and not well-formed parametrised theories, in accordance with normal Isabelle practice. Isabelle theories are usually built by conservative extensions; all theory-building operations provided by Isabelle except for the introduction of axioms (**axioms**) are conservative extensions. Hence, if the body of the parametrised theory is built as a conservative extension of the parameter, the parametrised theory is well-formed.

4. Application to Algorithm Design

The proposed structuring mechanisms allow *transformation rules* to be formalised and applied in the theorem prover. Transformation rules in our sense are formalised development patterns or design tactics. They represent non-automatable design decisions in the development process, such as the introduction of a recursive function definition. Using such rules has two advantages: they guide the development process, and they reduce proof effort, as once proven they are applicable any number of times.

Technically, a transformation rule is a well-formed parametrised theory. In the notation of Diagram 2.1, a transformation rule $\langle P, B\rangle$ is applied by constructing a theory morphism σ from P to I, and then instantiating B using σ. The result is a derived theory morphism σ' which can be used to automatically translate theorems from B. The first step may require proof work as the axioms of the parameter theory P need proofs in I. An

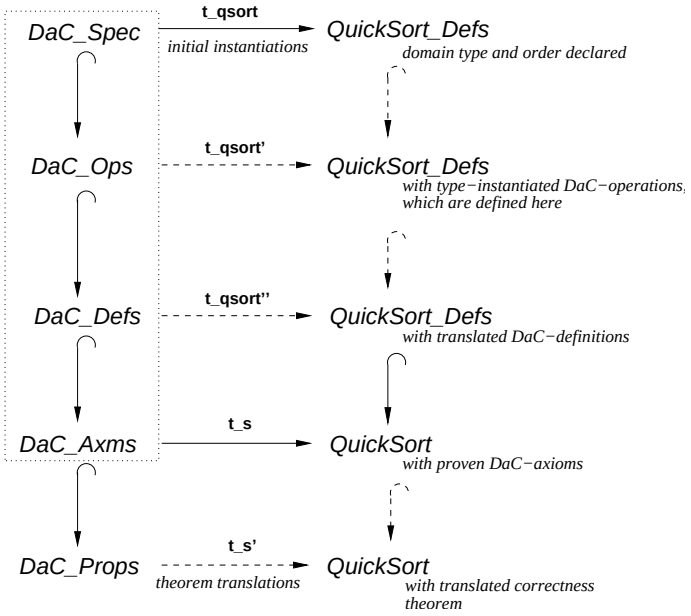
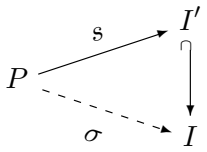


Figure 4.2: Stepwise development of `QuickSort`. The parameter of the transformation rule is given by the dotted rectangle. Dashed morphisms are automatically derived.

interactive command **make_instance** assists users in this task: if the user specifies the mapping of operations with a signature morphism s from P to I', **make_instance** constructs a theory text containing an extension of I' with the axioms of P as proof obligations (i.e., theorems with pending proofs). Once the proofs have been completed, the theory morphism σ is constructed automatically from the signature morphism s as shown in Diagram 4.1.



(4.1)

4.1 Example: Divide-and-Conquer

The *divide-and-conquer* design tactic, introduced by Smith [1985], synthesises a recursive function `F` and its correctness proof from a specification in pre- and postcondition style, if the specification and some auxiliary functions satisfy certain properties. This design tactic has been formalised in Isabelle, and will be applied to the specification of sorting to derive a quicksort algorithm, along with a proof of its correctness. The development is outlined in Figure 4.2. The dotted rectangle is the parameter of the tactic and starts with a theory `DaC_Spec`, declaring the types `D_F`, `R_F`, `D_G`, and `R_G` and the placeholders for pre- and postconditions:

```
F_pre        :: "D_F ⇒ bool"
F_post       :: "D_F ⇒ R_F ⇒ bool"
```

Next, the theory *DaC_Ops* contains the declaration of operation symbols:

```
Measure      :: "D_F ⇒ nat"
Primitive    :: "D_F ⇒ bool"
Dir_Solve    :: "D_F ⇒ R_F"
Decompose    :: "D_F ⇒ (D_G × D_F × D_F)"
Compose      :: "(R_G × R_F × R_F) ⇒ R_F"
G            :: "D_G ⇒ R_G"
G_pre        :: "D_G ⇒ bool"
G_post       :: "D_G ⇒ R_G ⇒ bool"
```

The function *Measure* ensures that the definition of *F* is well-founded. The function *Primitive* determines if *F* can be computed directly by *Dir_Solve*, otherwise the function *Decompose* transforms x into a tuple (a, b, c) of type *D_G* × *D_F* × *D_F*, for which the divide-and-conquer strategy recursively computes *Compose*(*G* a, *F* b, *F* c) (where *G* is an auxiliary function).

The theory *DaC_Defs* contains a schema for the recursive function to be synthesised, and *DaC_Axms* specifies the behaviour of the operation symbols from *DaC_Ops* by means of axioms; e.g., *Dir_Solve* must satisfy the same specification as *F* for primitive arguments. This completes the parameter.

The body of the transformation rule is the theory *DaC_Props* and contains the correctness proof for divide-and-conquer algorithms, namely that the synthesised function satisfies the specification:

```
theorem DaC : assumes "F_pre x" shows "F_post x (F x)"
```

The transformation rule is now applied to derive a sorting algorithm. We first specify the problem: a sorting function accepts as input all lists *il* of elements of type *D*, where *D* has a linear order. The function returns a sorted list *ol* which is a permutation of *il* (denoted *il* ↭ *ol*). Formally:

```
sort_pre_def  : "sort_pre l ≡ True"
sort_post_def : "sort_post il ol ≡ sorted ol ∧ il ↭ ol"
```

Let this specification be in an ancestor theory of *QuickSort_Defs*, say *Sorted*. In order to apply the transformation rule, a morphism *t_s* is constructed. The domain *D_F* and range *R_F* are instantiated with *D list*, while the domain *D_G* and range *R_G* of *G* are instantiated with *D*. The mapping of the pre- and postcondition gives the theory morphism *t_qsort*:

```
thymorph t_qsort : DaC_Spec ⟶ QuickSort_Defs
 maps [("DaC_Spec.D_F"    ↦ "Sorted.D list"),
       ("DaC_Spec.D_G"    ↦ "Sorted.D"),
       ("DaC_Spec.R_F"    ↦ "Sorted.D list"),
       ("DaC_Spec.R_G"    ↦ "Sorted.D")]
      [("DaC_Spec.F_pre"  ↦ "Sorted.sort_pre"),
       ("DaC_Spec.F_post" ↦ "Sorted.sort_post")]
```

The parameter operations from *DaC_Ops* are now translated along *t_qsort*,

which corresponds to the instantiation of their types:

t_instantiate *DaC_Ops* **mapping** *t_qsort*

A theory morphism *t_qsort' : DaC_Ops* $\longrightarrow$ *QuickSort_Defs* is derived, which *extends* *t_qsort* by mapping the operations from *DaC_Ops* to the newly translated ones. These functions are now defined: *G* and *Dir_Solve* are identities, *Measure* becomes *length*, decomposition splits a non-empty list *x#xs* into the elements of *xs* less than *x* and the elements equal to or larger than *x*, and *Compose* becomes list append. Instantiating the *Dac_Defs* definitions results in a recursive definition of *qsort* (renaming the translated function):

t_instantiate *DaC_Defs* **mapping** *t_qsort'*
renames: *[("DaC_Defs.F"* $\mapsto$ *"qsort")]*

In order to obtain a proof of the correctness of *qsort*, it remains to instantiate the body of the rule by proving the axioms from *DaC_Axms*. This can be done using **make_instance**, as described above, to generate a theory *QuickSort* with the proof obligations. Once these are discharged, a theory morphism *t_s* is obtained, which allows the automatic translation of the theorems from *DaC_Props*; in particular, the correctness of *qsort* is derived. (Remark that by different instantiations of these operations, we can obtain other sorting algorithms. For example, if *Decompose* splits lists in the middle and *Compose* merges them, the mergesort algorithm is synthesised.)

5. Application to Constructor Classes and Monads

Constructor classes combine higher-order polymorphism and overloading [Jones 1995]. The functional programming language Haskell [Peyton Jones 2003] uses the monad constructor class to model imperative features such as encapsulated state, exceptions, and I/O in a functional setting. By modeling constructor classes, and in particular monads, as parametrised theories, a similar model can be defined in higher-order logic. We now develop the general theory of monads and consider some instances of this theory.

5.1 Monads

A *computational monad* [Moggi 1989] is a type constructor *M* of arity 1 with two operations *bind* and *eta* (sometimes called *return*), such that *bind* is associative with *eta* as its left and right unit, and *eta* is injective:

```
typedecl 'a M
consts
  eta   :: "'a ⇒ 'a M"
  bind  :: "'a M ⇒ ('a ⇒ 'b M) ⇒ 'b M"  ("_ >>= _")
axioms
  mon_lunit:  "(eta x >>= t) = t x"
  mon_runit:  "(t >>= eta)   = t"
  mon_assoc:  "(s >>= t >>= u) = (s >>= (λx. t x >>= u))"
  mon_eta_inj: "eta x= eta y ⟹ x= y"
```

Additional mixfix syntax can be defined for an operation in Isabelle; e.g., the infix syntax `>>=` for `bind` above. Based on this definition, operations may be introduced which reflect imperative programming constructs. To facilitate partial instantiations, the monad theory is decomposed into a hierarchy of theories: `MonadType` contains the monad type declaration, `MonadOpEta` and `MonadOpBind` the `eta` and `bind` operations, `MonadOps` their union, `MonadAxms` the monad axioms, and `Monad` the derived operations, syntactic sugar, and theorems (see Figure 5.3). After instantiating the type constructor, one can either axiomatically assert a monad structure (which is not a conservative extension), or define `bind` and `eta`, and prove the monad laws (which is a conservative extension).

The monad's `bind` operation is used for the sequential composition of computations; a special case is when the second computation does not use the result of the first computation. Introducing the Haskell-inspired syntax `{a<- p; q; r}` for `p >>= λa.q >>= λb.r` (where `b` does not occur in `r`) allows a small imperative programming language to be generically embedded into any monad. In order to add control structures, the *conditional* is defined as

```
constdefs
  cond :: "bool S ⇒ 'a S ⇒ 'a S ⇒ 'a S" ("IF _ THEN _ ELSE _ FI")
 "cond c p q ≡ {b <- c; if b then p else q}"
```

A branch is selected after evaluating `c`. The condition `c` is a monadic value and not a Boolean, so the evaluation of `c` may have side effects. In this case the reduction cannot eliminate `c`. The following reduction rules (and similar rules for `False`) can now be proved:

```
(b = eta True) ⟹ (IF b THEN p ELSE q FI)= p
({b; eta True}= b) ⟹ (IF b THEN p ELSE q FI)= {b; p}
```

In order to add *iteration*, we distinguish *commands*, which have the return type `unit`, from *expressions*, which may have any return type. Only commands, of type `cmd = unit S`, may be iterated. Let `SKIP = eta ()` be the empty command. In a denotational semantics, iteration is usually defined as a fixpoint. However, in classical Church-style higher-order logic (HOL) all functions are total, and hence not all functions have a fixpoint. In this model iteration will always be terminating and defined by means of a well-founded recursion operator with a strictly decreasing order `ord` as parameter:

```
constdefs
  while :: "('a × 'a) set ⇒ ('a ⇒ bool) ⇒ ('a ⇒ 'a S) ⇒ 'a ⇒ cmd"
 "while ord p b i ≡
     wfrec ord (λF a. if p a then {a'<- b a; F a'} else SKIP) i"
```

The termination condition `p` is not allowed to have side effects. The following unfolding lemma shows that iteration behaves as expected:

```
wf ord  ⟹ ∀a. (result (b a), a) : ord
 ⟹ while ord p b i = (if p i then {a <- b i; while ord p b a} else SKIP)
```

To avoid repeatedly proving the decrease of the termination measure for

standard data structures, it is convenient to directly consider such structures. For example, a *for*-loop for lists of natural numbers may be defined:

```
consts
  mapM :: "('a ⇒ cmd) ⇒ 'a list ⇒ cmd"
primrec
  "mapM cmd []     = SKIP"
  "mapM cmd (i#is) = ({cmd i; mapM cmd is})"
constdefs
  for  :: "nat ⇒ nat ⇒ (nat ⇒ cmd) ⇒ cmd"
 "for start end body ≡ mapM body [start.. end]"
```

Unfolding lemmas for the *for*-loop may now be proved, including

```
 s<e ⟹ (FOR i:=s TO e DO b i OD) = ({b s; FOR i:=(s+1) TO e DO b i OD})
 s<e ⟹ (FOR i:=s TO e DO b i OD) = ({FOR i:=s TO (e-1) DO b i OD; b e})
```

General iteration is given by continuous functions, which are guaranteed to have a fixed point. These functions are not available in HOL but may be found in the logic of computable functions (LCF). Isabelle's object logic HOLCF [Müller *et al.* 1999] extends HOL with LCF. However, there are disadvantages: continuity requires that types used as domain or codomain for the functions form complete partial orders, including the Booleans.

```
constdefs
  loop    :: "tr S ⇒ cmd ⇒ cmd" ("WHILE _ DO _ OD")
 "loop c b ≡ fix (λF. IF c THEN {b ; F} ELSE SKIP FI)"
```

An unfolding lemma for general iteration in HOLCF can now be proved:

```
  (WHILE c DO p OD) = (IF c THEN {p; WHILE c DO p OD} FI)
```

Reasoning in HOLCF is not as accomplished as in HOL; a technical difficulty is that proof of the function's continuity is required (even for β-reduction). Thus, both models have their advantages and disadvantages. Both models are generic over the monad and allow standard programming language control structures to be encoded. The monad provides basic operations of the programming language, such as reading or writing variables and exception handling. These operations are now considered.

5.2 State Transformer Monads

A *state transformer* [Launchbury and Peyton Jones 1994] is a function $f : X \times S \Rightarrow Y \times S$ with a state S as additional input and output parameter. Each such f is equivalent to a curried function $f' : X \Rightarrow (S \Rightarrow Y \times S)$. Consequently state transformers may be modeled by the type constructor

```
types 'a st = S ⇒ ('a × S)
```

where `S` is the state parameter. To construct the monad structure, `MonadOps` is instantiated with `st` for `M`. This declares the monad operations, which are defined as follows: `bind` is composition (where `uncurry` maps a curried function f' as defined above to the corresponding uncurried f), and `eta`

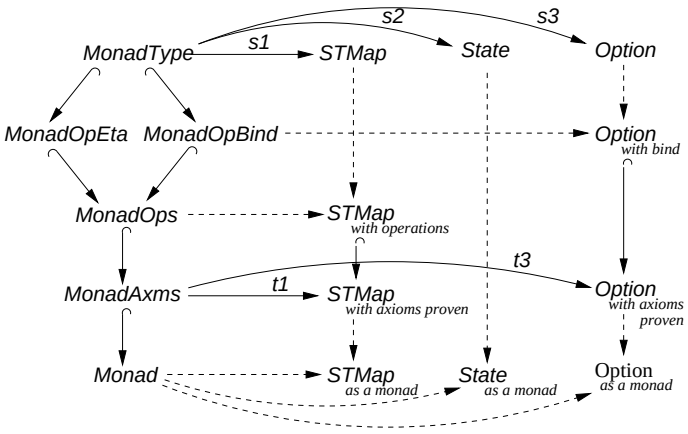


Figure 5.3: The different instantiations of *Monad*.

returns the identity.

thymorph *s1* : *MonadType* ⟶ *STMap* **maps** *[("MonadType.M"* ↦ *"STMap.st")]*
t_instantiate *MonadOps* **mapping** *s1*

defs
  bind_def: "bind f g ≡ *uncurry g o f"*
  eta_def: "eta x ≡ *λs. (x, s)"*

After proving the monad properties, a theory morphism *t1* instantiates the monad theory:

thymorph *t1* : *MonadAxms* ⟶ *STMap* **maps** *[("MonadType.M"* ↦ *"STMap.st")]*
t_instantiate *Monads* **mapping** *t1*

References are represented by a finite map of natural numbers to some value type *v*. The state is instantiated with the tuple consisting of the finite map and the value of the next free reference:

types *S* = *"(nat* × *(nat* ⇀ *v))"*
  ref = *nat*
  'a st = *"S* ⇒ *('a* × *S)"*

Finite maps from *a* to *b*, denoted *'a* ⇀ *'b*, are defined in Isabelle's library as *'a* ⇒ *'b option*. The option type *'b option* adds an extra element to *'b*, here representing 'undefined':

datatype *'b option = Some 'b | None*

The partial lookup function is written as normal application and the pointwise update of a map *f* at *x* with value *v* is written *f(x*↦ *v)*. The operations *lookup*, *update*, and *ref* to retrieve a value, store a value, and create a new reference are now defined (*fst* and *snd* are tuple projections, *case* is case distinction, and *arbitrary* is an arbitrary but unknown value):

```
constdefs
  lookup :: "ref ⇒ v st"
 "lookup r ≡ λs. (case (snd s) r of None ⇒ arbitrary | Some b ⇒ b, s)"
  upd    :: "ref ⇒ v ⇒ unit st"
 "upd r v  ≡ λs. ((), (fst s, (snd s) (r↦ v)))"
  ref    :: "v ⇒ ref st"
 "ref v ≡ λs. (fst s+1, (fst s+1, snd s (fst s↦ v)))"
```

This definition provides a consistent model of mutable state. Figure 5.3 shows the instantiating theory morphisms: `MonadOps` with `s1` to declare the monad operations, and `MonadAxms` with `t1` to express that `STMap` is a monad.

5.3 The Axiomatic State Monad

In the state transformer monad, the state is monomorphic; i.e., all values have the same type. A state with typed polymorphic references, as in Standard ML, uses an *axiomatic* characterisation of stateful computations, due to Plotkin and Power [2002]. In this approach, a monad `S` is *declared* with a polymorphic reference type `ref` and functions `ref`, `lookup`, and `update`:

```
typedecl 'a S
thymorph s2 : MonadType ⟶ State maps [("MonadType.M"↦ "State.S")]
t_instantiate Monad mapping s2

typedecl 'a ref
consts
  ref    :: "'a ⇒ ('a ref) S"
  lookup :: "'a ref ⇒ 'a S"          ("!_")
  update :: "'a ref ⇒ 'a ⇒ unit S" ("_ := _")
```

The behaviour of `lookup` and `update` and their interaction with `ref` are specified by *global* and *local state axioms*, respectively, including

```
  GS4:  "({r := v; w<- !r; t w}) = ({r:= v; t v})"
  LS2:  "({r<- ref v; w<- !r; t r w}) = ({r<- ref v; t r v})"
```

The axiomatisation is complete; in fact, any state transformer monad is a model of these axioms. Elements of any type may be stored and retrieved in an arbitrary way, as in this example computation:

```
{x<- ref 3; v<- ref "foo"; x := 5; v:= "bar"; !x}
```

Note that in contrast to `STMap` the theory `State` is not consistent by construction, but relies on metalogical arguments for consistency.

5.4 Exception Monads

The `option` datatype, modeling partiality, has already been introduced in Section 5.2. It can be extended to a monad which models computations with possible exceptions (or errors). The unit of the monad will be `Some`, and `bind` propagates the error value `None`. To derive the monad structure, instantiate `MonadOpBind` with a morphism from `MonadType` which maps `M` to

option. This declares the *bind* operation, which may be defined as follows:

```
thymorph s3 : MonadType ⟶ Option
    maps [("MonadType.M" ↦ "Datatype.option")]
t_instantiate MonadOpBind mapping s3

primrec
  "(None >>= f) = None"
  "((Some x) >>= f) = f x"
```

After proving the monad laws, a theory morphism from *Monad* to *Option* turns *Option* into a monad (automatically finding the only possible unit):

```
thymorph t3 : MonadAxms ⟶ Option
    maps [("MonadType.M" ↦ "Datatype.option")]
t_instantiate Monad mapping t3
```

The *Option* monad requires two instantiations, first of the declaration of *bind*, and then of the monad (see Figure 5.3). Errors are handled by a case distinction. If it is desirable to know *why* a computation has failed, the *Exception* monad may be obtained by a similar instantiation of the datatype

```
datatype 'a exn =  Some 'a | Error string
```

5.5 Imperative Programs in the State Monad

Some short examples now demonstrate how typical imperative programs may be captured using monads. We use the `State` monad (from Section 5.3). All expressions may have side effects, so functions *f*`: a => b` are lifted to functions *f*`: a => b M`. These can be applied to arguments of type `a M` using the Kleisli-application (the *bind* operation with the arguments exchanged). In particular, let `.<=` and `.=` denote the lifted relations `<=` and `=`. The increment, in-place addition, and multiplication
operations may now be defined:

```
  inc :: "nat ref ⇒ nat S"  ("_++")
 "inc i ≡ {iv<- !i; i:= iv+1; eta (iv+1)}"
  addto :: "nat ref ⇒ nat ⇒ unit S"  ("_+=_")
 "addto i c ≡ {iv<- !i; i:= iv+ c}"
  multo :: "nat ref ⇒ nat ⇒ unit S"  ("_*=_")
 "multo i f ≡  {iv<- !i; i:= iv* f}"
```

An imperative computation of the exponential may now be given as:

```
constdefs
  pow :: "nat ⇒ nat ⇒ nat S"
 "pow x n ≡ { p<- ref 1; FOR i:=1 TO n DO (p *= x) OD; !p }"
```

The correctness of this computation can be expressed as follows:

```
lemma pow_correct:
  "holds ({p<- pow x n; eta (p = x^n)})"
```

As an example in a different object logic, consider a *WHILE*-loop which computes the square root of an integer. In HOLCF, we can avoid the well-foundedness requirement of HOL, so we do not have to provide a termination measure. The flat cpo *int'* is used instead of *int*. (The corresponding lifted function definitions are omitted here.)

```
constdefs
  iroot :: "int' ⇒ int' S"
 "iroot a ≡
  { term <- ref 1; sum <- ref 1; i <- ref 0;
    WHILE ((!sum) .<= (eta a)) DO ({ term += 2; sum += 2; i++ }) OD;
    !i
  }"

lemma iroot_correct:
  "holds ({i<- iroot a; eta (i^2 <= a & a < (i+1)^2)})"
```

The examples show how monads and syntactic sugar allow imperative language features to be modeled in a natural way, and how we can benefit from the generic setting by choosing an appropriate object logic. The embedding is shallow; methods and procedures of the imperative language are captured by functions declared and defined in Isabelle, and Isabelle's proof tactics may be directly applied.

6. Related Work

6.1 Formal Program Development

Approaches to formal program development in theorem provers usually commit to a particular methodology and logic. In particular the refinement calculus and its variants have received attention, with encodings in HOL by Långbacka *et al.* [1995] and Butler *et al.* [1997], in Isabelle/HOL by Hemer *et al.* [2001], in Isabelle/ZF by Staples [1998], and in the Ergo prover by Carrington *et al.* [1998]. These systems do not take advantage of a logical framework prover. In contrast, the exploitation of the logical framework suggested in this paper allows a general methodology for reasoning reuse.

Specware [Srinivas and Jullig 1995] is a formal system for structured program development which supports refinement and specification morphisms (corresponding to theory morphisms) as the main structuring mechanism, and code generation for C, Java, and Lisp. An encoding of monads for Specware was recently developed by Pavlovic *et al.* [2003]. Specware relies on external provers (like Snark) for correctness proofs. In this sense, it is dual to our approach: Specware focuses on structuring and relies on external proof tools whereas we add structuring mechanisms to a theorem prover.

Other structured development systems which allow the manipulation of signatures and theories rather than object-logic formulae are VSE [Hutter *et al.* 2000], Maya [Autexier *et al.* 2002] and Hets [Mossakowski 2005]. VSE is based on a fixed object logic, whereas Maya and Hets are tools for reason-

ing in structured theories, relying on external provers for correctness proofs. In contrast to approaches based on algebraic specifications (such as VSE, Specware, and Hets), our approach enjoys the full benefit of Isabelle's expressivity, in particular higher-order functions and parametric polymorphism, in addition to the proof support offered by Isabelle.

Theory interpretations based on theory morphisms are supported in the theorem provers PVS and IMPS, see [Owre and Shankar 2001] and [Farmer 2000]. In contrast to these approaches, our work allows proofs to be automatically *converted* to the target theory: a proof in the target theory is derived which is independent of the morphism. Hence, our approach emphasises correctness in the sense that it directly constructs proofs.

6.2 Parametrisation in Isabelle

Other approaches to structuring theories in Isabelle include the representation of theories by meta- or object-logic formulae, and named proof contexts.

Isabelle has been used by Kolyang *et al.* [1996] and Lüth and Wolff [2000] to model transformation rules independent of the object logic, and by Anderson and Basin [2000] to model program schemata as meta-logic rules. Specifications are modeled as object-logic formulae. The prover provides good support for correctness reasoning in this setting, but the formulae become big and difficult to understand when specifications and transformation rules get more complex. Formulae quickly contain many free type variables. As a consequence, instantiation becomes error-prone and leads to unclear error messages. In our approach specifications are theories, enjoying the full prover support for modularisation. In addition, our approach is more general: in the previous approaches transformation rules are restricted by the object logic (e.g., allowing type or function variables), our approach allows any part of a theory to be transformed.

In order to use locally parametrised definitions, Isabelle provides a mechanism of named proof contexts called *locales* [Kammüller *et al.* 1999]. In contrast to our approach, locales can only be parametrised over polymorphic operations and axioms, not type constructors, and they can only be used to define a constant within a locale, i.e., local definitions for datatypes or recursive functions are not supported. (It is possible though to encode, e.g., the divide-and-conquer transformation rule presented in Section 4, by modeling the parameter types as type variables.) Compared to our approach, locales are better integrated into Isar (e.g., locales can be instantiated as needed during a proof). With regard to efficiency our approach requires that Isabelle is built with full proof objects (increasing space usage by approximately 30%), whereas locales are always available. In our approach, translating a theorem along a theory morphism requires replaying the proof object, which depends on the size of the proof. This process is less efficient than instantiating the definitions of a locale, which is done by forward resolution and thus independent of the size of the proof. In summary, locales are slightly more efficient and easier to use, but less powerful. One may roughly compare locales to

type classes (as in Haskell), and our parametrised theories to functors (as in Standard ML).

Constructor classes have recently been proposed for Isabelle by Huffman *et al.* [2005], using a universal domain construction in the HOLCF object logic. This approach elegantly handles infinite datatypes and continuous functions, but does not allow user defined definitions (generic syntactic sugar), and requires a large amount of manual proof work for transferring established theorems, which in our approach is automated. In addition, constructor classes are just one particular application of our approach.

7. Conclusions

This paper proposes extensions to generic theorem provers with proof terms which facilitate structured program development, namely theory morphisms and parametrised theories. For logical framework style theorem provers, the approach is independent of a particular object logic or specification language. Although conceptually simple the extensions have good expressive power, as demonstrated by the example applications to transformation rules and to denotational semantics based on monads.

A remaining challenge is to combine monads. For this, monad transformers [Liang *et al.* 1995] can be encoded as a constructor class in our setting. However, monad transformers may be criticised for their lack of modularity, as all possible combinations of monads need to be considered *a priori*. A better approach may be to define operations on monads, such as the coproduct [Lüth and Ghani 2002] or tensor product [Hyland *et al.* 2002]. The formalisation of such operations in Isabelle remains to be addressed.

Structured development techniques as proposed in this paper may also be combined with previous work by the authors on abstraction [Johnsen and Lüth 2004], which allows proofs to be generalised in order to derive inference rules. Using this technique, transformation rules may be derived from proven theorems rather than axiomatically assumed, and they are therefore correct by construction. The combination provides a method for deriving new transformation rules by abstracting existing developments, addressing the problem of how to derive transformation rules in new problem domains.

The proposed techniques have been implemented as conservative extensions to Isabelle, and as they do not require changes to Isabelle's logical kernel, they do not interfere with the prover's logical integrity. The techniques are fully integrated into Isabelle's proof and command language Isar. The implementation currently comprises about 7000 lines of SML code, half of which is the integration in Isar. The extensions run on Isabelle 2005, without any modifications to the prover. The source code, and the theory files for the examples presented in this paper are available at the project website `http://www.informatik.uni-bremen.de/~cxl/awe`.

A more elaborate case study is currently under way, as the approach and implementation presented in this paper (including the model of computa-

tional monads) are used in the ongoing development of a safe control program for the autonomous wheelchair *Rolland*.

References

Anderson, Penny and Basin, David. 2000. Program Development Schemata as Derived Rules. *Journal of Symbolic Computation 30*, 1 (July), 5–36.

Autexier, Serge, Hutter, Dieter, Mossakowski, Till, and Schairer, Axel. 2002. The Development Graph Manager Maya. In *Proc. 9th Int. Conf. Algebraic Methodology and Software Technology (AMAST'02)*, Volume 2422 of *Lecture Notes in Computer Science*. Springer, 495–501.

Berghofer, Stefan and Nipkow, Tobias. 2000. Proof terms for simply typed higher order logic. In *13th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'00)*, Volume 1869 of *Lecture Notes in Computer Science*. Springer, 38–52.

Bertot, Yves and Castéran, Pierre. 2004. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer.

Butler, Michael, Grundy, Jim, Långbacka, Thomas, Rukšėnas, Rimvydas, and von Wright, Joakim. 1997. The Refinement Calculator: Proof Support for Program Refinement. In *Proc. Formal Methods Pacific (FMP'97)*, Springer, 40–61.

Carrington, David A., Hayes, Ian J., Nickson, Ray, Watson, Geoffrey, and Welsh, Jim. 1998. A Program Refinement Tool. *Formal Aspects of Computing 10*, 97–124.

Farmer, William M. 2000. An Infrastructure for Intertheory Reasoning. In *17th International Conference on Automated Deduction (CADE-17)*, Volume 1831 of *Lecture Notes in Computer Science*. Springer, 115–131.

Hemer, David, Hayes, Ian, and Strooper, Paul. 2001. Refinement Calculus for Logic Programming in Isabelle/HOL. In *14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'01)*, Volume 2152 of *Lecture Notes in Computer Science*. Springer, 249–264.

Huffman, Brian, Matthews, John, and White, Peter. 2005. Axiomatic Constructor Classes in Isabelle/HOLCF. In *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, Volume 3603 of *Lecture Notes in Computer Science*. Springer, 147–162.

Hutter, Dieter, Langenstein, Bruno, Rock, Georg, Siekmann, Jörg H., Stephan, Werner, and Vogt, Roland. 2000. Formal software development in the Verification Support Environment (VSE). *Journal of Experimental and Theoretical Artificial Intelligence 12*, 4, 383–406.

Hyland, Martin, Plotkin, Gordon D., and Power, John. 2002. Combining Computational Effects: commutativity & sum. In *2nd IFIP International Conference on Theoretical Computer Science (TCS'02)*, Volume 223 of *IFIP Conference Proceedings*. Kluwer, 474–484.

Johnsen, Einar Broch and Lüth, Christoph. 2003. Abstracting Refinements for Transformation. *Nordic Journal of Computing 10*, 4, 313–336.

Johnsen, Einar Broch and Lüth, Christoph. 2004. Theorem Reuse by Proof Term Transformation. In *Proc. 17th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'04)*, Volume 3223 of *Lecture Notes in Computer Science*. Springer, 152–167.

Jones, Mark P. 1995. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal for Functional Programming 5*, 1 (Jan.), 1–35.

Kammüller, Florian, Wenzel, Markus, and Paulson, Lawrence C. 1999. Locales – A Sectioning Concept for Isabelle. In *Proc. 12th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'99)*, Volume 1690 of *Lecture Notes in Computer Science*. Springer, 149–166.

Kolyang, Santen, Thomas, and Wolff, Burkhart. 1996. Correct and User-Friendly Implementations of Transformation Systems. In *Formal Methods Europe (FME'96)*, Volume 1051 of *Lecture Notes in Computer Science*. Springer, 629–648.

Långbacka, Thomas, Rukšėnas, Rimvydas, and von Wright, Joakim. 1995. TkWinHOL: A Tool for Window Interference in HOL. In *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Volume 971 of *Lecture Notes in Computer Science*. Springer, 245–260.

Launchbury, John and Peyton Jones, Simon. 1994. Lazy Functional State Threads. In *Proc. ACM Conf. on Programming Languages Design and Implementation*, 24–35.

Liang, Sheng, Hudak, Paul, and Jones, Mark P. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*. ACM Press.

Lüth, Christoph and Ghani, Neil. 2002. Composing Monads Using Coproducts. In *Intl. Conf. on Functional Programming (ICFP'02)*. ACM Press, 133–144.

Lüth, Christoph and Wolff, Burkhart. 2000. TAS — A Generic Window Inference System. In *13th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'00)*, Volume 1869 of *Lecture Notes in Computer Science*. Springer, 405–422.

Moggi, Eugenio. 1989. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 14–23.

Mossakowski, Till. 2005. *Heterogeneous specification and the heterogeneous tool set*. Habilitationsschrift, Universität Bremen.

Müller, Olaf, Nipkow, Tobias, von Oheimb, David, and Slotosch, Oscar 1999. HOLCF = HOL + LCF. *Journal of Functional Programming 9*, 191–223.

Nipkow, Tobias, Paulson, Lawrence C., and Wenzel, Markus. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Volume 2283 of *Lecture Notes in Computer Science*. Springer.

Owre, Sam, Rushby, John, Shankar, Natarajan, and von Henke, Friedrich 1995. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering 21*, 2 (Feb.), 107–125.

Owre, Sam and Shankar, Natarajan 2001. Theory Interpretation in PVS. Tech. Report SRI-CSL-01-01, Computer Science Laboratory, SRI International, Menlo Park, CA.

Pavlovic, Dusko, Pepper, Peter, and Smith, Douglas R. 2003. Colimits for concurrent collectors. In *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, Volume 2772 of *Lecture Notes in Computer Science*. Springer, 568–597.

Peyton Jones, Simon (Editor). 2003. *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press.

Pfenning, Frank. 2001. Logical Frameworks. In *Handbook of Automated Reasoning*, Robinson, Alan and Voronkov, Andrei, Editors. Elsevier, 1063–1147.

Plotkin, Gordon and Power, John. 2002. Notions of Computation Determine Monads. In *Proc. 5th Intl. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*, Volume 2303 of *Lecture Notes in Computer Science*, 342–356.

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

Srinivas, Yellamraju V. and Jullig, Richard. 1995. Specware: Formal Support for Composing Software. In *Proc. Conf. Mathematics of Program Construction*, Volume 947 of *Lecture Notes in Computer Science*. Springer.

Staples, Mark. 1998. *A Mechanised Theory of Refinement*. PhD thesis, University of Cambridge.

Wenzel, Markus. 2001. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München.