An Approach to Transformational Development in Logical Frameworks

Maksym Bortin[1], Einar Broch Johnsen[2], and Christoph Lüth[1]

[1] FB 3 — Mathematics and Computer Science, Universität Bremen, Germany
email: {maxim,cxl}@informatik.uni-bremen.de
[2] Department of Informatics, University of Oslo, Norway
email: einarj@ifi.uio.no

Abstract This paper presents a formalisation of transformational program design tactics in a logical framework. Transformation rules are formalised as parameterised theories. To apply a transformation rule, the parameter part of the theory is matched to an instantiation by a theory morphisms. New transformation rules may be formulated directly in the logic, or they may be derived using proof-theoretical methods to generalise theorems. This formalisation is implemented in the theorem prover Isabelle, which has been extended to handle parameterised theories and theory morphisms for this purpose. The approach is demonstrated by applying the divide-and-conquer design tactic to derive a sorting algorithm, and by deriving and reapplying a transformation rule to introduce tail recursion for list traversal.

1 Introduction

Formal program development requires considerable proof effort, in addition to careful planning and design. Therefore, development tools should provide user support by automating proofs as far as possible, and aid in structuring developments. While there has been a lot of work in both theorem proving and algorithm design, these have to a large extent been separate efforts, and seldom been combined into one uniform system. Theorem provers such as PVS [15], Coq [4], and Isabelle [14] allow the formalisation of an impressive amount of mathematics. Algorithm design systems such as Specware [17] offer a large amount of algorithm theories, but their proof power is not comparable.

This paper sketches the design and implementation of a system combining algorithm design and transformational development with modern theorem proving. Our work takes a logical framework [7] as a starting point, and adds the notion of transformation rules to manipulate theories. We inherit the proof power and genericity of the logical framework, so our approach is in principle independent of the object logic or specification formalism used. New transformation rules may be formulated directly in the logic, but they may also be derived using proof-manipulation methods, extending previous work on generalisation of theorems by proof term transformation [10]. A distinct advantage of our work is its

Accepted for
Intl. Symposium on Logic-based Program Synthesis and Transformation (LOPSTR), Verona, Italy, Aug. 2004

light-weight approach: by adding some features to a powerful theorem proving environment, we obtain support for the design of algorithms.

The method has been implemented in Isabelle, with the advantage that we can use all of Isabelle's powerful proof tactics, libraries such as the datatype package, and logical encodings such as classical higher-order logic (HOL) and its extension with the logic of computable functions (HOLCF), which gives us access to a full functional language inside HOL. The method is directly applicable to any Isabelle-based transformational development method, e.g. [1, 8, 13, 18].

Technically, Isabelle is extended with a notion of parameterised theory, morphisms between theories, and abstraction. Parameterised theories model transformation rules, and theory morphisms model their application. Various datatypes and tactics are implemented in Standard ML to model theory morphisms, transformation rules, and the manipulation of theories. These are integrated in Isabelle to form the basic machinery for program development.

This paper is structured as follows: Sect. 2 considers a model of transformation rules as parameterised specifications. Sect. 3 briefly describes the implementation of key concepts such as theory morphisms, transformation rules and theorem abstraction in Isabelle. Sect. 4 sketches an example application using divide-and-conquer transformation, and Sect. 5 shows how to derive a transformation rule from the correctness proof of a transformation and reapply the rule to a different setting.

2 Transformation Rules as Parameterised Specifications

Signature and theory morphisms. A signature $\Sigma = \langle T, \Omega \rangle$ consists of type constructors T and operations Ω of given arities. A *signature morphism* $\sigma : \Sigma_1 \to \Sigma_2$ is a map between the type constructors and operations of the signatures, preserving the arities of the type constructors and the domain and range of the operations. A theory consists of a signature, a set of axioms, and a set of theorems proved in the theory. Given two theories Thy_1 and Thy_2, a *theory morphism* is a signature morphism together with a map of the axioms of Thy_1 to theorems or axioms of Thy_2.

Transformation rules. A transformation rule is a parameterised theory, given by a *parameter* P and a *body* B. The parameter contains the premises of the rule, e.g. type declarations, operations, and axioms which become *applicability conditions* when instantiated. The parameter are embedded into the body which contains the theory development, e.g. theorems and their proofs. In order to apply the rule, we provide a theory morphism from the parameter specification to a target theory I, instantiating parameter types and operations, and proving the validity of the applicability conditions in the target theory. Thus, a transform-

$$\begin{tikzcd} P \arrow[r, "\sigma"] \arrow[d, hook] & I \arrow[d, hook, dashed] \\ B \arrow[r, dashed, "\bar{\sigma}"] & R \end{tikzcd} \qquad (1)$$

ation $\Theta = (P, B)$ is instantiated with a target theory I by a theory morphism $\sigma : P \to I$. The resulting theory R extends I with the theorems and proofs of the rule body B, translated along σ (see diagram (1); the dashed morphisms are constructed automatically). This way, we reuse the proofs in the body of the transformation rule to automatically derive theorems about the instantiated parameter. This notion of parameterised specifications and their instantiation is also found in algebraic languages such as CASL [2].

Constructing new transformation rules. Transformation rules can be constructed from ordinary unparameterised theories as follows. Given a theory T, we first decide which part of T should become the parameter P, and which part remains in the body part B. (The parameter should contain type declarations and operations, the body the main proofs; exactly how to split T is a design decision.) We can then insert applicability conditions into the parameter, and use these to discharge premises of the theorems in the body, collecting various assumptions from the body into the parameter. Finally, we can replace free variables or type variables in the body by operations or type declarations from the parameter.

3 Implementing Transformation Rules

A logical framework is a meta-level inference system which can be used to specify other object level deductive systems. Object logic formulae are represented by abstract syntax and object logic derivability by a meta-logic predicate. In particular, object logic types are meta-logic terms and object logic proof rules are meta-logic axioms or theorems. The approach of this paper is implemented in the Isabelle prover. The implementation extends previous work on proof term transformations [9, 10], using Isabelle's proof terms [3]. Proof term transformations are used to map theorems from one theory to another, to apply a theory morphism, and to generalise theorems from a concrete to a more abstract setting.

Proof terms. The proof terms of Isabelle are defined in a typed λ-calculus corresponding to Isabelle's meta-logic under the Curry-Howard isomorphism. A term bound by a λ-abstraction corresponds to a proof rule premise and β-reduction to an inference. 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.

The manipulated meta-level proof terms are replayed to derive new object level inference rules. Hence, the logical framework ensures the correctness of the derived rules. We use this process to implement theory morphisms and abstraction. This way, we have a conservative implementation of abstraction and theory morphisms which does not compromise the integrity of Isabelle's logical kernel.

A representation of morphisms. We define abstract data types for signature morphisms and theory morphisms (*thy-morph*). The invariants of the abstract data types ensure that for signature morphisms a translated term typechecks if the original term did, and for theory morphisms that source theory axioms are provable in the target theory. Signature and theory morphisms are obtained using constructors which check the invariants, given source and target theories and the appropriate mappings.

Given a signature morphism $\sigma : \Sigma \to \Sigma'$, we can map any proof term in Σ to a proof term in Σ' by replacing operation symbols and types. Replaying the proof term in Σ', we get a translated theorem. Given a theory morphism, we additionally replace occurrences of axioms in the source theory by theorems of the target theory as given by the theory morphism. This also implies that theorems need to be translated in the order of their dependency, so if the proof of a theorem ϕ uses another theorem ψ, we need to translate ψ first.

Abstraction by proof transformation. A logical framework provides a good environment to design new transformation rules by systematic generalisation. In this process, we can generalise a given transformation rule into a more abstract form, making it more widely applicable.

Let π be a proof of a theorem ϕ, consisting of a series of inference steps in the meta-logic. The proposed abstraction process will transform π in a stepwise manner into a proof of a schematic theorem. In the following, let $\phi[x/t]$ and $\pi[x/t]$ denote substitution, replacing a term t of a given type by a term or variable x of the same type in a formula ϕ or proof π, renaming bound variables as needed to avoid variable capture. We have the following three basic abstractions (technical details may be found in [10]):

- *Making assumptions explicit:* A theorem ψ used in the proof π will appear as a proof constant $\mathit{Theorem} : \psi$. By the implication introduction rule (corresponding to abstraction in the λ-calculus), we can transform the proof π into the proof $\lambda h : \phi . \pi[h/\psi]$ of the theorem $\psi \Longrightarrow \phi$.
- *Abstracting function symbols:* When all implicit assumptions concerning a function symbol f have been made explicit, 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 f, and can be replaced by variables x throughout the proof. We obtain a proof $\pi[\mathsf{x}/f]$ of the theorem $\phi[\mathsf{x}/f]$. To emphasise the abstracted function symbols, schematic variables are denoted $\mathsf{x}, \mathsf{y}, \mathsf{z}$, etc.
- *Abstracting type constants:* 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.

These basic abstraction functions can be combined into more advanced *abstraction tactics*. For example, the tactic `abs_from_type`, used in Sect. 5, takes a theorem and a list of types as arguments and returns a theorem independent of the given types. In order to abstract over a type t, we have to find all operations $f_1, \ldots, f_n$ in the signature of which t occurs, and in turn all theorems $\phi_1, \ldots, \phi_m$ which contain any of these operations. We first abstract over all the theorems, then over all the operations and finally over the type.

A representation of transformation rules. In order to use transformation rules in Isabelle, we introduce a type *trafo-rule* of transformation rules between theories, consisting of parameter and body theories. The application of a transformation rule is given by a function on this type, and results in a new theory translating the definitions of the body along the theory morphism:

datatype *trafo-rule = TrafoRule of { param :: thy, body :: thy }*
val *apply : trafo-rule * thy-morph -> thy*

The new types, constants, and theorems are given generic names; a variant of the application function allows explicit naming.

4 Example: Divide and Conquer Transformation

The application of a transformation rule is illustrated by formalising the *divide-and-conquer* design tactic [16], which recursively synthesises a function $f : D_f \Rightarrow R_f$. If a value $f(x)$ cannot be given directly for $x \in D_f$, the computation is decomposed into two smaller parts, to which f is recursively applied, and the results are recomposed. An auxiliary value (and function g) may be needed for recomposition. The parameters of the tactic are the domain and range types of f and g, D_f, R_f, D_g, R_g; and functions *measure* for well-foundedness of the domain D_f, *primitive* to recognise values handled directly by *dir-solve*, and functions for composition and decomposition:

$$\begin{array}{llll} \mathit{measure} & :: D_f \Rightarrow \mathit{nat} & \mathit{decompose} & :: D_f \Rightarrow (D_g \times (D_f \times D_f)) \\ \mathit{primitive} & :: D_f \Rightarrow \mathit{bool} & \mathit{compose} & :: (R_g \times (R_f \times R_f)) \Rightarrow R_f \\ \mathit{dir\text{-}solve} & :: D_f \Rightarrow R_f & g & :: D_g \Rightarrow R_g \end{array}$$

The specification of the actual target function f (and the auxiliary function g) is given by four predicates, in a pre- and postcondition style:

$$\begin{array}{ll} f_{pre} :: D_f \Rightarrow \mathit{bool} & f_{post} :: D_f \Rightarrow R_f \Rightarrow \mathit{bool} \\ g_{pre} :: D_g \Rightarrow \mathit{bool} & g_{post} :: D_g \Rightarrow R_g \Rightarrow \mathit{bool} \end{array}$$

The functions have to satisfy a total of six axioms, including

$$\begin{array}{l} \textbf{axiom}\ \mathit{A\text{-}DCMP} :\ \forall\ x.\ f_{pre}\ x \wedge (\neg\ \mathit{primitive}\ x) \longrightarrow \\ \qquad\qquad \mathit{let}\ (x_1, x_2, x_3) = \mathit{decompose}\ (x)\ \mathit{in}\ g_{pre}\ x_1 \wedge f_{pre}\ x_2 \wedge f_{pre}\ x_3 \end{array}$$

and the requirement that composition preserves the specification: if we decompose a term in the domain, apply the function to the decomposed parts, and recompose, the result will satisfy the specification. Based on these functions, we can give a recursive definition of f in the body of the transformation rule, and prove that it satisfies the given specification (theorem DaC below). Note that we allow a tree-like recursion (two recursive calls) instead of a linear one.

consts $f :: \ D_f \Rightarrow R_f$
recdef $f\ x \equiv (\mathit{if}\ (\mathit{primitive}\ x)\ \mathit{then}\ (\mathit{dir\text{-}Solve}\ x)$
$\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ (\mathit{compose} \circ (\mathit{prod}\ g\ f\ f) \circ \mathit{decompose})\ x\ )$
theorem $\mathit{DaC} :\ \forall\ x.\ f_{pre}\ x \longrightarrow f_{post}\ x\ (f\ x)$

Well-foundedness of the recursive definition is ensured by the *measure* function, and *prod* is an auxiliary higher-order function which tuples functions, so $\mathit{prod}\,(f\,g\,h)\,x = (fx, gx, hx)$.

Sorting lists. We now apply the divide-and-conquer tactic to synthesise a sorting algorithm for *nat list.* We first instantiate the problem domain, i.e. the predicates specifying that lists are to be sorted. We further need to instantiate the six functions above. If decomposition splits a list into its head, a list of the elements smaller than the head, and a list of the elements larger or equal to the head, and composition concatenates the (recursively sorted) lists, we get *quicksort.*

The actual application of the transformation rule is achieved (at the ML level) by specifying a signature morphism σ from the parameter theory of divide-and-conquer to a theory *QuickSort* with a map for types

$$\{\ D_f \mapsto \mathit{nat\ list}, R_f \mapsto \mathit{nat\ list}, D_g \mapsto \mathit{nat}, R_g \mapsto \mathit{nat}\ \}$$

and a map for constants which in particular instantiates f_{post} with an appropriate predicate *sorted* on lists. In order to construct a theory morphism, we need to prove the applicability conditions of the tactic after translation to the *QuickSort* theory. We define a mapping from the names of the applicability conditions to theorems proved in *QuickSort*, and use the constructor to define a theory morphism σ_T, extending σ. The application of the transformation rule then gives us a recursive definition of *quicksort* together with a proof of its correctness. By decomposing a list into its head and tail, we get *insertsort* as another instance of the design tactic. In this case, the composition function has to insert an element back into a (recursively) sorted list, preserving sortedness; this function is in turn another instance of divide-and-conquer.

5 Example: Derivation and Reapplication of a Rule

In this section, we illustrate how to derive a generalised transformation rule by generalising a standard development, and then reapply it in different settings. We use a standard definition of natural numbers *nat* with an addition operation $+ :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$, and a type α *list* of lists with elements of type α. Following Isabelle convention, we denote the empty list by $[\,]$, and for any non-empty list l we let $\mathit{hd}(l)$ and $\mathit{tl}(l)$ denote the head and the tail of l, respectively. Now consider a function $\mathit{sum} :: \mathit{nat\ list} \Rightarrow \mathit{nat}$, which takes as argument a list of natural numbers and returns the sum of the elements in the list. This function can be recursively defined by

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

and the well-foundedness of the definition is ensured by the measure relation induced by the *length* of the function's list argument. It is well known that an equivalent tail-recursive definition can be given; here, we let $\mathit{sum2} :: \mathit{nat\ list} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat}$ be recursively defined by

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

with the same well-foundedness requirement as given above. For all x, $\mathit{sum}(x)$ is the same as $\mathit{sum2}(x,0)$. This is expressed by the formula

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

which states the correctness of the corresponding transformation. With Isabelle, we can prove this formula by induction on x using three lemmas. The most involved of these is $\forall x\, a\, l\,.\, \mathit{sum2}(l, x + a) = x + \mathit{sum2}(l,a)$.

We will now generalise this theorem to derive a transformation rule which can be instantiated to lists of other types other than *nat*. Applying the tactic `abs_from_type` outlined in Sect. 3, the type *nat* is replaced by a type variable α, 0 and $+$ by schematic variables z and p (typed as $\mathsf{z} :: \alpha$ and $\mathsf{p} :: \alpha \Rightarrow \alpha \Rightarrow \alpha$), and *sum* and *sum2* by schematic variables sum and $\mathsf{sum2}$, respectively. This results in the following generalised theorem *TailRec*:

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

In the abstracted theorem *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 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.

The abstracted theorem provides a correctness proof for a transformation rule, allowing us to transform a recursive definition into a tail-recursive one. In the notation of Sect. 2, the parameter of the transformation rule consists of a type declaration α, the operations $\mathsf{z} :: \alpha$, $\mathsf{p} :: \alpha \Rightarrow \alpha \Rightarrow \alpha$ and sum, and the monoid properties as application conditions. The body contains the recursive definition of $\mathsf{sum2}$ and the correctness theorem *TailRec*.

We now illustrate how to use the transformation rule. Let @ denote string concatenation and ε the empty string. Consider a function

$$\mathit{concat}\ x \equiv \mathit{if}\ x = [\,]\ \mathit{then}\ \varepsilon\ \mathit{else}\ (\mathit{hd}\ x)\ @\ (\mathit{concat}\ (\mathit{tl}\ x))$$

which concatenates all strings in a list of strings. Instantiating *TailRec* with

$$\sigma = [\, \mathit{concat}/\mathsf{sum}, \mathit{fastcat}/\mathsf{sum2}, [\,]/\mathsf{z}, @/\mathsf{p}\,]$$

also instantiates the type variable α to *string*. In practice, we provide the instantiation of sum and $\mathsf{sum2}$, and Isabelle's higher-order unification instantiates the remaining variables for us. In the instantiated theorem, the conclusion states that $\mathit{concat}\ (x) = \mathit{fastcat}\ (x, \varepsilon)$. The first premise is the definition of *concat*, the second is the proposed tail-recursive definition

$$\mathit{fastcat}\ (x, y) \equiv \mathit{if}\ x = [\,]\ \mathit{then}\ y\ \ \mathit{else}\ \mathit{fastcat}\ (\mathit{tl}\ x,\ \ y\ @\ (\mathit{hd}\ x)),$$

and the remaining premises are the applicability conditions,which are routinely discharged by the theorem prover:

$$\forall u.\, \varepsilon\ @\ u = u, \quad \forall u.\, u\ @\ \varepsilon = u, \quad \text{and} \quad \forall u\, v\, c.\, u\ @\ (v\ @\ c) = (u\ @\ v)\ @\ c.$$

The theorem *TailRec* can also be instantiated to show that for a monoid (f, e), the functions which fold from the right (*foldr*) and fold from the left (*foldl*) coincide. With the substitution

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

we get the following theorem:

$$\begin{aligned}&(\forall u\, .\, \mathsf{f}\ \mathsf{e}\ u = u \wedge \forall u\, .\, \mathsf{f}\ u\ \mathsf{e} = u \wedge \forall u\ v\ c\, .\, \mathsf{f}\ u\ (\mathsf{f}\ v\ c) = \mathsf{f}\ (\mathsf{f}\ u\ v)\ c)\\ &\qquad \longrightarrow\ \mathit{foldr}\ \mathsf{f}\ \mathsf{x}\ \mathsf{e} = \mathit{foldl}\ \mathsf{f}\ \mathsf{e}\ \mathsf{x}\end{aligned}$$

Note that here the operations f and e remain schematic variables, and the type variable α remains free. The first two applicability conditions of the instantiated *TailRec* theorem are discharged automatically by the prover.

6 Summary and Future Work

This paper presents an approach to transformational development in logical framework style theorem provers, extending previous work by the authors [9,10]. The resulting system combines powerful tactical proof support with algorithm design. Whereas theorem provers have been used previously to implement formal software development systems [5,6,8,11–13,18], these systems do not exploit the advantages of logical frameworks. In particular, proof term manipulation in the logical framework provides support for automated generalisation of developments while preserving the correctness proofs, and support for structuring mechanisms such as theory morphisms.

In contrast to previous work on transformational development in theorem provers [1,11,13], we manipulate the structure of the theories, and do not reflect the structure back into the object level. The latter can lead to big, unreadable formulae. For example, the divide-and-conquer rule as formalised in [13] is given by a single formula with six free variables for the operations, and four different free type variables. Furthermore the instantiation of such a formula is very error-prone due to the number of free type variables, and results in correspondingly unclear error messages. Our system treats theories as transformation objects, similar to Specware [17].

Presently, the application of transformation rules requires programming in Isabelle's low-level SML interface. The integration of our approach into Isabelle's high-level proof language Isar [14] is currently under development. The integration with Isar will allow formal developments in a more high-level, user-friendly way. At this level, transformation rule are represented in a single theory containing both parameter and body. The target theory will include the theory morphism and apply the transformation rule, renaming the resulting functions as desired. The integration of the approach with Isar may be illustrated as follows, using the example of Sect 4:

theory $DaC = Main:$
param
typedecl $D_f, \ldots$
consts
  $compose :: (R_g \times (R_f \times R_f)) \Rightarrow R_f$
  $\ldots$
axiom *A-DCMP*
  $\ldots$
body
recdef $f\ x = \ \ldots$

theory $QuickSort = Main:$
constdefs
$\ldots$

apply
  $DaC\ [\ D_f \longmapsto nat\ list,$
    $Compose \longmapsto qsort\text{-}decompose,$
    $A\text{-}DCMP \longmapsto a\text{-}dcmp, \ldots\ ]$
rename $[\ f \ \longmapsto quicksort\ ]$

A disadvantage here is that the synthesised functions do not textually occur in the theory. There are two ways to remedy this: one is to add theory transformation tactics which manipulate the theory text itself, but we prefer the second: extending Isabelle's document generation system to generate documentation for the generated theory.

An advantage of our work is the light-weight approach: by adding little more than a thousand lines of SML to a standard tactical prover, we can do algorithm design in it combined with the prover's proof support facilities. The approach should be evaluated with more elaborate case studies. This remains future work.

References

1. P. Anderson and D. Basin. Program development schemata as derived rules. *Journal of Symbolic Computation*, 30(1):5–36, July 2000.
2. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.
3. 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.
4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* EATCS Texts in Theoretical Computer Science. Springer, 2004.
5. 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 (FMP'97)*, pages 40–61. Springer, 1997.

6. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10:97–124, 1998.
7. 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.
8. 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.
9. E. B. Johnsen and C. Lüth. Abstracting refinements for transformation. *Nordic Journal of Computing*, 10(4):313–336, 2003.
10. E. B. Johnsen and C. Lüth. Theorem reuse by proof term transformation. In *17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'04)*, Lecture Notes in Computer Science. Springer, 2004. To appear.
11. 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*, number 1051 in Lecture Notes in Computer Science, pages 629– 648. Springer, 1996.
12. 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. Springer, 1995.
13. C. Lüth and B. Wolff. TAS — a generic window inference system. 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 405–422. Springer, 2000.
14. 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.
15. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.
16. D. R. Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5(1):37–58, Feb. 1985.
17. 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.
18. M. Staples. *A Mechanised Theory of Refinement.* PhD thesis, University of Cambridge, 1998.