System Support for the Interactive Transformation of Functional Programs

Helmuth Partsch Wolfram Schulte Ton Vullinghs
Universität Ulm
Fakultät für Informatik
D-89069 Ulm, Germany
+49 731 5024161
{partschh,wolfram,ton}@informatik.uni-ulm.de

ABSTRACT
This paper describes the program transformation system *Ultra*. The intended use of *Ultra* is to assist programmers in the formal derivation of correct and efficient programs from high-level descriptive or operational specifications. The most salient features of *Ultra* are its sound theoretical foundation, its lean, and portable implementation, its extendability, its flexible and convenient way to express transformation tasks, and its comfortable user interface.

Keywords
Transformational programming, functional programming, equational reasoning.

1 INTRODUCTION

There are many reasons to advocate the use of formal methods to support the construction of complex and technically advanced software systems. Transformational programming is such a technique [10]. In essence, it is the systematic manipulation of programs, a mechanical task that can be automated using *Ultra* (*Ul*m *Tra*nsformation System). *Ultra* basically supports equational reasoning about functional programs using defining equations, algebraic laws of underlying data structures, and transformation rules. Writing correct and efficient programs in *Ultra* is divided in two phases. First an initial, maybe inefficient or even non-operational program is developed, of which correctness is easy to show. In the second phase, correctness preserving transformation rules are applied to transform the initial program into a semantically equivalent but more efficient version.

To derive new functions from existing ones *Ultra* supports the unfold-fold strategy [4]. The basic elements of this strategy are unfolding (replace a call to a function by its instantiated body), folding (the inverse of unfolding), instantiation (introduce an instance of an already existing equation), abstraction (introduce local names for (common) subexpressions), rule application (apply properties that hold in the problem domain (usually axioms of the underlying data structures) or for the particular algorithm at hand), and definition (add a new defining equation to the actual program).

The unfold-fold paradigm is a basic technique that is often used in combination with other more advanced strategies. Strategies are supported in *Ultra* in the form of specialised transformation rules or combinations of transformation rules (called tactics). *Ultra* provides for example tactics and rules to transform functions into tail recursive form, for differencing (also called strength reduction), for function tupling, for the elimination of function compositions (also called fusion or deforestation), for tabulation and memoization, and for partial evaluation.

Furthermore, *Ultra* can be used to develop operational algorithms from descriptive (non-operational) ones. The descriptive constructs that are supported are the qualified expressions *forall*, *exists*, *some* (select one element from many alternatives) and *that* (select a uniquely characterised element). Operational programs can be derived from a descriptive specification by generalisation, by enumeration or by specialised strategies, such as divide and conquer.

Ultra does not only support modifying terms but is also useful for bookkeeping- and development-navigating tasks. In the rest of this paper we describe the transformation system in more detail. In Sect. 2 we present the basic principles of program transformation. Section 3 discusses the user interface of *Ultra*. Section 4 shows a sample session with the system. Section 5 gives a brief summary of first experiences.

2 THE TRANSFORMATION CALCULUS

The transformation calculus has its roots in the transformation semantics of the CIP system [1, 13] and is based on a two-level Horn clause logic.

Terms. The main purpose of a program transformation system is the (interactive) manipulation of terms. In

the case of *Ultra*, we use an extended[1] subset[2] of Gofer [6] to formulate the target program. Gofer is a higher-order, referential transparent, strongly typed functional language with lazy semantics. Using a concrete functional language as the object for transformations has a number of obvious benefits. First of all, specifications that are transformed are executable, which enables a direct prototyping. Referential transparency implies that the meaning of an expression is denoted by its value and that there are no side effects when computing this value. As a consequence, subexpressions may be replaced freely by other expressions having the same value, thus providing a simple but sound basis for equational reasoning.

Examples of terms are given in Fig. 1. The $(+\!\!+)$ operator concatenates two lists of elements of type a. The function *wrap* converts an element into a singleton list. The expression $[]$ denotes the empty list, $(:)$ is the constructor for non-empty lists.

$$
\begin{array}{lcl}
(+\!\!+) & :: & [a] \to [a] \to [a] \\
[] +\!\!+ ys & = & ys \\
(x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys) \\
\\
wrap & :: & a \to [a] \\
wrap\ x & = & x : []
\end{array}
$$

Figure 1: Examples of terms

Program Schemes. Program schemes are a generalisation of terms. A program scheme is a term that may contain free variables. An example of a program scheme is the expression $wrap\ x +\!\!+ xs$. A program scheme is instantiated by mapping its free variables to program schemes. For example, $wrap\ 1 +\!\!+ [2,3,4]$ is an instance of the above program scheme.

Transformation Rules. A transformation rule is a special kind of logical inference where the conclusion either denotes an equivalence or descendency relation between two program schemes. The premises of the inference denote the applicability conditions for the rule. We use the following notation for transformation rules:

$$cs \models i \Longleftrightarrow o \qquad \text{or} \qquad cs \models i \Longrightarrow o$$

where i is called the input scheme, o is called the output scheme and cs represents a list of applicability conditions. Applicability conditions are positive implicational formulas and usually restrict the possible values of the variables in i and o by some semantical predicate.

The definitive goal of a transformation session is to derive a new transformation rule. This rule reflects the relation between the initial and final term of the derivation process. The new rule can be added to the knowledge base of the transformation system and can be used in a future session.

Basic transformation rules are either derived from the programming language semantics or are given by the user to describe properties of the underlying data structures. Typical language defined rules are for example β-reduction and case simplification. The definition of user-defined transformation rules is similar to the declaration of ordinary Gofer functions. For example, the unconditional law *wrap_append* is written in the following way:

$$
\begin{array}{l}
wrap\_append\ x\ xs = \\
\quad [] \models wrap\ x +\!\!+ xs \Longleftrightarrow x : xs
\end{array}
$$

where $[]$ denotes the empty list of applicability conditions.

The scheme variables of a transformation rule are exactly those variables that are listed as parameters in the rule's declaration. In the example above, x and xs correspond to the scheme variables in the input and output scheme of the transformation rule. If this rule is applied, first-order pattern matching is used to instantiate the scheme variables.

Algebraic Properties. Consider the following transformation rule[3]:

$$
\begin{array}{l}
intro\_right\_neutral\ op\ a\ e = \\
\quad [right\_neutral\ op\ e] \models a \Longleftrightarrow a\ `op`\ e
\end{array}
$$

This rule states that it is safe to replace some expression a by some new expression $a\ `op`\ e$, provided e is right neutral for op. Just like $\Longleftrightarrow$, *right_neutral* is a semantical predicate on terms. Another example of a rule is *apply_associativity*.

$$
\begin{array}{l}
apply\_associativity\ op\ a\ b\ c = \\
\quad [associative\ op] \models \\
\qquad a\ `op`\ (b\ `op`\ c) \Longleftrightarrow (a\ `op`\ b)\ `op`\ c
\end{array}
$$

Instances of algebraic properties like *associative* are called *facts*. If a rule is applied that contains an algebraic property in its premises, *Ultra* tries to derive proper instantiations for the scheme variables by searching the *fact-base* (see Fig. 2). If no, or no unique matching instances are found, the user is asked to enter the values for the scheme variables.

[1] To support the *forall-*, *exists-*, *some-* and *that-* expressions we extended the transformation language with a non-deterministic choice operator.

[2] The current *Ultra*-type system is based on the standard Hindley/Milner system. In particular, *Ultra* does not yet support type or constructor classes.

[3] Any function *op* can be used as an infix operator by writing *op* between single quotes.

$$right\_neutral\ (+)\ 0$$
$$right\_neutral\ (+\!\!+)\ []$$

$$associative\ (+)$$
$$associative\ (+\!\!+)$$

Figure 2: A simple fact-base

Using a prolog-like syntax, we can formulate additional algebraic properties for terms. For example, the *monoid* property is defined in the following way:

$$monoid\ op\ e \vdash neutral\ op\ e, associative\ op$$

Tactics. The user can transform a program by repeatedly searching and applying adequate transformation rules. This labour-intensive process can be partly controlled and automated using *tactics* and *tactic combinators* [11]. A tactic is a function that maps some term into a new term. From an abstract point of view, any transformation rule can be regarded as a tactic. Tactic combinators handle the composition of tactics. The most common combinators are:

- $t_1\ andT\ t_2$: try to apply tactic t_1, if successful try to apply tactic t_2,
- $t_1\ orT\ t_2$: try to apply tactic t_1, if not successful try to apply tactic t_2,
- $repeatT\ t$: try to apply tactic t as many times as possible
- $dfsT\ t$: try to apply tactic t recursively on all subterms of the actual term in a depth-first order.

Using these combinators we can write complex tactics that carry out larger transformation tasks. An example of a tactic is shown in Fig. 3. This tactic tries to repeat

$$simplify\ =\ dfsT\ (repeatT\ simple\_step)$$
$$\textbf{where}\ simple\_step\ =\ case\_simple\ `orT`$$
$$distr\_simple\ `orT`\ const\_simple\ `orT`\ldots$$

Figure 3: The simplify-tactic

the application of a single simplification step (e.g., a case simplification) on every subterm of the actual term. Another powerful tactic is the solve tactic. This tactic simplifies or even eliminates descriptive constructs in favour of operational ones. If, for instance, a qualified expression is given that describes only a single result, the solve tactic tries to find it by performing a special kind of resolution.

Theories. Programs, rules and algebraic properties are logically organised into theories. The program transformation system offers a number of operations to apply and manipulate these theories.

3 THE SYSTEM

Figure 4 gives an impression of the graphical user interface (GUI) of the actual implementation of *Ultra*. The interface consists of the following components:

- The Editor: This interface component supports context sensitive editing of terms. Context sensitive editing means that the editor offers functions to select and highlight parts of the displayed term. What part of the term is exactly highlighted depends on the syntactical structure of the term. By selecting a part of the displayed term, the user indicates that this part will be the input for the next transformation step. For some transformation tasks (e.g., for a *let*-abstraction) the system supports multiple selections.
- The Controls: The user interface defines command buttons to support the basic tasks of the unfold-fold paradigm. For example, if the user presses the Fold button, the system performs a fold step on the actually marked term. Furthermore, general tactics like the simplifier and the solver, or a tactic to rearrange associative and commutative operators can be directly invoked by pressing the corresponding buttons. Specialised tactics can be found in the pull-down menu Tactics. The other menus mainly hide operations for organisational tasks like loading new theories, starting or stopping derivations, or generating (LaTeX) output for the actual derivation.
  Because a derivation process consists of a number of transformation steps, the system offers buttons to support a number of primitive navigation tasks, like Back (undo) and Forward (undo an undo step). Additionally, the user can focus on a particular subterm by zooming in on this term. To return to a surrounding term, Zoom out has to be invoked.
- The Database: Various objects involved in a transformation session (e.g., terms and rules) have to be administered by the system. These objects are contained in the knowledge base of the system. The user interface provides facilities to access and update the information in the knowledge base. The user can select terms and rules to indicate that they serve as input for subsequent system actions.

Ultra is written in TkGofer [17] (an extension of Gofer), which is especially suited for the definition of GUIs in a functional language. The convenient and flexible way to define and modify the layout and functionality of GUI systems in TkGofer enables a rapid integration of new features (e.g., add new tactics) in *Ultra*.

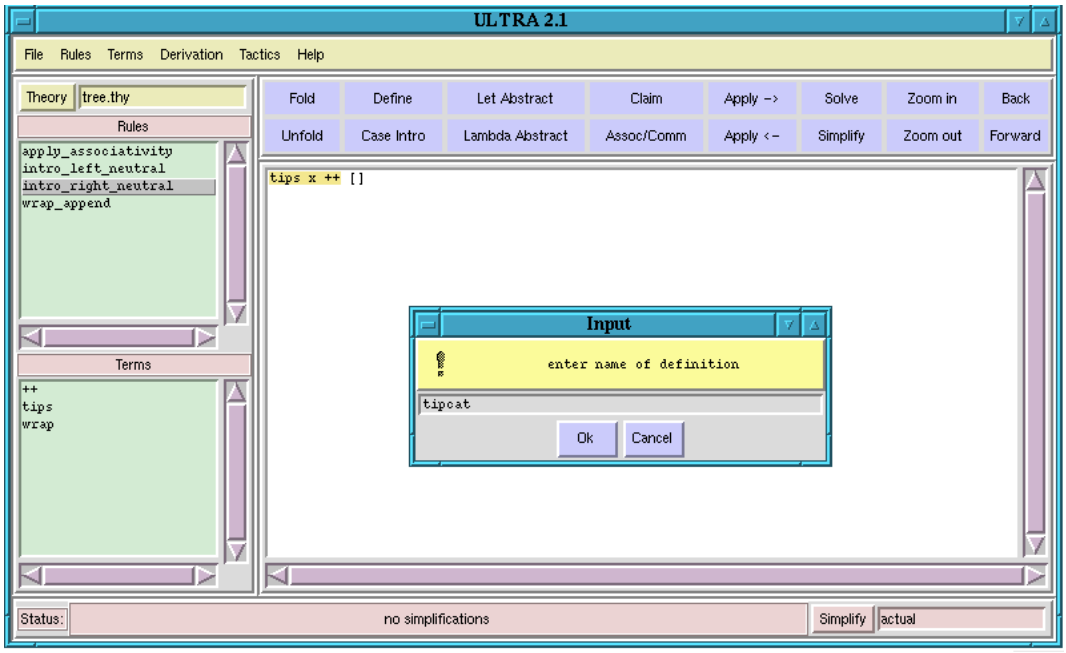

Figure 4: The main window of the *Ultra* system.

4 A SAMPLE SESSION

In this section we describe a sample session with the *Ultra* system. We try to derive an efficient solution for traversing a binary tree and concatenating the tips (leaves) of this tree. The example can also be found in [2] and illustrates the *generalisation* technique.

Typically, we start a transformation session with the formalisation of the problem theory, that is, the definition of the relevant data structures, operations and properties. The tree traversal theory is shown in Fig. 5. After loading this theory, *Ultra* parses and type-checks it. Since this theory uses the functions *wrap* and $+\!\!+$, we

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree}\ a & = & \mathit{Tip}\ a \\
 & | & \mathit{Bin}(\mathit{Tree}\ a)(\mathit{Tree}\ a) \\
 & & \\
\mathit{tips} & :: & \mathit{Tree}\ a \to [a] \\
\mathit{tips}(\mathit{Tip}\ a) & = & \mathit{wrap}\ a \\
\mathit{tips}(\mathit{Bin}\ l\ r) & = & \mathit{tips}\ l +\!\!+ \mathit{tips}\ r
\end{array}
$$

Figure 5: The tree traversal theory

assume that the user has already loaded the theory containing definitions, rules and facts about these functions (see Sect. 2).

The function *tips* returns the inorder list of tips of the tree. The function is easy to understand and fulfils the required behaviour. Unfortunately, it is not very efficient. In Gofer, like in most functional languages, the computation of $x +\!\!+ y$ takes time proportional to the length of x. Therefore, the above definition of *tips* is a quadratic-time program.

Our goal is to improve the efficiency of *tips*. We develop a function *tipcat* such that $\mathit{tipcat}\ t\ x = \mathit{tips}\ t +\!\!+ x$. Since the empty list is a neutral element for append we have $\mathit{tips}\ t = \mathit{tipcat}\ t\ []$, so *tipcat* is a generalisation of our problem. The addition of an extra parameter is known as accumulation and is a well-known technique for improving the efficiency of functional programs [2].

Our first goal is to define the function *tipcat* by generalising *tips*. We enter the term $\lambda x \to \mathit{tips}\ x$ at the console to start the derivation. If we press the enter key, this term is parsed and checked and displayed in the transformation editor. Now the derivation can start. The derivation process is shown in Fig. 6. Selected terms are underlined.

$$
\begin{array}{l}
\lambda\, x \to \underline{\mathit{tips}\ x} \\
\Longleftrightarrow \{\!|\ \text{apply}\ \mathit{intro\_right\_neutral}\ |\!\} \\
\quad \lambda\, x \to \underline{\mathit{tips}\ x +\!\!+ []} \\
\Longleftrightarrow \{\!|\ \text{define}\ \mathit{tipcat}\ |\!\} \\
\quad \lambda\, x \to \mathit{tipcat}\ x\ []
\end{array}
$$

Figure 6: Generalising *tips*

The system automatically finds the right neutral element for $+\!\!+$ (the user just has to press the Apply button). The define step is done by marking the partial application of $+\!\!+$ and pressing the button Define. This opens a dialogue where we enter the name of the new definition (*tipcat*). The selected term is immediately folded to this new definition. We stop the derivation. The system asks for the name of the derived rule. We enter *fastTips* and the rule

$$fastTips = [] \models \lambda\, x \to tips\ x \Longleftrightarrow \lambda\, x \to tipcat\ x\ []$$

is added to the system catalog. This rule reflects the intended generalisation.

Our next goal is to derive a recursive version of *tipcat* that is independent of the functions $+\!\!+$ and *tips*. For this we use the standard unfold-fold strategy. Figure 7 shows the steps we have to make in *Ultra*. We start

$$\begin{array}{l}
\quad \lambda\, x\ y \to \underline{tipcat\ x\ y} \\
\Longleftrightarrow \{|\ \text{unfold } tipcat\ |\} \\
\quad \lambda\, x\ y \to \underline{tips\ x} +\!\!+ y \\
\Longleftrightarrow \{|\ \text{unfold } tips\ |\} \\
\quad \lambda\, x\ y \to \underline{(\textbf{case}\ x\ \textbf{of}} \\
\qquad \underline{Tip\ v \to wrap\ v} \\
\qquad \underline{Bin\ l\ r \to tips\ l +\!\!+ tips\ r) +\!\!+ y} \\
\Longleftrightarrow \{|\ \text{simplify}\ |\} \\
\quad \lambda\, x\ y \to \textbf{case}\ x\ \textbf{of} \\
\qquad Tip\ v \to wrap\ v +\!\!+ y \\
\qquad Bin\ l\ r \to (tips\ l +\!\!+ tips\ r) +\!\!+ y
\end{array}$$

Figure 7: Derivation of *tipcat*: Initial unfolding steps

the derivation with unfolding *tipcat*, followed by unfolding *tips*. The simplify step will distribute $+\!\!+\ y$ in the branches of the case expression.

We treat the two cases independently by using the focus (zoom in/zoom out) feature of the system. First, we concentrate on the terminating *Tip* case (see Fig. 8). We

$$\begin{array}{l}
\Longleftrightarrow \{|\ \ldots \text{ zoom } \textbf{case}\ Tip\ |\} \\
\quad \underline{wrap\ v +\!\!+ y} \\
\Longleftrightarrow \{|\ \text{apply } wrap_append\ |\} \\
\quad v : y
\end{array}$$

Figure 8: Derivation of *tipcat*: The *Tip* case

just have to apply the rule *wrap_append* (as introduced in Sect. 2). If we select this rule in the Rules window, the system will directly apply this rule. Alternatively, we could just press the Apply button. The system will then present all applicable rules.

The essential step of the *Bin* case (see Fig. 9) is to rebracket the $+\!\!+$ operator. This is allowed because we declared $+\!\!+$ as an associative operator.

$$\begin{array}{l}
\Longleftrightarrow \{|\ \ldots \text{ zoom } \textbf{case}\ Bin\ |\} \\
\quad (tips\ l +\!\!+ \underline{tips\ r}) +\!\!+ \underline{y} \\
\Longleftrightarrow \{|\ \text{apply associativity tactic}\ |\} \\
\quad tips\ l +\!\!+ (\underline{tips\ r +\!\!+ y}) \\
\Longleftrightarrow \{|\ \text{fold } tipcat\ |\} \\
\quad \underline{tips\ l +\!\!+} (tipcat\ r\ y) \\
\Longleftrightarrow \{|\ \text{fold } tipcat\ |\} \\
\quad tipcat\ l\ (tipcat\ r\ y)
\end{array}$$

Figure 9: Derivation of *tipcat*: The *Bin* case

Rebracketing is done by marking two terms and pressing the button Assoc/Comm. The associativity property of $+\!\!+$ is used to reorganise the term in such a way that the two selected terms become operands of the same operator. The two following steps replace the calls to *tips* and $+\!\!+$ by calls to *tipcat*. This is done by pressing the Fold button. If the function to fold can not be uniquely determined, the system pops up a dialogue and asks the user to select the right instance.

As a result of the above derivation, we obtain the following rule:

$$\begin{array}{l}
fastTipcat = [] \models \\
\quad \lambda\, x\ y \to tipcat\ x\ y \Longleftrightarrow \\
\qquad \lambda\, x\ y \to\ \ \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad Tip\ v \to v : y \\
\qquad\qquad\qquad Bin\ l\ r \to tipcat\ l\ (tipcat\ r\ y)
\end{array}$$

The derived rule can be transformed into a recursive function (select Rule to Term in the Rules menu). In contrast to the original definition of *tips*, this new function is a linear time program. Rules and functions are stored in the system theory and can be executed in Gofer.

5 DISCUSSION

Program transformation has a long tradition. Nevertheless, searching adequate techniques for automated support still is an important research item. A large number of prototypical systems for specific programming areas exists. One of the most eye-catching results is the KIDS [15] system and its successor Specware [7]. Both systems provide high-level operations for transformational development of programs from specifications. Powerful built-in tactics are used to simplify and optimise programs.

Recent developments that are comparable to *Ultra* are for example the UniForM Workbench [8] and the MAP [14] system. The UniForm Workbench provides a general framework for tool-supported formal program development. Using Isabelle [12] as tool basis, a program transformation system for functional programs has been developed. MAP is a system that is also based on the unfold-fold strategy and supports the interactive derivation of logic programs.

Conceptually, the design of *Ultra* is based on CIP-S [1]. However, a number of design decisions were deliberately modified. For instance, whereas CIP-S is object language independent, *Ultra* uses a fixed language. This enables a much better support for language specific transformations. *Ultra* is completely written in the functional language Gofer. It uses TkGofer [17] for the functional implementation of the graphical user interface. Although *Ultra* offers significantly more automation than CIP-S did, it is written in considerably less lines of code (about 9000 lines without library, which is about 1/3 of the size of CIP-S). One of the main reasons for this increase in functionality and decrease in lines of code is the use of modern concepts of functional languages like constructor classes and monads [5]. Constructor classes allow overloading of operators and functions, and help to reduce and structure your code. Furthermore, they support code reuse in the form of default instance declarations. Monads are a mechanism to handle imperative concepts like I/O and global state in a purely functional way. They are useful to structure larger applications (like *Ultra*). For instance, in *Ultra* monads are used to integrate systematic exception handling and to separate the user interface from the application core [16].

Although rather simple, the derivation of *tips* showed the basic principles of program transformation using *Ultra*. If we use variants of the presented steps (e.g., automatically perform a simplification on the complete term after every derivation step) we can achieve a higher degree of automation. However, experience has shown that in many cases, the user still wants to have the flexibility to perform derivation steps by hand. If the degree of automation is too high, alternative derivation steps are easily eliminated.

So far, we mainly used the system to have tool support in our lectures on formal methods. We replayed many exercises and examples from several textbooks (e.g., [3, 10]), ranging from the obligatory 'sum of squares' to more complicated ones like unification. Students did not encounter any real problems using the system. They favoured the use of *Ultra* compared to 'pencil and paper derivations'. Moreover, the system has been used for the derivation of a complex layout algorithm for block-structured documents [16].

In every example we did, the size of the transformed programs did not exceed 30 lines of code. This has nothing to do with the capacity of *Ultra*, but mainly depends on the special nature of functional programs. Functional programs are known for expressing complex ideas in a very concise way. Furthermore, many transformations were only applied on small fragments of larger programs. Only these fragments had to be displayed in the editor window.

One of *Ultra*'s current limitations is the restriction to first-order pattern matching. The matching of rules often requires a rearrangement of the selected term. This problem is currently handled by tactics for the introduction and elimination of lambda abstractions, rebracketing, and swapping of operands. Higher-order unification (as for example supported in Isabelle [12]) would solve this problem and improve usability.

A recent development is the support of a reduction mode to prove unresolved premises. Closely related is the support for induction. In this context *Ultra* was used for the deductive derivation of hardware algorithms in Haskell [9]. In this case study, *Ultra* showed its value by revealing a number of omissions in hand-written derivations.

Ultra is freely available via `ftp` for any platform that runs TkGofer (e.g., Unix, Linux, Windows). For more information, please contact the authors.

Acknowledgements
We thank Mark Dettinger, Walter Guttmann, Tobias Haeberlein, Thorsten Quell and Joachim Schmid for designing and implementing substantial parts of *Ultra*.

REFERENCES

[1] F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP. Volume II: The Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.

[2] R. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall International, 1997.

[3] R.S. Bird and Ph. Wadler. *Introduction to Functional Programming*. Prentice Hall International, Hemel Hempstead, 1988.

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

[5] J. Jeuring and E. Meijer, editors. *Advanced Functional Programming. First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 1995*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[6] M. Jones. *An introduction to Gofer (draft)*, 1993. Included as part of the standard Gofer distribution.

[7] R. Juellig, Y. Srinivas, and J. Liu. SPECWARE: An advanced environment for the formal development of complex software systems. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 551–555, Munich, 1996. Springer-Verlag.

[8] C. Lüth, E. Karlsen, Kolyang, S. Westmeier, and B. Wolff. Tool integration in the UniForM-Workbench. In *Tool Support for System Specification, Development and Verification*, Advances in Computing Science. Springer-Verlag, 1998.

[9] B. Möller. Deductive hardware design: A functional approach. In *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[10] H.A. Partsch. *Specification and Transformation of Programs – A Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.

[11] L. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[12] L Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[13] P. Pepper. A simple calculus for program transformation (inclusive of induction). *Science of Computer Programming*, 9(3):221–262, 1987.

[14] S. Renault, A. Pettorossi, and M. Prioetti. Design, implementation and use of the Map transformation system. Technical Report 491, Instituto di Analisi dei Sistemi ed Informatica del CNR, Roma, 1998.

[15] D.R. Smith. KIDS – a knowledge-based software development system. In *Proc. Workshop on Automating Software Design*, pages 129–136. Morgan-Kaufmann, 1988.

[16] T. Vullinghs. *Functional Abstractions for Imperative Actions*. PhD thesis, Universität Ulm, 1998.

[17] T. Vullinghs, W. Schulte, and Th. Schwinn. The design of a functional GUI library using constructor classes. In D. Bjorner, M. Broy, and I. Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 398–409, Novosibirsk, 1996. Springer-Verlag.