Journal of Universal Computer Science, vol. 9, no. 2 (2003), 173-188
submitted: 14/10/02, accepted: 14/2/03, appeared: 28/2/03 © *J.UCS*

Tool Support for the Interactive Derivation of Formally Correct Functional Programs

Walter Guttmann
(University of Ulm, Germany
walter@informatik.uni-ulm.de)

Helmuth Partsch
(University of Ulm, Germany
partsch@informatik.uni-ulm.de)

Wolfram Schulte
(Microsoft Research, Redmond, USA
schulte@microsoft.com)

Ton Vullinghs
(University of Ulm, Germany
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. We illustrate its utility by deriving a version of the Heapsort algorithm from a non-deterministic specification.

Ultra supports equational reasoning about functional programs using defining equations, algebraic laws of underlying data structures, and transformation rules. The system does not only support modifying terms, but is also useful for bookkeeping and development-navigating tasks.

The most salient features of *Ultra* are its sound theoretical foundation, its extendability, its flexible and convenient way to express transformation tasks, its comfortable user interface, and its lean and portable implementation. *Ultra* itself is written in the functional language Gofer.

Key Words: Constructive Program Development, Equational Reasoning, Functional Programming, Heapsort, Non-deterministic Specification, Program Transformation, Ultra, Unfold-Fold

Category: D.1.1, D.2.2, F.3.1, I.2.2

1 Introduction

The construction of programs from formal problem specifications is one method to guarantee the correctness of solutions. The technique, transformational programming, is thoroughly described in [Partsch 1990]. This is the basis for the program transformation system *Ultra*.

Writing correct and efficient programs in *Ultra* is divided into 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 more efficient, yet semantically equivalent or refined version.

In particular, *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 a large choice), and *that* (select a uniquely characterized element). Operational programs can be derived from a descriptive specification, for instance, by generalization, by enumeration, or by specialized strategies such as divide and conquer.

As an example for a derivation starting with a non-deterministic specification we develop a variant of the Heapsort algorithm in [Section 7]. Although the formal development of sorting algorithms has a long tradition, not much attention has been devoted to a transformational derivation of Heapsort so far. Indeed, to our knowledge, we describe the first derivation of Heapsort in a functional language that persistently proceeds in a transformational manner, and refer to [Morgan 1994] for the development in an imperative language using the refinement calculus.

To derive new functions from existing ones *Ultra* supports the unfold-fold methodology of [Burstall and Darlington 1977]. This is extended by advanced strategies that are supported by *Ultra* in the form of specialized transformation rules or combinations of transformation rules, called tactics [see Section 4]. As we will demonstrate, tactics themselves can be combined to form yet more powerful strategies capable of performing complex transformation tasks.

2 Theoretical Foundation

The transformation calculus supported by *Ultra* has its roots in the transformation semantics of the CIP system [Bauer et al. 1987]. In this section we informally describe the objects of the calculus. A detailed treatment and a soundness proof is given by [Pepper 1987].

2.1 Object Language

The main purpose of a program transformation system is the (interactive) manipulation of terms. The formulation of target programs in *Ultra* is based on the functional language Haskell. To support the *forall*-, *exists*-, *some*-, and *that*-expressions we have extended the specification language with a non-deterministic choice operator. The current type system of *Ultra*, however, is based on the standard Hindley/Milner system (e.g., *Ultra* does not yet support type or constructor classes). The semantics of the object language is presented in [Schmid 1998] and is strongly based on [Paterson 1992].

For instance, we can define the algebraic data type of binary, node-valued trees accompanied by a few functions:

$$\begin{array}{lcl}
\textit{data Tree a} & = & \textit{Leaf} \\
 & \mid & \textit{Node}\ (\textit{Tree a})\ a\ (\textit{Tree a}) \\
\\
\textit{leroot} & :: & a \rightarrow \textit{Tree a} \rightarrow \textit{Bool} \\
\textit{leroot x Leaf} & = & \textit{True} \\
\textit{leroot x}\ (\textit{Node l e r}) & = & x \leq e \\
\\
\textit{heapinv} & :: & \textit{Tree a} \rightarrow \textit{Bool} \\
\textit{heapinv Leaf} & = & \textit{True} \\
\textit{heapinv}\ (\textit{Node l e r}) & = & \textit{heapinv l} \wedge \textit{heapinv r} \wedge \textit{leroot e l} \wedge \textit{leroot e r}
\end{array}$$

where *heapinv* t asserts that the tree t satisfies the heap property (i.e., the value of each node is not greater than the values of its children). We will use such trees in our Heapsort derivation [see Section 7.1].

Using a concrete functional language as the object for transformations has a number of 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.

2.2 Transformation Rules

The main tools of the calculus are transformation rules. Their semantics is based on a two-level Horn clause logic. The most general form of a transformation rule looks like:

$$[[P_{1,1}, \ldots, P_{1,k_1} \vdash P_{1,0}], \ldots, [P_{n,1}, \ldots, P_{n,k_n} \vdash P_{n,0}]] \quad \models \quad i \Longleftrightarrow o$$

where i and o are program schemes (i.e., terms that may contain free variables) and $P_{i,j}$ are predicates over program schemes. We call $P_{i,j}$ an antecedent if $j \geq 1$ and a consequent if $j = 0$. Together, $[P_{i,1}, \ldots, P_{i,k_i} \vdash P_{i,0}]$ is called a premise and $i \Longleftrightarrow o$ the conclusion of the rule. Finally, i and o are called the input and output schemes, respectively.

The notation $i \Longleftrightarrow o$ of the conclusion is short for $[\,] \vdash i \equiv o$ where $\equiv$ is a special predicate, namely the equivalence of terms. Additionally, *Ultra* also supports the descendance relation where $i \Longrightarrow o$ abbreviates $[\,] \vdash i \sqsupseteq o$. The predicate $\sqsupseteq$ describes the (reversed) ordering in the lower powerdomain construction of [Paterson 1992] used to model non-determinism.

The free variables in a transformation rule are either defined globally (e.g., like *heapinv* in [Section 2.1]) or parameters of the rule (if they occur in the conclusion), or local to the premise they occur in.

A transformation rule is valid if for all ground instances of its parameters, whenever its premises are valid so is its conclusion. Thus, the premises of a transformation rule denote its applicability conditions.

A premise is valid if for all ground instances of its local variables, whenever its antecedents are valid so is its consequent. The validity of the conclusion (that has no antecedents and no local variables) is defined the same way.

Currently, predicates may denote either the equivalence or descendance of two program schemes or user-defined algebraic properties of variables. For ground instances of program schemes (i.e., terms) equivalence and descendance are defined according to the object language semantics and the validity of algebraic properties is derived as described in [Section 2.3].

The transformation calculus ensures that all occurring rules can be formulated using the two-level Horn clause schema just shown. The calculus is designed to provide sufficient expressive power for transformational program development while being as simple as possible. More about this trade-off and completeness considerations can be found in [Pepper 1987].

Ultra has several built-in transformation rules that are derived from the object language semantics, e.g., β-reduction, distribution over *case*-expressions, and rearrangement of associative and commutative operators.

Further transformation rules are given by the user to state additional (not built-in) language properties or to describe domain-specific properties of the underlying data structures. For instance, we can define the rules:

$$\begin{array}{l} \mathit{add\_neutral\_left}\ x_1\ f\ x_2 = \\ \qquad [[\,] \vdash \mathit{lneutral}\ f\ x_1] \ \ \models \ \ x_2 \Longleftrightarrow x_1\ `f`\ x_2 \end{array}$$

$$\begin{array}{l} \mathit{some\_split}\ p_1\ p_2 = \\ \qquad [\,] \ \ \models \ \ \mathit{some}\ (\lambda(x : xs) \rightarrow p_1\ x \wedge p_2\ xs) \Longleftrightarrow \mathit{some}\ p_1 : \mathit{some}\ p_2 \end{array}$$

$$\begin{array}{l} \mathit{tree\_exists}\ b = \\ \qquad [\,] \ \ \models \ \ \mathit{exists}\ (\lambda t \rightarrow \mathit{heapinv}\ t \wedge b\ `\mathit{equals}`\ \mathit{abstraction}\ t) \Longleftrightarrow \mathit{True} \end{array}$$

The rule *add_neutral_left* states that it is safe to replace some expression x_2 by the expression x_1 ‘f‘ x_2 (where the backquotes make f an infix operator), provided x_1 is left neutral for f. The rule *some_split* has no applicability conditions and explains the construction of a non-deterministically characterized list by the construction of its head and tail, respectively. Finally, the domain-specific rule *tree_exists* asserts the existence of a heap t that contains a given collection b of elements [see Section 7.3].

If a rule is applied to a term t, first-order pattern matching between the input scheme and t is used to instantiate the parameters of the rule. If the matching succeeds, the resulting substitution is applied to the output scheme that replaces t. The applicability conditions are resolved, if possible, as shown in [Section 2.3].

2.3 Algebraic Properties

The applicability conditions of a transformation rule may restrict the possible values of its variables by some algebraic properties. The rule *add_neutral_left* of [Section 2.2], for instance, requires that x_1 is a left neutral for f. Algebraic properties are formulated by the user with a Prolog-like syntax stating facts and clauses. Examples of such definitions occur in the small property base:

$$\begin{array}{lll} \textit{associative} & \wedge & \\ \textit{lneutral} & \wedge\ \textit{True} & \\ \textit{rneutral} & \wedge\ \textit{True} & \\ \\ \textit{neutral} & f\ x & :\!-\ \textit{lneutral}\ f\ x, \textit{rneutral}\ f\ x \\ \textit{monoid} & f\ x & :\!-\ \textit{associative}\ f, \textit{neutral}\ f\ x \end{array}$$

The validity of algebraic properties over terms is inductively defined with regard to such a property base. Facts express basic instances and clauses are used to derive additional properties.

If a rule is applied that contains an algebraic property in its premises, *Ultra* tries to infer proper instantiations for the scheme variables by searching its property base. If no, or no unique matching instances are found, the user is asked to enter the values for the scheme variables.

For instance, application of the rule *add_neutral_left* of [Section 2.2] to a term t of boolean type replaces t automatically by $\textit{True} \wedge t$ in the presence of the above property base. The applicability condition of *add_neutral_left* is resolved by this instantiation and does not appear in the resulting rule.

3 Extendability

Data structures, functions, rules, and algebraic properties are logically organized into theories. Physically, they are stored in theory files that can be freely accessed and edited by the user. Beyond that, the program transformation system offers a number of operations to apply and manipulate these theories.

The definitive goal of a transformation session is to derive a new transformation rule. Such a rule reflects the semantic relation between the initial and final term of the derivation process. Definitions of new functions are obtained from expressions that occur during the session. The new rule and the new functions are added to the knowledge base of the transformation system and can be used in a future session.

The procedure for deriving a new transformation rule in *Ultra* is:

1. Enter the start term, which becomes the input scheme of the new rule. Start the derivation.

2. Perform several transformation steps. To this end, interactively apply built-in or user-defined transformation rules, assisted by automatic simplifications and tactics [see Section 4].

3. Stop the derivation. The end term becomes the output scheme of the new rule. Those applicability conditions that could not be resolved are preserved for the new rule.

The new rule is then stored in the knowledge base and is available for further applications. The transformation session can now be continued with the derivation of another transformation rule.

Finally, extending *Ultra* by new tactics [see Section 4], as well as integrating them into the graphical user interface is readily accomplished.

4 Tactics

The user can transform a program by repeatedly searching and applying adequate transformation rules. This labor-intensive process can be partly controlled and automated using *tactics* and *tactic combinators* [Paulson 1983]. A tactic is a function that maps some term to 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. Using these combinators we can write complex tactics that carry out larger transformation tasks. For instance, here is a simplification tactic of *Ultra*:

$$\begin{aligned} \mathit{simplify} \;&=\; \mathit{dfsT}\ (\mathit{repeatT}\ \mathit{simple\_step}) \\ \mathit{where}\ \mathit{simple\_step} &= \mathit{case\_simple}\ `\mathit{orT}` \\ &\ \mathit{arith\_simple}\ `\mathit{orT}` \\ &\ \mathit{const\_simple}\ `\mathit{orT}`\ \ldots \end{aligned}$$

This tactic tries to repeat as many times as possible (combinator *repeatT*) the application of a single simplification step on every subterm of the parameter term in a depth-first order (combinator *dfsT*). Every atomic step may be one of several simplifications (combinator *orT*), e.g., arithmetic evaluation.

Another powerful tactic is the *solve* tactic. This tactic simplifies or even eliminates descriptive constructs in favor 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. Both *simplify* and *solve* are used repeatedly in the derivation of Heapsort [see Section 7.4].

Even elaborate tactics that support transformational methodologies can be composed. For example, *Ultra* has a built-in tactic that can perform certain non-trivial unfold-fold transformations in one step. Its organization can be briefly described as:

$$\begin{aligned} \mathit{unfold\_fold}\ t\ &=\ \mathit{if}\ \ \mathit{is\_function}\ t\ \mathit{then}\ \mathit{rewrite}\ (\mathit{name\_of}\ t)\ t\ \mathit{else}\ t \\ \mathit{where}\ \mathit{rewrite}\ n\ &=\ \mathit{unfold}\ \text{`}\mathit{thenT}\text{`} \\ &\ \ \mathit{unfold\_subterms}\ \text{`}\mathit{thenT}\text{`} \\ &\ \ \mathit{simplify}\ \text{`}\mathit{thenT}\text{`} \\ &\ \ \mathit{fold}\ n \end{aligned}$$

Note that a tactic is a function taking terms as input. This tactic first checks whether the parameter denotes a function. If so, it unfolds the top-level function and, subsequently, function applications in subterms. The resulting term is simplified and, finally, instances of the body of the start term are folded.

The benefits are obvious: instead of tediously repeating the same small steps, the derivation is done in a comfortable and automated way. The transformation knowledge is captured in a single, reusable, and combinable tactic.

5 User Interface

The graphical user interface of *Ultra* [see Fig. 1] is designed to provide clear and easy access to the operations that perform transformation tasks or manipulate theories, and the required data. 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 currently 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. Specialized tactics can be found in the pull-down menu *Tactics*. The other menus mainly hide operations for organizational tasks like loading new theories, starting or stopping derivations, or generating output for the current derivation.

Because a derivation process consists of several transformation steps, the system offers buttons to support 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.

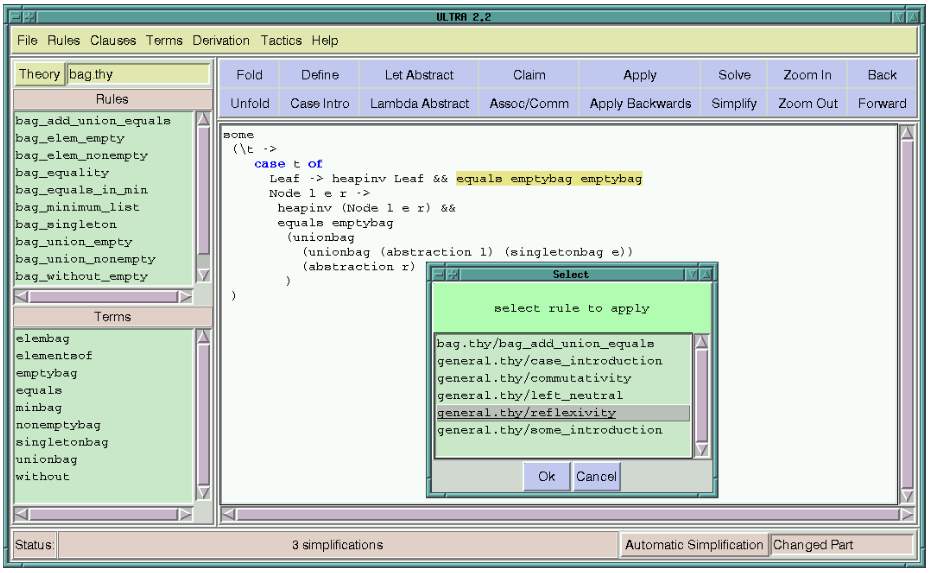

Figure 1: *The main window of the Ultra system.*

– 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.

6 Design and Implementation

Conceptually, the design of *Ultra* is based on CIP-S [Bauer et al. 1987]. Several design decisions, however, were deliberately changed. 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 programming language Gofer. It uses TkGofer [Vullinghs et al. 1996] for the implementation of the graphical user interface. 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., adding new tactics) in *Ultra*.

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 [Jeuring and Meijer 1995].

Constructor classes allow overloading of operators and functions, and help to reduce and structure the code. Furthermore, they support code reuse in the form of default instance declarations. Monads are a mechanism to deal with imperative concepts like I/O and global state in a purely functional way. They are useful to structure larger applications. For instance, in *Ultra* monads are used to integrate systematic exception handling and to separate the user interface from the application core [Vullinghs 1998].

Ultra is freely available for any platform that runs TkGofer (e.g., Linux, Solaris, Windows). It comes equipped with a manual [Guttmann 2000] that describes how to use the system, demonstrates the interaction by performing several derivations, and is a reference to the functionality provided via the GUI.

7 Derivation of Heapsort

We use the program transformation system *Ultra* to derive a version of the Heapsort algorithm from a non-deterministic specification. This section presents an overview of the derivation and focuses on some details. A complete, technical description and evaluation is given by [Guttmann 2002].

7.1 Specification of Sorting

We start with the following (traditionally) correct description of sorting:

$$\begin{array}{lcl}
\mathit{sort} & :: & [a] \to [a] \\
\mathit{sort}\ \mathit{xs} & = & \mathit{some}\ (\lambda \mathit{ys} \to \mathit{issorted}\ \mathit{ys} \wedge \\
 & & \qquad\qquad \mathit{elementsof}\ \mathit{ys}\ \text{`}\mathit{equals}\text{`}\ \mathit{elementsof}\ \mathit{xs}) \\
\mathit{issorted} & :: & [a] \to \mathit{Bool} \\
\mathit{issorted}\ [\,] & = & \mathit{True} \\
\mathit{issorted}\ (x : \mathit{xs}) & = & x == \mathit{minimum}\ (x : \mathit{xs}) \wedge \mathit{issorted}\ \mathit{xs}
\end{array}$$

where the function *minimum* on lists is given by the standard prelude.

The function *elementsof* is due to the requirement that the sorted list is a permutation of the input list. It constructs a bag (or multiset) of the elements of its parameter list. We assume that bags are specified as an abstract data type with conditional equational laws. Such a specification can be translated in a straight-forward manner into an *Ultra* theory file where (primitive) functions and transformation rules play the role of operations and laws, respectively. A short excerpt of that file is:

$$\begin{array}{lcl} \mathit{elementsof} & :: & [a] \rightarrow \mathit{Bag}\ a \\ \mathit{elementsof}\ [\,] & = & \mathit{emptybag} \\ \mathit{elementsof}\ (x : xs) & = & \mathit{nonemptybag}\ x\ (\mathit{elementsof}\ xs) \end{array}$$

$$\begin{array}{l} \mathit{bag\_minimum\_list}\ x\ xs\ = \\ \qquad [\,]\ \models\ \mathit{minbag}\ (\mathit{elementsof}\ (x : xs)) \Longleftrightarrow \mathit{minimum}\ (x : xs) \end{array}$$

$$\begin{array}{ll} \mathit{reflexive} & \mathit{equals} \\ \mathit{commutative} & \mathit{equals} \\ \mathit{transitive} & \mathit{equals} \end{array}$$

where *emptybag* and *nonemptybag* are generators of bags and *minbag* finds the minimum element of a non-empty bag according to the rule *bag_minimum_list*. Moreover, the definitions of the function *sort* (above) and the rule *tree_exists* in [Section 2.2] use the equality (on bags) *equals*. A small property base contains facts that declare *equals* an equivalence relation and are used in [Section 7.4].

The main design decision of Heapsort is to represent bags by binary trees. We already saw parts of the *Ultra* theory file that contains the specification of trees in [Section 2]. Further details are given in the following excerpt:

$$\begin{array}{lcl} \mathit{abstraction} & :: & \mathit{Tree}\ a \rightarrow \mathit{Bag}\ a \\ \mathit{abstraction}\ \mathit{Leaf} & = & \mathit{emptybag} \\ \mathit{abstraction}\ (\mathit{Node}\ l\ e\ r) & = & \\ \multicolumn{3}{l}{\qquad \mathit{abstraction}\ l\ `\mathit{unionbag}`\ \mathit{singletonbag}\ e\ `\mathit{unionbag}`\ \mathit{abstraction}\ r} \end{array}$$

$$\begin{array}{l} \mathit{tree\_minimum}\ l\ e\ r\ f\quad = \\ \qquad [\,]\ \models\ \mathit{heapinv}\ (\mathit{Node}\ l\ e\ r) \wedge f(\mathit{minbag}\ (\mathit{abstraction}\ (\mathit{Node}\ l\ e\ r))) \\ \qquad\quad \Longleftrightarrow\ \mathit{heapinv}\ (\mathit{Node}\ l\ e\ r) \wedge f\ e \end{array}$$

where *singletonbag* and *unionbag* construct a one-element bag and the union of two bags, respectively, and *abstraction* collects all elements of a tree into a bag. The rule *tree_minimum* states that a non-empty heap has a minimum element at its root (which is a consequence of the heap property). Since *Ultra* does not yet support resolving applicability conditions other than algebraic properties, the heap property is encoded as a conjunctive assertion in this rule.

7.2 Main Development

The relevant data structures, operations, and properties are formalized by now. We start the transformation session with *Ultra* by loading the involved theory files that are parsed and type-checked.

The development of Heapsort is centered around the introduction of trees as intermediate data structures and can be separated into two parts. The first part deals with the composition of an intermediate tree from the input sequence. The second part deals with the decomposition of the intermediate tree into the sorted output sequence. Both parts are connected by the main development.

The main development starts with the descriptive specification of *sort* given above. It exhibits several kinds of transformation steps that *Ultra* can perform (where we mark the ►selected subterms◄ that serve as the target for each step):

$$
\begin{array}{ll}
 & \blacktriangleright sort\ xs \blacktriangleleft \\
\equiv & \{\!|\text{unfold}|\!\} \\
 & some\ (\lambda ys \rightarrow issorted\ ys\ \wedge\ \blacktriangleright elementsof\ ys\ `equals`\ elementsof\ xs \blacktriangleleft) \\
\equiv & \{\!|\text{apply } introduce\_composetree\text{, see [Section 7.3]}|\!\} \\
 & some\ (\lambda ys \rightarrow\ \blacktriangleright issorted\ ys\ \wedge \\
 & \qquad\qquad elementsof\ ys\ `equals`\ abstraction\ (composetree\ xs) \blacktriangleleft) \\
\equiv & \{\!|\text{apply } add\_neutral\_left|\!\} \\
 & some\ (\lambda ys \rightarrow\ \blacktriangleright True \blacktriangleleft \wedge issorted\ ys\ \wedge \\
 & \qquad\qquad elementsof\ ys\ `equals`\ abstraction\ (composetree\ xs)) \\
\equiv & \{\!|\text{apply } heap\_composetree \text{ backwards, see [Section 7.3]}|\!\} \\
 & some\ (\lambda ys \rightarrow\ heapinv\ \blacktriangleright (composetree\ xs) \blacktriangleleft \wedge issorted\ ys\ \wedge \\
 & \qquad\qquad elementsof\ ys\ `equals`\ abstraction\ \blacktriangleright (composetree\ xs) \blacktriangleleft) \\
\equiv & \{\!|\lambda\text{-abstraction (with multiple selection)}|\!\} \\
 & \blacktriangleright (\lambda t \rightarrow some\ (\lambda ys \rightarrow heapinv\ t \wedge issorted\ ys\ \wedge \\
 & \quad elementsof\ ys\ `equals`\ abstraction\ t)) \blacktriangleleft (composetree\ xs) \\
\equiv & \{\!|\text{define } decomposetree \text{ (that is automatically folded)}|\!\} \\
 & \blacktriangleright decomposetree\ (composetree\ xs) \blacktriangleleft \\
\equiv & \{\!|\text{define } heapsort \text{ (again, automatically folded)}|\!\} \\
 & heapsort\ xs
\end{array}
$$

As described in [Section 2.3], the system automatically finds the right instance for the application of *add_neutral_left* (the user just has to press the *Apply* button). Not only can multiple occurrences of a term be selected for the λ-abstraction, but also that term relative to which the abstraction is being performed.

The derivation of Heapsort still needs to be completed by deriving recursive, i.e., operational versions for the descriptive specifications of the introduced functions *composetree* and *decomposetree*.

7.3 Introduction of Heaps

Let us first discuss the derivation of the rule *introduce_composetree*, although we will not present the transformation steps in detail. It starts with the scheme

b ‘*equals*‘ *elementsof xs* that matches a subterm of the unfolded definition of *sort*. By virtue of the rule *tree_exists* of [Section 2.2] there exists a heap that represents the bag *elementsof xs*. Thus, we may introduce a function that constructs this heap:

$$\begin{array}{lll} \mathit{composetree} & :: & [a] \rightarrow \mathit{Tree}\ a \\ \mathit{composetree}\ xs & = & \mathit{some}\ (\lambda t \rightarrow \mathit{heapinv}\ t \wedge \\ & & \qquad\qquad \mathit{elementsof}\ xs\ \text{‘}\mathit{equals}\text{‘}\ \mathit{abstraction}\ t) \end{array}$$

Note that this definition of *composetree* is not supplied by the user, but arises during the derivation (just as the definition of *decomposetree* in the main development). To complete the introduction of heaps we exploit the fact that *equals* is transitive. The resulting rule that is used in [Section 7.2] is:

$$\begin{array}{l} \mathit{introduce\_composetree}\ b\ xs = \\ \quad [\,]\ \models\ b\ \text{‘}\mathit{equals}\text{‘}\ \mathit{elementsof}\ xs \\ \quad\ \Longleftrightarrow\ b\ \text{‘}\mathit{equals}\text{‘}\ \mathit{abstraction}\ (\mathit{composetree}\ xs) \end{array}$$

From the definition of *composetree* it follows (since all terms are well-defined) that the resulting tree is indeed a heap. We record this fact in the transformation rule:

$$\begin{array}{l} \mathit{heap\_composetree}\ xs = \\ \quad [\,]\ \models\ \mathit{heapinv}\ (\mathit{composetree}\ xs) \Longleftrightarrow \mathit{True} \end{array}$$

that is used in the main development, too.

7.4 From Descriptive Specifications to Operational Functions

Our next goal is to derive a recursive version of *composetree* that no longer contains descriptive constructs. We proceed according to the unfold-fold methodology. After unfolding *composetree* and the inside occurring call to *elementsof*, we have sufficient detail to distinguish two cases (according to the definition of *elementsof* in [Section 7.1]).

We will deal with both cases in turn. This procedure is supported by *Ultra*'s ability to focus on a subterm. To this end, the user selects that subterm and presses the *Zoom in* button.

In the first case we have to find some heap that represents the empty bag. This derivation illustrates the automation capabilities of *Ultra*:

$$\begin{array}{ll} & \mathit{some}\ (\lambda t \rightarrow \mathit{heapinv}\ t \wedge \mathit{emptybag}\ \text{‘}\mathit{equals}\text{‘}\ \blacktriangleright \mathit{abstraction}\ t \blacktriangleleft) \\ \equiv & \{\!|\text{unfold}|\!\} \\ & \mathit{some}\ (\lambda t \rightarrow \blacktriangleright \mathit{heapinv}\ t \wedge \mathit{emptybag}\ \text{‘}\mathit{equals}\text{‘}\ \mathit{case}\ t\ \mathit{of} \\ & \qquad \mathit{Leaf} \rightarrow \mathit{emptybag} \\ & \qquad \mathit{Node}\ l\ e\ r \rightarrow \mathit{abstraction}\ l\ \text{‘}\mathit{unionbag}\text{‘}\ \mathit{singletonbag}\ e \\ & \qquad\qquad\qquad \text{‘}\mathit{unionbag}\text{‘}\ \mathit{abstraction}\ r \blacktriangleleft) \end{array}$$

$\equiv$ $\{\!|$simplify$|\!\}$
$\quad$ *some* $(\lambda t \to$ *case t of*
$\qquad$ $Leaf \to heapinv\ Leaf \wedge \blacktriangleright emptybag\ `equals`\ emptybag\blacktriangleleft$
$\qquad$ $Node\ l\ e\ r \to heapinv\ (Node\ l\ e\ r) \wedge emptybag\ `equals`$
$\qquad\quad$ $abstraction\ l\ `unionbag`\ singletonbag\ e\ `unionbag`\ abstraction\ r)$
$\equiv$ $\{\!|$apply *reflexivity*$|\!\}$
$\quad$ *some* $(\lambda t \to$ *case t of*
$\qquad$ $Leaf \to heapinv\ Leaf \wedge True$
$\qquad$ $Node\ l\ e\ r \to heapinv\ (Node\ l\ e\ r)\ \wedge \blacktriangleright emptybag\ `equals`$
$\qquad\quad$ $abstraction\ l\ `unionbag`\ singletonbag\ e\ `unionbag`\ abstraction\ r\blacktriangleleft)$
$\equiv$ $\{\!|$apply catalog, viz. all rules derived from the specification of bags$|\!\}$
$\quad$ $\blacktriangleright some\ (\lambda t \to case\ t\ of$
$\qquad$ $Leaf \to heapinv\ Leaf \wedge True$
$\qquad$ $Node\ l\ e\ r \to heapinv\ (Node\ l\ e\ r) \wedge False)\blacktriangleleft$
$\equiv$ $\{\!|$solve$|\!\}$
$\quad$ *Leaf*

In the second case we have to find some heap that represents the non-empty bag that consists of the first element of the sequence xs and the bag containing the remaining elements. For the latter bag there exists a representing heap, and from [Section 7.3] we already know how to construct it, namely with *composetree*. Before *introduce_composetree* can be applied, however, some terms have to be rearranged using the commutativity properties of *equals* (by selecting the arguments and pressing the *Assoc/Comm* button).

It remains to find some heap that represents a bag containing one element and the elements of another heap. This is a descriptive specification of the insertion of an element into a heap which can be used to define a corresponding function *insert*. The derivation of *composetree* is then finished:

$$\begin{array}{lll} composetree\ xs \iff case\ xs\ of & [\,] \to Leaf \\ & y : ys \to insert\ y\ (composetree\ ys) \end{array}$$

The definition of *insert* is still descriptive. The unfold-fold paradigm is employed to derive a recursive version for *insert*, too, and this results in:

$$\begin{array}{ll} insert\ x\ t \implies case\ t\ of & Leaf \to Node\ Leaf\ x\ Leaf \\ & Node\ l\ e\ r \to Node\ (insert\ (x\ `max`\ e)\ r)\ (x\ `min`\ e)\ l \end{array}$$

The descendance relation reflects the fact that there are other possible ways to combine x, l, e, and r to a tree (maybe even involving further information such as the sizes or depths of the subtrees). Our decision leads to the variant just shown that constructs so-called Braun trees. Their structure is very similar to that of classical heaps. The important property is that they are balanced and guarantee access with $O(\log n)$ time complexity.

The derivation of an operational version for the other part of Heapsort, namely *decomposetree*, is also driven by the unfold-fold methodology. It uses the rule *tree_minimum* of [Section 7.1] to isolate the minimum element of a bag. The rest of the sorted list is then constructed recursively. Since we have already demonstrated the major transformation capabilities of *Ultra*, we do not present more details here, but refer to [Guttmann 2002].

8 Experiences and Related Work

So far, we have mainly used the system to have tool support in our lectures on formal methods. We have replayed many exercises and examples from several textbooks (e.g., [Bird and Wadler 1988] and [Partsch 1990]), ranging from simple ones like sum-of-squares [Wadler 1990] to more complicated ones like unification. Students did not encounter any real problems using the system, but favored 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 [Vullinghs 1998].

Ultra is a semi-automatic transformation system. This means that the user is responsible for performing the derivation whereas the system assists the process by, e.g., automatically carrying out simplifications after each derivation step. The degree of automation can be extended by providing further tactics and, to some extent, the user can also adjust it during runtime.

Difficulties arise when the degree of automation becomes too high. On the one hand, the user is less flexible since alternative derivation steps are easily eliminated. On the other hand, when several transformation rules are automatically applied, their order generally influences the outcome. This is especially significant within the scope of optimizing compilers, where fully automated rewrite is required. There, the use of transformation rules is emerging for functional languages [Peyton Jones et al. 2001] as well as for imperative ones [Dettinger 2000]. Subtle interactions between the application of user-defined rules and other optimizations pose a problem.

For semi-automatic systems, a further issue is how to capture the user's intention at controlling transformational developments. For example, in the High Assurance Transformation System [Winter 1999] the transformations are not performed manually, but by supplying *transformation programs*. They control the rewrite engine and can be seen as “metaprograms” whose use has been advocated earlier [Feather 1982].

Different variants of program transformations, e.g., translation, refactoring, optimization, and normalization, require varying traversal and rewriting strategies. Unified support for the specification of the necessary, fully automatic transformations is provided by the language Stratego [Visser 2001]. Finally, the experimental MAP system [Renault et al. 1998] applies the unfold-fold paradigm enriched with strategies to interactively transform (constraint) logic programs.

Compared with other program transformation systems, the characteristics of *Ultra* are:

- Support for non-deterministic, descriptive operations through the constructs *forall*, *exists*, *some*, and *that*.
- Based on a well-explored transformation calculus that is expressive enough while remaining computationally simple.
- Capability of semi-automatically performing complex transformations with specifications of the size of several modules.
- Concise, portable, and easily extendable implementation using modern functional concepts.

9 Future Work

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 λ-abstractions, rebracketing, and swapping of operands. Higher-order unification as, e.g., supported in Isabelle, would solve this problem and improve usability.

An ongoing development is the implementation of a reduction mode to prove unresolved premises. Closely related is the support for proofs by induction. In this context *Ultra* was used for the deductive derivation of hardware algorithms in Haskell [Möller 1998]. In this case study, *Ultra* showed its value by revealing a number of omissions in hand-written derivations.

It has been suggested [de Moor and Sittampalam 1999] to achieve induction by using catamorphisms and the related fusion rules, together with an extended matching algorithm. A prototype, the fully automatic transformation system MAG, has been used to perform optimizations through the application of accumulation and tupling techniques. We have in mind to evaluate the benefits of this approach with respect to a possible integration into *Ultra*.

References

[Bauer et al. 1987] F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, P. Pepper: "The Munich project CIP, volume II: The program transformation system CIP-S"; Lecture Notes in Computer Science, 292, Springer-Verlag, Berlin (1987).

[Bird and Wadler 1988] R. Bird, Ph. Wadler: "Introduction to functional programming"; Prentice Hall International, Hemel Hempstead (1988).

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

[de Moor and Sittampalam 1999] O. de Moor, G. Sittampalam: "Generic program transformation"; in S.D. Swierstra, P.R. Henriques, J.N. Oliveira (eds.): "Advanced Functional Programming"; Lecture Notes in Computer Science, 1608, Springer-Verlag, Berlin (1999), 116–149.

[Dettinger 2000] M. Dettinger: "Erweiterte Compilierung von C/C++"; PhD thesis, Universität Ulm (2000).

[Feather 1982] M.S. Feather: "A system for assisting program transformation"; ACM Transactions on Programming Languages and Systems, 4, 1 (Jan 1982), 1–20.

[Guttmann 2000] W.N. Guttmann: "An introduction to Ultra"; Universität Ulm (Dec 2000); http://www.informatik.uni-ulm.de/pm/ultra/.

[Guttmann 2002] W.N. Guttmann: "Deriving an applicative Heapsort algorithm"; technical report UIB-2002-02, Universität Ulm (Dec 2002).

[Jeuring and Meijer 1995] J. Jeuring, E. Meijer (eds.): "Advanced Functional Programming"; Lecture Notes in Computer Science, 925, Springer-Verlag, Berlin (1995).

[Möller 1998] B. Möller: "Deductive hardware design: A functional approach"; in B. Möller, J.V. Tucker (eds.): "Prospects for hardware foundations"; Lecture Notes in Computer Science, 1546, Springer-Verlag, Berlin (1998), 421–468.

[Morgan 1994] C. Morgan: "Programming from specifications"; second edition, Prentice Hall International, Hemel Hempstead (1994).

[Partsch 1990] H.A. Partsch: "Specification and transformation of programs: A formal approach to software development"; Springer-Verlag, Berlin (1990).

[Paterson 1992] R. Paterson: "A tiny functional language with logical features"; in J. Darlington, R. Dietrich (eds.): "Declarative programming"; Springer-Verlag, New York (1992), 66–79.

[Paulson 1983] L. Paulson: "A higher-order implementation of rewriting"; Science of Computer Programming, 3, 2 (Aug 1983), 119–149.

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

[Peyton Jones et al. 2001] S. Peyton Jones, A. Tolmach, T. Hoare: "Playing by the rules: Rewriting as a practical optimisation technique in GHC"; in R. Hinze (ed.): "Preliminary proceedings of the 2001 ACM SIGPLAN Haskell workshop"; Universiteit Utrecht (2001), 203–233.

[Renault et al. 1998] S. Renault, A. Pettorossi, M. Proietti: "Design, implementation, and use of the MAP transformation system"; technical report 491, Instituto di Analisi dei Sistemi ed Informatica del CNR (Dec 1998).

[Schmid 1998] J. Schmid: "Nichtdeterminismus und Programmtransformation"; Master's thesis, Universität Ulm (1998).

[Visser 2001] E. Visser: "Stratego: A language for program transformation based on rewriting strategies"; in A. Middeldorp (ed.): "Rewriting techniques and applications"; Lecture Notes in Computer Science, 2051, Springer-Verlag, Berlin (2001), 357–361; http://www.stratego-language.org/.

[Vullinghs 1998] T. Vullinghs: "Functional abstractions for imperative actions"; PhD thesis, Universität Ulm (1998).

[Vullinghs et al. 1996] T. Vullinghs, W. Schulte, Th. Schwinn: "The design of a functional GUI library using constructor classes"; in D. Bjørner, M. Broy, I.V. Pottosin (eds.): "Perspectives of System Informatics"; Lecture Notes in Computer Science, 1181, Springer-Verlag, Berlin (1996), 398–409.

[Wadler 1990] Ph. Wadler: "Deforestation: Transforming programs to eliminate trees"; Theoretical Computer Science, 73, 2 (1990), 231–248.

[Winter 1999] V.L. Winter: "An overview of HATS: A language independent high assurance transformation system"; Proceedings of the IEEE symposium on application-specific systems and software engineering & technology (1999), 222–229.