Tool Support for the Interactive Derivation of Formally Correct Functional Programs

Walter Guttmann, Helmuth Partsch,

Wolfram Schulte[*] and Ton Vullinghs[†]

Universität Ulm, Fakultät für Informatik, 89069 Ulm, Germany
{walter,partsch,wolfram,ton}@informatik.uni-ulm.de

(Extended 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. *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.

Program Transformation

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.

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 many alternatives) 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.

[*]Currently affiliated with Microsoft Research, Redmond, USA
[†]Currently affiliated with DaimlerChrysler Research, Ulm, Germany

To derive new functions from existing ones *Ultra* supports the unfold-fold strategy [3]. This is extended by advanced strategies that are supported in *Ultra* in the form of specialized transformation rules or combinations of transformation rules (called tactics, see below). As we will demonstrate, tactics themselves can be combined to form yet more powerful strategies capable of performing complex transformation tasks.

Theoretical Foundation

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* is however based on the standard Hindley/Milner system (e.g., *Ultra* does not yet support type or constructor classes).

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.

The transformation calculus supported by *Ultra* has its roots in the transformation semantics of the CIP system [1] and is based on a two-level Horn clause logic. A transformation rule is a special kind of logical inference where the conclusion denotes an equivalence relation between two program schemes (terms that may contain free variables). The premises of the inference denote the applicability conditions for the rule. We use the following notation for transformation rules:

$$cs \models i \Longleftrightarrow o$$

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, frequently an algebraic property.

Extendability

Algebraic properties are formulated with a Prolog-like syntax where 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 the *fact-base*. If no, or no unique matching instances are found, the user is asked to enter the values for the scheme variables.

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. The rule is 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 Haskell functions.

Extending *Ultra* by new tactics (see next section), as well as integrating them into the graphical user interface is easily accomplished.

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* [10]. 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, afterwards 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,
- t_1 '*thenT*' t_2: try to apply tactic t_1, if 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 present term in a depth-first order.

Using these combinators we can write complex tactics that carry out larger transformation tasks. The following example shows a simplification tactic:

```
simplify = dfsT (repeatT simple_step)
    where simple_step = case_simple `orT`
                        distr_simple `orT`
                        const_simple `orT` ...
```

This tactic tries to repeat the application of a single simplification step (e.g., a *case* simplification) on every subterm of the parameter term. 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.

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. 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.

User Interface

Programs, rules, and algebraic properties are logically organized into theories. The program transformation system offers a number of operations to apply and manipulate these theories. The graphical user interface of *Ultra* is designed to provide clear and easy access to these operations and the required data.

The editor supports context sensitive editing of terms, i.e., the user can select parts of the displayed term according to its syntactical structure, thereby indicating the input for the next transformation step. The user interface defines command buttons to support the basic tasks of the unfold-fold paradigm, general tactics like the simplifier, and navigation tasks like *undo* and *zoom* that ease the accomplishment of complex derivations. Furthermore, *Ultra* provides facilities to access the information in the knowledge base (e.g., terms and rules).

Design and Implementation

Conceptually, the design of *Ultra* is based on CIP-S [1]. A number of design decisions were deliberately modified, however. 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 [13] for the functional 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 [7].

Ultra is freely available for any platform that runs TkGofer (e.g., Unix, Linux, Windows). It comes equipped with a manual [6] that describes how to use the system, demonstrates the interaction by performing several derivations, and is a reference to the functionality provided through the GUI.

Experiences

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., [2, 9]), ranging from the simple ones like *sumsquares* 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 [12].

Ultra is a semi-automatic transformation system, which 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 [11] as well as for imperative ones [4]. 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 [14], transformations are not performed manually but by providing *transformation programs.* They control the rewrite engine and can be seen as "metaprograms", whose use has been advocated earlier [5].

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 lambda abstractions, rebracketing, and swapping of operands. Higher-order unification as, for example, 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 [8]. In this case study, *Ultra* showed its value by revealing a number of omissions in hand-written derivations.

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 Ph. Wadler. *Introduction to Functional Programming.* Prentice Hall International, Hemel Hempstead, 1988.

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

[4] M. Dettinger. *Erweiterte Compilierung von C/C++.* PhD thesis, Universität Ulm, 2000.

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

[6] W. Guttmann, H. Partsch, and T. Vullinghs. *An Introduction to Ultra.* Universität Ulm, 2000. http://www.informatik.uni-ulm.de/pm/ultra/.

[7] 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.

[8] B. Möller. Deductive hardware design: A functional approach. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 421–468. Springer-Verlag, 1998.

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

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

[11] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In R. Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 203–233. Universiteit Utrecht, 2001.

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

[13] T. Vullinghs, W. Schulte, and Th. Schwinn. The design of a functional GUI library using constructor classes. In D. Bjørner, 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.

[14] V.L. Winter. An overview of HATS: A language independent high assurance transformation system. In *Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, 1999.