Geniaal programmeren
– Generic programming at Utrecht –

Johan Jeuring

Lambert Meertens

Technical Report UU-CS-2009-001
January 2009

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Geniaal programmeren[*]
– Generic programming at Utrecht –

Johan Jeuring and Lambert Meertens

Generic programs are programs that abstract over types. Generic programs can be used to address several common programming patterns. This paper discusses why generic programming is useful, gives applications of generic programming, and discusses the history of generic programming and current generic programming activities at Utrecht University.

Constructing software.
One of the central issues in software technology, studied at the computer science programme of Utrecht University and many other universities, is the efficient construction of correct software. How do you construct software that does what it should do? There are many aspects to software construction. First you have to know *what* to construct. What are the user requirements of the software? Such requirements are functional requirements, such as that the software sorts a list of book titles in dictionary order, and non-functional requirements, such as that it must return the sorted result within a reasonable amount of time. Once the requirements are known, the actual construction can start — assuming that the software is not available somewhere, or is too expensive to buy. *How* do you construct software that satisfies the given requirements?

Generic programming.
In the development of software, *structuring data* plays an important role. Many programming methods and software development tools center around creating a datatype (or XML schema, UML model, class, grammar, etc.). Once the structure of the data has been designed, a software developer adds *functionality* to the datatypes. There is always some functionality that is specific for a datatype, and part of the reason why the datatype has been designed in the

[*]This title was shamelessly stolen from Arthur Baars, Martin Broere, Bastiaan Heeren, and Erwin Jansen, who used it as a title for their project in the 1999 version of the course on Generic Programming at Utrecht Unversity.

first place. Other functionality is similar or even the same on many datatypes, following common programming patterns. Examples of such patterns are:

- in a possibly large value of a complicated datatype (for example for representing the structure of a company), applying a given action at all occurrences of a particular constructor (e.g., adding or updating zip codes at all occurrences of street addresses) while leaving the rest of the value unchanged;
- serializing a value of a datatype, or comparing two values of a datatype for equality, functionality that depends only on the *structure* of the datatype;
- adapting data access functions after a datatype has changed, something that often involves modifying large amounts of existing code.

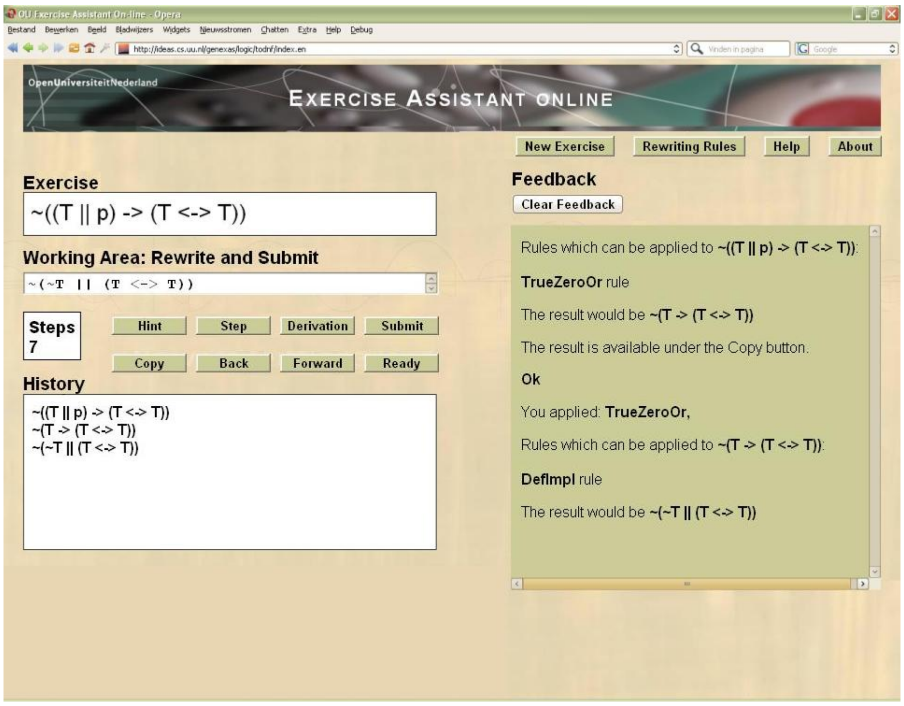
Generic programming addresses these high-level programming patterns.[1] Using generic programming, we can easily implement traversals that apply an action only to relevant parts of a possibly large value, functions thar are naturally defined by induction on the structure of datatypes, and functions that automatically adapt to a changing datatype.

Of course, other programming paradigms aim to address similar programming patterns. In the object-oriented programming world, classes have been introduced to promote reuse and data abstraction, and design patterns to describe often occurring programming patterns. Generic programming addresses most of these patterns, but sometimes in an entirely different way [3].

Applications of generic programming.
As a programmer, you don't want to write the same code twice. Ideally, you describe the essential aspects of the problem you want to solve in your code, and all other "boilerplate" code is derived automatically for you. Generic programming supports focussing on the essential aspects in your code, and in that sense, applications of generic programming are everywhere. The remainder of this section describes a concrete example of such an application. In the next section we work out another example.

At the Open University NL (OUNL) and Utrecht University, we are developing several so-called exercise assistants: programs that offer assistance to a student doing an exercise, typically in some mathematically formalized domain, requiring problem solving using precisely defined methods. We have developed an exercise assistant that supports interactively solving a system of linear equations [15], and another assistant that supports calculating the disjunctive normal form (DNF) of a logical expression [10]. Extensions of the former tool are used in linear algebra courses at the technical universities of Delft and Eindhoven, while the second tool is used in the discrete mathematics course in the Computer Science curriculum of the OUNL. A screenshot of

[1]We also use the term *datatype-generic programming* to distinguish the field from Java generics, Ada generic packages, generic programming in C++ STL, etc. Another term used for the same is *polytypic programming*.


OU Exercise Assistant On-line - Opera
Bestand Bewerken Beeld Bladwijzers Widgets Nieuwsstromen Chatten Extra Help Debug
http://ideas.cs.uu.nl/genexas/logic/todnf/index.en
OpenUniversiteitNederland
EXERCISE ASSISTANT ONLINE
New Exercise
Rewriting Rules
Help
About
Hint
Step
Derivation
Submit
Copy
Back
Forward
Ready
Clear Feedback

Exercise

```
~((T || p) -> (T <-> T))
```

Working Area: Rewrite and Submit

```
~(~T || (T <-> T))
```

Steps
7

History

```
~((T || p) -> (T <-> T))
~(T -> (T <-> T))
~(~T || (T <-> T))
```

Feedback

Rules which can be applied to **~((T || p) -> (T <-> T))**:

TrueZeroOr rule

The result would be **~(T -> (T <-> T))**

The result is available under the Copy button.

Ok

You applied: **TrueZeroOr,**

Rules which can be applied to **~(T -> (T <-> T))**:

DefImpl rule

The result would be **~(~T || (T <-> T))**

Figure 1: The Online Exercise Assistant

the assistant for calculating DNFs of logical expressions is shown in Figure 1. Besides these exercise assistants we have built assistants for several other domains. A distinguishing feature of our assistants is that they give very good feedback. Although giving good feedback is generally acknowledged to be vital for learning [14], current e-learning systems that support incrementally solving exercises typically lack sophisticated techniques for giving feedback. We hope to improve upon this situation.

To develop an exercise assistant, we need three domain-specific main components: a domain description (for example: systems of linear equations over the field of real numbers, or first-order logical expressions), together with a concrete representation of the domain (how are the expressions presented to the student); rules for reasoning about the domain (for example: multiplication distributes over addition, de Morgan for logical expressions); and one or more strategies for solving exercises in the domain (for example: first move occurrences of $\neg$ inwards until they are in front of a logical variable, and then distribute $\wedge$ over $\vee$ to obtain an expression in DNF). We do not want to implement a separate exercise assistant for each domain, of which there are many: in mathematics, logic, computer science, physics, biology, statistics, etc. Furthermore, the implementation of the different exercise assistants is very similar: the user interface, the web application, the communication between the different components, etc., are all exactly the same. The only difference between the assistants is in the domain-dependent parts. We want to *generate* an exercise assistant given the three domain-specific main components, in particular the domain description. The components that differ per exercise assistant are components that deal with:

- generating arbitrary expressions,
- rewriting expressions,
- determining the distance between two expressions,
- traversing expressions,
- selecting within expressions.

We have written generic programs for each of these components.

Example: generic data compression.
Intelligent Compression Technologies Inc. (ICT), a subsidiary of ViaSat, is a company that develops compressors that exploit advance knowledge of standard file formats to decompose and analyze files on-the-fly, determining and applying the best algorithm, or sequence of algorithms, for optimal compression. The result is the highest compression rates possible on first ("cold cache") download. One of the most important idea behind these compressors is to take the format (or the type) of the data into account when compressing data.

The main idea behind ICTs compressors is as follows. As markup is added to the content, XML documents may become (very) large. Fortunately, due to the repetitive structure of many XML documents, these documents can be compressed by quite a large factor. This can be achieved if we use information from the DTD (or Schema) of the input document in the XML compressor. For example, consider the following small XML file (we consider only XML files which are valid with respect to a DTD):

```
<book lang="English">
<title>   Dead famous  </title>
<author>  Ben Elton    </author>
<date>    2001         </date>
<chapter> Nomination   </chapter>
<chapter> Eviction     </chapter>
<chapter> One Winner   </chapter>
</book>
```

In this file, 130 bytes are used for markup (not counting line breaks), and only 90 bytes for content (or 52 when not counting spaces purely serving a layout purpose). This file may be compressed by separating the structure (markup) from the contents, and compressing the two parts separately. For compressing the structure we can make good use of the DTD. If we know how many different elements and attributes, say n, appear in the DTD, we can replace each occurrence of the markup of an element in a valid XML file by $\lceil \log_2 n \rceil$

bits.[2] The DTD for the above document contains 6 elements and attributes, so we need 3 bits per element or attribute. Since there are eight occurrences of elements and attributes in the above document, 24 bits (3 bytes) are needed for the markup, compared to 130 bytes in the original file. The (small) price that has to be paid is that the strings that appear in the data have to be separated by a special separator symbol. Even further reduction is possible. Markup only needs to be recorded if a choice between different tags can be made. In the above document, we have choices for the language of the book and the number of its chapters. All other elements are compulsory and can be inferred from the DTD, so they do not need to be encoded. Using this idea, we need only 5 bits (1 byte) to represent the markup in the above document.

We now develop an encoding function implementing this idea in Haskell. A first step is to use an XML data binding to map an XML document onto a value of a Haskell datatype that is used to represent values of a particular schema. For example, for a DTD or an XML schema describing the structure of books, we might get the following datatypes:

```
data Book          = Book BookAttrs Title Author Date [Chapter]
data BookAttrs     = BookAttrs{bookLang :: Lang}
data Lang          = English | Dutch
newtype Title      = Title   String
newtype Author     = Author  String
newtype Date       = Date    String
newtype Chapter    = Chapter String
```

An encoding function maps a value of the datatype Book to a pair consisting of the list of content strings of the document, and a value denoting the structure of the XML document. The structure is represented by a list of choices: each time there is a choice in the structure, for example between the *English* or *Dutch* language, a value of type Choices tells which choice has been made.

```
data Choice   = L | R
type Choices  = [Choice]

encodeBook :: Book → ([String],     [Choices])
encodeBook    book  = (contents book, structure book)
```

The *contents* of a Book is simply obtained by collecting all strings that appear in a value of type Book.

```
contents :: Book → [String]
contents   (Book bookAttrs (Title title) (Author author) (Date date) chapters) =
  [title, author, date] ++ map (λ(Chapter chapter) → chapter) chapters
```

The resulting list of strings can be post-processed by some standard string compression algorithm, such as LZW. For a single book, this will not pay off, but the gain for a whole library catalogue may be considerable.

[2]Using a variable length, as for example in Huffman coding, sometimes a lower average can be reached.

The *structure* of a BOOK is obtained by registering the choices that have been made in constructing a value. For a BOOK the only choices that have to be registered are the choice between the two languages, and the choice between a non-empty and an empty list.

```
structure :: BOOK → [CHOICES]
structure  (Book (BookAttrs lang) title author date chapters) =
  choicesLang lang : choicesChapters chapters
choicesLang :: LANG   → CHOICES
choicesLang  English = [L]
choicesLang  Dutch   = [R]
choicesChapters :: [CHAPTER] → [CHOICES]
choicesChapters  []          = [[R]]
choicesChapters  (c : cs)    = [L] : choicesChapters cs
```

The resulting list of choices can be efficiently represented by viewing the constructors L and R as bits, and storing the bits in bytes.

A generic program for encoding a value of an arbitrary datatype can be defined in Haskell or in Generic Haskell [5]. Several students taking the Generic Programming course in the Software technology master have implemented variants of encoders, for example using arithmetic coding or Huffman coding, for their practical assignments.

How does it work?

How can we define a generic function: a function that works on arbitrary datatypes? It would lead too far to explain the details of how we define generic functions, but we will explain the main idea behind generic functions in this section.

Datatypes possess structure. Complex datatypes are defined by constructing them from basic types such as integers, characters, etc., by operations such as *sum* (for a choice between different component types) and *product* (for combining several component types). Other (static) type-building operations are type abstraction and type application. For example, a PUBLICATION datatype could be defined as being a BOOK, or a JOURNALARTICLE, or a REPORT (using the sum operation), where a BOOK may be defined as comprising BOOKATTRS, a TITLE, an AUTHOR, a DATE, and a list of CHAPTERS (using the product operation). To define a generic function, it suffices to specify how to handle sum and product, and how to deal with the basic types. This provides sufficient information to generate a specialization of the generic function for any given datatype defined as described above.

We have developed several languages and libraries in which generic functions can be defined. The tools for these languages and libraries take care of automatically transforming values of datatypes to some internal representation which can be used by generic functions, and back again.

More information about how to define generic functions can be found in several tutorials [8, 1, 6, 9].

History of generic programming.
Traditionally, the Netherlands has had a strong position in program construction methods and programming languages. Dijkstra developed his calculus for program construction in the sixties, and some of the important people behind Algol 68, notably van Wijngaarden, were Dutch.

Meertens also contributed to Algol 68, and since then started thinking about how to further improve and simplify program construction. He started with designing more abstract programming languages, but slowly moved to program transformation methods. In the beginning of the eighties he joined forces with Richard Bird of Oxford University to develop what later became known as the "Bird-Meertens formalism". The main idea of this formalism is to specify a programming problem as an obviously correct, but possibly very inefficient program, and then to transform this program into an efficient program by means of program transformation laws.

For example, sorting a list can be specified as: calculate all permutations of the input list, and return the first of those that is sorted. This is a runnable specification, but it is terribly inefficient. We can transform this program into a more efficient program by applying programming laws on the datatype of lists. One of the simplest such transformations results in the insertion-sort algorithm.

The development of programs from specifications and the development of program transformation laws was a very active research area in the eighties and early nineties. The STOP (Specification and Transformation Of Programs) project in the Netherlands was the centre of this research from 1988 to 1992. Many PhD students were funded by the STOP project, and several groups joined the meetings of the project to also work on these topics. The computer science department of Utrecht University, in particular Doaitse Swierstra and Lambert Meertens, played a central role in the STOP project. Many people received part of their PhD education within the STOP project at Utrecht University, such as Erik Meijer (now at MicroSoft), Johan Jeuring (who returned to Utrecht after a period at Chalmers in Sweden), and Oege de Moor (now at Oxford University).

Another community in which program transformation played an important role was the IFIP Working Group 2.1. This working group originally developed Algol 68, but since then moved on to program calculation. Meertens, Bird, and many other researchers are members of this group.

Together, Richard Bird and Lambert Meertens developed a theory of lists, including program transformation laws for programs on lists [2, 13]. Examples of such laws are that the composition of two maps is a map again, and that the composition of a function with a fold is a fold again, provided the function satisfies some distributivity conditions — in algebraic terms, it has to be a homomorphism. Using concepts from category theory, in particular functors and functor-algebras, this work was generalized by Grant Malcolm in Groningen [12] and others to programming laws on arbitrary datatypes. These generic laws are about functions that work on arbitrary datatypes, such as (generic) folds and maps.

The program calculus thus developed could be used to calculate solutions to many software problems. As a spin-off, the theory described programs that could be implemented in a standard, but different, way on datatypes that can be described as initial functor-algebras. No general-purpose programming language supported such typed, generic functions, so these functions had to be implemented over and again for different datatypes.

Using the structure of functors, Jansson and Jeuring at Chalmers, Gothenburg, designed PolyP [7], an extension of the lazy, higher-order functional programming language Haskell [16]. A generic function is defined by means of induction on the structure of functors. Using this programming language, they not only defined the recursive combinators from the program calculus, such as folds and unfolds, but also generic programs for unification, term rewriting, pattern matching, etc.

PolyP supported the definition of generic functions on datatypes that can be described as initial functor-algebras but do not involve mutual recursion. While sufficient for proof-of-concept demonstration purposes, this last restriction was a severe limitation on the practical applicability of PolyP. Generic programming is particularly attractive in situations with large datatypes, such as the abstract syntax of programming languages, and such datatypes are usually mutually recursive. To support generic functions on sets of mutually recursive datatypes, Hinze, Jeuring, and Löh developed Generic Haskell from 2000 onwards at the computer science department of Utrecht University [4, 11]. Generic functions defined in Generic Haskell can be applied to values of almost any datatype definable in Haskell. The fixed-point structure of datatypes is lost however, and thereby the capability of defining the generic fold function.

Meanwhile, Haskell, or, more precisely, compilers supporting various Haskell extensions, evolved considerably since PolyP was developed. With respect to types, GHC, the Glasgow Haskell Compiler, now supports multiple-parameter type classes, generalized algebraic datatypes (GADTs), type families, etc. Using these extensions, it is now possible to define generic functions in Haskell itself, using a library for generic programming. Since 2000, around 10 of such libraries have been developed world-wide [18]. Three of these libraries are maintained by the Software Technology group in Utrecht. Since — from a generic programming perspective — the expressiveness of these libraries is almost the same as the special purpose language-extensions, and since such libraries are much easier to develop, maintain, and ship, these libraries may make generic programming available to a wider audience.

Using type families and GADTs, it is possible to solve another open generic programming problem. Alexey Rodriguez and others [17] showed how to obtain a fixed-point representation of possibly mutually recursive datatypes, bringing the generic fold function back into the fold. Thus we can define the fold function for the abstract syntax of a programming language, bringing generic programming in reach for compiler writers.

Challenges.
New ideas in program development take time to find their place in mainstream program development methods. For example, the concept of parametric polymorphism existed more than 25 years before it was added to a mainstream programming language like Java, in the form of Java Generics. We expect that the appearance of generic programming techniques in mainstream programming language will likewise take a while. We see the following as the main challenges for generic programming:

- To show how generic programming can be applied in many situations, and how using generic programming in software development improves productivity, maintainability, and safety.
- To show that generic programming is not just tied to Haskell, but is a general concept that can be applied in any high-level programming language with a developed type concept.
- To further develop the theory of generic programming.

Generic programming captures a number of important programming patterns that appear in many programs. It makes it easier to construct, maintain, and adapt software. It will probably find its way to Java in about 15 years from now.

Generic programming at Utrecht University.
Within the Software Technology group at Utrecht University a very active group is working on generic programming, currently consisting of Johan Jeuring, Andres Löh, and Lambert Meertens, four PhD students (Sean Leather, José Pedro Magalhães, Alexey Rodriguez Yakushev, and Americo Vargas), and several more MSc students. Andres Löh completed his PhD thesis within the group in 2004, and Alexey Rodriguez will finish in 2009. We are involved in several research projects related to generic programming, and are currently working on libraries for generic programming, generic programming using dependently typed programming languages, and applications of generic programming. Each year we teach a course on generic programming within the Software Technology master programme.

References

[1] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *Advanced Functional Programming, AFP'98*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.

[2] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer-Verlag, 1987.

[3] J. Gibbons. Datatype-generic programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[4] R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.

[5] R. Hinze and J. Jeuring. Generic Haskell: applications. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 57–97. Springer-Verlag, 2003.

[6] R. Hinze and J. Jeuring. Generic Haskell: practice and theory. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlag, 2003.

[7] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[8] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming, AFP'96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.

[9] J. Jeuring, S. Leather, J. P. Magalhães, and A. Rodriguez Yakushev. Libraries for generic programming in Haskell. Technical Report UU-CS-2008-025, Department of Information and Computing Sciences, Utrecht University, 2008.

[10] J. Lodder, J. Jeuring, and H. Passier. An interactive tool for manipulating logical formulae. In M. Manzano, B. Pérez Lancho, and A. Gil, editors, *Proceedings of the Second International Congress on Tools for Teaching Logic*, 2006.

[11] A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In O. Shivers, editor, *Proceedings of the International Conference on Functional Programming, ICFP'03*, pages 141–152. ACM Press, August 2003.

[12] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[13] L. Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer*

Science, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[14] E. Mory. Feedback research revisited. In D. Jonassen, editor, *Handbook of research for educational communications and technology*, 2003.

[15] H. Passier and J. Jeuring. Feedback in an interactive equation solver. In M. Seppälä, S. Xambo, and O. Caprotti, editors, *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 53–68. Oy WebALT Inc., 2006.

[16] S. Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. A special issue of the Journal of Functional Programming.

[17] A. Rodriguez Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. Technical Report UU-CS-2008-019, Department of Information and Computing Sciences, Utrecht University, 2008.

[18] A. Rodriguez Yakushev, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM.