Generic Programming for Software Evolution

Johan Jeuring
Department of Information and Computing Sciences
Utrecht University
The Netherlands

Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University Nijmegen
The Netherlands

1 Introduction

Change is endemic to any large software system. Business, technology, and organization usually frequently change in the life cycle of a software system. However, changing a large software system is difficult: localizing the code that is responsible for a particular part of the functionality of a system, changing it, and ensuring that the change does not lead to inconsistencies or other problems in other parts of the system or in the architecture or documentation is usually a challenging task. Software evolution is a fact of life in the software development industry, and leads to interesting research questions [21, 22, 31].

Current approaches to software evolution focus on the software development process, and on analyzing, visualizing, and refactoring existing software. These approaches support changing software by, for example, recognizing structure in code, and by making this structure explicitly visible, by means of refactoring or renovating the code. By making structure explicit, it becomes easier to adapt the code.

A *generic program* is a program that works for values of any type for a large class of data types (or DTDs, schemas, class hierarchies). If data types change, or new data types are added to a piece of software, a generic program automatically adapts to the changed or new data types. Take as an example a generic program for calculating the total amount of salaries paid by an organization. If the structure of the organization changes, for example by removing or adding an organizational layer, the generic program still calculates the total amount of salaries paid. Generic programming is a new research field [14, 16, 9, 11, 6, 3], and its implications for software development and software evolution have hardly been investigated.

Since a generic program automatically adapts to changed data types, generic programming is a promising approach to the software evolution problem, in particular for software where type formalisms and types play an important role, such as data-centric software. Assume structure has been recognized in software, for example by means of refactoring or renovating the code. Then a generic program makes those parts of the code which depend on the structure of data *independent* of that structure.

This position paper

- explains why we think generic programming is useful for software evolution,
- describes the kind of software evolution problems for which generic programming is useful,
- and discusses the research challenges for using generic programming for software evolution.

This paper is organized as follows. Section 2 briefly introduces generic programming. Section 3 discusses some scenarios in which generic programming is useful for software evolution, and Section 4 discusses the research problems we want to solve in order to make generic programming a viable approach to software evolution. Section 5 concludes.

2 Generic programming

Software development often consists of designing a data type[1], to which functionality is added. Some functionality is data type specific, other functionality is defined on almost all data types, and only depends on the type structure of the data type. Examples of generic (sometimes also called *polytypic*) functionality defined on almost all data types are storing a value in a database, editing a value, comparing two values for equality, pretty-printing a value, etc. A function that works on many data types is called a generic function. Applications of generic programming can be found not just in the rather small programming examples mentioned, but also in

- XML tools such as XML compressors [12], and type-safe XML data binding tools [5];
- test set generation for automatic testing [18, 17];
- constructing 'boilerplate' code that traverses a value of a rich set of mutually recursive data types (for example representing the abstract syntax of a programming language) applying real functionality (for example collecting the variables that appear in expressions) at a small portion of the data type (the case for variables in the abstract syntax) [19, 24, 20, 32];
- structure editors such as XML editors [10], and generic graphical user interfaces [1, 30];
- data conversion tools [15] which for example store a data type value in a database [10], or output it as XML, or in a binary format [33].

This section introduces generic programming in Generic Haskell. We give a brief introduction to Generic Haskell, assuming the reader has some knowledge of Haskell [28] or ML. Generic Haskell is an extension of the lazy, higher-order, functional programming Haskell that supports generic programming; more details can be found in [13, 23]. Generic programs can also be written in Clean [3], ML [8, 7], Maude [25], Java [34], and some other programming languages.

[1] or DTDs, schemas, class hierarchies, etc. In the rest of this paper the word data type is used as a concept that represents all of these concepts.

2.1 Generic programming in Generic Haskell

A generic program is a program that works for a large class of data types. A generic program takes a type as argument, and is usually defined by induction on the type structure. As an example, we define a very simple generic function `content` that extracts the strings and integers (shown as strings) that appear in a value of an arbitrary data type. The instance of `content` on the type of binary trees with integers in the leaves, and strings in the internal nodes, defined by

```
data Tree  =  Leaf Int | Node Tree String Tree
```

returns `["3","Bla","7"]` when applied to `Node (Leaf 3) "Bla" (Leaf 7)`. The generic function `content` returns the document's content not only for the type `Tree`, but for any data type one can define. In particular, it will still work when a data type definition is changed[2].

```
content {| t :: *  |}               ::  t -> [String]
content {| Unit    |}  Unit         =   []
content {| Int     |}  int          =   [show int]
content {| String  |}  str          =   [str]
content {| a :+: b |}  (Inl a)      =   content {|a|} a
content {| a :+: b |}  (Inr b)      =   content {|b|} b
content {| a :*: b |}  (a :*: b)    =   content {|a|} a ++ content {|b|} b
```

Function `content {| t |}` is a type-indexed function. The type argument appears in between special parentheses `{|`, `|}`. The different type arguments are explained below. An instance of `content` is obtained by applying `content` to a type, for example, `content{|Tree|}`. The type of function `content` is given for a type `t` of kind `*`. This does not mean that `content` can only be applied to types of kind `*`; it only gives the type information for types of kind `*`. The type of function `content` on types with kinds other than `*` can automatically be derived from this base type. Note that the single type given for this function ensures that all instances of this function on particular types are type correct. A type-correct generic function generates type correct code [23]. Using an accumulating parameter we can obtain a more efficient version of function `content`.

To apply a program to (values of) different types, each data type that appears in a source program is mapped to its structural representation. This representation is expressed in terms of a limited set of data types, called structure types. A generic program is defined by induction on these structure types. Whenever a generic program is applied to a user-defined data type, the Generic Haskell compiler takes care of the mapping between the user-defined data type and its corresponding structural representation. If we want a generic function to exhibit non-standard behavior for a particular data type we can add an extra case expressing this behavior to the definition of the generic function.

The translation of a data type to a structure type replaces a choice between constructors by a sum, denoted by `:+:` (nested to the right if there are more than two constructors), and a sequence of arguments of a constructor by a product, denoted by `:*:` (nested to the right if there are more than two arguments). A nullary constructor is replaced by the structure type `Unit`. The arguments of the constructors are not translated. For example, for the data type `Tree` we have

[2]The (infix) operator `++` returns the concatenation of its two input strings.

```
data Tree       =  Leaf Int | Node Tree String Tree
type Str(Tree)  =  Int :+: (Tree :*: String :*: Tree)
```

Here `Str` is a meta function that given an argument type generates a new type name. The structural representation of a data type only depends on the top level structure of a data type. The arguments of the constructors, including recursive calls to the original data type, appear in the representation type without modification. A type and its structural representation are isomorphic (ignoring undefined values). The isomorphism is witnessed by a so-called *embedding-projection pair*: a value of the data type

```
data EP a b  =  EP  (a -> b) (b -> a)
```

The Generic Haskell compiler generates the translation of a type to its structural representation, together with the corresponding embedding projection pair.

From this translation, it follows that it suffices to define a generic function on sums (`:+:`, with values of the form `Inl l` or `Inr r`), products (`:*:`, with values of the form `l :*: r`, and `Unit`, with as only value `Unit`) and on base types such as `Int` and `String`. Function `content` has been defined on these structure types.

One might wonder whether there is an efficiency price to be paid for using generic programming. In principle this technique indeed introduces additional overhead. To apply the generic function, the original data structure first is converted to its structural representation. Then the generic function is applied, yielding another structural representation that is converted back to the data structure of the resulting domain. Due to these conversions the technique may indeed lead to inefficient code when it is frequently being applied to large data structures. Fortunately, there exists an optimization technique [2, 4] which completely removes the generic overhead for almost all cases, resulting in code which is as efficient as hand written code [32]. This makes it possible to use generic programming for real world practical applications.

3 Generic programming for software evolution

We claim generic programming is useful for software evolution. This section sketches a number of scenario's in which generic programming can be applied successfully to support software evolution.

Webshops. There are many webshops on the internet. The functionality of all these shops varies very little: product descriptions have to be retrieved from a database, and shown in a friendly way to the users, user actions have to be translated to database transactions, etc. Much of the software written for webshops only depends on the structure of the data (the products sold). Implementing a webshop using generic programming technology [30, 29] supports reuse of software for different webshops. Furthermore, whenever a product catalogue changes, which happens very frequently for most webshops, there is no need to rewrite any software.

Actually, generic programming can be used for any kind of data controlled web site. Using generic programming techniques arbitrary complicated interactive web forms, the shape and content of which depend on (stored) data or the contents of other web forms, can be generated automatically.

DTDs. A DTD (XML Schema) is used to describe the structure of a particular kind of documents. An example of a DTD is the TEI (Text Encoding Initiative) DTD. The Text Encoding Initiative Guidelines are an international and interdisciplinary standard that facilitates libraries, museums, publishers, and individual scholars to represent a variety of literary and linguistic texts for online research, teaching, and preservation. The first version of the TEI DTD appeared in 1990, and since then, updated versions have appeared in 1993, 1994, 1999, 2002, and 2005. Of course, with each new release, software that deals with TEI documents has to be updated. If the software uses a data binding [26], it is likely that with each new version of the DTD, much of the 'business logic' has to be rewritten, because the structure of the data has changed. Using generic programs, only those parts of the software that deal with the new features of the DTD have to be added.

Traversing large data structures. Programming languages evolve. For most programming languages, the abstract syntax used in a compiler is rather large. A lot of the functionality on this abstract syntax only addresses a small subpart of the abstract syntax. For example, we might want to collect all variables in a program. However, to reach a variable in the abstract syntax, large traversal functions have to be written. Using generic programming we can write a single traversal function, which we specialize for different special purposes. For example, for collecting all variables in the abstract syntax, we add a single line that deals with variables to the generic traversal function. If the abstract syntax evolves, such a function still works as expected. Many phases in a compiler can be implemented as generic programs [32]. Generic programming is not only useful for a typical generic phase like parsing, there is also gain for algorithms that are more specific, such as a type checker. Surprisingly many algorithms can be expressed conveniently in a generic way, by defining "exceptions to the general case".

Information Systems Another important area in which generic programming might be very useful, is Information Systems [27]. A well designed Information System is constructed from an Information Model that is the result of a requirements analysis. An Information Model can be seen as a type specification defining the structure and properties of the information that can be stored in the Information System. Information Systems are often derived systematically from Information Models. All the information about such a system is present in the model, and generating an Information System fully automatically from a given Information Model should therefore be possible. The functionality for storing, retrieving and changing information is directly deduced from the corresponding types. Information systems are not only accessed by applications, they are also inspected interactively by human beings. The webshop technology described above can be applied here as well. The graphical user interfaces needed for viewing and changing any part of the stored data can be generated automatically. When an organization changes, this usually has consequences for the Information Model and the corresponding Information System. By using generic programming techniques, the consequences of a change will be minimal. It is not clear yet what kind of programming effort will still be required when a Information Model is changed. Clearly, some of the old information stored in the old Information System has to be converted to information in the new format. But again such a conversion can be defined using generic techniques considerably reducing the amount of programming that has to be done.

Generic programming is useful for software evolution, and in order to be useful, software on data that changes frequently has to be implemented in terms of generic functions. Existing software that does not use generic programming techniques first has to be refactored or renovated such that it uses generic programming techniques for data that changes frequently.

Of course, generic programming does not solve all software evolution problems. For example, changes in the modular structure of programs, or changes in the API of a program are unrelated, and have to be solved using different methods. Furthermore, the approach only partially works for data types with properties, such as the data type of ordered lists. Finally, in dynamically typed languages the approach cannot be applied directly, and has to be simulated in some way, probably using soft typing techniques. Alternatively, reflection can be used to achieve similar results, but is much more difficult since it lacks the proper abstractions for defining generic programs. We think that using a typed programming language is a first step towards producing robust code.

4 Research challenges for generic programming

There are a number of research challenges that have to be investigated to make generic programming a viable tool for supporting software evolution.

- *Prototype applications.* We have to develop a number of prototype applications to demonstrate the approach. At the moment we are working on web applications [30, 29], in the near future we hope to work on database connection and migration tools. We will use these and maybe other tools to demonstrate the usefulness of generic programming for software evolution.

- *Software process.* To use generic programming techniques optimally, information systems have to be designed around the data from the design phase on. This probably means that the software process for developing systems that use generic programming techniques has to be slightly adapted. Furthermore, there should be a relatively straightforward mapping between the data modeling language and the implementation. The distinction between design and implementation is reduced considerably.

- *Programming language.* Although there exist a number of programming languages that have added support for generic programming in the last five years, all of these extensions are recent, and none of the extensions can be called mature: the generated code is often inefficient, generic functions are not first class, the structural representation of data types cannot be changed, etc.

We are in the process of writing a research proposal that addresses the first two research challenges for applying generic programming for software evolution.

5 Conclusions

Generic Programming is a recent programming technique that is useful for building software that needs to work with evolving data types. The implications of using generic programming for software development and software evolution have hardly been investigated, but first results are promising. A number of research challenges have to be investigated to make

generic programming an integral part of the software development process for developing software that has to deal with frequently changing data types.

References

[1] Peter Achten, Marko van Eekelen, and Rinus Plasmeijer. Generic Graphical User Interfaces. In *The 15th International Workshop on the Implementation of Functional Languages, IFL 2003, Selected Papers*, volume 3145 of *LNCS*, pages 152–167. Springer-Verlag, 2004.

[2] Artem Alimarine. *Generic Functional Programming - Conceptual Design, Implementation and Applications*. PhD thesis, University of Nijmegen, The Netherlands, 2005. ISBN 3-540-67658-9.

[3] Artem Alimarine and Rinus Plasmijer. A generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, volume 2312 of *LNCS*, pages 168–186. Älvsjö, Sweden, Springer, September 2002.

[4] Artem Alimarine and Sjaak Smetsers. Improved Fusion for Optimizing Generics. In Manuel Hermenegildo and Daniel Cabeza, editors, *Proceedings of Seventh International Symposium on Practical Aspects of Declarative Languages*, number 3350 in LNCS, pages 203 – 218. Long Beach, CA, USA, Springer, January 2005.

[5] Frank Atanassow, Dave Clarke, and Johan Jeuring. Scripting XML with Generic Haskell. In *Proceedings of the 7th Brazilian Symposium on Programming Languages, SBLP 2003*, 2003. An extended version of this paper appears as ICS, Utrecht University, technical report UU-CS-2003-023.

[6] Roland Backhouse and Jeremy Gibbons, editors. *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*. Springer-Verlag, 2003.

[7] Juan Chen and Andrew W. Appel. Dictionary passing for polytypic polymorphism. Technical Report TR-635-01, Princeton University, March 2001.

[8] Jun Furuse. Generic polymorphism in ML. In *Journées Francophones des Langages Applicatifs*, January 2001.

[9] Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002*. Kluwer Academic Publishers, 2003.

[10] Paul Hagg. A framework for developing generic XML Tools. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2002.

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

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

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

[14] Patrik Jansson and Johan 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.

[15] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[16] Johan Jeuring, editor. *Workshop on Generic Programming*. Utrecht University, 2000. Technical report UU-CS-2000-19.

[17] Pieter Koopma and Rinus Plasmeijer. Testing reactive systems with gast. In S. Gilmore, editor, *Proceedings Fourth symposium on Trends in Functional Programming, TFP03*, pages 111–129, Edinburgh, Scotland, 2003. ISBN 1-84150-122-0.

[18] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic Automated Software Testing. In Ricardo Peña, editor, *The 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Selected Papers*, volume 2670 of *LNCS*. Springer-Verlag, 2003.

[19] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. TLDI'03.

[20] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. *Proceedings ICFP'05*, 40(9):204–215, 2005.

[21] M.M. Lehman. Programs, life cycles and the laws of software evolution. *Proc. IEEE*, 68(9):1060–1078, 1980.

[22] M.M. Lehman and L.A. Belady. *Program Evolution: Processes of Software Change*. Academic Press, London, 1985.

[23] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

[24] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Olin Shivers, editor, *Proceedings of the International Conference on Functional Programming, ICFP'03*, pages 141–152. ACM Press, August 2003.

[25] N. Marti-Oliet M. Clavel, F. Duran. Polytypic programming in maude. In *WRLA 2000*, 2000.

[26] Brett McLaughlin. *Java & XML data binding*. O'Reilly, 2003.

[27] Betsy Pepels and Rinus Plasmeijer. Generating Applications from Object Role Models. In R. Meersman, editor, *Proceedings of the OTM Workshops 2005, OnTheMove - OTM 2005 Federated Conferences and Workshops*, volume 3762 of *LNCS*, pages 656–665, Agia Napa, Cyprus, Oct 31-Nov 4 2005. Springer.

[28] Simon 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.

[29] Rinus Plasmeijer and Peter Achten. The Implementation of iData - A Case Study in Generic Programming. In A. Butterfield, editor, *Proceedings Implementation and Application of Functional Languages, 17th International Workshop, IFL05*, Dublin, Ireland, September 19-21 2005. To Appear in Springer LNCS.

[30] Rinus Plasmeijer and Peter Achten. iData For The World Wide Web - Programming Interconnected Web Forms. In *Proceedings Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, Fuji Susono, Japan, Apr 24-26 2006. To Appear.

[31] A.L. Powell. A literature review on the quantification of software change. Technical Report YCS 305, Computer Science, University of York, 1998.

[32] Arjen van Weelden, Sjaak Smetsers, and Rinus Plasmeijer. A Generic Approach to Syntax Tree Operations. In A. Butterfield, editor, *Proceedings Implementation and Application of Functional Languages, 17th International Workshop, IFL05*, Dublin, Ireland, September 19-21 2005. To Appear in Springer LNCS.

[33] M. Wallace and C. Runciman. Heap compression and binary I/O in Haskell. In *2nd ACM Haskell Workshop*, 1997.

[34] Stephanie Weirich and Liang Huang. A design for type-directed programming in Java. In *Workshop on Object-Oriented Developments (WOOD 2004*, 2004.