Chapter 1

Generic Programming for Domain Reasoners

Johan Jeuring[1], José Pedro Magalhães[2], Bastiaan Heeren[3]
Category: Evaluation

Abstract: An exercise assistant is a tool that supports students learning procedural skills, such as solving systems of linear equations, or rewriting a logic expression to disjunctive normal form. The domain reasoner is the component of an exercise assistant that deals with tasks that depend on the semantics of a particular domain. An exercise assistant typically has multiple domain reasoners, and the behavior of each of these is, to a large extent, determined by the domain.

Generic programming techniques claim to reduce code duplication, to make it easier to change the structure of data, and to provide implementations of many useful functions on most datatypes. There are many libraries for generic programming, all with different datatype support, expressiveness, and ease of use. While a lot of effort has been put into developing new libraries in the past few years, there are few examples of real-life applications of generic programming. In this paper we describe our experience with applying generic programming techniques to our domain reasoners implemented in Haskell. We assess and compare the resulting generic domain reasoners with each other and with the initial, non-generic version, using the software quality factors of the ISO 9126 international standard.

1.1 INTRODUCTION

An exercise assistant helps a student with solving exercises, for example about solving equations, or determining the derivative of a function. There exist hundreds of exercise assistants, for many mathematical domains, logic, physics, etc.

[1]Universiteit Utrecht and Open Universiteit Nederland, The Netherlands; johanj@cs.uu.nl

[2]Universiteit Utrecht, The Netherlands; jpm@cs.uu.nl

[3]Open Universiteit Nederland, The Netherlands; bastiaan.heeren@ou.nl

An important part of an exercise assistant is the domain reasoner: a component which tracks the steps the student takes, gives hints, diagnoses errors, records progress, shows worked-out solutions, etc. The functionality of the domain reasoner fundamentally depends on the domain, the rules that hold for the domain, and the strategies for solving exercises within the domain.

The core components of a domain reasoner are:

- A description of the domain, consisting of an abstract syntax together with a parser and a pretty-printer.
- Rules with which expressions in the domain are manipulated. For example, for the domain of logic expressions, the distribution rule $x \vee (y \wedge z) \rightsquigarrow (x \vee y) \wedge (x \vee z)$, De Morgan's law $\neg(x \vee y) \rightsquigarrow \neg x \wedge \neg y$, etc.
- Strategies for solving exercises in the domain. For example, for rewriting a term into disjunctive normal form: first remove boolean constants, implications and equivalences, then push negations inside, and finally move occurrences of $\vee$ upwards.

Additionally, for each domain we need functionality for unifying and rewriting terms, generating exercises, traversing terms, determining the top-level equality of two terms, etc.

In the past few years we have developed domain reasoners which help with diagnosing student behavior in exercise assistants for calculating with fractions, performing Gaussian elimination, solving systems of linear equations and other linear algebra exercises, factoring polynomials, rewriting a logical term to disjunctive normal form, rewriting relation algebra terms, developing (extended) lambda-calculus programs, and determining the derivative or integral of a function [8, 9, 15, 19].

Our domain reasoners offer the functionality as described above as web services [6]. Several exercise assistants, such as MathDox [5], ActiveMath [7], and the Freudenthal Institute digital mathematics environment (DWO) [3], use our domain reasoning web services. At the moment we develop and maintain our own domain reasoners, but in the future we expect users of our services to add or adapt domains. Furthermore, we are involved in two recently funded research projects about bridging the knowledge gap in mathematics between secondary schools and universities: Math-bridge (from the EU eContentPlus programme), and NKBW2 (from the SURF Foundation, the Dutch higher education and research partnership organization for ICT). For these projects we have to deliver a number of domain reasoners each year.

To avoid reimplementing the same functionality for every domain, and to simplify the process of creating a new domain reasoner, we have used *generic programming* techniques [2] to implement rewriting, traversals, etc. once and for all domains. The application of generic programming techniques to our domain reasoners is the project that we evaluate in this paper: how can generic programming be used in a large, realistic application? What are the benefits and the shortcomings of the solution that uses generic programming techniques? We expect

generic programming techniques to simplify the addition of new domain reasoners or to change existing ones, and to reduce the code base and code duplication. However, application of generic programming techniques also poses some challenges. We have to choose a suitable approach, with the right expressiveness and datatype support. Additionally, we must ensure that we can keep all of the original functionality, and that the generic solution is indeed an improvement over the non-generic version.

We have implemented our domain reasoners in Haskell. Initially we chose Haskell because it is a language we feel comfortable in programming, and because domain reasoners manipulate expression trees, at which Haskell is good. Later, when we decided to make our domain reasoners generic, this turned out to be the right choice, since there are many approaches to generic programming in Haskell [11, 23]. Types (domains) play an important role in our software, and we also want to use typed approaches to generic programming, which discards an approach like DrIFT[4] and other approaches based on metaprogramming or code generation. The essential technology used in our domain reasoners is rewriting (together with matching). When rewriting, we may apply any rewriting rule at any position in a term. Of course, a rewriting rule can only be applied to a term within the domain. It follows that we need to know which subterms of a term belong to the domain again, which implies that we need to know the recursive structure of the domain. There are only a few generic programming libraries in Haskell that give the user access to the recursive structure of values: Uniplate [17], Regular [18], and Multirec [22] in a straightforward way, and Scrap Your Boilerplate (SYB) [14] in a rather cumbersome way (see the Uniplate module Data.Generics.PlateData which implements Uniplate using SYB). We have three implementations of a generic domain reasoner, one using Uniplate, one using Multirec, and one using Regular.

This evaluation paper discusses where and how generic programming techniques impact the quality of our software. To structure the discussion, we use the quality characteristics as defined in the ISO 9126 [13] international standard for the evaluation of software quality. We use these characteristics because they help guaranteeing completeness and independence in our comparison. When we discuss the characteristics, we will give examples of how they play a role in our domain reasoners. We will compare four versions of our domain reasoners: a version without generic programming techniques and three versions using generic programming, one for each of the Uniplate, Multirec and Regular libraries. We hope that our experience can be used as a reference for other projects that consider using generic programming.

1.2 USING GENERIC PROGRAMMING

We have refactored our existing domain reasoners to split off components that can be implemented by means of generic components. For example, the second

[4] `http://repetae.net/computer/haskell/DrIFT/`

version of our first domain reasoner (2005–2006) implemented linear equations using the following datatypes:

```
type Equations = [Equation]
data Equation  = Expr :=: Expr
data Expr      = Con Rational
               | EVar String
               | Expr :+: Expr
               | Expr :-: Expr
               | Expr :×: Expr
               | Expr :/: Expr
```

The domain reasoner contained functionality for rewriting linear equations and mathematical expressions that are used in equations:

```
module EquationsRewriteAnalysis where

equationsRewrite = ...
exprRewrite      = ...
equationsSubst   = ...
exprSubst        = ...
```

in a 449 line module. In our current domain reasoner, the domain of linear expressions does not contain specific functionality for rewriting and substitution anymore. Instead, generic functions for rewriting and substitution are called when the generic domain reasoner is used on linear expressions. The generic rewriting and substitution functions themselves are implemented in terms of basic generic functions such as *fold*, *zip*, *crush*, and *map* [18]. All in all, the generic modules for rewriting consist of 370 lines. The generic rewriting functionality is used in all other domains as well.

There are several ways in which generic programming can be used in our domain reasoners:

- To obtain instances of functions which naturally depend on the structure of the data type, such as the rewriting function, but also basic functions like *fold* and *map*.
- To use generic traversal functions to apply some function at a particular position in a possibly large value of a family of datatypes.
- To not have to adjust many functions when extending or redefining a domain.

We have used generic functions for all of these purposes.

1.3 GENERIC PROGRAMMING LIBRARIES

There are several libraries for generic programming in Haskell. These differ substantially in expressiveness, datatypes supported, and usability. Recently, a detailed comparison of generic programming libraries has been performed [21, 23]. We have used this comparison when choosing a generic programming library.

1.3.1 Datatypes in the domain reasoners

Our domain reasoners currently represent several domains, such as logic expressions, relation algebra and programming in a simple functional language. Each domain is represented by one or more datatypes. To provide generic functionality on a domain, we have to guarantee that the datatypes representing the domain can be treated generically. Different libraries for generic programming have different levels of support for datatypes.

Our datatypes have different levels of complexity. The domain of logic expressions, for instance, is represented by a single, recursive, non-parametric datatype:

```
data Logic = Logic :→: Logic   -- implication
           | Logic :↔: Logic   -- equivalence
           | Logic :∧: Logic   -- conjunction (and)
           | Logic :∨: Logic   -- disjunction (or)
           | Not Logic         -- negation (not)
           | Var String        -- variables
           | T                 -- true
           | F                 -- false
```

The domains of relation algebra, programming, and calculating with fractions are represented by datatypes of similar complexity.

Our representation of mathematical expressions uses lists:

```
data Expr = Expr :+: Expr      -- addition
          | Expr :×: Expr      -- multiplication
          | Expr :−: Expr      -- subtraction
          | Negate Expr        -- negation
          | Expr :/: Expr      -- division
          | Sqrt Expr          -- square root
          | Nat Integer        -- natural number
          | EVar String        -- variable
          | Sym String [Expr]  -- symbol
```

Due to their pervasiveness in Haskell, lists should be given a particular treatment by the generic programming library. For instance, we expect the algebra for the *Sym* constructor of the Expr datatype to have type

```
algSym :: String → [r] → r
```

This will not happen if, for instance, we treat the [Expr] occurrence as a constant type.

The domain of systems of linear equations uses a family of parametric datatypes:

```
type LinearSystem = [Equation Expr]
data Equation a   = a :=: a
```

We need the parametricity because we can represent equations of different kinds of expressions.

The domain for polynomials uses an abstract datatype:

```
newtype Polynomial a = P (IntMap a)
```

Using abstract datatypes increases efficiency and conciseness of the code, but precludes the application of generic programming techniques.

The expressiveness of a generic programming library is orthogonal to its support for datatypes. Some libraries can express more generic functions than others. For our domain reasoners, we need functionality such as rewriting, data generation, and folding. This clearly restricts our choice to libraries which have a notion of the recursive structure of the datatypes: Uniplate, Regular, and Multirec. We have used each of these three libraries at different stages in our domain reasoners. We will give a brief overview of each of the libraries and our reasons for using them.

1.3.2 Uniplate

The Uniplate library [17] provides combinators for generic traversals. It is similar in style to SYB, but simpler since it does not use rank-2 polymorphism. In the simplified version we use for our domain reasoners, it consists of a single function in a type class:

```
class Uniplate a where
  uniplate :: a → ([a], [a] → a)
```

The *uniplate* function provides a way to access the immediate children of a term (the first component of the result pair) and to rebuild a term using a list of replacement children (the second component of the pair). Every type to be used with this library has to be given an instance of the class *Uniplate*. We show part of the instantiation for the Logic datatype:

```
instance Uniplate Logic where
  uniplate (p :→: q) = ([p, q], λ[p, q] → p :→: q)
  uniplate (p :↔: q) = ...
  ...
```

The other constructors have a similar definition.

Generic functions in Uniplate are written using the *uniplate* combinator. The function *transform* below, for instance, applies a transformation function on every subelement of a term in a bottom-up fashion:

```
transform :: (Uniplate a) ⇒ (a → a) → a → a
transform f a = let (children, builder) = uniplate a
                in f $ builder $ map (transform f) children
```

In this function, *children* stands for the recursive children of the input term, and *builder* is the function that, given a list of replacement children, rebuilds the input term by replacing its children with elements from the list. Both are combined to apply the transformation function f to every recursive element of the input term.

We used Uniplate as the first generic programming library in our domain reasoners. Its simplicity allowed for easy integration and implementation of functionality such as top-level equality and a form of rewriting. However, functionality such as data generation or folding cannot be expressed in Uniplate.

1.3.3 Multirec

Multirec [22] is a recently developed generic programming library. It has the unique ability to represent mutually recursive datatypes in a fixed-point view, therefore allowing the definition of generic folds on mutually recursive datatypes, for instance. It is based on the representation of the pattern functor of a family of types using type families. Consider a possible representation for the domain of programming in a simple functional language:

```
data LExpr = Lambda String LExpr
           | LVar String
           | Apply LExpr LExpr
           | Fix LExpr
           | LInt Int
           | Let Decl LExpr
data Decl  = String := LExpr
           | Seq Decl Decl
```

To illustrate the generic representation used by Multirec, we show the pattern functor of this family of mutually recursive datatypes:

```
data AST :: * → * where
  LExpr :: AST LExpr
  Decl  :: AST Decl
type instance PF AST  =
      (     C Lambda (K String :×: I LExpr)
        :+: C LVar    (K String)
        :+: C Apply   (I LExpr  :×: I LExpr)
        :+: C Fix     (I LExpr)
        :+: C LInt    (K Int)
        :+: C Let     (I Decl   :×: I LExpr)
      ) :▷: LExpr
  :+: (     C Assign  (K String :×: I LExpr)
        :+: C Seq     (I Decl   :×: I Decl)
      ) :▷: Decl
```

Here, AST is an auxiliary datatype defined to represent the family. In its pattern functor PF, sums (:+:) and products (:×:) encode the choice between alternative

constructors and multiple arguments to a constructor, respectively. Constructors are encoded by C, constant types (outside the family) by K, and recursive occurrences (tagged with the recursive type) by I. A group of alternatives belonging to a single datatype is tagged with :▷:.

Also necessary are the conversion functions to and from the original datatypes into the generic view:

```
instance Fam AST where
  from LExpr (Lambda s e) = L (Tag (L (C (K s :×: I (I* e)))))
  from...
  to LExpr (L (Tag (L (C (K s :×: I (I* e)))))) = Lambda s e
  to...
```

Note that the recursive occurrences need to be wrapped in the I_* constructor because of the higher kind of the I constructor.

We used Multirec in our second version of the generic domain reasoners to increase the expressiveness. With Multirec we can write generic folds and datatype generation, as well as an improved rewriting mechanism. However, Multirec cannot represent parametric datatypes such as Equation, or datatypes with lists such as Expr (Section 1.3.1).[5] This reduces the number of domains on which we can use generic programming.

1.3.4 Regular

The Regular library is the underlying mechanism of the rewriting framework described in Van Noort et al. [18]. It can be seen as a simplification of Multirec, representing a single type at a time:

```
type instance PF Logic = C Impl  (I :×: I)      -- implication
                       :+: C Equiv (I :×: I)    -- equivalence
                       :+: C Conj  (I :×: I)    -- conjunction (and)
                       :+: C Disj  (I :×: I)    -- disjunction (or)
                       :+: C Not   I            -- negation (not)
                       :+: C Var   (K String)   -- variables
                       :+: C T     U            -- true
                       :+: C F     U            -- false

instance Regular Logic where
  from (p :→: q) = L (I p :×: I q)
  from...
  to L (I p :×: I q) = p :→: q
  to...
```

[5] As we mentioned previously, we want to treat lists as a special case. We could represent a datatype with lists in Multirec by considering a (monomorphic) list datatype as being part of the family, but this changes the semantics of the generic functions. For example, the algebras for the fold become less natural.

The pattern functor representation is simpler than in Multirec, since we no longer require tagging of recursive occurrences.

Regular does not support parametric datatypes or datatypes containing lists. However, we have investigated the possibility of changing the library to add support for parameters [10] and functor composition (of which datatypes with lists are a special case). Both extensions seem possible and do not lead to much increased code complexity. After having used Multirec, we realized our domain reasoners would benefit more from supporting parameters and datatypes with lists than from supporting families. Moreover, the Regular library is easier to use.

1.4 QUALITY CHARACTERISTICS

The ISO 9126 international standard introduces a quality model for the evaluation of software quality. This model lists six characteristic areas of importance. We use this set of characteristics to structure our discussion on the quality characteristics of our domain reasoners.

Functionality We have split off several components from the individual domain reasoners to be replaced by generic components. This has made it clear which functionality is common to all domains, and which functionality is specific for a particular domain. Examples of generic components are the *fold* function, functions for rewriting and matching, functions for generating arbitrary terms, etc. Splitting off generic components leads to an increased separation of concerns: improvements in rewriting functionality are solved in the rewriting component, and lead to improvements in rewriting for every domain. Furthermore, the functionality of the generic components can now be used by other applications as well. In addition to the generic components there is domain-specific functionality, for example functionality for determining the semantics of an expression in the domain.

Our first generic variant of the domain reasoners used Uniplate for the generic functionality. Uniplate allowed us to specify rewriting and traversals generically. However, its simplicity limits its functionality. In Uniplate one cannot write generic functions that build values from scratch. In both Multirec and Regular we can write generic QuickCheck [4] *Arbitrary* instances, which are used to generate exercises and test data, whereas using Uniplate we cannot. Uniplate also cannot define the generic fold, or even top-level equality. Both Multirec and Regular support the definition of these functions.

Multirec supports families of (possibly mutually recursive) datatypes. We have families of datatypes in some of our domain reasoners, but currently we do not need access to the recursive structure of a family of types: in general, we can focus on a single type and treat all the other occurring types as constants. For instance, for the domain reasoner of systems of linear equations described in Section 1.3.1, we only have rewrite rules for the Equations, and not for LinearSystems. However, having support for families of datatypes might be desirable in general.

We could describe our domain for simple functional programming by the LExpr and Decl datatypes shown in Section 1.3.1. Neither Uniplate nor Regular can represent the rewrite rule for inlining in this domain, Let $(x := e)$ (LVar x) $\rightsquigarrow e$, since it requires matching of both expressions and declarations. Currently our domain for simple functional programming deals with let declarations as syntactic sugar, and we therefore do not need the above mutually recursive datatypes. Using Multirec, however, would allow for such a change in the future.

The current version of Multirec has no support for type variables. The domain reasoner for systems of equations uses parametrized datatypes, as we have seen in Section 1.3.1. We expect support for parametric datatypes to be available in Multirec soon, but for now we cannot use it on this particular domain reasoner. For Regular, the extension necessary to support parametric datatypes is more or less straightforward, and described in detail by Hesselink [10].

Reliability The generic domain reasoners are more reliable than the original domain reasoners. One reason is that the generic domain reasoners make more use of library code, which is generally more reliable than datatype-specific code. Furthermore, the original domain reasoners contained some ad-hoc, and sometimes dangerous, solutions, which have been removed in the generic domain reasoners.

An example of a dangerous solution is the way metavariables were introduced and used in the original domain reasoners. To describe rewriting rules on the domains, the variables of the domains themselves were used as metavariables. This was implemented using a type class:

```
class MetaVar a where
  metaVar   :: Int → a
  isMetaVar :: a   → Maybe Int
```

The instantiation for the domain of logic expressions, for instance, represented metavariables as variables with a particular name:

```
instance MetaVar Logic where
  metaVar n    = Var ("_" ++ show n)
  isMetaVar (Var ('_' : s)) | ¬ (null s) ∧ all isDigit s
               = return (read s)
  isMetaVar _  = Nothing
```

Other domains had similar instances. This led to the possibly dangerous situation where a domain-level variable could be confused with a rewriting rule metavariable. Additionally, this only works for domains which have a notion of variables.

The above solution is part of the domain reasoners that use Uniplate. Using either Multirec or Regular, we represent metavariables at the generic structure representation level. The type synonym Scheme t represents an alternative (indicated by the sum :+:) between the t type and a metavariable:

```
type Scheme t = K MVar :+: PF t
type MVar     = Int
```

The rewriting functions now operate on Scheme t instead of t, guaranteeing that no metavariables can escape from the rewriting mechanism. Scheme t adds a new case to the pattern functor of t, namely a constant case for integers. This also lifts the restriction that the domain has to have some sort of variable constructor: every type t for which we have a pattern functor can now be extended to Scheme t.

Usability From the perspective of a programmer who is adding a new domain, customizing our domain reasoners is much easier using generic programming. We only need to specify the abstract syntax for the domain and its semantics-dependent operations (parsing and pretty-printing, rewriting rules, associative and commutative operators, and the strategies). We get functionality such as folding, top-level equality, rewriting with type-safe metavariable extension, traversals, and exercise generation for free.

With generic programming we also get a higher degree of composability. If we have specified the domains of rationals, floats, and matrices, combining them to obtain matrices of rationals or matrices of floats, with the correct set of rewrite rules, requires no effort.

Efficiency Generic programs tend to be less efficient than their handwritten counterparts. Compilers do not always perform the necessary optimizations to remove the generic representation types and conversion functions from the compiled code [16]. While this has been shown to be possible in at least the approach of Alimarine and Smetsers [1], in the general case it remains an open problem.

In Rodriguez [21] a simple benchmark of generic programming libraries is performed. The two Uniplate functions tested are approximately 5 times slower than their handwritten counterparts. For Multirec, performance varies from 1.7 to 47 times slower. Rewriting in Regular was benchmarked by Van Noort et al. [18], revealing that it is, in general, 3 to 4 times slower than type-specific rewriting. More recently, Magalhães et al. [16] show that stimulating inlining in GHC can dramatically increase the performance of Regular, but not of Multirec.

However, efficiency is generally not a problem for our domain reasoners. Since the terms represent exercises to be worked with by humans, they tend to be small. Hence, no penalty is noticeable for using any of the generic versions. Nevertheless, this could pose a problem for other large applications, such as abstract syntax tree traversals on compilers.

Maintainability Maintaining the code of the new domain reasoners is easier: not only is there less code to maintain, but the code is also more clearly structured. Initially we developed a naive generic *Arbitrary* generator that did not take term size or constructor frequency into account. Later we improved the implementation to take those into account. Since the generator is generic, it is only defined once, and all the domains immediately benefit from the advantages. With handwritten instances, we would have had to go through each domain implementing the same improvement everywhere.

In particular, repetitive code fragments like *Arbitrary* and *CoArbitrary* instances, folds, etc. lead to copy and paste programming which can give rise to subtle bugs. By using generic functions we can move (and reduce) a lot of testing to the generic programming library.

We looked at the number of lines of source code for defining the logic domain, together with essential functionality such as top-level equality, generation of data and rewriting. This is a simple but representative way of comparing the domain reasoners. The logic domain is not chosen because it is particularly suited for improvement, but simply because it is one of our earlier domains, so we have a non-generic domain reasoner for it. Our initial, non-generic version consisted of 496 lines of code, including type-specific code for a zipper and handwritten traversals over the domain. The Uniplate version, which used generic traversals and rewriting, consisted of 282 lines of code. Using Multirec, we can remove some remaining type-specific code for rewriting, and get the *fold* and data generation functions for free, reducing the total line count to 214, less than 50% of the original line count. The reduction with Regular is the same, since these libraries have very similar characteristics. The number of lines of source code is not necessarily a representative metric, but we think that in our case they faithfully express the source code reduction we achieved. In general, a datatype with n constructors will be defined in n lines of code. Using Uniplate, we still had to write the *arbitrary*, *coarbitrary*, top level equality, *fold*, and *uniplate* functions. Each of these is generally defined by pattern-matching on the domain datatype, therefore requiring n lines each. Additionally we had to define the *MetaVar* instance. With Multirec or Regular, each of the functions which required n lines require one line only, and there is no longer a *MetaVar* class. We only need to add 4 lines of code to invoke the Template Haskell generation of generic representations.

Turning our tool into a generic program has also made the code in the tool much more consistent. We had to be precise about our decisions in the generic code, so that all generic functionality is conceptually exactly the same for the different domains. Furthermore, there is now a single implementation for many functions which previously had one implementation per domain.

The only negative implication in maintainability is, possibly, the increased use of advanced Haskell programming techniques in the code. The initial version of the domain reasoners was written by relatively inexperienced Haskell programmers. As the number of domains grew, maintenance of the project was handed over to an experienced Haskell programmer, who introduced the generic version using Uniplate. The version using Multirec would probably not have been possible without close contact with the library creators. For Regular we have had to develop some features ourselves to increase the datatype support in the library. We suspect that only a seasoned Haskell programmer with broad experience in generic programming can maintain our current domain reasoners. However, bigger programs become naturally more complex, and our domain reasoners now offer much more functionality than in the initial version.

Nevertheless, care has to be taken to avoid proliferation of generics-related code throughout the entire code base. Multirec, for instance, introduces the con-

cepts of "family type" and "index type." In most other generic programming libraries, there is a single generic type, like the a in *Uniplate* a. In Multirec, we have *Fam* φ ix, with φ representing the family and ix a particular index on this family. This causes propagation of parameters in type classes, and increases the overall complexity of the code in the domain reasoners. For instance, instead of an exercise on expressions (Exercise Expr), with Multirec we have an exercise on the expression type in the family of datatypes for expressions (Exercise ExprFam Expr). However, for many domains, the family of datatypes contains a single type. There is no additional generality but the family type has to be propagated through the entire code. This reduces the overall readability and maintainability of the code.

Portability The original domain reasoners were developed in Haskell 98 [20], which made the portability of the domain reasoners good. When we moved to generic domain reasoners we first used an embedded implementation of Uniplate. This implementation was a subset of the Uniplate package available on HackageDB which was entirely Haskell 98, so it did not influence portability. Then we used Multirec and Regular, which introduce dependencies on a number of advanced extensions of Haskell (such as type families) only available in GHC, the main compiler for Haskell. These extensions are liable to change when new versions of the compiler are released, which means we might have to adapt our code. As a consequence, portability is worse.

1.5 CONCLUSIONS

The use of generic programming libraries in our domain reasoners has given us increased functionality, reliability, usability, and maintainability. This is caused by a clearer separation of concerns, decreased number of lines of code, higher composability, and making it simpler to add a new domain. Using generic programming possibly leads to worse performance, but this is not noticeable for users. Using Multirec and Regular we depend on more language extensions and can only compile with GHC.

We have also seen that using a simple generic programming library (Uniplate) gave us a considerable improvement in our code quality, but it eventually became a limitation in itself. The rewriting mechanism, for instance, required the domains to have variables, and we could not write generic producers. Moving to more complex generic programming libraries lifted these restrictions, but made us give up on compiler portability.

We believe the use of generic programming and particularly Regular will also enable us to improve the ease of use and functionality of the domain reasoners further. We are currently looking into rewriting expressions taking into account associativity and commutativity of (some) binary operators. This is useful for two reasons: it reduces the number of rules of the domain[6] and it simplifies rec-

[6] Specifying $x+0 \rightsquigarrow x$ should be enough to express the neutral element of addition. The extra rule $0+x \rightsquigarrow x$ should not be necessary.

ognizing application of rules to a term by the user. Since pretty-printed expressions almost never contain unnecessary parentheses, a user is generally unaware of associativity. For increased reliability in the traversals, we want to use the generic zipper [12]. There is an implementation of a zipper for Multirec using type-indexed datatypes through type families. This implementation can easily be converted to Regular. This would replace the current untyped approach using a list of integers to represent a location in a datatype. We also plan to look into the generic selection problem in the exercise assistant: the user should be able to select a subexpression and apply a rule to that part of the expression. Apart from some domain-specific information regarding the way terms are pretty-printed and which terms are selectable, this problem can also be solved with a generic algorithm.

The use of generic programming techniques in our domain reasoners provides us with a piece of software of better overall quality. The advantages clearly outweigh the disadvantages, and the effort invested into translating to a generic approach pay off when adding multiple new domains. We expect other projects to potentially benefit from the application of generic programming techniques too. However, not all projects are suitable for this. In our case, we have several domains which require similar functionality. Each domain is represented by one or more datatypes, on top of which the functionality is implemented. Other projects which manipulate several datatypes (e.g. structure editors), or perhaps a single, large datatype (e.g. compilers) are also well-suited for the application of generic programming, and should benefit in the same way as our domain reasoners did.

Acknowledgements This work has been partially funded by the Portuguese Foundation for Science and Technology (FCT), via the SFRH/BD/35999/2007 grant, and by the Netherlands Organisation for Scientific Research (NWO), through its project on "Real-life Datatype-Generic Programming" (612.063.613). We thank Andres Löh for his help with using Multirec and Erik Hesselink for his work on its extension. Alex Gerdes provided feedback on an early version of this paper. Harrie Passier and Arthur van Leeuwen contributed to older versions of the exercise assistants. We also thank the anonymous referees for their helpful feedback.

REFERENCES

[1] A. Alimarine and S. Smetsers. Optimizing generic functions. In D. Kozen and C. Shankland, editors, *MPC*, volume 3125 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

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

[3] P. Boon and P. Drijvers. Algebra en applets, leren en onderwijzen (algebra and applets, learning and teaching, in Dutch). `http://www.fi.uu.nl/publicaties/literatuur/6571.pdf`, 2005.

[4] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, ICFP 2000*, pages 268–279. ACM, 2000.

[5] A. Cohen, H. Cuypers, E. Reinaldo Barreiro, and H. Sterk. Interactive mathematical documents on the web. In *Algebra, Geometry and Software Systems*, pages 289–306. Springer, 2003.

[6] A. Gerdes, B. Heeren, J. Jeuring, and S. Stuurman. Feedback services for exercise assistants. In *ECEL 2007: Proceedings of the 7th European Conference on e-Learning*, 2008. Also available as Technical report Utrecht University UU-CS-2008-018.

[7] G. Goguadze, A. González Palomo, and E. Melis. Interactivity of exercises in ActiveMath. In *International Conference on Computers in Education, ICCE 2005*, 2005.

[8] B. Heeren and J. Jeuring. Recognizing strategies. In A. Middeldorp, editor, *WRS 2008: Reduction Strategies in Rewriting and Programming, 8th International Workshop*, 2008.

[9] B. Heeren, J. Jeuring, A. van Leeuwen, and A. Gerdes. Specifying strategies for exercises. In S. Autexier et al., editor, *MKM 2008: Mathematical Knowledge Management*, volume 5144 of *LNAI*, pages 430–445. Springer, 2008.

[10] E. Hesselink. Generic programming with fixed points for parametrized datatypes. Master's thesis, Utrecht University, 2009.

[11] R. Hinze, J. Jeuring, and A. Löh. Comparing approches to generic programming in Haskell. In *Datatype-Generic Programming*, LNCS 4719, pages 72–149. Springer, 2007.

[12] G. Huet. The zipper. *JFP*, 7(5):549–554, 1997.

[13] ISO/IEC. 9126-1:2001: Software engineering—product quality—part 1: Quality model. Technical report, International Organization for Standardization, Geneva, Switzerland, 2001.

[14] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *TLDI '03*, pages 26–37, 2003.

[15] J. Lodder, H. Passier, and S. Stuurman. Using IDEAS in teaching logic, lessons learned. In *CSSE 2008: International Conference on Computer Science and Software Engineering*, pages 553–556, 2008.

[16] J. P. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing generics is easy! Technical Report UU-CS-2009-022, Department of Information and Computing Sciences, Utrecht University, 2009.

[17] N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Haskell'07*, 2007.

[18] T. van Noort, A. Rodriguez Yakushev, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. In *WGP '08*, pages 13–24. ACM, 2008.

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

[20] S. Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. A special issue of JFP.

[21] A. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. PhD thesis, Utrecht University, 2009.

[22] A. Rodriguez Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2009*, 2009.

[23] 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*, pages 111–122, 2008.