Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Generic Programming for Domain Reasoners

José Pedro Magalhães[1]

joint work with

Johan Jeuring[1,2] Bastiaan Heeren[2]

[1] Department of Information and Computing Sciences, Utrecht University
[2] School of Computer Science, Open University of the Netherlands

Trends in Functional Programming
June 2–4, 2009

Overview

Domain Reasoners

Generic Programming

Evaluation

Future work and conclusions

1. Domain Reasoners

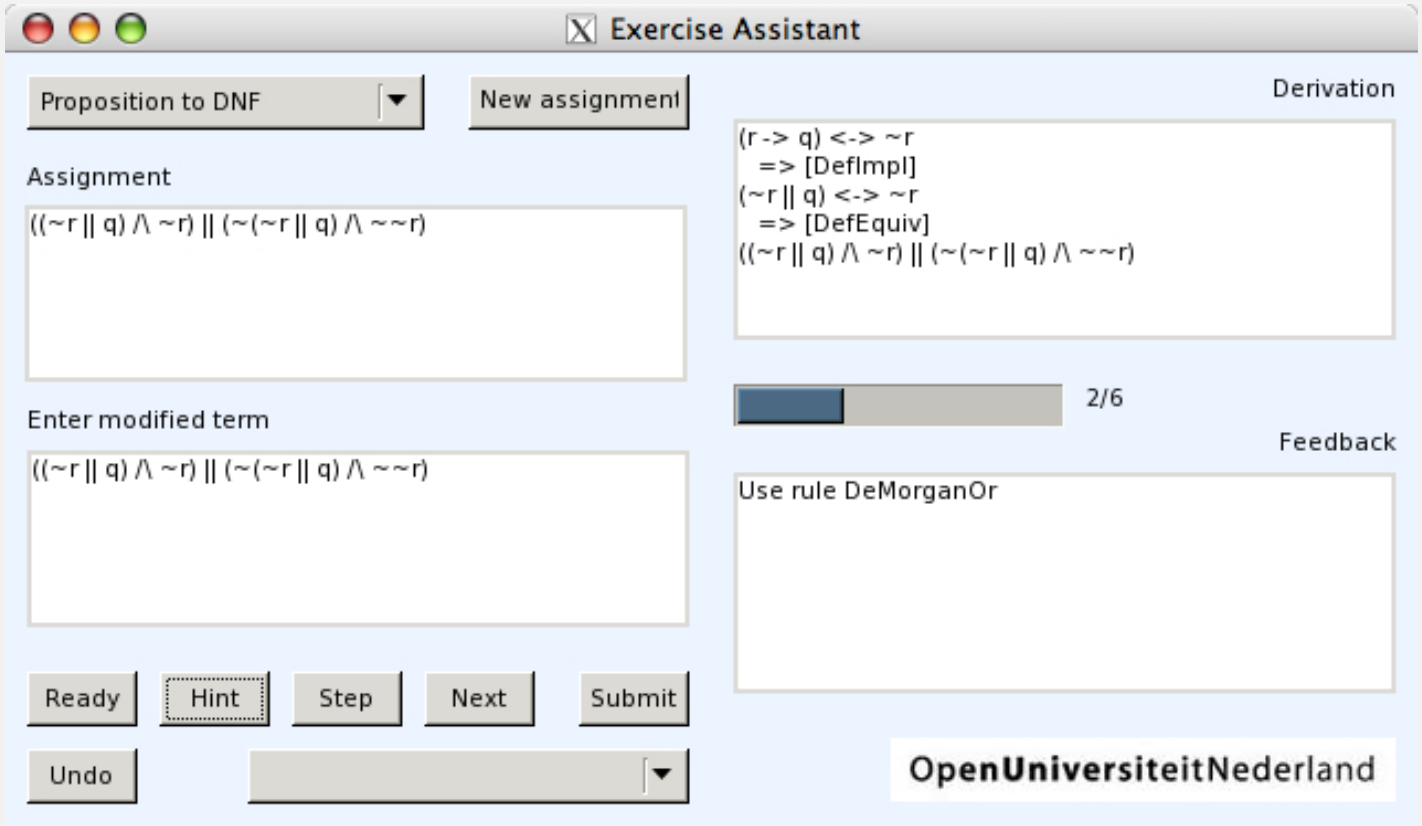
Exercise assistants I

An exercise assistant helps a student with some procedural skill:

- Turning a logical proposition into disjunctive normal form
- Solving linear equations
- Performing Gaussian elimination

There exist hundreds of exercise assistants, for many mathematical domains, logic, physics, etc.

Exercise assistants II



Also available online (demo).

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Domain reasoners I

The domain reasoner is the core component of an exercise assistant:

- Track the steps the student takes
- Give hints
- Diagnose errors
- Record progress
- Show worked-out solutions

Domain reasoners II

The functionality of the domain reasoner fundamentally depends on the domain:

- The rules that hold for the domain
- The strategies for solving exercises within the domain

Example: the logic domain

```
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
```

Example: the programming domain

```
data LExpr = Lambda String LExpr    -- lambda abstraction
           | LVar String            -- variables
           | Apply LExpr LExpr      -- function application
           | Fix LExpr              -- fixpoint
           | Int Int                -- integers
           | Let Decl LExpr         -- let ... in

data Decl = String := LExpr         -- variable declaration
```

2. Generic Programming

Generic behavior

Many parts of the domain reasoners can be implemented in a datatype-generic way: its behavior does not depend on the domain in question. Examples of such parts are:

- Folding
- Top-level equality
- Rewriting (which requires metavariable extension)
- Traversals
- Exercise generation

Generic programming

Generic programming techniques claim to:

- Reduce code duplication
- Make it easier to change the structure of data
- Provide implementations of many useful functions on most datatypes

We have applied generic programming techniques to our domain reasoners and compared the initial, non-generic version with two versions using different libraries for generic programming in Haskell.

Type-specific domain reasoners

We started our project with a single domain reasoner for solving linear equations. All required functionality was implemented specifically for this domain.

Rewriting, for instance, was implemented in a 449 line module:

$$\begin{aligned}&\textbf{module } \mathit{EquationsRewriteAnalysis} \textbf{ where}\\ &\quad \mathit{equationsRewrite} = \ldots\\ &\quad \mathit{exprRewrite} \quad\;\; = \ldots\\ &\quad \mathit{equationsSubst} \;\; = \ldots\\ &\quad \mathit{exprSubst} \quad\;\; = \ldots\end{aligned}$$

A package for generic rewriting consists of only 370 lines of source code.

Uniplate

As the necessity to implement other domains arose, we moved to a simple generic programming library to simplify traversals and rewriting. For this we used Uniplate (Mitchell and Runciman, 2007).

Uniplate

As the necessity to implement other domains arose, we moved to a simple generic programming library to simplify traversals and rewriting. For this we used Uniplate (Mitchell and Runciman, 2007).

$$\textbf{class } \textit{Uniplate}\ \mathsf{a}\ \textbf{where}$$
$$\quad \textit{uniplate} :: \mathsf{a} \to ([\mathsf{a}], [\mathsf{a}] \to \mathsf{a})$$

$$\textbf{instance } \textit{Uniplate}\ \mathsf{Logic}\ \textbf{where}$$
$$\quad \textit{uniplate}\ (l_1 :\!\!\to\!\!: l_2) = ([l_1, l_2], \lambda[l_3, l_4] \to l_3 :\!\!\to\!\!: l_4)$$
$$\quad \textit{uniplate}\ (l_1 :\!\!\leftrightarrow\!\!: l_2) = \ldots$$
$$\quad \ldots$$

Multirec I

However, Uniplate was too limited for all the functionality we needed:

- No generic producers
- Does not integrate nicely with type-indexed datatypes

As the number of domains grew, we moved to Multirec (Rodriguez *et al.*, 2009):

- Fixed point view on data, which we need for functionality such as type-safe rewriting
- Supports mutually recursive datatypes, which we need for some of our domains (like programming)

Multirec II

```
type instance PF Programming =
        (K String :×: I LExpr) :▷: LExpr
    :+: (K String)             :▷: LExpr
    :+: (I LExpr :×: I LExpr)  :▷: LExpr
    :+: (I LExpr)              :▷: LExpr
    :+: (K Int)                :▷: LExpr
    :+: (I Decl :×: I LExpr)   :▷: LExpr
    :+: (K String :×: I LExpr) :▷: Decl
```

```
class Fam φ where
  from :: φ ix → ix → PF φ I* ix
  to   :: φ ix → PF φ I* ix → ix
```

3. Evaluation

ISO 9126 quality model

To compare our initial, non-generic domain reasoners with the versions using Uniplate and Multirec, we have used the quality characteristics described in the ISO 9126 international standard for the evaluation of software quality:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Functionality

Using Uniplate, we can separate the functionality for generic rewriting and traversals. This leads to increased modularity and separation of concerns.

Using Multirec, we can further separate the functionality for folds, top-level equality and exercise generation.

However, Multirec does not (yet) support parametric datatypes. This means we cannot use it with our domain reasoner for systems of equations.

Reliability I

The generic domain reasoners are more reliable since they make use of more library code.

Another example of increased reliability is the way metavariables were handled:

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

The idea was to represent metavariables as variables in the domain with a particular name.

Reliability II

For the logic domain:

$$
\begin{aligned}
&\textbf{instance } \mathit{MetaVar} \text{ Logic } \textbf{where} \\
&\quad \mathit{metaVar}\ n \quad = \mathit{Var}\ (\text{"\_"} +\!\!+ \mathit{show}\ n) \\
&\quad \mathit{isMetaVar}\ (\mathit{Var}\ (\text{'\_'} : s)) \mid \neg\ (\mathit{null}\ s) \wedge \mathit{all}\ \mathit{isDigit}\ s \\
&\qquad\qquad = \mathit{return}\ (\mathit{read}\ s) \\
&\quad \mathit{isMetaVar}\ \_ = \mathit{Nothing}
\end{aligned}
$$

Reliability II

For the logic domain:

$$
\begin{aligned}
&\textbf{instance } \mathit{MetaVar}\ \text{Logic } \textbf{where}\\
&\quad \mathit{metaVar}\ n \quad = \mathit{Var}\ (\text{"\_"} +\!\!+ \mathit{show}\ n)\\
&\quad \mathit{isMetaVar}\ (\mathit{Var}\ (\text{'\_'} : s)) \mid \neg\ (\mathit{null}\ s) \wedge \mathit{all}\ \mathit{isDigit}\ s\\
&\qquad\qquad = \mathit{return}\ (\mathit{read}\ s)\\
&\quad \mathit{isMetaVar}\ \_ = \mathit{Nothing}
\end{aligned}
$$

The version with Multirec uses type-safe metavariable extension:

$$
\begin{aligned}
&\textbf{type}\ \text{Scheme s} = \text{K MVar :+: PF s}\\
&\textbf{type}\ \text{MVar} \quad = \text{Int}
\end{aligned}
$$

This ensures that no metavariables can escape and it works on domains that have no notion of variables.

Usability

From the perspective of a programmer who has to add a new domain, customizing our domain reasoners is now much easier. A programmer only has to specify:

- Abstract syntax
- Semantics-dependent operations:
  - Parsing and pretty-printing
  - Rewriting rules
  - Associative and commutative operators
  - The strategies for the exercises

Generic functionality comes for free.

Efficiency

The generic domain reasoners are probably less efficient than their handwritten counterparts.

We have not yet performed detailed benchmarks, but recent research (Rodriguez 2009) shows that while Uniplate performs reasonably well, Multirec can be more than 40 times slower than a handwritten solution.

Further work is required to optimize the performance of generic programming libraries.

Maintainability

Maintaining the code of the generic domain reasoners is easier, since there is less code to update. If a domain changes, the generic functionality does not have to be adapted.

The number of lines of source code decreases dramatically when using generic programming (defining the abstract syntax and essential functionality):

Handwritten	Uniplate	Multirec
496	282	214

However, advanced generic programming techniques are now used, so the maintainer needs to be a skilled functional programmer.

Portability

The original domain reasoners were developed in Haskell 98, which is a standard portable across many compilers.

For the domain reasoners with Uniplate, we used an embedded, simplified implementation of the library which is also Haskell 98.

The use of Multirec introduces dependencies on a number of advanced extensions of Haskell only available in GHC, so portability is worse.

4. Future work and conclusions

Future work I

Currently we are working on using a new version of Multirec which supports parametric mutually recursive datatypes.

This should allow us to use Multirec in the domain reasoner for systems of equations.

We still expect to report on the results of the new version, since this is a considerable limitation.

Future work II

Other improvements using the new generic framework are still possible:

- Associativity- and commutativity-aware rewriting
  - Reduce the number of rules
  - Simplify recognition of application of rules by the user
- Use of generic zipper for traversals for increased reliability
- Generic selections

Conclusions

The use of generic programming libraries in our domain reasoners has given us increased functionality, reliability, usability, and maintainability.

Using generic programming probably leads to worse performance, but this is, in our case, not noticeable for users. Using Multirec we depend on more language extensions and can only compile with GHC.

Overall, the advantages outweigh the disadvantages and the effort invested into translating to a generic approach pays off when adding multiple new domains.