DGG: Datatype-generic Generator

Jurriën Stutterheim

January 24, 2011

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Overview

1. DGG: Datatype-generic Generator

1.1 Datatype-generic Programming

Nota Bene

§1.1

- The following slides give a very short introduction to generic programming for the purpose of setting a context for this presentation
- This introduction is incomplete and does not aim to provide you with a full understanding of generic programming

Datatype-generic Programming

§1.1

- Datatype-generic programming (DGP) allows functions to be defined once for a wide range of different data types
- Functions are programmed over the structure of a data type, rather than the data types themselves
- No more countless numbers of class instances

Sum of Products

Regular datatype:

$$\mathbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

Sum of products view:

$$\mathbb{1} + (a \times List\, a)$$

EMGM embedding-projection pair

Converting from and to a generic view is done using an embedding-projection pair (EP).

$$
\begin{array}{l}
\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a) \\
\textbf{type}\ ListRep\ a = Unit : + : a : * : List\ a \\
listEP :: EP\ (List\ a)\ (ListRep\ a) \\
listEP = EP\ from\ to \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
from\ Nil & = L\ Unit \\
from\ (Cons\ x\ xs) & = R\ (x : * : xs) \\
to\ (L\ Unit) & = Nil \\
to\ (R\ (x : * : xs)) & = Cons\ x\ xs
\end{array}
\end{array}
$$

Observation

- A lot of the generic representation code is highly regular and can be automatically generated
- Most generic programming libraries do so using Template Haskell

1.2 Introducing DGG

The problem

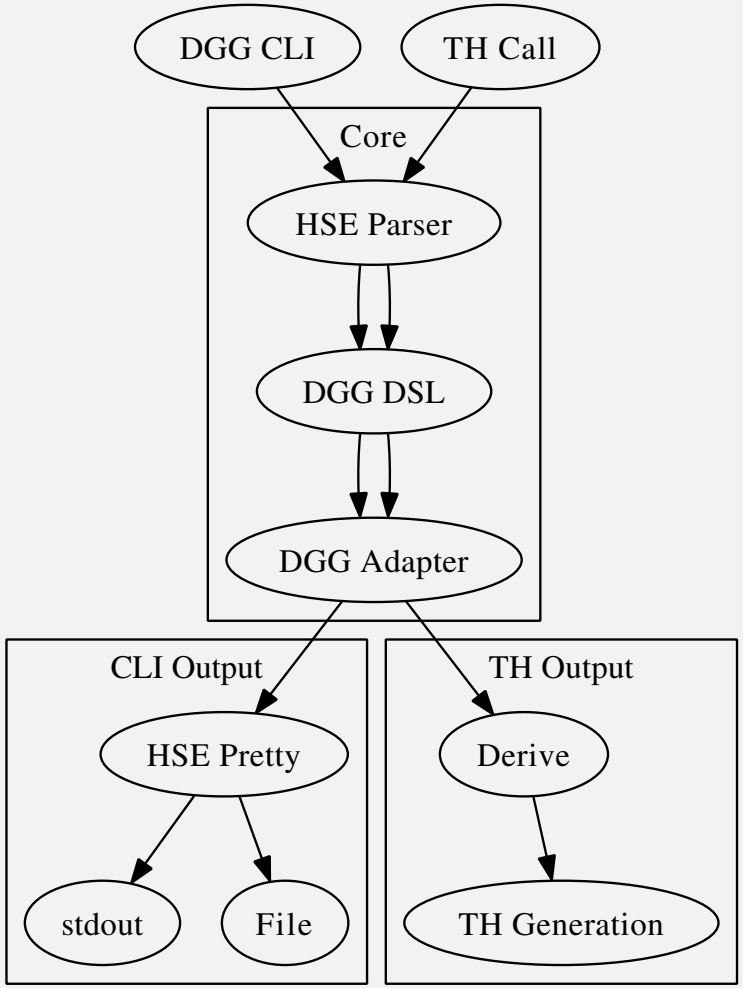
- Individual DGP libraries often have similar functionality for gathering information about data types
- Problems with Template Haskell
  - Changes frequently with new GHC releases
    - DGP libraries need to be updated to the new API
    - Hard to maintain BC with older versions
  - Debugging TH code is hard: `-ddump-splices`
  - TH is only available on GHC

The solution

- Parse data types to a custom DSL which individual DGP libraries can use to generate code, reducing duplicate work
- Abstract from Template Haskell
  - Using `haskell-src-exts` and the DSL
  - Derive supports using TH with `haskell-src-exts`
  - Offers a more stable API to DGP library developers
- Offer a command line tool
  - Output generated code to file or stdout
  - Easy for debugging when developing a DGP library
  - Compiler independent

Architecture

§1.2



1.3 Using DGG

Current support

§1.3

- Regular data types
- EMGM's embedding-project pair and (very) limited support for `Rep`, `FRep` etc.
- SYB `Data` and `Typeable`

Template Haskell usage

The `derive` function is from the Derive library

The `deriveEMGM` function is from DGG

$$\$\,(derive\ deriveEMGM\ ''\ MyDataType)$$

CLI usage

Parse all datatypes in `DataTypes.hs` and generate code for SYB, output code to stdout

```
dgg -i ‘‘DataTpes.hs’’ -a ‘‘SYB’’
```

Parse all datatypes in `DataTypes.hs` and generate code for EMGM, write code to `Reps.hs`

```
dgg -i ‘‘DataTypes.hs’’ -a ‘‘EMGM’’ -o ‘‘Reps.hs’’
```

Example output

norm2782@Dalwhinnie: ~/Documents/UU/AFP/Project...

```
➜ src git:(master) ✗ runhaskell Main.hs -i "List.hs" -a "emgm"
module GenericReps where
import Generics.EMGM
dggEP_List = EP from' to'
  where from' Nil = L Unit
        from' (Cons a1 a2) = R (a1 :*: a2)
        to' (L Unit) = Nil
        to' (R (a1 :*: a2)) = Cons a1 a2
➜ src git:(master) ✗
```

1.4 Extending DGG

Extending DGG

- DGG is easily extended to support new DGP libraries
- Each adapter module exports four functions:

$$
\begin{aligned}
&imports* &&:: [ImportDecl] \\
&derive* &&:: Derivation \\
&make* &&:: CodeGenerator \\
&isSupp* &&:: UnivSupp \rightarrow Bool
\end{aligned}
$$

Where $*$ is replaced by the name of the DGP library

CLI tool

- After writing the adapter, only the `Main.hs` file needs to be modified.
- The following line is added to make the library selectable via the `-a` flag:

$$(\texttt{"foogp"}, \mathit{Adapter}\ \mathit{makeFooGP}\ \mathit{isSuppFooGP}\ \mathit{importsFooGP})$$

DGG DSL

DGP library specific code is generated from these data types

$$
\begin{array}{l}
\textbf{data}\ TCInfo = TCInfo\ \{\, tcName :: Name \\
\qquad\qquad\qquad\qquad\quad ,\, tcType\ \ :: TypeType \\
\qquad\qquad\qquad\qquad\quad ,\, tcVars\ \ :: [\,TCVar\,] \\
\qquad\qquad\qquad\qquad\quad ,\, tcDCs\ \ :: [\,DCInfo\,] \\
\qquad\qquad\qquad\qquad\quad \}\\
\textbf{data}\ DCInfo = DCInfo\ \{\, dcName :: Name \\
\qquad\qquad\qquad\qquad\quad ,\, dcIndex :: Int \\
\qquad\qquad\qquad\qquad\quad ,\, dcFixity :: ConFixity \\
\qquad\qquad\qquad\qquad\quad ,\, dcAssoc :: Associativity \\
\qquad\qquad\qquad\qquad\quad ,\, dcVars\ \ :: [\,DCVar\,] \\
\qquad\qquad\qquad\qquad\quad \}
\end{array}
$$

1.5 Wrapping up

Future work

- Truly detect whether data types are supported by the DGP library
- Code generation up to standards of the existing TH solutions
- Add support for GADTs and type synonyms
- Improved kind analysis
- Infix operators and associativity information

Conclusion

- Reduces the amount of duplicate work for DGP library developers
- Abstracts from Template Haskell
- Offers CLI utility
  - Improved adapter debugging
  - Compiler independent
- Available on GitHub: https://github.com/norm2782/DGG

1.6 Questions?