Type Classes in Functional Logic Programming

Enrique Martín Martín
emartinm@fdi.ucm.es
Dpto. Sistemas Informáticos y Computación
Universidad Complutense de Madrid

20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation

January 24-25, 2011
Austin, Texas, USA
Combination of the best of declarative paradigms in a single model:

- **Functional languages**: HO functions, type systems, demand-driven evaluation.
- **Logic languages**: partial structures, non-deterministic search, unification
- **Constraint languages**: efficient constraint solving.

Systems: Toy, Curry
Call-time choice semantics: all copies of a non-deterministic expression created during reduction must be shared.

\[
\begin{align*}
\text{coin} &= 0 \\
\text{coin} &= 1 \\
\text{dup } X &= (X, X) \\
\end{align*}
\]

\( > \text{dup } \text{coin} \rightarrow (\text{coin}, \text{coin}) \rightarrow (0,0) \)

\( > \text{dup } \text{coin} \rightarrow (\text{coin}, \text{coin}) \rightarrow (1,1) \)

\((0,1)\) and \((1,0)\) are not values of \(\text{dup } \text{coin}\)
Type classes and FLP

- Type classes provide a clean and modular way of writing overloaded functions.
- Type classes are usually implemented using dictionaries.
- However, dictionaries present a problem of bad interaction with non-determinism and call-time choice when used in FLP.
class arb A where
    arb :: A

instance arb bool where
    arb = true
    arb = false

arbL2 :: arb A => [A]
arbL2 = [arb, arb]
Translation with dictionaries (I): classes

- Each **class declaration** generates a data declaration for **dictionaries** and **projecting functions** to extract from dictionaries.

```haskell
class arb A where
  arb :: A
```

```haskell
data dictArb A = dArb A
arb :: dictArb A -> A
arb (dArb Farb) = Farb
```
Translation with dictionaries (II): instances

- Each **instance declaration** generates a **concrete dictionary**.

```haskell
instance arb bool where
  arb = true
  arb = false

arbBool :: bool
arbBool = true
arbBool = false

dictArbBool :: dictArb bool
dictArbBool = dArb arbBool
```
Overloaded functions are transformed to accept dictionaries as arguments.

```haskell
arbL2 :: arb A => [A]
arbL2 = [arb, arb]
```

```haskell
arbL2 :: dictArb A -> [A]
arbL2 Darb = [arb Darb, arb Darb]
```
Bad interaction with call-time choice and non-determinism

- **Missing answers** in FLP with non-determinism and nullary member functions due to **sharing**.
  [Wolfgang Lux: Type-classes and call-time vs. run-time (Curry mailing list)]

```haskell
> arbL2 :: [bool]
arbL2 dictArbBool → [arb dictArbBool, arb dictArbBool]
→* [arb (dArb true), arb (dArb true)]
→* [true, true]

> arbL2 :: [bool]
arbL2 dictArbBool → [arb dictArbBool, arb dictArbBool]
→* [arb (dArb false), arb (dArb false)]
→* [false, false]
```

- **[true, false]**, **[false, true]** are not reached
This paper

- A **type-passing** translation of type classes in FLP using **type-indexed functions** (TIF): functions with a different behaviour for each different type.

- The translation is **well-typed** in a **new liberal type system for FLP**.
  
  [Liberal Typing for Functional Logic Programs (APLAS'10)]

- This translation **solves** the problem of **missing answers** and it also has **other advantages**.
Outline

- Our liberal type system.
- Translation using TIFs and type witnesses.
- Advantages of the translation.
- Conclusions.
- Future work.
Our Liberal Type System
Type system

- New type system for FLP [APLAS'10].
  - Type declarations are mandatory.
  - Similar to Damas-Milner for deriving/inferring types for expressions.
  - Well-typedness of a program proceeds rule by rule.
  - Possibilities for generic programming: generic functions, type-indexed functions.
**Definition of well-typed rule**

A program rule is **well-typed** if the right-hand side fixes the types of the variables and the result **less** than the left-hand side.

**It guarantees type preservation.**
Type system: example

\[
\begin{align*}
\text{size} & : \ \textbf{A} \rightarrow \textbf{nat} \\
\text{size}\ \text{false} & = s\ z \\
\text{size}\ \text{true} & = s\ z \\
\text{size}\ z & = s\ z \\
\text{size}\ (s\ \text{X}) & = s\ (\text{size}\ \text{X})
\end{align*}
\]
Type system: example

\[
\text{size :: } A \rightarrow \text{nat}
\]

\[
\text{size false} = \text{s z}
\]

\[
\text{size true} = \text{s z}
\]

\[
\text{size z} = \text{s z}
\]

\[
\text{s z :: nat}
\]

\[
\text{size false :: nat}
\]
Type system: example

\[
\text{size} :: A \rightarrow \text{nat} \\
\text{size} \ false = s \ z \\
\text{size} \ true = s \ z \\
\text{size} \ z = s \ z \\
\text{size} \ (s \ X) = s \ (\text{size} \ X)
\]
Type system: example

\[
\text{size} \::\ A \rightarrow \text{nat} \\
\text{size} \; \text{false} = s \; z \\
\text{size} \; \text{true} = s \; z \\
\text{size} \; z = s \; z \\
\text{size} \; (s \; X) = s \; (\text{size} \; X)
\]

\[
X :: \text{nat} \\
\text{size} \; (s \; X) :: \text{nat}
\]

\[
X :: A \\
s \; (\text{size} \; X) :: \text{nat}
\]

\((A, \text{nat})\) is more general than \((\text{nat}, \text{nat})\)
Type system: ill-typed example

\[
f : \texttt{bool} \rightarrow A
\]

\[
f \texttt{true} = z
\]

\[
f \texttt{false} = \texttt{true}
\]
Type system: ill-typed example

\[ f :: \text{bool} \rightarrow A \]
\[ f \text{ true} = z \]
\[ f \text{ false} = \text{true} \]

ill-typed because \text{nat} is not more general than \text{A}
Type system: ill-typed example

\[ f :: \text{bool} \rightarrow A \]

\[ f \text{ true} = z \]

\[ f \text{ false} = \text{true} \]

\[ f \text{ false} :: A \]

\[ \text{true} :: \text{bool} \]

ill-typed because \text{nat} is not more general than \text{A}
Type system: ill-typed example

\[ f :: \text{bool} \rightarrow A \]

- \( f \text{ true} = z \)
- \( f \text{ false} = \text{true} \)

Type preservation is not guaranteed:
- \( f \text{ true} :: \text{bool} \rightarrow z :: \text{nat} \)
- \( f \text{ false} :: [\text{int}] \rightarrow \text{true} :: \text{bool} \)
Translation using TIFs and type witnesses
Translation: intuition

- **Type-passing translation**: passes type information to overloaded functions
  - Replace each **overloaded function** by a **TIF**.
  - Each **rule** of an overloaded function in an **instance** is a **rule** of the corresponding **TIF**.
  - The TIF uses **type witnesses** to determine which rules to apply.
Type witnesses

- A way of representing types as values.
- Extend its data type with a new constructor.
- Examples:
  - `data nat = z | s nat | #nat`
  - `data bool = true | false | #bool`
  - `data [A] =
    nil | cons A (list A) | #list A`

  - `[bool] → #list #bool :: [bool]`
  - `[[nat]] → #list (#list #nat) :: [[nat]]`
The translation uses type information obtained by a previous type checking phase which decorates function symbols.

\[ g \ X = \text{eq} \ X \ [\text{true}] \]

\[ g :: \text{[bool]} \rightarrow \text{bool} \quad X = \]

\[ \text{eq} :: \langle \text{eq} \ [\text{bool}]) \Rightarrow \text{[bool]} \rightarrow \text{[bool]} \rightarrow \text{bool} \quad X \ [\text{true}] \]
Translation: classes

- Each **member function** generates a **TIF** that accepts a **type witness**.

```haskell
class arb A where
    arb :: A -> A
```
Each **rule** of a **member function** generates a **rule of the TIF** that accepts the corresponding **type witness**.

```
instance arb bool where
   arb = true
   arb = false
```

```
arb #bool = true
arb #bool = false
```
Type witnesses are passed to overloaded functions (which will have a type decoration with a class context).

\[
\begin{align*}
arbL2 & :\ arb\ A \Rightarrow [A] \\
arbL2 & ::\langle arb\ A\rangle \Rightarrow A \rightarrow [A] = \\
& [arb::\langle arb\ A\rangle \Rightarrow A,\ arb::\langle arb\ A\rangle \Rightarrow A]
\end{align*}
\]

\[
\begin{align*}
arbL2 & :\ A \rightarrow [A] \\
arbL2 & \text{ WitnessA} = \\
& [arb\ \text{WitnessA, arb\ WitnessA}]
\end{align*}
\]
Advantages of the translation
Adequacy to call-time choice

> arbL2::[bool]
  > arbL2::<arb bool> => [bool]
  arbL2 #bool → [arb #bool, arb #bool]
  → [true, arb #bool]
  → [true, true] 😊

> arbL2::[bool]
  > arbL2::<arb bool> => [bool]
  arbL2 #bool → [arb #bool, arb #bool]
  → [false, arb #bool]
  → [false, true] 😞

> arbL2::[bool]
  > arbL2::<arb bool> => [bool]
  arbL2 #bool → [arb #bool, arb #bool]
  → [true, arb #bool]
  → [true, false] 😊

Missing answers recovered
Tests: fragments of real programs which use type classes `eq`, `ord` and `num`. Some adapted from `nobench` suite for Haskell.

- Speedup measured in the Toy system in the evaluation of 100 random expressions.

<table>
<thead>
<tr>
<th>Program</th>
<th>Speedup (Time dict / Time TIF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eqlist</td>
<td>1.6414</td>
</tr>
<tr>
<td>fib</td>
<td>2.3063</td>
</tr>
<tr>
<td>galeprimes</td>
<td>1.4885</td>
</tr>
<tr>
<td>memberord</td>
<td>2.2802</td>
</tr>
<tr>
<td>mergesort</td>
<td><strong>1.0476</strong></td>
</tr>
<tr>
<td>permutsort</td>
<td>1.7186</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.0743</td>
</tr>
</tbody>
</table>
Optimizations

- Known optimizations for dictionaries
  [Lennart Augustsson: Implementing Haskell overloading (FPCA '93)]

- There are also optimizations for the new translation:
  - Specialized versions from instances

```haskell
instance arb bool where
  arb = true
  arb = false
```

Instead of

```haskell
f :: bool = not :: bool -> bool
arb :: <arb bool> => bool
```

```haskell
f = not arb_bool
```
Optimizations

- The speedup decreases in general but is still favorable even considering optimizations in both translations:

<table>
<thead>
<tr>
<th>Program</th>
<th>Speedup (Time dict opt / Time TIF opt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eqlist</td>
<td>1.3627</td>
</tr>
<tr>
<td>fib</td>
<td>2.3777</td>
</tr>
<tr>
<td>galeprimes</td>
<td>1.0016</td>
</tr>
<tr>
<td>memberord</td>
<td>2.2386</td>
</tr>
<tr>
<td>mergesort</td>
<td>1.0453</td>
</tr>
<tr>
<td>permutsort</td>
<td>1.7259</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.0005</td>
</tr>
</tbody>
</table>
Conclusions and future work
Conclusions

- Type-passing translation of type classes for FLP relying on a new liberal type system.
- Adequate to the call-time choice semantics of FLP.
- Performs faster or equal than dictionaries even when optimizations are considered.
- Resulting programs are simpler.
- Supports easily multiple modules and separate compilation.
Future work

- **Implement** and integrate into the Toy system.
- Once integrated, **test** the **efficiency** results **automatically** with a larger set of problems.
- Study other **extensions of type classes** (multi-parameter type classes, constructor classes) and how they fit in the type-passing translation.
- Study **more optimizations** for the new TIF translation.
Thanks!