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

Functional Logic Programming (I)

- **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

Functional Logic Programming (II)

- **Call-time choice semantics**: all copies of a non-determinic expression created during reduction must be shared.

```
coin = 0
coin = 1
dup X = (X,X)
```

> dup **coin** → (**coin**, **coin**) → (**0**,**0**)
> dup **coin** → (**coin**, **coin**) → (**1**,**1**)

(0,1) and **(1,0)** are not values of *dup 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.**

Our running example

```
class arb A where
  arb :: A

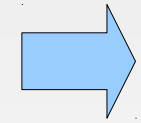
instance arb bool where
  arb = true
  arb = false

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.

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



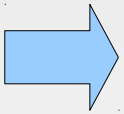
```
data dictArb A = dArb A

arb :: dictArb A -> A
arb (dArb Farb) = Farb
```

Translation with dictionaries (II): instances

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

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



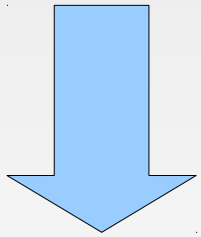
```
arbBool :: bool
arbBool = true
arbBool = false

dictArbBool :: dictArb bool
dictArbBool = dArb arbBool
```

Translation with dictionaries (III): functions

- **Overloaded functions** are transformed to **accept dictionaries** as arguments.

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



```
arbL2 :: dictArb A -> [A]
arbL2 Darb = [arb Darb, arb Darb]
```

Bad interation 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)]

```
> 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**.
  - Posibilities for **generic programming**: generic functions, type-indexed functions.

The heart of our type system

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

```
size :: A -> nat
size false = s z
size true = s z
size z = s z
size (s X) = s (size X)
```

Type system: example

```
size :: A -> nat
size false = s z
size true = s z
size _ = s z
size (X) = s (X)
```

size false :: nat

s z :: nat

Type system: example

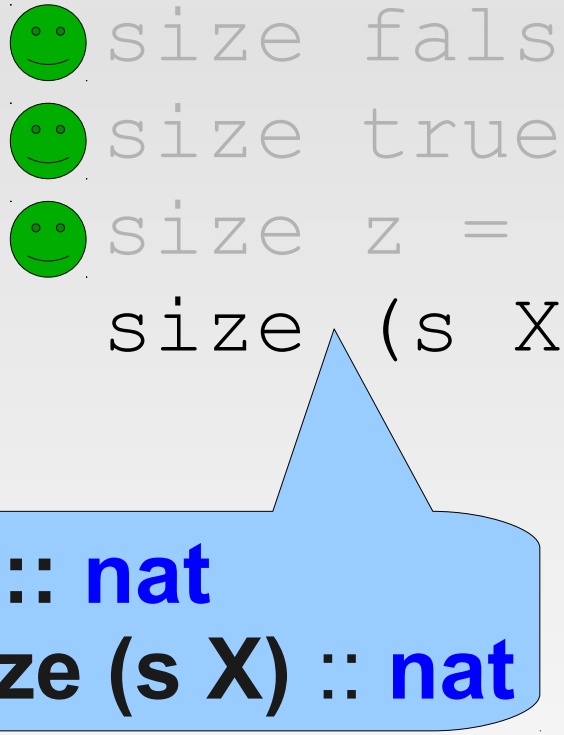
```
size :: A -> nat
size false = s z
size true = s z
size z = s z
size (s X) = s (size X)
```

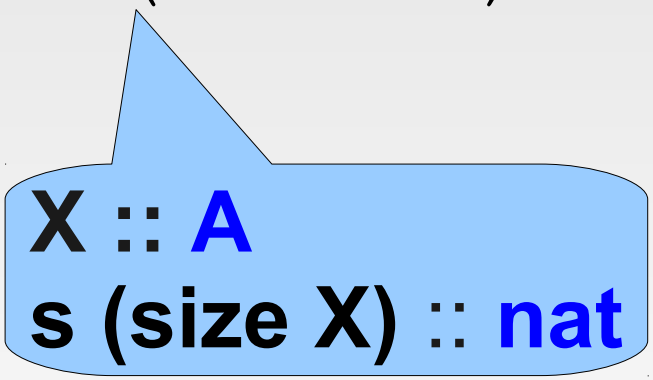
size true :: nat

s z :: nat

Type system: example

```
size :: A -> nat
size false = s z
size true = s z
size z = s z
size (s X) = s (size X)
```





(A, nat) is more general than **(nat, nat)**

Type system: ill-typed example

```
f :: bool -> A
f true = z
f false = true
```

Type system: ill-typed example

```
f :: bool -> A
f true = z
f false = true
```

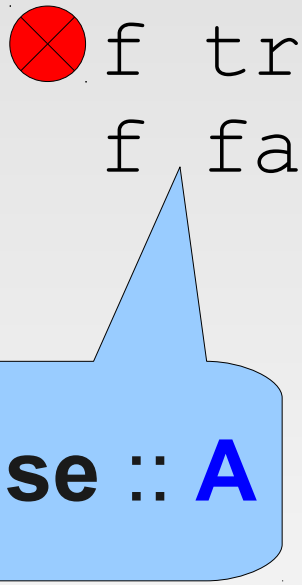
f true :: A

z :: nat

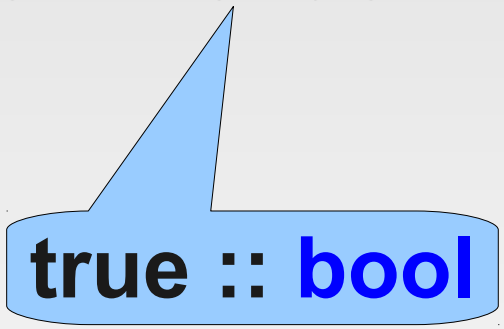
ill-typed because **nat** is not more general than **A**

Type system: ill-typed example

```
f :: bool -> A
f true = z
f false = true
```



f false :: A

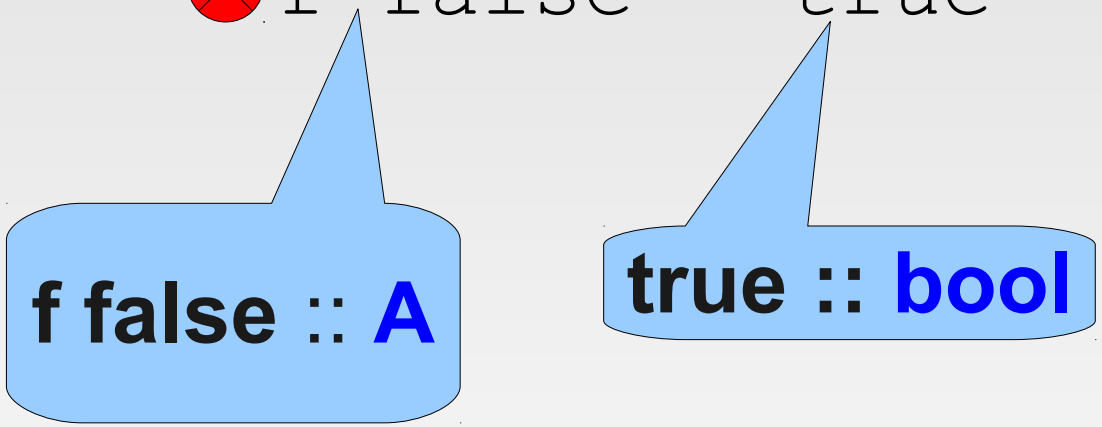


true :: bool

ill-typed because **nat** is not more general than **A**

Type system: ill-typed example

```
f :: bool -> A
f true = z
f false = true
```



Type preservation is not guaranteed:

`f true` **:: bool** → `z` **:: nat**

`f false` **:: [int]** → `true` **:: 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.

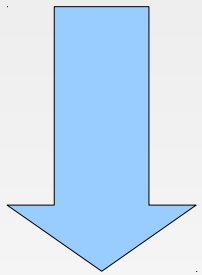
Type witnesses

- 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]]`**
    ```

Translation: decorations

- The translation uses type information obtained by a previous type checking phase which **decorates function symbols**.

```
g X = eq X [true]
```

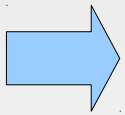


```
g:: [bool] → bool X =
eq::<eq [bool]> => [bool] → [bool] → bool X [true]
```

Translation: classes

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

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

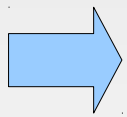


```
arb :: A -> A
```

Translation: instances

- 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
```

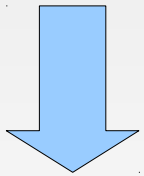


```
arb #bool = true
arb #bool = false
```

Translation: functions

- **Type witnesses** are passed to **overloaded functions** (which will have a type decoration with a class context).

```
arbL2 :: arb A => [A]
arbL2::<arb A> => A → [A] =
   [arb::<arb A> => A, arb::<arb A> => A]
```



```
arbL2 :: A -> [A]
arbL2 WitnessA =
  [arb WitnessA, arb WitnessA]
```

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

Speedup

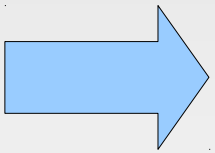
- **Tests**: fragments of real programs which use type classes `eq`, `ord` and `num`. Some adapted from *nobench* suite for Haskell.
- Speedup measured in the Toy system in the evaluation of 100 random expressions.

Program	Speedup (Time dict / Time TIF)
eqlist	1.6414
fib	**2.3063**
galeprimes	1.4885
memberord	2.2802
mergesort	**1.0476**
permutsort	1.7186
quicksort	1.0743

Optimizations

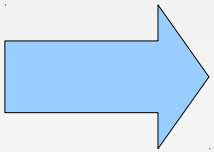
- Known optimizations for dictionaries
  [Lennart Augustsson: Implementing Haskell overloading (FPCA '93)]
- There are also optimizations for the new translation:
  - Specialized versions from instances

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



```
arb_bool = true
arb_bool = false
```

```
f::bool = not::bool → bool
          arb::<arb bool> => bool
```



```
f = not arb_bool
```

Instead of

```
f = not (arb #bool)
```

Optimizations

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

Program	Speedup (Time dict opt / Time TIF opt)
eqlist	1.3627
fib	**2.3777**
galeprimes	1.0016
memberord	2.2386
mergesort	1.0453
permutsort	1.7259
quicksort	**1.0005**

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!