Transparent Function Types:
Clearing up Opacity

Enrique Martín-Martín and
Juan Rodríguez-Hortalá
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

14th International Symposium on Principles and Practice of
Declarative Programming (PPDP 2012)

Leuven, Belgium. September 21st, 2012
1. Context: functional-logic programming
2. Motivation of the paper: opacity problems
3. Proposed solution
4. Approach 1: complete decorations
5. Approach 2: variables in decorations
6. Conclusions and future work
Outline

1. Context: functional-logic programming
2. Motivation of the paper: opacity problems
3. Proposed solution
4. Approach 1: complete decorations
5. Approach 2: variables in decorations
6. Conclusions and future work
Functional-logic programming

**Logic programming**
- Non-deterministic search
- Logic variables

**Functional programming**
- Higher-order functions
- Damas-Milner polymorphism
- Lazy evaluation

**Constraint programming**

- **Systems:** Curry (PAKCS, Münster, KICS), **Toy**.
This paper focuses on Toy, which is based on the Higher-Order Constructor Based Rewriting Logic (HO-CRWL) [González-Moreno et al., 2001]

The HO-CRWL semantics supports HO patterns, which are partial applications of constructor and function symbols.

- HO patterns are values: they cannot be evaluated
- HO patterns can appear in left-hand sides of rules
- They allow to distinguish extensionally equal expressions
- Examples: id, not, and false, map id
Higher-order patterns (II)

- Boolean circuits [González-Moreno et al., 2001]

```
alias circuit = bool → bool → bool

x1, x2 :: circuit
x1 X Y = X     x2 X Y = Y

notGate :: circuit → bool → bool → bool
notGate C X Y = not (C X Y)

size :: circuit → nat
size x1 = zero
size x2 = zero
size (notGate C) = succ (size C)
```

- \( x_1 \) and \( \text{notGate} \ (\text{notGate} \ x_1) \) are extensionally equal (same behavior for the same inputs), however they are distinguished by the function size
Equality in HO-CRWL

- In HO-CRWL the notion of equality (==) corresponds to joinability:
  
  \[ \text{two expressions } e \text{ and } e' \text{ are equal } (e == e') \text{ iff they can be reduced to the same value} \]

- Note that \(x1\) and notGate (notGate \(x1\)) are not joinable because they are different HO patterns.

- Equality in FLP systems is a primitive because it cannot be defined by rules. It proceeds structurally:

  \[ c \ e_1 \ldots e_n == c \ e'_1 \ldots e'_n \rightarrow e_1 == e'_1, \ldots, e_n == e'_n \]
Most FLP systems adopt the Damas-Milner type system directly:

- Inherited from Functional Programming (Haskell, ML)
- Well established
- Good properties: principal types, inference, ...

However, there are some opacity situations involving HO patterns and equality that are not properly handled by this type system.
Outline

1. Context: functional-logic programming
2. Motivation of the paper: opacity problems
3. Proposed solution
4. Approach 1: complete decorations
5. Approach 2: variables in decorations
6. Conclusions and future work
Polymorphic casting:

\[
snd :: A \rightarrow B \rightarrow B \\
snd X Y = Y
\]

\[
unpack :: (B \rightarrow B) \rightarrow A \\
unpack (snd X) = X
\]

\[
cast :: A \rightarrow B \\
cast X = unpack (snd X)
\]

- not (cast zero) is **well-typed** (it has type \texttt{bool}) because of the type of cast
- However, the evaluation does not preserve types: not (cast zero) $\rightarrow$ not (unpack (snd zero)) $\rightarrow$ not zero

ill-typed!!
Where is the problem?

The type of the HO pattern \( \text{snd} \ X \ (B \to B) \) does not reflect the type of its argument \( X \ (A) \). The variable \( X \) in opaque in the HO pattern \( \text{snd} \ X \).

The \text{cast} function is an evil identity function: it returns the same element \( X \), but with any type.
Opaque decomposition:

\[
\begin{align*}
\text{snd} &: \ A \to B \to B \\
\text{snd} \ X \ Y &= Y \\
(==) &: \ A \to A \to \text{bool} \\
% \text{primitive function}
\end{align*}
\]

The expression \((\text{snd} \ \text{true}) == (\text{snd} \ \text{zero})\) is **well-typed**, since both sides can have the same type:

- \((\text{snd} \ \text{true}) :: \ \text{bool} \to \text{bool}\)
- \((\text{snd} \ \text{zero}) :: \ \text{bool} \to \text{bool}\)

To resolve the equality a **decomposition step** is performed:

\((\text{snd} \ \text{true}) == (\text{snd} \ \text{zero}) \to \text{true} == \text{zero}\)

The resulting expression \text{true} == \text{zero} is **ill-typed**
Opacity problems (II)

- Where is the problem?

\[
\text{snd :: } A \to B \to B \\
\text{snd \(X\ Y\ =\ Y\) \% primitive function}
\]

- As before, the type of \(\text{snd \(e\)}\) does not contain any information about the type of \(e\):
  - \((\text{snd \(true\)})\ ::\ \text{bool} \to \text{bool}\)
  - \((\text{snd \(zero\)})\ ::\ \text{bool} \to \text{bool}\)

- Therefore a decomposition step of expressions of the same type can produce a comparison of expression of different types
1. Context: functional-logic programming
2. Motivation of the paper: opacity problems
3. Proposed solution
4. Approach 1: complete decorations
5. Approach 2: variables in decorations
6. Conclusions and future work
**Proposed solution**

- **Problem**: HO patterns containing function symbols generate opacity

- **Solution**: use a Damas-Milner type system but **enrich** the type information of **function types** so that partial applications contain the types of all the previous arguments. We place that extra information as **decorations** in the **arrows** of the functional type.

**Example:**

\[
\text{snd} :: A \rightarrow \ (B \rightarrow (A)) \ B \\
\text{snd} \ X \ Y = Y
\]

Now the type of a HO pattern reflects the type of all its components:

\[
\text{snd} \ true :: \text{bool} \rightarrow \ (\text{bool}) \ \text{bool} \\
\text{snd} \ zero :: \text{bool} \rightarrow \ (\text{nat}) \ \text{bool}
\]
Proposed solution

- Decorations solve polymorphic casting

\[
\begin{align*}
\text{snd} &:: A \rightarrow (\_ \_A) B \\
\text{snd} X Y &= Y \\
\text{unpack} &:: (B \rightarrow (\_ \_A) B) \rightarrow (\_ \_A) A \\
\text{unpack} (\text{snd} X) &= X \\
\text{cast} &:: A \rightarrow (\_ \_A) A \\
\text{cast} X &= \text{unpack} (\text{snd} X)
\end{align*}
\]

- \text{unpack} returns an element of the same type that is packed in the argument
- The only valid type for \text{cast} is \( A \rightarrow (\_ \_A) A \), as expected for an identity function
- Now the expression \( \text{not} \ (\text{cast} \ 0) \) is simply \textit{ill-typed}
Proposed solution

- Decorations solve opaque decomposition

\[
\begin{align*}
\text{snd :: } & A \rightarrow (B \rightarrow (A) B) \\
\text{snd } X Y &= Y \quad \% \text{ primitive function}
\end{align*}
\]

- Now the expression \((\text{snd } \text{true}) \equiv (\text{snd } \text{zero})\) is **ill-typed**: 
  - \(\text{(snd true)} :: \text{bool} \rightarrow (\text{bool})\) \text{bool}
  - \(\text{(snd zero)} :: \text{bool} \rightarrow (\text{nat})\) \text{bool}

- Then \((\text{snd } \text{true}) \equiv (\text{snd } \text{zero})\) is **rejected** instead of producing ill-typed decomposition steps
1. Context: functional-logic programming
2. Motivation of the paper: opacity problems
3. Proposed solution
4. **Approach 1:** complete decorations
5. Approach 2: variables in decorations
6. Conclusions and future work
Approach 1: full decorations

- **Idea**: decorations in arrows contain the complete types of all the previous arguments.

- **Examples**:
  - `snd :: A → () B → (A) B`
    
    ```latex
    \text{snd } X \ Y = Y
    ```
  - `and :: bool → () bool → (bool) bool`
    
    ```latex
    \text{and } true \ Y = Y
    \text{and } false \ Y = false
    ```
  - `f :: bool → () nat → (bool) A → (bool,nat) A`
    
    ```latex
    \text{f } true \ zero \ X = X
    ```
Approach 1: full decorations

**Idea:** decorations in arrows contain the complete types of all the previous arguments.

**Examples:**

- \( \text{snd} :: A \rightarrow (\_\_\_) B \rightarrow (A) B \)
  \( \text{snd} \ X \ Y = Y \)

- \( \text{and} :: \text{bool} \rightarrow (\_\_\_) \text{bool} \rightarrow (\text{bool}) \text{bool} \)
  \( \text{and} \ \text{true} \ Y = Y \)
  \( \text{and} \ \text{false} \ Y = \text{false} \)

- \( \text{f} :: \text{bool} \rightarrow (\_\_\_) \text{nat} \rightarrow (\text{bool}) A \rightarrow (\text{bool,nat}) A \)
  \( \text{f} \ \text{true} \ \text{zero} \ X = X \)
Idea: decorations in arrows contain the complete types of all the previous arguments.

Examples:

- \( \text{snd} :: A \to (B \to (A) B) \)
  \[ \text{snd } X \ Y = Y \]

- \( \text{and} :: \text{bool} \to (\text{bool} \to (\text{bool}) \text{bool}) \)
  \[ \text{and } \text{true} \ Y = Y \] \[ \text{and } \text{false} \ Y = \text{false} \]

- \( \text{f} :: \text{bool} \to (\text{nat} \to (A \to (\text{bool}, \text{nat}) A) \)
  \[ \text{f } \text{true} \ \text{zero} \ X = X \]
Approach 1: type system

- **Good properties:**
  - Closure under type substitutions
  - Type preservation during evaluation
  - Sound and complete type inference (based on unification)

- **Drawback:**
  - Some interesting (and harmless) programs using HO patterns are considered as ill-typed
Approach 1: drawback

- Boolean circuits [González-Moreno et al., 2001]

```
alias circuit = bool → () bool → (bool) bool

x1, x2 :: circuit
x1 X Y = X    x2 X Y = Y

notGate :: circuit → () bool → (circuit) bool → (circuit, bool) bool
notGate C X Y = not (C X Y)

size :: ??
size x1 = zero
size x2 = zero
size (notGate C) = succ (size C)
```

- x1 and x2 have type circuit (bool → () bool → (bool) bool)
- (notGate C) has type bool → (circuit) bool → (circuit, bool) bool
- Incompatible types in rules size cannot have any valid type!!
Outline

1. Context: functional-logic programming
2. Motivation of the paper: opacity problems
3. Proposed solution
4. Approach 1: complete decorations
5. Approach 2: variables in decorations
6. Conclusions and future work
Approach 2: variables in decorations

- **Idea:** decorations in arrows contain the **type variables** of the types of the previous arguments.

- **Examples:**
  - \( \text{snd :: } A \to \text{B} \to \text{B} \)
    \[
    \text{snd X Y = Y}
    \]
  - \( \text{and :: bool} \to \text{bool} \to \text{bool} \)
    \[
    \text{and true Y = Y}
    \text{and false Y = false}
    \]
  - \( \text{f :: bool} \to \text{nat} \to A \to A \)
    \[
    \text{f true zero X = X}
    \]
**Approach 2: variables in decorations**

- **Idea**: decorations in arrows contain the **type variables** of the types of the previous arguments.

- **Examples**:
  - \( \text{snd :: } A \to B \to (A)B \)
    \[
    \text{snd } X \; Y = Y
    \]
  - \( \text{and :: } \text{bool} \to \text{bool} \to \text{bool} \)
    \[
    \text{and } \text{true} \; Y = Y
    \]
    \[
    \text{and } \text{false} \; Y = \text{false}
    \]
  - \( \text{f :: } \text{bool} \to \text{nat} \to A \to A \)
    \[
    \text{f } \text{true} \; \text{zero} \; X = X
    \]
Approach 2: variables in decorations

- **Idea**: decorations in arrows contain the type variables of the types of the previous arguments.

- **Examples**:
  - \( \text{snd} :: A \rightarrow (B \rightarrow A) B \)
    \( \text{snd} \ X \ Y = Y \)
  - \( \text{and} :: \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool} \)
    \( \text{and} \ \text{true} \ Y = Y \)
    \( \text{and} \ \text{false} \ Y = \text{false} \)
  - \( \text{f} :: \text{bool} \rightarrow (\text{nat} \rightarrow A) \rightarrow (A \rightarrow A) \)
    \( \text{f} \ \text{true} \ \text{zero} \ \text{X} = \text{X} \)

  \[ \text{var}(	ext{bool}) = \{\} \]
Idea: decorations in arrows contain the type variables of the types of the previous arguments.

Examples:

- \( \text{snd} :: A \rightarrow B \rightarrow (A)B \)
  \[ \text{snd} \ X \ Y = Y \]
- \( \text{and} :: \text{bool} \rightarrow \text{bool} \rightarrow (\text{bool}) \)
  \[ \text{and} \ \text{true} \ Y = Y \]
  \[ \text{and} \ \text{false} \ Y = \text{false} \]
- \( \text{f} :: \text{bool} \rightarrow \text{nat} \rightarrow A \rightarrow A \)
  \[ \text{f} \ \text{true} \ \text{zero} \ X = X \]

\( \text{var(bool)} = \{\} \)

\( \text{var(bool,nat)} = \{\} \)
Approach 2: variables in decorations

- **Idea**: decorations in arrows contain the **type variables** of the types of the previous arguments.

- **Examples**:
  - \[\text{snd} :: A \to B \to (A) B\]
    \[
    \text{snd} \ X \ Y = Y
    \]
  - \[\text{and} :: \text{bool} \to \text{bool} \to \text{bool}\]
    \[
    \text{and} \ \text{true} \ Y = Y
    \]
    \[
    \text{and} \ \text{false} \ Y = \text{false}
    \]
  - \[\text{f} :: \text{bool} \to \text{nat} \to A \to A\]
    \[
    \text{f} \ \text{true} \ \text{zero} \ X = X
    \]

- The **type variables** of the previous arguments is the only information we need to **avoid opacity**
Approach 2: boolean circuits

- Original boolean circuits [González-Moreno et al., 2001]

```plaintext
alias circuit = bool → () bool → () bool

x1, x2 :: circuit
x1 X Y = X  x2 X Y = Y

notGate :: circuit → () bool → () bool → () bool
notGate C X Y = not (C X Y)

size :: circuit → nat
size x1 = zero
size x2 = zero
size (notGate C) = succ (size C)
```

- $x_1$ and $x_2$ have type circuit ($\text{bool} \rightarrow () \text{bool} \rightarrow () \text{bool}$)
- $(\text{notGate } C)$ has type $\text{bool} \rightarrow () \text{bool} \rightarrow () \text{bool}$
- The size function is now well-typed
Approach 2: boolean circuits

- **Original boolean circuits** [González-Moreno et al., 2001]

```
alias circuit = bool -> () bool -> () bool

x1, x2 :: circuit
x1 X Y = X  x2 X Y = Y

notGate :: circuit -> ()
notGate C X Y = not (C X Y)

size :: circuit -> nat
size x1 = zero
size x2 = zero
size (notGate C) = succ (size C)
```

- x1 and x2 have type `circuit (bool -> () bool -> () bool)`
- `(notGate C)` has also type `circuit (bool -> () bool -> () bool)`
- The `size` function is now well-typed
Approach 2: type system

- It must be done with care: if you include directly only the type variables you obtain a type system that lacks **closure under type substitutions**
  - This property is **essential** to develop a **type inference** algorithm based on unification
- **Mixed approach**: derive types with complete type decorations (as in approach #1) but use a flattening process that leave only type variables in arrow decorations
  - This mixed approach guarantees **type preservation** and recovers **closure under type substitutions**, thus allowing a sound and complete **type inference**
1. Context: functional-logic programming
2. Motivation of the paper: opacity problems
3. Proposed solution
4. Approach 1: complete decorations
5. Approach 2: variables in decorations
6. Conclusions and future work
Conclusions

- We extend the Damas-Milner type system to avoid unintended opacity caused by HO patterns in FLP systems.
- The main idea is to decorate the arrows of functional types with type information about the previous arguments.
  - We have tried different approaches, depending on the information placed in the decorations.
- The proposed type system solves the well-known FLP problems of polymorphic casting and opaque decomposition.
- To the best of our knowledge, it is the first type systems for FLP that handles correctly polymorphic casting as an identity function.
  Other type systems can only forbid it [González-Moreno et al.,2001], [López-Fraguas et al.,2009], [López-Fraguas et al.,2010]
Future work

- The integration of this type system into the Toy system would be very valuable.
  - We could test the type system automatically with a big set of programs.
  - As arrow decorations are a novelty, it would help us to appreciate if they fit easily in programmers intuition

- In this paper we only consider rewriting reductions in our type preservation result. It would be very interesting to study how it works when considering narrowing and logic variables
Thank you!
Approach 1: type system

- **Well-typed rules:**
  
  A rule $f \ t_1 \ldots \ t_n = e$ is well-typed if $\mathcal{A} \vdash \lambda t_1 \ldots \lambda t_n. e : \tau$
  
  and $\tau$ is a variant of $\mathcal{A}(f)$

  A rule is well-typed if the type of the associated $\lambda$-abstraction matches the type of the function

- **Well-typed programs:**

  A program is well-typed if all its rules are well-typed

- Then we need to focus on how to derive **types** for $\lambda$-abstractions
Approach 1: type system

- Similar to the rule for λ-abstractions in the Damas-Milner type system

\[ \phi \vdash \{X_n : \tau_n\} \vdash t : \tau \]
\[ \phi \vdash \{X_n : \tau_n\} \vdash e : \tau \]
\[ (\Lambda) \quad A \vdash \lambda \tau. e : \tau \rightarrow (\cdot) \quad \text{anArgs}(k, \tau, \tau') \]

where \( \overline{X_n} = \text{var}(t) \) and \( \lambda depth(e) = k \)

- \( \lambda depth \) counts the number of arguments in a λ-abstraction

\[ \lambda depth(s) = 0 \quad \lambda depth(\text{let } X = e_1 \text{ in } e_2) = 0 \]
\[ \lambda depth(e_1 \ e_2) = 0 \quad \lambda depth(\lambda t. e) = 1 + \lambda depth(e) \]

- \( \text{anArgs}(n, \tau, \tau') \) adds \( \tau \) as a decoration in the arrows of \( \tau' \) up to some depth \( n \)

\[ \text{anArgs}(0, \tau, \tau') = \tau' \]
\[ \text{anArgs}(n, \tau, \tau_1 \rightarrow_{\tau'} \tau_2) = \tau_1 \rightarrow_{(\tau, \tau')} \text{anArgs}(n - 1, \tau, \tau_2) \text{ where } n > 0 \]
Existential types

- An adaptation of existential constructors [Läufer & Odersky, 1994] to HO patterns would solve the polymorphic casting. However, it would be rejected and not treated as an identity.
  
  \[
  \text{unpack} \ (\text{snd } X) = X
  \]
  The rule is rejected because \textit{in this use} \( X \) is existentially quantified, and appears in the returned type.

- However, opaque decomposition would happen:
  
  \[
  (\text{snd true}) == (\text{snd zero})
  \]
  is a well-typed expression considering an adaptation of existential constructors, so it will be reduced to
  
  \[
  \text{true} == \text{zero}
  \]
Extensionality in FLP

- [López-Fraguas et al., 2008]

\[
\begin{align*}
g(X) &= \text{zero} & f &= g & f'(X) &= f(X) \\
h(X) &= \text{succ zero} & f &= h
\end{align*}
\]

\[
\begin{align*}
f\text{double } F \ G \ X &= (F \ X) + (G \ X) \\
f\text{double } F &= f\text{add } F \ F
\end{align*}
\]

- The functions \( f \) and \( f' \) are extensionally equal
- However they can be distinguished in the context \( f\text{double } _0 \)
  - \((f\text{double } f \ 0) \rightarrow^* \text{zero, succ (succ zero)}\)
  - \((f\text{double } f' \ 0) \rightarrow^* \text{zero, succ zero, succ (succ zero)}\)
- Reason: combination of HO, non-determinism, and the call-time choice parameter passing semantics