Restoring Confluence for Functional Dependencies

Tom Schrijvers    Martin Sulzmann

Catholic University of Leuven, Belgium

ITU, Denkmark

Overview

Problem Overview
New TF-Inspired Encoding
The Propagation Encoding
Comparison
Conclusion
Overview

1. Problem Overview
2. New TF-Inspired Encoding
3. The Propagation Encoding
4. Comparison
5. Conclusion
Multi-parameter type class with FD

```haskell
class Collects c e | c -> e where
    insert :: e -> ce -> ce
    delete :: e -> ce -> ce
    member :: e -> ce -> Bool
    empty :: ce
```

- FD models invariant / constraint on instances
- only one possible element type for any collection type
FDs allow more precise types, e.g.

```
inserttwo x y c = insert x (insert y c)
```

What’s the signature? Normally:

```
(Collects c e1, Collects c e2) => e1 -> e2 -> c -> c
```

With FD: e1 and e2 same type: “improvement” e1 = e2

```
Collects c e => e -> e -> c -> c
```
What is the FD type checking algorithm? What are its properties?

- Why do Hugs and GHC behave differently?
- Does the type checker terminate?
- Is the type checker sound & complete?
- What instances am I allowed to write?

Hugs/GHC are black boxes: type checking FDs implemented/not formalized
Formalization as constraint rewriting [Sulzmann et al. 2007]:
Type class and instances yield three types of rules:

--- class Collects c e / c -> e
Collects c e1, Collects c e2 ==> e1 = e2 (Fd)

--- instance Collects [e] e
Collects [e] e <=> True (Inst)
Collects [e1] e2 <=> e1 = e2 (Imp)
Important properties:

- **Termination** of rewrite rules
- **Confluence** of rewrite rules: type checker is reliable
  each rewrite sequence yields the same result

\[
\text{Collects } [e] e_1, \text{ Collects } [e] e_2 \\
/ \ \\
\text{IMP} \times 2 / \ \ \ \\
\text{FD} (e_1 = e_2) \\
/ \ \\
\text{Collects } [e] e \\
/ \ \\
\text{Inst} \times 2 \ \\
\text{IMP, Inst} \\
/ \\
\text{True}
\]
Confluence holds under the Weak Coverage Condition *if the FDs are full.*

Our main issues:
- Why doesn’t it hold for non-full FDs?
- Can we achieve confluence for non-full FDs at all?
What are non-full FDs?

A non-full FD is the opposite of a full FD:

Definition (Full Functional Dependencies)
We say the functional dependency

\[ \text{class } TC \ a_1 \ldots \ a_n | a_{i_1}, \ldots, a_{i_k} \rightarrow a_{i_0} \]

for a type class TC is full iff

\[ k = n - 1. \]

E.g.

- \textbf{class Collects c e | c \rightarrow e} is full
- \textbf{class C a b c | a b \rightarrow c} is full
- \textbf{class C a b c | a \rightarrow c} is non-full
Are non-full FDs useful?

Yes! E.g. HackageDB contains several packages with non-full FDs

Non-full FD in parsec

```haskell
class (Monad m) => Stream s m t | s -> t where
  -- s: stream
  -- m: monad
  -- t: token

  uncons :: s -> m (Maybe (t,s))

instance (Monad m) => Stream [t] m t where ...

instance Stream FileCursor IO Char where ...
```
The FD-CHR encoding for a non-full FD:

```
class F a b c | a -> b
instance F a b Bool => F [a] [b] Bool

F a b1 c, F a b2 d ==> b1 = b2  \hspace{1cm} (Fd)
F [a] [b] Bool <= F a b Bool \hspace{1cm} (INST)
F [a] b c ==> b = [b1] \hspace{1cm} (IMP)
```
Non-confluence: two derivations yield different results

\[ F \ [a] \ [b] \ Bool, F \ [a] \ b2 \ d \]
\[ \overset{\text{(FD)}}{\longrightarrow} \]
\[ F \ [a] \ [b] \ Bool, F \ [a] \ [b] \ d, b2 = [b] \]
\[ \overset{\text{(INST)}}{\longrightarrow} \]
\[ F \ a \ b \ Bool, F \ [a] \ [b] \ d, b2 = [b] \]

\[ F \ [a] \ [b] \ Bool, F \ [a] \ b2 \ d \]
\[ \overset{\text{(INST)}}{\longrightarrow} \]
\[ F \ a \ b \ Bool, F \ [a] \ b2 \ d \]
\[ \overset{\text{(IMP)}}{\longrightarrow} \]
\[ F \ a \ b \ Bool, F \ [a] \ [c] \ d, b2 = [c] \]
Overview

1. Problem Overview
2. New TF-Inspired Encoding
3. The Propagation Encoding
4. Comparison
5. Conclusion
Type families: functions at the level of types.

```haskell
-- declaration
type family Elem c

-- instances
type instance Elem BitVector = Bit
type instance Elem [e] = e
type instance Elem (Trie e) = e
```

Represent a functional relation between types.
Idea: Express the FD with a TF!

```
-- class Collects c e / c -> e
class Elem c ~ e => Collects c e where ...
type family Elem c

instance Collects [e] e where ...
type instance Elem [e] = e
```

Equality constraint `Elem c ~ e` captures FD
Surprisingly: The TF encoding is confluent for non-full FDs.

See paper for details.
The TF-CHR Encoding

Encode TFs with FDs [IFL 2007]
⇒ alternative FD encoding of TF encoding
⇒ TF-CHR encoding

Step 1. Non-Full FD with TF

class TF a ~ b => F a b c

type family TF a

instance F a b Bool => F [a] [b] Bool
type instance F [a] = [F a]
Encode TFs with FDs [IFL 2007]
⇒ alternative FD encoding of TF encoding
⇒ TF-CHR encoding

Step 2. TF as full FD

```
class FD a b => F a b c
class FD a b | a -> b

instance F a b Bool => F [a] [b] Bool
instance FD a b => FD [a] [b]
```

TF-CHR encoding: confluent!
Overview

1. Problem Overview
2. New TF-Inspired Encoding
3. The Propagation Encoding
4. Comparison
5. Conclusion
Based on suggestion by Claus Reinke.

- Always use \( \Rightarrow \) instead of \( \Leftarrow \)
- Confluence is trivial (no critical pairs)

\[
\text{class } F \ a \ b \ c \ | \ a \rightarrow b \\
\text{instance } F \ a \ b \ \text{Bool} \Rightarrow F \ [a] \ [b] \ \text{Bool}
\]

\[
F \ a \ b1 \ c, \ F \ a \ b2 \ d \Rightarrow b1 = b2 \quad \text{(Fd)}
\]

\[
F \ [a] \ [b] \ \text{Bool} \Rightarrow F \ a \ b \ \text{Bool} \quad \text{(Inst)}
\]

\[
F \ [a] \ b \ c \Rightarrow b = [b1] \quad \text{(Imp)}
\]

See paper for details.
Properties of Interest

1. confluence (of non-full FDs)
2. type-preserving translation
3. performance
Comparison of Confluence

CONFLUENCE OF NON-FULL FDS

- non-confluent
  - FD-CHR encoding
- confluent
  - TF encoding
  - TF-CHR encoding
  - PROP-CHR encoding

Non-confluence is an artefact of the FD-CHR encoding!
Some FD improvements cannot be expressed in System F:

```haskell
class F a b | a -> b
instance F Int Int

f :: F Int a => a -> Int
   -- F Int a ==> a = Int
f = x -> x + 1
```

yields in System F:

```
Λa.λd:FDict Int a.λx:a. x + 1
```

**Ill-typed!**: `x` cannot be of type `a` and `Int` at the same time
System $F_C$ extends System $F$ with type coercions.

- $a \sim \text{Int}$: coercion kind (equality of types)
- $\text{co} : a \sim \text{Int}$: coercion variable (evidence of equality)
- $x \downarrow \text{co}$: cast expression (based on evidence)

Evidence combinators:

- $\text{sym} \text{co} : \text{Int} \sim a$: symmetry of equality
- $\text{sym} \text{co} \circ \text{co} : \text{Int} \sim \text{Int}$: transitivity of equality
Evidence for type functions:

```haskell
class (TF x ~ y) => F x y

type family TF x

instance F Int Int

type instance TF Int = Int

-- Dictionary type for class F x y
data FDict x y = FDict (TF x ~ y)

-- Dictionary constructor for F Int Int
fdictIntInt = FDict coInt

-- Coercion axiom for F Int = Int
coInt :: F Int ~ Int
```
TF encoding yields well-typed System $F_C$ code:

\[
\Lambda a. \lambda d: \text{FDict} \text{ Int a.} \lambda x:a. \text{case d of}
\begin{array}{l}
\{ \text{FDict co:(FD Int } \sim a) \rightarrow \\
(x \triangleright \text{sym co } \circ \text{coInt}) + 1 \}
\end{array}
\]

See [Schrijvers et al. IFL’07; ICFP’08 submission] for type checking/evidence inference.

Only works for type functions, not for CHR encodings!
Comparison of Type Preservation

TYPE PRESERVATION POSSIBLE

- type preservation in System $F_C$
  - TF encoding
- no type preservation
  - FD-CHR encoding
  - TF-CHR encoding
  - PROP-CHR encoding

Type preservation depends on the core language!
Memoing possible
  
  - in GHC (close to FD-CHR)
  - PROP-CHR

Unexplored
  
  - TF
  - TF-CHR

No conclusive evidence of better performance yet.
Overview

1. Problem Overview
2. New TF-Inspired Encoding
3. The Propagation Encoding
4. Comparison
5. Conclusion

T. Schrijvers and M. Sulzmann

Restoring Confluence for Functional Dependencies
In conclusion:

- unexplored design space for FD type checking,
- non-confluence is due to the FD-CHR approach, and
- two new approaches don’t suffer from non-confluence.
- After 8 years we still don’t have a complete picture of FDs.
- Don’t think a language feature is bad because its implementations are.
Future Work

- Further study of FD encodings
- multi-range FDs: \( a \rightarrow b, \ a \rightarrow c \) vs. \( a \rightarrow b\ c \)
- System \( \text{F}_{\text{CHR}} \): type preserving translation of CHR
- Unified algorithm for type classes and type functions
Questions?