Fixing Idioms

A recursion primitive for Applicative DSLs

Dominique Devriese       Ilya Sergey
Dave Clarke              Frank Piessens
Functional DSLs

- Functional languages are a good host for elegant DSLs
- Shallow functional embeddings inherit desirable features: abstraction, types, reasoning.
- Missing: a *typed, functional representation of cyclic structures*?
- This problem is holding DSLs back, e.g. parser DSLs:
  - Why only parse? Why not analyse, visualise, debug?
  - Less optimisation than parser generators?
Representations of Cyclic Structures

- Mutable references, referential identity: imperative 😞
- Deep embeddings: not shallow 😞
- Reduce cyclic to infinite + laziness:
  - Makes recursion unobservable for DSL algorithms 😞
  - In other words: DSL restricted to least fixpoints 😞
- Previous work:
  - implicitly take fixpoint at top-level (like CFGs)
  - represent DSL terms as open recursive
  - no recursion inside term, modularity disadvantages: 😞
Add a fixpoint primitive \( \mu x \ldots x \ldots \) to DSL.
Shallow functional representation of binding? HOAS?
Correct version of HOAS: PHOAS or Finally Tagless
Applicative DSLs:

- good for DSLs representing computations with hidden effects or hidden inputs (e.g. parsers)
- contrary to **Monads**: still analysable (less power to user, more power to library)
- effect-value separation:
  - **Monad**: $(\gg=) :: m a \to (a \to m b) \to m b$
  - **Applicative**: $(\otimes) :: m (a \to b) \to m a \to m b$
- natural setting for effectful recursion (not **Monad**ic value recursion)
Different fixpoint primitives for different DSLs?

- Applicative DSLs differ from lambda calculi (e.g. Oliveira and Löh):
  - Add \( \text{pure} :: a \rightarrow p \ a. \)
  - Subtract \( \text{lam} :: (p \ a \rightarrow p \ b) \rightarrow p \ (a \rightarrow b). \)

Note: adding Lam in an Applicative DSL is not a solution, e.g. parsing.

- Observation: finally tagless fixpoint primitive not enough for advanced parser transformations!

- Need to specify and exploit value-effects-separation during transformation!

- Surprising: re-specify what already follows?
Contributions

- Fixpoint primitive \texttt{afix}:

\begin{verbatim}
\textbf{class} Applicative \( p \Rightarrow \) ApplicativeFix \( p \) \textbf{where}
\texttt{afix ::} \( (\forall q. \text{Applicative } q \Rightarrow (p \circ q) \ a \rightarrow (p \circ q) \ a) \rightarrow p \ a \)
\end{verbatim}

- Properties:
  - Rank-2 type specifies effect-values separation for \texttt{afix}'s argument
  - Axiom specifying fixpoint behaviour

- Practicality:
  - Reduce mutual recursion to simple (uses generic programming)
  - \texttt{alet}-notation: shallow syntactic sugar implemented in GHC

- Applications:
  - Left-recursion removal for \textit{Applicative} parser combinators
  - Analyse cyclicity in FRP model of circuits
A Closer Look

- Composing *Applicative* Functors: \((p \circ q)\)
- *afix*’s type
Composing Applicative Functors

```haskell
class Applicative p where
    pure :: a → p a
    (∗) :: p (a → b) → p a → p b

newtype (p ◦ q) a = Comp { comp :: p (q a) }

instance (Applicative p, Applicative q) ⇒
    Applicative (p ◦ q) where ...
```
**afix’s type**

```haskell
class Applicative p ⇒ ApplicativeFix p where
  afix :: (∀ q. Applicative q ⇒
    (p ◦ q) a → (p ◦ q) a) → p a
```

The type

```haskell
f :: ∀ q. Applicative q ⇒ (p ◦ q) a → (p ◦ q) a
```

specifies `Applicative` effects-values separation for `f` (see paper).

Crucial: a restricted equivalent of lambda...

```haskell
coapp :: Applicative p ⇒ (∀ q . Applicative q ⇒
    (p ◦ q) a → (p ◦ q) b) → p (a → b)
```
Practicality

- *nafix*: arity-generic version of *afix* for mutual recursion
- *alet*-notation: shallow syntactic sugar implemented in GHC

```
\begin{align*}
\text{alet} \ expr &= (+) \ $\ expr \ \&\ token \ '+' \ \&\ factor \\
& \quad \&\ factor \\
\text{factor} &= (*) \ $\ factor \ \&\ token \ '*' \ \&\ term \\
& \quad \&\ term \\
\text{term} &= token \ '(' \ \&\ expr \ \&\ token \ ')' \\
& \quad \&\ \text{decimal}
\end{align*}
```

in \ expr

Desugars into application of *nafix*.
Applications

- Test circuits for correct cyclicity (see paper).
- Left-recursion removal:

\[ \text{exprParse} :: \text{String} \rightarrow \text{Int} \]
\[ \text{exprParse} = \text{parseUU} (\text{transformPaull expr}) \]
\[ \text{testParse} = \text{exprParse} "1+7*3+(8*1+2*6)" \]
(Intuition behind need for coapp in left-recursion removal)

\[
\begin{align*}
\text{expr} :: \ldots & \Rightarrow p \text{ Int} \\
\text{expr} &= \text{afix } \$(\lambda s \rightarrow \text{digit} \odot (+) \$ s \odot \text{digit})
\end{align*}
\]

is transformed (essentially) into

\[
\begin{align*}
\text{expr} :: \ldots & \Rightarrow p \text{ Int} \\
\text{expr} &= \text{flip } (\$) \$ \text{digit} \odot \text{many \ exprD} \\
\text{exprD} :: \ldots & \Rightarrow p (\text{Int} \rightarrow \text{Int}) \\
\text{exprD} &= \text{flip } (+) \$ \text{digit}
\end{align*}
\]

To derive \(\text{exprD}\), we go from type

\[
(\forall q. \text{Applicative } q \Rightarrow (p \circ q) \text{ Int} \rightarrow (p \circ q) \text{ Int}) \text{ to } p (\text{Int} \rightarrow \text{Int}).
\]

This is \textit{coapp}!
Conclusion

- Shallow functional DSLs need shallow functional representation of recursion
- Applicative DSLs have special needs
- We show one suitable solution with
  - a new finally tagless primitive `afix` whose type enforces effects-values separation
  - support for mutual recursion using generically programmed `nafix`
  - shallow syntactic sugar through `alet` with implementation in GHC
  - applications to parsing and circuit design
- Read our paper if you want to know more!