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: ☹

Functional Representations of Cyclic Structures

- 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

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*: $(\circledast) :: m\ (a \to b) \to m\ a \to m\ b$
- 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 $\quad pure :: a \to p\ a$.
  - Subtract $\quad lam :: (p\ a \to p\ b) \to p\ (a \to b)$.

  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 *afix*:

$$\begin{array}{l} \textbf{class}\ \mathit{Applicative}\ p \Rightarrow \mathit{ApplicativeFix}\ p\ \textbf{where} \\ \quad \mathit{afix} :: (\forall\, q.\, \mathit{Applicative}\ q \Rightarrow \\ \qquad\qquad\qquad (p \circ q)\ a \to (p \circ q)\ a) \to p\ a \end{array}$$

- Properties:
  - Rank-2 type specifies effect-values separation for *afix*'s argument
  - Axiom specifying fixpoint behaviour
- Practicality:
  - Reduce mutual recursion to simple (uses generic programming)
  - **alet**-notation: shallow syntactic sugar implemented in GHC
- Applications:
  - Left-recursion removal for *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

$$
\begin{array}{l}
\textbf{class}\ \mathit{Applicative}\ p\ \textbf{where} \\
\quad \mathit{pure} :: a \to p\ a \\
\quad (\circledast)\ :: p\ (a \to b) \to p\ a \to p\ b \\
\textbf{newtype}\ (p \circ q)\ a = \mathit{Comp}\ \{\mathit{comp} :: p\ (q\ a)\} \\
\textbf{instance}\ (\mathit{Applicative}\ p, \mathit{Applicative}\ q) \Rightarrow \\
\qquad\qquad \mathit{Applicative}\ (p \circ q)\ \textbf{where}\ \ldots
\end{array}
$$

afix's type

$$\begin{array}{l}\textbf{class } \mathit{Applicative}\ p \Rightarrow \mathit{ApplicativeFix}\ p\ \textbf{where} \\ \quad \mathit{afix} :: (\forall\ q.\, \mathit{Applicative}\ q \Rightarrow \\ \qquad\qquad\qquad (p \circ q)\ a \to (p \circ q)\ a) \to p\ a\end{array}$$

The type

$$f :: \forall\ q.\, \mathit{Applicative}\ q \Rightarrow (p \circ q)\ a \to (p \circ q)\ a$$

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

Crucial: a restricted equivalent of lambda...

$$\begin{array}{l}\mathit{coapp} :: \mathit{Applicative}\ p \Rightarrow (\forall\ q\ .\ \mathit{Applicative}\ q \Rightarrow \\ \qquad\qquad\qquad (p \circ q)\ a \to (p \circ q)\ b) \to p\ (a \to b)\end{array}$$

Practicality

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

$$
\begin{array}{lll}
\textbf{alet}\ \mathit{expr} & = & (+) \circledS \mathit{expr} \circledast\!\!< \mathit{token}\ \texttt{'+'} \circledast \mathit{factor} \\
 & & \oplus\ \mathit{factor} \\
\quad \mathit{factor} & = & (*) \circledS \mathit{factor} \circledast\!\!< \mathit{token}\ \texttt{'*'} \circledast \mathit{term} \\
 & & \oplus\ \mathit{term} \\
\quad \mathit{term} & = & \mathit{token}\ \texttt{'('} \circledast\!\!> \mathit{expr} \circledast\!\!< \mathit{token}\ \texttt{')'} \\
 & & \oplus\ \mathit{decimal} \\
\textbf{in}\ \mathit{expr} & &
\end{array}
$$

Desugars into application of *nafix*.

Applications

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

$$
\begin{aligned}
&\mathit{exprParse} :: \mathit{String} \to \mathit{Int} \\
&\mathit{exprParse} = \mathit{parseUU}\ (\mathit{transformPaull}\ \mathit{expr}) \\
&\mathit{testParse} = \mathit{exprParse}\ \texttt{"1+7*3+(8*1+2*6)"}
\end{aligned}
$$

(Intuition behind need for coapp in left-recursion removal)

$$
\begin{aligned}
&\mathit{expr} :: \ldots \Rightarrow p\ \mathit{Int} \\
&\mathit{expr} = \mathit{afix}\ \$\ \lambda s \to \mathit{digit} \oplus (+) \circledS\ s \circledast \mathit{digit}
\end{aligned}
$$

is transformed (essentially) into

$$
\begin{aligned}
&\mathit{expr} &&:: \ldots \Rightarrow p\ \mathit{Int} \\
&\mathit{expr} &&= \mathit{flip}\ (\$) \circledS\ \mathit{digit} \circledast \mathit{many}\ \mathit{exprD} \\
&\mathit{exprD} &&:: \ldots \Rightarrow p\ (\mathit{Int} \to \mathit{Int}) \\
&\mathit{exprD} &&= \mathit{flip}\ (+) \circledS\ \mathit{digit}
\end{aligned}
$$

To derive *exprD*, we go from type
$(\forall\ q.\ \mathit{Applicative}\ q \Rightarrow (p \circ q)\ \mathit{Int} \to (p \circ q)\ \mathit{Int})$ to $p\ (\mathit{Int} \to \mathit{Int})$.
This is *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!