Fixing Idioms

A recursion primitive for applicative DSLs

Dominique Devriese Ilya Sergey Dave Clarke Frank Piessens
iMinds-DistriNet, KU Leuven
{firstname.lastname}@cs.kuleuven.be

Abstract

In a lazy functional language, the standard encoding of recursion in DSLs uses the host language's recursion, so that DSL algorithms automatically use the host language's least fixpoints, even though many domains require algorithms to produce different fixpoints. In particular, this is the case for DSLs implemented as *Applicative* functors (structures with a notion of pure computations and function application). We propose a recursion primitive *afix* that models a recursive binder in a finally tagless HOAS encoding, but with a novel rank-2 type that allows us to specify and exploit the effects-values separation that characterises *Applicative* DSLs. Unlike related approaches for *Monad*s and *Arrow*s, we model effectful recursion, not value recursion.

Using generic programming techniques, we define an arity-generic version of the operator to model mutually recursive definitions. We recover intuitive user syntax with a form of shallow syntactic sugar: an **alet** construct that syntactically resembles the **let** construct, which we have implemented in the GHC Haskell compiler. We describe a proposed axiom for the *afix* operator. We demonstrate usefulness with examples from *Applicative* parser combinators and functional reactive programming. We show how higher-order recursive operators like *many* can be encoded without special library support, unlike previous approaches, and we demonstrate an implementation of the left recursion removal transform.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Recursion

Keywords Applicative functors, observable recursion, HOAS

1. Introduction

Let us start with an embedded domain-specific language (EDSL) of parser rules, modelled as the GADT (see e.g. [29]) *Rule*. The data type is parameterised by the type a of parse results:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{data}\ \mathit{Rule}\ a\ \textbf{where}} \\
\quad \mathit{Pure} & :: a \to \mathit{Rule}\ a \\
\quad \mathit{Seq} & :: \mathit{Rule}\ (a \to b) \to \mathit{Rule}\ a \to \mathit{Rule}\ b \\
\quad \mathit{Disj} & :: \mathit{Rule}\ a \to \mathit{Rule}\ a \to \mathit{Rule}\ a \\
\quad \mathit{Fail} & :: \mathit{Rule}\ a \\
\quad \mathit{Token} & :: \mathit{Char} \to \mathit{Rule}\ \mathit{Char}
\end{array}
$$

Rule provides DSL primitives *Pure* (match the empty string, return a fixed result), *Seq* (sequence two rules, apply the result of the first to that of the second), *Disj* (choose between two rules), *Fail* and *Token* (parse and return a specified character).

Rule uses the *Applicative* parser combinator style introduced and popularised by Swierstra et al. [31]. Readers may recognise *Seq* and *Pure* as *pure* and $\circledast$ operators of an *Applicative* functor. With Haskell's $a\,`f`\,b$ for $f\ a\ b$, the following bs and $bs' :: \mathit{Rule\ String}$ model a language of arbitrary-length sequences of `b`:

$$
\begin{array}{l}
bs\ = (\mathit{Pure}\ (:)\ `\mathit{Seq}`\ \mathit{Token}\ \texttt{'b'}\ `\mathit{Seq}`\ bs)\quad `\mathit{Disj}`\ \mathit{Pure}\ \texttt{""} \\
bs' = (\mathit{Pure}\ \mathit{snoc}\ `\mathit{Seq}`\ bs'\ `\mathit{Seq}`\ \mathit{Token}\ \texttt{'b'})\ `\mathit{Disj}`\ \mathit{Pure}\ \texttt{""}
\end{array}
$$

According to rule bs, matches either start with token `'b'`, followed by another match of bs or they are empty. The rule $\mathit{Pure}\ (:)$ has no parse behaviour of bs, but produces the list cons operator (:) as a result, so that the parsed token `'b'` is consed with the result of the recursive match. For an empty match, the empty string `""` is returned. Parser rule bs' defines the same language and parse results but expects the recursive match *first*, and the token 'b' second. bs' is *left-recursive*: it refers to itself in a left-most position.

The algorithm $\mathit{nullable} :: \mathit{Rule}\ a \to \mathit{Bool}$ checks if a rule accepts the empty string:

$$
\begin{array}{ll}
\mathit{nullable}\ (\mathit{Pure}\ _) & = \mathit{True} \\
\mathit{nullable}\ (\mathit{Seq}\ a\ b) & = \mathit{nullable}\ a \wedge \mathit{nullable}\ b \\
\mathit{nullable}\ (\mathit{Disj}\ a\ b) & = \mathit{nullable}\ a \vee \mathit{nullable}\ b \\
\mathit{nullable}\ \mathit{Fail} & = \mathit{False} \\
\mathit{nullable}\ (\mathit{Token}\ _) & = \mathit{False}
\end{array}
$$

This definition is satisfactory for finite rules, but a problem arises for infinite, recursive production rules like bs and bs' above.

A known problem of parser DSLs like *Rule* is that left-recursive rules are not treated well. Unlike $\mathit{nullable}\ \ bs$ (which is *True*), $\mathit{nullable}\ \ bs'$ is $\perp$: computation loops forever. Computationally, $\mathit{nullable}\ \ bs'$ loops when considering the left-most part of its first alternative. Denotationally, $\mathit{nullable}\ \ bs'$ corresponds to the fixpoint of a certain function, and the *least fixpoint* $\perp$ we get from Haskell is not the one we would like.

This example shows that for DSLs like our grammar model, algorithms need more control of how fixpoints are calculated. As such, it is inappropriate to rely on Haskell's least fixpoints. Otherwise, parsing libraries are restricted to top-down parsing algorithms, left-recursion is difficult (although some algorithms deal with it anyway, e.g. [16]) and some algorithms are impossible (e.g. print a representation of a rule's parsing structure). Also for DSLs with re-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PEPM'13, January 21–22, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1842-6/13/01…$15.00

cursion in other domains, useful analyses and transformations are rendered impossible by the *implicit* or *unobservable* recursion.

1.1 Overview and Contributions

Like Oliveira and Cook [25], we propose to represent recursive definitions using a recursive binder μ, e.g.:

$$bs = \mu\ s.(\mathit{Pure}\ (:)\ `\mathit{Seq}`\ \mathit{Token}\ \texttt{'b'}\ `\mathit{Seq}`\ s)\ `\mathit{Disj}`\ \mathit{Pure}\ \texttt{""}$$

After some more background in section 2, we encode such a primitive in HOAS style in section 3. We don't use standard HOAS as it allows certain illegal terms, but use Carette et al.'s *finally tagless* style instead. We adapt the technique from lambda terms to applicative DSLs (no Lambda binder but a pure primitive instead) with a new representation of the recursive binder: the afix primitive.

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

The primitive differs from Carette et al.'s Lambda because of its rank-2 type. We show how it models a separation between values and effects of the DSL. This reflects the separation that distinguishes applicative DSLs from monadic ones and makes them suited for analysis and optimisation. In section 4, we discuss an axiom that governs the intended semantics (recursion) of afix.

A downside of an explicit fixpoint primitive like afix is that we lose some of the elegance of Haskell's standard recursive equations. In section 5, we define a shallow form of syntactic sugar to recover Haskell's standard elegance in the form of the $\mathbf{alet}$[1] construct, e.g.:

$$\textbf{alet}\ bs = (:)\ \circledS\ \mathit{token}\ \texttt{'b'} \circledast bs \oplus \mathit{pure}\ \texttt{""}$$

We define scoping and typing rules for $\mathbf{alet}$ and a desugaring into standard Haskell, all implemented in our branch of the Glasgow Haskell Compiler (GHC). Using generic programming techniques, we implement a mutual recursion primitive that reduces mutually recursive bindings to a series of recursive bindings.

Finally, in section 6, we demonstrate the usefulness of our techniques with two larger examples. One example concerns a simple functional reactive programming (FRP) model of electronic hardware and the second example shows a non-trivial parser transformation that removes left-recursion from a parser.

2. Background, examples and machinery

2.1 More fixpoints

Consider how $\mathit{cbe}' = \mathit{nullable}\ \mathit{bs}'$ produces $\bot$ as the wrong solution of a recursive equation. Denotationally (see e.g. [36]), $\mathit{cbe}' = \bot$ is the least fixpoint of cbef' under the standard complete partial order on values.[2]

$$\mathit{cbef}'\ s = (s \wedge \mathit{False}) \vee \mathit{True}$$

The fact that this least fixpoint is $\bot$ is especially unfortunate because cbef' does have a non-$\bot$ fixpoint: $\mathit{cbef}'\ \mathit{True} \equiv \mathit{True}$.

Based on $\mathit{nullable}$, we can also calculate the first set of a rule: the set of characters that can start a successful parse.

$$\begin{array}{l}\mathit{firstSet} :: \mathit{Rule}\ a \rightarrow \mathit{Set}\ \mathit{Char}\\ \mathit{firstSet}\ (\mathit{Pure}\ _) \quad = \mathit{Set.empty}\\ \mathit{firstSet}\ (\mathit{Seq}\ a\ b)\ =\\ \quad \textbf{if}\ \mathit{nullable}\ a\ \textbf{then}\ \mathit{firstSet}\ a \cup \mathit{firstSet}\ b\ \textbf{else}\ \mathit{firstSet}\ a\\ \mathit{firstSet}\ (\mathit{Disj}\ a\ b) = \mathit{firstSet}\ a \cup \mathit{firstSet}\ b\\ \mathit{firstSet}\ \mathit{Fail} \qquad = \mathit{Set.empty}\\ \mathit{firstSet}\ (\mathit{Token}\ t)\ = \mathit{singleton}\ t\end{array}$$

If $\mathit{nullable}\ \mathit{bs}'$ were True instead of $\bot$, then $\mathit{fs}' = \mathit{firstSet}\ \mathit{bs}'$ also gives us the least fixpoint of a function fsf':

$$\mathit{fsf}'\ s = (s \cup \mathit{singleton}\ \texttt{'b'}) \cup \mathit{Set.empty}$$

fsf' has several fixpoints: $\mathit{singleton}\ \texttt{'b'}$, but also $\bot$ (chosen by Haskell) and $\mathit{Set.fromList}\ \texttt{"abc"}$. Only one corresponds to intuition and language theory: $\mathit{singleton}\ \texttt{'b'}$. To conclude, algorithm authors should use domain knowledge to choose appropriate fixpoints and the representation of recursion should support this.

2.2 Applicative Functors

Rule's constructors fit the pattern of an $\mathit{Applicative}$ functor [23], a type class modelling structures with a notion of pure computations (pure) and application ($\circledast$):[3]

$$\begin{array}{l}\textbf{class}\ \mathit{Functor}\ p \Rightarrow \mathit{Applicative}\ p\ \textbf{where}\\ \quad \mathit{pure} :: a \rightarrow p\ a\\ \quad (\circledast)\ :: p\ (a \rightarrow b) \rightarrow p\ a \rightarrow p\ b\\ \textbf{instance}\ \mathit{Applicative}\ \mathit{Rule}\ \textbf{where}\\ \quad \mathit{pure} = \mathit{Pure}\\ \quad (\circledast)\ = \mathit{Seq}\end{array}$$

Rule also instantiates $\mathit{Alternative}$: a class of $\mathit{Applicative}$ functors with a notion of disjunction ($\oplus$) and failure (empty).

$$\begin{array}{l}\textbf{class}\ \mathit{Applicative}\ p \Rightarrow \mathit{Alternative}\ p\ \textbf{where}\\ \quad (\oplus)\quad :: p\ a \rightarrow p\ a \rightarrow p\ a\\ \quad \mathit{empty} :: p\ a\\ \textbf{instance}\ \mathit{Alternative}\ \mathit{Rule}\ \textbf{where}\ \cdots\end{array}$$

2.3 Applicative functors vs. Monads

The reason we work with $\mathit{Applicative}$ DSLs and not for example the better known Monads is that for an analysis to handle recursive equations (our main goal), it must be able to observe the structure of the recursive equation. Monadic DSLs use a monadic bind $\gg\!=$ typed $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ that allows the effects structure of a term to depend on the result value of a previous term, rendering the computation impossible to analyse. Consider the term

$$\begin{array}{l}\mathit{evilMonadic} = \textbf{do}\ n \leftarrow \mathit{evilMonadic}\\ \qquad\qquad\qquad\qquad \textbf{if}\ n \equiv 100\ \textbf{then}\ \mathit{evilMonadic}\\ \qquad\qquad\qquad\qquad \textbf{else}\ \mathit{return}\ (100 - n)\end{array}$$

It is unclear if this recursive equation has a non-$\bot$ solution, but even if it has, it is hopeless to find it. The $\mathbf{do}$-notation translates to $\mathit{evilMonadic} \gg\!= (\lambda n\ \rightarrow\ \textbf{if}\ \ n\ \equiv\ 100 \cdots)$ and this second argument is a function that may return entirely different rules for different n. The monadic bind can not inspect all cases, so that finding non-least fixpoints is hopeless in all but the simplest cases.

Unlike $\gg\!=$, the $\mathit{Applicative}$ apply operator $\circledast$'s type ($p\ (a \rightarrow b) \rightarrow p\ a \rightarrow p\ b$) prevents computations' *values* from influencing the *structure* of subsequent computations. This separates a computation's effects from its values, corresponding for parsers to context-free-ness (roughly, see [21]). This makes $\mathit{Applicative}$ parser libraries well-suited for analysis and optimisation [1, 31] and makes the goal of finding non-least fixpoints feasible.

[1] The word "applet" was already taken...

[2] Note: this is not the standard meaning of cbe' but it can be shown because of the structure of $\mathit{nullable}$ and bs'.

[3] We consistently omit $\mathit{Functor}$ instances because its fmap or $\circledS$ method should satisfy $f\ \circledS\ p = \mathit{pure}\ f \circledast p$.

2.4 Composing Applicative functors

An important tool for us is the composition of two *Applicative* functors p and q [23]. The composed functor $(p \circ q)$ is again *Applicative*.

```
newtype (p ∘ q) a = Comp { comp :: p (q a) }
instance (Applicative p, Applicative q) ⇒
           Applicative (p ∘ q) where
  pure  = Comp · pure · pure
  f <*> v = Comp $ (<*>) <$> comp f <*> comp v
```

For a composed functor $(p \circ q)$, we call p and q the outer resp. inner functor. Terms in either functor can be lifted to $(p \circ q)$. Also useful is a function *withInner*, lifting operations on the inner functor q:

```
liftOuter :: (Functor p, Applicative q) ⇒ p a → (p ∘ q) a
liftInner :: Applicative p ⇒ q a → (p ∘ q) a
withInner :: Functor p ⇒ (q a → h a) →
                    (p ∘ q) a → (p ∘ h) a
```

2.5 Effectful recursion, not value recursion

We study DSL terms defined as the solution of recursive equations, like the examples bs and bs'. This is a different kind of recursion than the *value recursion* studied by Erkök and Launchbury [14]. They study monads that support recursive values in the parameters and results of effectful computations. For example, GHC's *Data.IORef* (a mutable references interface in the *IO* Monad) supports the cells' values to be defined lazily. Using the recursive **do** syntax of GHC's `DoRec` extension (evolved from the original proposal), a function can create mutually recursive reference cells:

```
data Node = Node Int (IORef Node)
mk2nodes = do rec p ← newIORef (Node 0 r)
                  r ← newIORef (Node 1 p)
              return p
```

Value recursion is only interesting in the context of monadic computations. For *Applicative* functors, the separation of values and effects limits the value recursion entirely to the value level. The same effect can then be achieved by using the standard *fix* at the value level (see section 4.2). It is the monadic interplay of effects and values that makes value recursion interesting. In this paper, we study *effectful* recursion, a different form of recursion that is more naturally associated with applicative functors. Effectful recursion is used to define effectful recursive DSL terms, like bs and bs'.

3. A recursion primitive

Let us define our recursion primitive *afix*. We start from more naive definitions and iteratively adapt it to satisfy our requirements.

In our *Applicative Rule* DSL, we might represent a recursive binder in a naive HOAS style with a new *Rule* primitive *Fix0*:

```
Fix0 :: (Rule a → Rule a) → Rule a
bs  = Fix0 $ λbs₀ → (:) <$> Token 'b' <*> bs₀    <|> pure ""
bs' = Fix0 $ λbs'₀ → snoc <$> bs'₀ <*> Token 'b' <|> pure ""
```

Unfortunately, this representation allows meaningless definitions:

```
badBind = Fix0 $ λself → case self of Pure _ → self
                                      _      → Pure 0
```

badBind treats its recursive parameter non-parametrically and does not model a usage of the recursive binder μ. Such definitions are not excluded by *Fix0*'s type. Chlipala explains how this complicates DSL algorithms like *nullable*, requiring an inverse algorithm mapping result values (here: booleans) back to DSL terms [6].

We use Carette et al.'s *finally tagless* style [5] to solve this. Chlipala's alternative Parametric HOAS [6] is discussed in section 7.

3.1 Finally tagless style (second attempt)

In Carette et al.'s finally tagless style, a parser rule with result type a is not a value of data type *Rule* a, but of type *FinalRule0* a:

```
class Applicative p ⇒ CharParser p where
  token :: Char → p Char
type FinalRule0 a =
  ∀ p. (Alternative p, CharParser p) ⇒ p a
```

Under this definition, a rule is something that can be interpreted in any functor that supports the required primitives, e.g. bs and bs':

```
bs, bs' :: FinalRule0 String
bs  = (:) <$> token 'b' <*> bs      <|> pure ""
bs' = snoc <$> bs' <*> token 'b' <|> pure ""
```

The quantification over p in *FinalRule0* rules out definitions like *badBind*. Since a term in the DSL has to support any functor p, it has no way of inspecting recursive references of type $p\ a$.

Writing algorithms in this final style is reminiscent of programming with Church encodings of data types. The polymorphism of *FinalRule0* is used by instantiating p with a special-purpose interpretation functor carrying intermediate analysis results. In the instances of *Alternative* and *CharParser* for this functor, parsing primitives like *pure*, `<|>` and *token* are handled. The analysis function *nullableF* just unwraps the result from the functor:

```
newtype NullableInterp a = NullI Bool
instance Applicative NullableInterp where
  pure _                = NullI True
  (NullI a) <*> (NullI b) = NullI (a ∧ b)
instance Alternative NullableInterp where ···
instance CharParser NullableInterp where ···
nullableF :: FinalRule0 a → Bool
nullableF (NullI r) = r
```

3.2 Finally Recursive (third attempt)

So how can we add a recursive binder to such a final DSL? The obvious solution mimics Carette et al.'s *lambda* construct:

```
class ApplicativeFix₀ p where afix₀ :: (p a → p a) → p a
bs  = afix₀ $ λbs₀ → (:) <$> token 'b' <*> bs₀    <|> pure ""
bs' = afix₀ $ λbs'₀ → snoc <$> bs'₀ <*> token 'b' <|> pure ""
```

We can extend our *nullableF* analysis to support $afix_0$:

```
instance ApplicativeFix₀ NullableInterp where
  afix₀ pf = pf (NullI False)
```

This instance passes *NullI False* to *pf*, specifying that recursive occurrences are assumed not to accept the empty string.

3.3 Applicative problems

Unfortunately, a problem with $afix_0$ remains in complex transformations like the left-recursion removal transform to be described in section 6.2. It will transform for example the left-recursive rule bs' into an equivalent, non-left-recursive rule:

```
transformPaull bs' ≡ foldr ($) <$> bsHead' <*> many bsTail'
  where bsHead' = pure ""
        bsTail' = (:) <$> token 'b'
```

More details follow, but essentially bs' is split in two parts: $bsHead'$ is what remains of the rule with leading self-references removed and $bsTail'$ contains the parts of the rule that follow leading self-references. The transformed rule parses $bsHead'$ first and then iteratively parses $bsTail'$ using $many :: p\ a \to p\ [a]$. The initial result of $bsHead'$ has type $String$, and the parse results of $bsTail'$, typed $String \to String$, are iteratively applied to it using $foldr\ (\$)$, to obtain the same parse results as before.

Actually performing this translation presents several challenges. For a rule like bs' of the form $afix_0\ bsf'$, we have to derive rules $bsHead'$ and $bsTail'$ from bsf'. Distinguishing *leading* from non-leading recursive references is a first problem, but let us assume for now they are all leading (like in bs'). It is then easy enough to find $bsHead'$ as $bsf'\ empty$. Deriving $bsTail'$ from bsf' is harder.

To derive $bsTail'$ (of type $p\ (String \to String)$) from bsf' (of type $p\ String \to p\ String$), we can try to somehow turn bsf' into a value of type $p\ (String \to String) \to p\ (String \to String)$, i.e. make all rules produce a value that can depend on the $String$ result of a previous match. The rules in bsf'' ignore this value, but we can now apply bsf'' to $pure\ id$, i.e. instantiate leading self-references to a pure parser returning the previous parse result.

Generalising slightly, what we crucially need is a *coapplicative* operator $coapp0$ (note the symmetry with the type of $\circledast$):

$$coapp0 :: (p\ a \to p\ b) \to p\ (a \to b)$$

$coapp0$ should satisfy $pf\ (pure\ v) \equiv (\$v) \circledS\ coapp0\ pf$ for $pf :: p\ a \to p\ b$ and $v :: a$. However, it cannot be implemented[4].

We might say that the polymorphism of $FinalRule0$ has to be instantiated too early. If we could instantiate p to $(p \circ ((\to)\ a))$[5] then we could lift the $Applicative$ and related instances to this composed functor, lift $pure\ id$ to type $(p \circ ((\to)\ a))\ a$ and call pf with this value to obtain a value of $(p \circ ((\to)\ a))\ b$ which is essentially $p\ (a \to b)$. However, even though we can instantiate a $FinalRule0$'s p parameter as we like, $(p \circ ((\to)\ a))$ is not suited as the type a becomes known only during the analysis of a term.

We note that $coapp0$'s type is precisely that of Carette et al.'s $Lambda$ constructor. So why can't we simply add a $lambda$ constructor to our applicative DSL to solve our problem? Unfortunately, such a constructor is a bad idea for many DSLs, including our parsing example. Even for a simple recursive descent (RD) parse algorithm, this lambda operator cannot be handled:

```
lambdaRD :: ((String → [(String, a)]) →
  String → [(String, a)]) → String → [(String, a → b)]
lambdaRD pf s = ?
```

3.4 Rank-2 types to the rescue (our final proposal)

The solution we propose is to change the type signature of $afix_0$:

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

$afix$'s type requires that the value of the recursive variable of $afix$'s argument can be wrapped in an arbitrary $Applicative$ functor q with the recursive definition's result value coming out in the same functor at the end.

This restriction is strong enough for the definition of $coapp$. We can instantiate the type constructor argument q to the *environment* functor $((\to)\ a)$, and exploit its standard $Applicative$ instance:

```
coapp :: Applicative p ⇒ (∀ q . Applicative q ⇒
           (p ∘ q) a → (p ∘ q) b) → p (a → b)
coapp p = comp $ p $ liftInner id
```

Our experience shows that $afix$'s type is not too restrictive though: $Applicative$-style primitives on a functor p can be lifted to $(p \circ q)\ a$ for any $Applicative\ q$, e.g. $Alternative$ and $CharParser$:[6]

```
instance (Alternative p, Applicative q) ⇒
           Alternative (p ∘ q) where
  empty   = Comp empty
  va ⦶ vb = Comp (comp va ⦶ comp vb)
instance (CharParser p, Applicative q) ⇒
           CharParser (p ∘ q) where
  token = liftOuter · token
```

We can adapt the previous examples bs and bs' as follows.

```
type FinalRule a = ∀ p.
  (Alternative p, ApplicativeFix p, CharParser p) ⇒ p a
bs, bs' :: FinalRule String
bs  = afix $ λs → (:) <$> token 'b' <*> s   ⦶ pure ""
bs' = afix $ λs → snoc <$> s <*> token 'b' ⦶ pure ""
```

The additional power of $afix$ over $afix_0$ is not always needed. In such cases (like our $nullable$), the quantified functor q can be instantiated to $Identity$:

```
runIdComp :: Functor p ⇒ (p ∘ Identity) a → p a
runIdComp p = runIdentity <$> comp p
wrapIdComp :: Applicative p ⇒ (∀ q . Applicative q ⇒
                 (p ∘ q) a → (p ∘ q) a) → p a → p a
wrapIdComp f s = runIdComp $ f $ liftOuter s
instance ApplicativeFix NullableInterp where
  afix pf = wrapIdComp pf (NullI False)
```

With $firstSet$ similarly adapted, the results for bs are unchanged: $nullableF\ bs \equiv True$, $firstSetF\ bs \equiv singleton$ `'b'`, but left-recursion is now supported: $nullableF\ bs' \equiv True$ and $firstSetF\ bs' \equiv singleton$ `'b'`.

3.5 The meaning of $afix$'s type

Let us try and gain more insight into the meaning of $afix$'s type:

$$afix :: \forall\ p. ApplicativeFix\ p \Rightarrow (\forall\ q. Applicative\ q \Rightarrow p\ (q\ a) \to p\ (q\ a)) \to p\ a$$

A useful tool is Reynolds' notion of parametricity [32, 34]. Because we just want to gain insight, we freely use imprecise formulations and ignore intricacies like strictness [8], although we do believe the results could be made more exact if necessary.

Let us consider the type of $afix$'s arguments:

$$\forall\ q. Applicative\ q \Rightarrow (p \circ q)\ a \to (p \circ q)\ a$$

We conjecture that the following *free theorem* holds for values pf of this type:

[4] We ignore unsafe techniques like dynamic typing and partiality.

[5] $(\to)$ is Haskell's curried arrow type constructor: $(\to)\ a\ b = a \to b$.

[6] This instance of $Alternative\ (p \circ q)$ arbitrarily lifts p's $Alternative$ instance and not q's, but in our case, q's special role warrants this.

Theorem 3.1. *For all q_1, q_2 Applicative, for $k : q_1\ a \to q_2\ a$ respecting the Applicative operations and $u :: (p \circ q1)\ a$,*

$$withInner\ k\ (pf\ u) \equiv pf\ (withInner\ k\ u).$$

This theorem states that for any k, applying $withInner\ k$ before or after pf has the same effect. Consider the constant functor K:

$\mathbf{data}\ K\ a = K$
$\mathbf{instance}\ Applicative\ K\ \mathbf{where}\ \{pure\ _ = K; _ \circledast _ = K\}$

With $k = \lambda_ \to K$, $withInner\ k$ maps values of $(p \circ q)\ a$ to $(p \circ K)\ a$, effectively erasing the value of the computation in p. Theorem 3.1 then implies that applying pf to an argument u and erasing the value of the result is equivalent to first erasing u's value and then applying pf to it. This means that the above theorem states a familiar property, namely that $afix$'s arguments pf respect the separation between values and effects of applicative functors.

4. An $ApplicativeFix$ Axiom

With the key points of our approach established, let us consider the properties a reasonable implementation of $ApplicativeFix$ should satisfy. Many Haskell type classes are associated with such axioms that can sometimes even be linked to category theory or mathematics in general. We have not identified such a relation, but we do propose an axiom that should hold for $afix$.

4.1 Fixpoint law

The $ApplicativeFix$ *fixpoint law* states that an implementation of $afix$ should always deliver a fixpoint:

$$afix\ pf \equiv wrapIdComp\ pf\ \$\ afix\ pf$$

Remember that denotationally, Haskell recursive definitions produce the *least* fixpoint of a function pf [36]. This law states that $afix$ must also return a fixpoint, but not necessarily the least. Note by the way that it also implies a behaviour on constant functions:

Corollary 4.1 (Constant preservation)**.** *If x is not free in g, $afix\ (\lambda x \to liftOuter\ g) \equiv g$.*

4.2 Interesting non-axioms

There are two axioms that we want to explicitly *not* propose. We discuss them, nevertheless, as they shed more light on the meaning of our $afix$ primitive.

Left shrinking considered harmful The first such property is an analogon of Erkök and Launchbury's *Left Shrinking* law for $MonadFix$ [14] or the equivalent for Paterson's $ArrowLoop$ [27]. For $ApplicativeFix$, one might expect the following left shrinking:

$$afix\ (\lambda x \to liftOuter\ a \circledast f\ x) \overset{\text{(not an axiom)}}{=} a \circledast afix\ f$$

with x not free in a and $a :: p\ (v \to v)$ for some p and v.

Let us consider what this proposed axiom would mean for an example parser for an infinite number of as, defined as follows:

$$as = afix\ \$\ \lambda s \to (:)\ \circledS\ token\ \texttt{'a'} \circledast s$$

Under the left-shrinking assumption, the above is equivalent to a single token parser followed by an infinite effect-free parser:

$$\begin{aligned} as &= afix\ \$\ \lambda as' \to (:)\ \circledS\ token\ \texttt{'a'} \circledast as' \\ &= (:)\ \circledS\ token\ \texttt{'a'} \circledast afix\ \$\ \lambda as' \to as' \end{aligned}$$

However, this new expression is not at all equivalent to the original. The left shrinking property states that computations not depending on recursive occurrences can be lifted out of the recursive equation, their effects no longer taking part in the recursion. As such, the axiom is defining for the value recursion that Erkök and Launchbury model and rules out the effectful recursion we aim for.

Fixing what is pure? A second property that *not* all reasonable instances of $ApplicativeFix$ satisfy, prescribes a form of fixpoints for recursive definitions that do not add effects:

$$afix\ (fmap\ h) \overset{\text{(not an axiom)}}{\equiv} pure\ (fix\ h)$$

for any h typed $a \to a$. The right hand side is a fixpoint of $fmap\ h$, but not necessarily the one we want. In an $Alternative$ functor p, $empty$ is also a fixpoint and it is in fact the natural one for most examples in this text. For instance, in a parser DSL, the left-hand side models a non-terminal X with a single production rule $X \to X$, equivalent in language theory to an $empty$ rule.

5. Making $afix$ practical

We turn our attention to some tools that make it practical to implement and work with DSLs using $afix$.

5.1 Some Tools

First, unlike alternative solutions [2, 11], $afix$ supports higher-order combinators without primitive support. These are observably recursive analogues of the standard $many$ and $some$ combinators:

$$\begin{aligned} &manyAF, someAF :: (Alternative\ p, ApplicativeFix\ p) \Rightarrow \\ &\qquad p\ a \to p\ [a] \\ &someAF\ f = (:)\ \circledS\ f \circledast manyAF\ f \\ &manyAF\ f = afix\ \$\ \lambda s \to (:)\ \circledS\ liftOuter\ f \circledast s \oplus pure\ [\,] \end{aligned}$$

Note that $manyAF$ and $someAF$'s types only differ from their standard analogues in the $ApplicativeFix$ constraint.

A standard function $afixInf$ implements $afix$ by going back to Haskell's unobservable recursion. This sometimes makes sense, for example when interfacing with existing libraries that are designed to work with Haskell's implicit recursion. We show an instance for `uu-parsinglib`'s parser representation. Also useful is $afixKill$, which replaces recursive occurrences with a failing $empty$ rule.

$$\begin{aligned} &afixInf :: Applicative\ p \Rightarrow (\forall\ q\ .\ Applicative\ q \Rightarrow \\ &\qquad (p \circ q)\ a \to (p \circ q)\ a) \to p\ a \\ &afixInf\ f = fix\ \$\ wrapIdComp\ f \end{aligned}$$

$\mathbf{instance}\ ApplicativeFix\ (P\ st)\ \mathbf{where}\ afix = afixInf$

$$\begin{aligned} &afixKill :: Alternative\ p \Rightarrow (\forall\ q\ .\ Applicative\ q \Rightarrow \\ &\qquad (p \circ q)\ a \to (p \circ q)\ a) \to p\ a \\ &afixKill\ f = runIdComp\ \$\ f\ empty \end{aligned}$$

5.2 Arity-Genericity

With $afix$ modelling effectful recursion on one variable, the next question is what to do about *mutual* recursion. Instantiating $afix$ at type $(\forall\ b\ .\ ApplicativeFix\ b \Rightarrow (f \circ b)\ (a1, a2) \to (f \circ b)\ (a1, a2)) \to f\ (a1, a2)$ is not a solution because the type $f\ (a1, a2)$ does not allow both recurrands to produce different effects. What we would like is a primitive of the following type:

$$assocComp_1 :: Applicative\ p \Rightarrow ((p \circ q1) \circ q2)\ a \to (p \circ (q1 \circ q2))\ a$$
$$assocComp_2 :: Applicative\ p \Rightarrow (p \circ (q1 \circ q2))\ a \to ((p \circ q1) \circ q2)\ a$$
$$liftComposed :: (Applicative\ p, Applicative\ q2) \Rightarrow (p \circ q1)\ a \to (p \circ (q2 \circ q1))\ a$$

Figure 1. Utility functions for working with functors $(p \circ q)$ (implementations omitted).

$$afix_2 :: ApplicativeFix\ p \Rightarrow (\forall\ q\ .\ Applicative\ q \Rightarrow ((p \circ q)\ a_1 \to (p \circ q)\ a_2 \to (p \circ q)\ a_1, (p \circ q)\ a_1 \to (p \circ q)\ a_2 \to (p \circ q)\ a_2)) \to (p\ a_1, p\ a_2)$$

This is not an instance of *afix*'s type, but we can construct such a primitive from it, taking inspiration from Bekić's theorem [36]. This theorem relates mutually recursive bindings to repeated applications of the *fix* operator. We imitate the form for *afix*, exploiting the polymorphism of $afix_2$'s argument and juggling the $(\cdot \circ \cdot)$ type constructor in the types (using utility functions from Figure 1).

$$
\begin{aligned}
& afix_2\ pf = (fp_1, fp_2)\ \mathbf{where} \\
& \quad fp_1 = afix\ \$\ \lambda fp_1' \to \\
& \qquad \mathbf{let}\ fp_2' = Comp\ \$\ afix\ \$\ \lambda fp_2'' \to comp\ \$ \\
& \qquad\qquad assocComp_2\ \$\ snd\ pf\ (liftComposed\ fp_1') \\
& \qquad\qquad (assocComp_1\ \$\ Comp\ fp_2'') \\
& \qquad \mathbf{in}\ fst\ pf\ fp_1'\ fp_2' \\
& \quad fp_2 = \text{... (analogous)}
\end{aligned}
$$

This definition is obscured by technicalities. In the first call to *afix*, the recursive argument fp_1' is of type $(p \circ q1)\ a_1$ for an arbitrary $q1$. However, in the second call to *afix*, fp_2'' is of type $(p \circ q2)\ (q1\ a_2)$ for an arbitrary $q2$. The trick is then to apply pf with its type parameter q instantiated to $(q2 \circ q1)$ and exploit associativity of the $(\cdot \circ \cdot)$ type constructor to fit everything together.

Now, we can do this for any arity, but this is not very effective: we can only hope to implement a finite number of $afix_i$s and additionally, the size of the definition of $afix_i$ increases exponentially with i. Luckily, GHC's type system allows us to do better.

Omitting details for space reasons, we have used techniques described by McBride [22][7] to develop an arity-generic or polyvariadic version of *afix*. With encoding artefacts removed and using ellipses in a not fully formal notation for type lists, this is its type:

$$
\begin{aligned}
& nafix :: \forall\ (p :: * \to *)\ ([t_1 \cdots t_n] :: [*]).\ ApplicativeFix\ p \Rightarrow \\
& \quad ((\forall q : * \to *. Applicative\ q \Rightarrow \\
& \qquad (p \circ q)\ t_1 \to \cdots \to (p \circ q)\ t_n \to (p \circ q)\ t_1), \cdots, \\
& \quad (\forall q : * \to *. Applicative\ q \Rightarrow \\
& \qquad (p \circ q)\ t_1 \to \cdots \to (p \circ q)\ t_n \to (p \circ q)\ t_n)) \to \\
& \quad (p\ t_1, \ldots, p\ t_n)
\end{aligned}
$$

This arity-generic version of *afix* gives us effectful *mutual* recursion at any arity without primitives beyond *afix*. The implementation of *nafix* can be found in our GHC branch (see Section 5.4).

5.3 Syntactic sugar

By writing recursive definitions in terms of this new fixpoint primitive, we lose some of the elegance of Haskell's standard **let** bindings. But because our *ApplicativeFix* and *afix* are a general tool for *Applicative* functors requiring observable effectful recursion, a shorthand notation to recover this elegance makes sense. Since we work in an applicative style, a syntax that resembles standard **let** bindings is natural: we propose the **alet**-notation:

$$\mathbf{alet}\ bs = pure\ \texttt{""} \oplus (:) \circledS\ token\ \texttt{'b'} \circledast bs$$

Note the use of a *top-level* **alet**. We allow mutual recursion, like in the following simple expression parser[8]

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

Syntactic and scoping rules for **alet** are analogous to normal **let** bindings. Its typing rule is presented in Figure 2, using notation from Vytiniotis et al. [33]. Specifically, their typing judgement $Q\,;\Gamma \vdash e : \tau$, where Q is a set of constraints, Γ is a typing environment, e an expression and τ a type. We write some kind annotations, typing context entries for type variables and explicit type applications of polymorphic values for clarity. The typing rule for **alet** is not standard, but it reflects the requirements of our *afix* and *nafix* primitives: the definitions of bound variables x_i are type-checked against type $(p \circ b)\ t_i$ where b is a fresh *Applicative* functor, and all x_j are bound at type $(p \circ b)\ t_j$. The body of the **alet** construct is type-checked with all x_i bound at type $p\ t_i$.

$$
\frac{
\begin{array}{l}
Q; \Gamma \vdash p : * \to * \qquad Q \Vdash ApplicativeFix\ p \qquad \text{q fresh} \\
\forall i = 1..n,\ \ Q \wedge Applicative\ q; \\
\qquad \Gamma, q : * \to *, \{x_j : (p \circ q)\ t_j\}_{j=1..n} \vdash e_i : (p \circ q)\ t_i \\
Q\,; \Gamma, \{x_i : p\ t_i\}_{i=1..n} \vdash e : T
\end{array}
}{
Q\,; \Gamma, \vdash \mathbf{alet}\ \{x_i = e_i\}_{i=1..n}\ \mathbf{in}\ e : T
}\ \text{(T-ALET)}
$$

Figure 2. Typing **alet** bindings

With this typing rule, Figure 3 defines a type- and syntax-directed desugaring $\mathcal{A}$, transforming **alet** to regular Haskell. Essentially, all recursively bound variables x_i are converted into open recursive functions $x_i^\sharp$. We generate a call to the n-ary *nafix* primitive and project out $\hat{x}_i$ from the resulting n-ary tuple. Finally, we replace occurrences of x_i in the body with $\hat{x}_i$. Type soundness of the translation follows easily from type checking the right-hand side of the translation $\mathcal{B}$ against the type of the original **alet** construct.

Theorem 5.1 ($\mathcal{A}$ is type-preserving)**.**

$$Q\,; \Gamma \vdash e : T \Rightarrow Q\,; \Gamma \vdash \mathcal{A}\langle\Gamma\rangle\,(e) : T.$$

We don't formalise *top-level* **alet***s*, a declaration form of **alet**s.

5.4 Implementation

We have extended GHC to parse, type-check and desugar **alet** expressions. We perform the translation in Fig. 3 during the *desugaring* phase, which translates scope- and type-checked Haskell code into GHC's explicitly typed core language (a variant of System F_ω extended with features like equality coercions [30]). We currently support **alet** expressions as defined in this text. Still missing is support for top-level **alet**s and type signatures for **alet** bindings

[7] We use type families instead of multi-parameter type classes though.

[8] Note: we use shorthands $\circledast\!\!\!\!<$ and $\circledast\!\!\!\!>$ that behave like $\circledast$ but ignore the result of their second resp. first argument, e.g. $v \circledast\!\!\!\!<\ w = const \circledS v \circledast w$.

$$\mathcal{A}\langle Q;\Gamma\rangle\left(\textbf{alet}\ \{x_i = e_i\}_{i=1..n}\ \textbf{in}\ e\right) = \mathcal{B}\langle\{t_i\}_{i=1..n}\rangle\left(\textbf{alet}\ \{x_i = e_i\}_{i=1..n}\ \textbf{in}\ e\right)$$
$$\text{with}\begin{cases} Q;\Gamma \vdash p : * \to *, \qquad Q \Vdash \mathit{ApplicativeFix}\ p \\ \forall i\ \ Q \wedge \mathit{Applicative}\ b;\Gamma, b : * \to *, \{x_j : (p \circ b)\ t_j\}_{j=1..n} \vdash e_i : (p \circ b)\ t_i \end{cases}$$
$$\mathcal{A}\langle Q;\Gamma\rangle\,(e) = \cdots \quad \text{(compositionally apply } \mathcal{A}\langle Q;\Gamma\rangle \text{ on components of } e\text{)}$$

$$\mathcal{B}\langle\{t_i\}_{i=1..n}\rangle\left(\textbf{alet}\ \{x_i = e_i\}_{i=1..n}\ \textbf{in}\ e\right) = \begin{array}{ll}\textbf{let} & \left\{x_i^\sharp = \lambda\, x'_1 \cdots x'_n \to [x'_j/x_j]_{j=1..n} e_i\right\}_{i=1..n} \\ & f = \mathit{nafix}\ @[t_1..t_n]\ \left(x_1^\sharp, \cdots, x_n^\sharp\right) \\ & \{\hat{x}_i = \pi_i @[(p\ t_1) \cdots (p\ t_n)]\ f\}_{i=1..n} \\ \textbf{in} & [\hat{x}_i/x_i]_{i=1..n}\ e\end{array}$$

Figure 3. The type-directed translation of **alet** bindings. We write some explicit type applications for clarity.

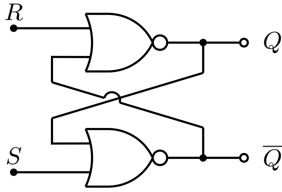

Figure 4. An SR-latch, using a pair of cross-coupled NOR gates.

$$\begin{array}{l} \mathit{nor}\ x\ y = \neg\,(x \vee y) \\ \mathit{srLatch} :: \mathit{Delayable}\ f \Rightarrow f\ (\mathit{Bool}, \mathit{Bool}) \to f\ (\mathit{Bool}, \mathit{Bool}) \\ \mathit{srLatch}\ \mathit{inputs} = \textbf{let}\ \mathit{qi}\ = \mathit{nor}\ \circledS\ r \circledast \mathit{delay}\ \mathit{False}\ \mathit{qi}' \\ \qquad\qquad\qquad\qquad \mathit{qi}' = \mathit{nor}\ \circledS\ s \circledast \mathit{delay}\ \mathit{False}\ \mathit{qi} \\ \qquad\qquad\qquad\quad \textbf{in}\ (,)\ \circledS\ \mathit{qi} \circledast \mathit{qi}' \\ \quad \textbf{where}\ r = \mathit{fst}\ \circledS\ \mathit{inputs} \\ \qquad\qquad s = \mathit{snd}\ \circledS\ \mathit{inputs} \end{array}$$

Figure 5. An FRP implementation of an SR latch's without **alet**.

(but type annotations in the bodies can be used instead). The examples in this text without top-level **alet**s are supported. The code is available on GitHub with instructions for building[9] and pointers for trying out examples and navigating the code.

6. Examples

6.1 Functional Reactive Programming

A domain where our techniques are useful is functional reactive programming (FRP), a declarative programming model that describes a system in terms of time-varying functions instead of mutable state with applications in robotics [28], animation [13] and graphical user interfaces [19].

A simple FRP model may consider behaviours as functions of time, making them an instance of the environment functor [23]:

$$\begin{array}{l} \textbf{type}\ \mathit{Behaviour} = (\to)\ \mathit{Time} \\ \textbf{instance}\ \mathit{Applicative}\ ((\to)\ a)\ \textbf{where} \\ \quad \mathit{pure}\ = \mathit{const} \\ \quad f \circledast g = \lambda t \to f\ t\ (g\ t) \end{array}$$

FRP provides a form of state through the delaying of behaviours. For discrete time models, the behaviour $\mathit{delay}\ v\ \mathit{bhv}$ produces value v first and the *delayed* values of bhv next. We assume functions $\mathit{prevT} :: \mathit{Time} \to \mathit{Time}$ and $\mathit{initT} :: \mathit{Time}$.

$$\begin{array}{l} \textbf{class}\ \mathit{Applicative}\ f \Rightarrow \mathit{Delayable}\ f\ \textbf{where} \\ \quad \mathit{delay} :: a \to f\ a \to f\ a \\ \textbf{instance}\ \mathit{Delayable}\ \mathit{Behaviour}\ \textbf{where} \\ \quad \mathit{delay}\ v\ b = \lambda t \to \textbf{if}\ t \equiv \mathit{initT}\ \textbf{then}\ v\ \textbf{else}\ b\ \$\ \mathit{prevT}\ t \end{array}$$

6.1.1 Modelling electronic circuits with flip-flops

Let us use FRP to model event networks and, in particular, electronic circuits. Figure 4 shows an SR-latch, a simple flip-flop circuit with two stable states, sometimes used to store information. The circuit has two input wires: R and S, and two output wires: Q and $\overline{Q}$ (opposite in correct states). If S and R are high, they *set* resp. *reset* the state of the circuit (the value of Q).

Figure 5 shows an FRP model of the SR-latch over an arbitrary $\mathit{Applicative}$ functor with a $\mathit{Delayable}$ instance, reacting to events of type $(\mathit{Bool}, \mathit{Bool})$ (the values applied to the input wires). The latch's recursion is modelled in the standard way, and there is no problem to simulate the circuit:

$$\mathit{sampleInput} = \mathit{delay}\ (\mathit{True}, \mathit{False})\ \$\ \mathit{const}\ (\mathit{False}, \mathit{False})$$

The simulation shows that our sample input correctly initialised the circuit: $\mathit{map}\ (\mathit{srLatch}\ \mathit{sampleInput})\ [\mathit{initT}\,..]$ gives us $[(\mathit{False}, \mathit{True}), (\mathit{False}, \mathit{True}), ...]$.

6.1.2 Delayable applicative functors

In Figure 5, the recursive calls to qi and qi' are guarded by delays, essential for the proper functioning of the example. If we remove the delays, our example no longer terminates. For more complex examples, such errors can be less obvious and lead to bugs.

Our approach can already do better. If we model the recursion using $\mathit{ApplicativeFix}$, we can scan for erroneous loops upfront and report errors early on. We just need to lift the $\mathit{Delayable}$ class to composed functors (omitted) and change $\mathit{srLatch}$ to use **alet**. We use $\mathit{liftOuter}$ to lift a and b into the composed functor:

$$\begin{array}{l} \mathit{srLatch2} :: (\mathit{ApplicativeFix}\ f, \mathit{Delayable}\ f) \Rightarrow \\ \qquad\qquad f\ (\mathit{Bool}, \mathit{Bool}) \to f\ (\mathit{Bool}, \mathit{Bool}) \\ \mathit{srLatch2}\ \mathit{inputs} = \\ \quad \textbf{alet}\ \mathit{qi}\ = \mathit{nor}\ \circledS\ \mathit{liftOuter}\ r \circledast \mathit{delay}\ \mathit{False}\ \mathit{qi}' \\ \qquad\quad \mathit{qi}' = \mathit{nor}\ \circledS\ \mathit{liftOuter}\ s \circledast \mathit{delay}\ \mathit{False}\ \mathit{qi} \\ \quad \textbf{in}\ (,)\ \circledS\ \mathit{qi} \circledast \mathit{qi}' \\ \quad \textbf{where}\ \ldots \quad \text{-- see Figure 5} \end{array}$$

In a final style, we can analyse this definition with a custom functor: $\mathit{TestValid}\ a$ ignores its parameter a, but keeps track of the valid-

[9] http://github.com/ilyasergey/GHC-XAppFix/wiki

ity of a circuit and its current minimum delay. The *Applicative* and *Delayable* instances initialise and combine the validity and minimum delay values in the obvious way.

```
data ValidInterp a = VI {isValid :: Bool, viDelay :: Int}
instance Applicative ValidInterp where
  pure _               = VI True 0
  VI va da ⊛ VI vb db = VI (va ∧ vb) (min da db)
instance Delayable ValidInterp where
  delay _ (VI s d) = VI s (d + 1)
```

In the *ApplicativeFix* instance, we test if the recursive function properly introduces a delay: we pass it a recursive occurrence with a delay of -1. If what comes back doesn't have an increased delay, the circuit is invalid:

```
instance ApplicativeFix ValidInterp where
  afix f = case wrapIdComp f $ VI True (−1) of
    VI t d | d < 0      → VI False d
    VI t d | otherwise → VI t d
```

Reassuringly, *isValid* (*srLatch2* (*VI True* 0)) is *True*.

6.2 Left-recursion removal

The largest example in this paper is the previously mentioned algorithm that transforms left-recursive parsers into an equivalent non-left-recursive form. It is inspired by a previous implementation of a uniform version of the Paull transformation [17, p. 304] in the `grammar-combinators` library [11]. This is a complex transformation and we think it shows other transformations like the ones needed for parsing with derivatives [24] are possible as well.

In section 3.3, we discussed how *transformPaull* transforms the example left-recursive rule bs' into a non-left-recursive equivalent:

```
transformPaull bs′ ≡
  foldr ($) ⓢ bsHead′ ⊛ many bsTail′ where
    bsHead′ = pure ""
    bsTail′  = (:) ⓢ token 'b'
```

The core of the transformation is in the implementation of *afix*, where we transform a parser like $bs' = afix\ bsf'$ to the above. The difficult part is deriving $bsHead'$ and $bsTail'$ from *pf* and we have already discussed some of the techniques for doing this in section 3. To derive the first, we pass *pf* a recursive occurrence that behaves differently in different positions: it always fails in a head position and behaves as an actual recursive occurrence in tail positions. In a similar but more complicated way, we construct $bsTail'$, but not without the *coapp* function from section 3.4.

We model different positions for a parsing rule as three contexts: *PurePos* (a position where a parser must not consume any input), *HeadPos* (the left-most position of a non-terminal definition where a parser must consume input) and *TailPos* (a position where input has already been consumed). The *HeadPos* context can additionally specify that primitive parsers in this position are to be either *killed* (replaced with *empty*) or left unmodified:

```
data RuleCtx = PurePos
             | HeadPos HeadMod
             | TailPos deriving Eq
data HeadMod = KillHeads | DontTouch deriving Eq
modHead :: Alternative p ⇒ HeadMod → p a → p a
modHead KillHeads   _ = empty
modHead DontTouch p = p
```

For an underlying parser functor p, we define a type of transformable parsers that support these three contexts. This type of transformable rules plays the role of the interpretation functor that we instantiate the polymorphism of the rules with. We define the algorithm by providing instances for the relevant type classes *Applicative*, *Alternative*, *CharParser* and *ApplicativeFix*.

```
data PaullT p a = PaullT {paullT :: RuleCtx → p a}
```

The *Alternative* instance just lifts p's operations in all contexts:

```
instance Alternative p ⇒ Alternative (PaullT p) where
  empty = PaullT $ const empty
  a ⦶ b  = PaullT $ λctx → paullT a ctx ⦶ paullT b ctx
```

In the *Applicative* instance, we define the behaviour of the parsers in the three contexts. The rule for sequencing is the most complex, because care has to be taken to correctly define how the two sequenced parts can each be in the left-most position of a rule:

```
instance (Alternative p, Applicative p) ⇒
  Applicative (PaullT p) where
  pure v = PaullT r where r PurePos       = pure v
                          r (HeadPos _) = empty
                          r TailPos        = pure v
  rf ⊛ rv = PaullT r
    where r ctx@(HeadPos _) =
                  paullT rf PurePos ⊛ paullT rv ctx
              ⦶ paullT rf ctx ⊛ paullT rv TailPos
          r ctx = paullT rf ctx ⊛ paullT rv ctx
```

The *CharParser* instance is self-explanatory:

```
instance (Alternative p, CharParser p) ⇒
    CharParser (PaullT p) where
  token c = PaullT r
    where r PurePos         = empty
          r (HeadPos kh) = modHead kh $ token c
          r TailPos          = token c
```

Finally, the *ApplicativeFix* instance handles recursive references:

```
instance (Alternative p, ApplicativeFix p) ⇒
  ApplicativeFix (PaullT p) where
  afix (pf :: ∀ q . Applicative q ⇒
    (PaullT p ∘ q) a → (PaullT p ∘ q) a) = PaullT r
    where ...
```

In a pure context, we just kill recursive occurrences:

```
r PurePos = paullT (afixKill pf) PurePos
```

In other contexts, we implement the transformation rule described above (taking into account the head modification in a head context). We use an omitted function *manyComp*: a version of *manyAF* from Section 5.1, lifted to composed functor $(p \circ q)$:

```
r (HeadPos hm) = rNP hm
r TailPos           = rNP DontTouch
rNP :: HeadMod → p a
rNP hm = afix $ λself →
  foldr ($) ⓢ rStart hm self ⊛ manyComp (rDeriv self)
```

rStart and *rDeriv* play the role of $bsHead'$ and $bsTail'$ in our earlier explanation. We define *rStart* by calling the function *pf* with a recursive occurrence that behaves as *empty* in pure and head contexts, and as an actual recursive reference elsewhere:

```
rStart :: Applicative q ⇒ HeadMod → (p ∘ q) a → (p ∘ q) a
rStart hm self =
```

```
Comp $ flip paullT (HeadPos hm) $ comp $ pf $
  Comp $ PaullT $ λctx →
    if ctx ≡ TailPos then comp self else empty
```

Finally, as discussed in Section 3.4, $rDeriv$ is a prime example for the need of the $coapp$ function described. It produces a parser with result type $a \to a$ to be used in the definition of rNP above. In addition to the recursive occurrence $self$ from the call to $afix$, $coapp$ gives us a placeholder $prev$ to substitute for left-recursive occurrences. Using our $RuleCtx$s, we call f with a recursive reference behaving as $prev$ in head positions and as $self$ elsewhere:

```
rDeriv :: Applicative q ⇒ (p ∘ q) a → (p ∘ q) (a → a)
rDeriv self = coapp $ rDeriv′ self
rDeriv′ :: ∀ q q2 . (Applicative q, Applicative q2) ⇒
  (p ∘ q) a → ((p ∘ q) ∘ q2) a → ((p ∘ q) ∘ q2) a
rDeriv′ self prev =
  assocComp₂ $ Comp $
  flip paullT (HeadPos KillHeads) $ comp $
  pf $ Comp $ PaullT rself where
    rself :: RuleCtx → p (q ∘ q2) a
    rself PurePos     = empty
    rself HeadPos { } = comp $ assocComp₁ prev
    rself TailPos     = liftOuter ⊛ comp self
```

With this machinery, the function $transformPaull$ applies the left-recursion removal transformation to a parser:

```
transformPaull :: Alternative q ⇒ PaullT q a → q a
transformPaull p = paullT p PurePos
                   ⦶ paullT p (HeadPos DontTouch)
```

Omitting some uninteresting glue code to the standard $Applicative$ parsing library `uu-parsinglib`, we can now apply its standard error-correcting parsing algorithm to our transformed and no longer left-recursive parser $expr$ from Section 5.3:

```
exprParse :: String → Int
exprParse = parseUU $ transformPaull expr
testParse = exprParse "1+7*3+(8*1+2*6)"
```

7. Related Work

Applicative functors For background on applicative functors, we refer to McBride and Paterson [23]. Note that their bracket notation (translating $[\![f\ u_1 \ldots u_n]\!]$ to $pure\ f \circledS u_1 \circledast \ldots \circledast u_n$) is orthogonal to our **alet** syntax and would fit well in **alet** right-hand sides. Lindley et al. clarify the relation between $Applicative$ functors, $Arrow$s and $Monad$s [20]. $Applicative$ parser combinators were popularised by Swierstra and colleagues [1, 31], who have shown them better suited for analysis and optimisation. A better handling of recursion is still a missing piece of the puzzle, and has been a motivation for work on observable recursion (see below).

HOAS representations There is a wide variety of research on HOAS representations and how to prevent the problem discussed in section 3. We have already discussed Carette et al. [5]'s finally tagless style, built on the work by Washburn and Weirich [35]. Chlipala has proposed Parametric HOAS, also exploiting parametricity in a better HOAS encoding [6]. Contrary to Carette et al., this results in an initial encoding instead of a final one. PHOAS has been used by Oliveira and Cook to obtain observable monomorphic recursion [25] in graph structures. In follow-up work developed in parallel with ours, Oliveira and Löh extend this to typed DSLs with mutually recursive bindings, leading to a solution with many similarities to ours [26]. In addition to observable recursion, they also provide observable sharing, which we have not considered. Their model of mutually recursive binders uses techniques similar to those in section 5.2, but they keep mutual bindings as primitive instead of reducing them to sequences of simple binders like us. For achieving a usable end-user syntax, they demonstrate the use of the impure `data-reify` library, while we propose the **alet** syntax.

We have experimented with using an initial encoding based on PHOAS instead of our finally tagless encoding. This involves a version of our $Rule$ data type parameterised by a type constructor v that represents variables of a certain type. We also add an additional Var constructor and a different recursion primitive Fix:

```
data PRule v a where
  Var :: v a → PRule v a
  Fix :: (v a → PRule v a) → PRule v a
  Seq :: PRule v (a → b) → PRule v a → PRule v b
  ...
type Rule a = ∀ v · PRule v a
```

Parametricity of bindings is enforced by requiring the terms to support any variable type constructor v. However, this representation also suffers from the problem discussed in section 3.2 and does not permit the definition of an analogon of $coapp$. As in section 3.4, this can be solved by changing Fix's type to the rank-2 type

```
Fix :: (∀ w. Applicative w ⇒
  v :<: w → w a → PRule w a) → PRule v a
```

$v :<: w$ is a synonym for $\forall\ a \cdot v\ a \to w\ a$, i.e. an embedding of variables $v\ a$ into a wider set $w\ a$. We have not found reasons to prefer this encoding over ours, although some people may prefer an initial encoding over a final one. This encoding may translate better to predicative languages like Agda.

Observable recursion through fixpoint primitives We have already seen Erkök and Launchbury's $MonadFix$ type class and fixpoint primitive $mfix$, typed $MonadFix\ m \Rightarrow (a \to m\ a) \to m\ a$ [14, 15]. They focus on value recursion; $mfix$'s type and its axioms (see Section 4.2) are not suited for effectful recursion. Erkök and Launchbury extend Haskell's do-notation with recursive value bindings that are desugared into applications of $mfix$.

Hughes proposed the use of $Arrow$s in functional languages [18] as a generalisation of monads, similar to applicative functors. In a paper proposing a do-notation for arrows, Paterson also proposed $ArrowLoop$: a type class modelling value recursion in arrows [27]. Similar to $MonadFix$, the $ArrowLoop$ axioms prescribe a value-recursion semantics and do not allow effectful recursion.

Observable recursion through typed references Both Baars and Swierstra [2, 3] and Devriese and Piessens [10, 11, 12] each define observably recursive encodings of grammars based on a well-typed representation of references. Baars and Swierstra employ a form of de Bruijn-indices into a type-level encoding of a type environment and Devriese and Piessens require the user to define an encoding of the grammar's non-terminals at both type and value level. Baars and Swierstra implement a fixpoint primitive and propose a general form of syntax macros for Haskell to recover a nice syntax for their grammar definitions. Brink et al. [4] demonstrate a deeper encoding of parsers than ours, representing grammars as a set of production rules parameterised by a set of non-terminals in the dependently-typed language Agda. Compared to these approaches, our fixpoint primitive is more powerful. For example, we support higher order

recursive operators like $manyAF$ from Section 5.1 without hard-coded support in the parsing library.

Dependently typed recursion In an unpublished draft, Danielsson and Norell [9] show how left-recursive parsers can be excluded in a dependently typed parser combinator DSL. They restrict the recursion in the DSL while we support other interpretations of the recursion that is there. Danielsson [7] uses mixed induction-coinduction to define parser combinators that support left-recursion, but (impressively) remain provably total and correct. Danielsson's technique does not seem to allow the choice of non-standard fixpoints in the same way as we do. For the following parser $test$

$$
\begin{aligned}
&test : Co\mathbb{N} \rightarrow P\ false\\
&test\ zero \quad\ \ = sat\ (\stackrel{?}{=}\ \texttt{'a'})\\
&test\ (suc\ n) = \sharp\ test_2\ (\flat\ n)\ \cdot\ \sharp\ ?\ (sat\ (\stackrel{?}{=}\ \texttt{'b'}))
\end{aligned}
$$

we suspect $test\ \infty$ is indistinguishable from finite $test\ n$ making first set calculation and other algorithms hopeless.

8. Conclusion

We have shown that effectful recursion is an important problem in several domains. We propose the class $ApplicativeFix$ of $Applicative$ functors with a recursion primitive $afix$. It uses a finally tagless HOAS encoding of a recursive binder μ, adapted to $Applicative$ functors with a rank-2 type that allows to exploit the $Applicative$ values-effects separation. We use generic programming techniques to derive mutual recursion primitives and propose the **alet** construct as a shallow form of syntactic sugar for $afix$ with an implementation in GHC. We show that our approach is useful for (at least) two domains: parsing and functional reactive programming. Our approach supports higher-order operators like $manyAF$ without ad hoc support in the DSL encoding.

Acknowledgements

This research is partially funded by the Research Foundation - Flanders (FWO), and by the Research Fund KU Leuven. Dominique Devriese holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO). The authors thank Bruno Oliveira and Andres Löh for an interesting discussion about their related work.

References

[1] A. I. Baars, A. Löh, and S. D. Swierstra. Parsing permutation phrases. *J. Funct. Program.*, 14(6):635–646, 2004.

[2] A. I. Baars and S. D. Swierstra. Type-safe, self inspecting code. In *Haskell*, pages 69–79, 2004.

[3] A. I. Baars, S. D. Swierstra, and M. Viera. Typed transformations of typed abstract syntax: The left corner transform. In *LDTA*, pages 18–33, 2009.

[4] K. Brink, S. Holdermans, and A. Löh. Dependently typed grammars. In *MPC*, 2010.

[5] J. Carette, O. Kiselyov, and C.-c. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(05):509–543, 2009.

[6] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *ICFP*, pages 143–156, 2008.

[7] N. A. Danielsson. Total parser combinators. In *ICFP*, pages 285–296, 2010.

[8] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *POPL*, pages 206–217, 2006.

[9] N. A. Danielsson and U. Norell. Structurally recursive descent parsing. Draft, 2008.

[10] D. Devriese and F. Piessens. Explicitly recursive grammar combinators - Implemention of some grammar algorithms. Technical Report CW594, KULeuven CS, 2010.

[11] D. Devriese and F. Piessens. Explicitly recursive grammar combinators. In *PADL*, pages 84–98. Springer, 2011.

[12] D. Devriese and F. Piessens. Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming*, 22(06):757–796, 2012.

[13] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, pages 263–273, 1997.

[14] L. Erkök and J. Launchbury. Recursive Monadic Bindings. In *ICFP*, pages 174–185, 2000.

[15] L. Erkök and J. Launchbury. A recursive do for Haskell. In *Haskell*, pages 29–37, 2002.

[16] R. Frost, R. Hafiz, and P. Callaghan. Parser combinators for ambiguous left-recursive grammars. In *PADL*, 2008.

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2003.

[18] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.

[19] N. R. Krishnaswami and N. Benton. A semantic model for graphical user interfaces. In *ICFP*, pages 45–57, 2011.

[20] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In *MSFP*, pages 97–117, 2008.

[21] P. Ljunglöf. Pure functional parsing - an advanced tutorial. Master's thesis, Chalmers, 2002.

[22] C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(4–5):375–392, 2002.

[23] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18:1–13, January 2008.

[24] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: a functional pearl. In *ICFP*, pages 189–195, 2011.

[25] B. C. Oliveira and W. R. Cook. Functional programming with structured graphs. In *ICFP*, pages 77–88, 2012.

[26] B. C. Oliveira and A. Löh. Abstract syntax graphs for domain specific languages. In *PEPM*, 2013.

[27] R. Paterson. A new notation for arrows. In *ICFP*, pages 229–240, 2001.

[28] J. Peterson, P. Hudak, and C. Elliott. Lambda in Motion: Controlling Robots with Haskell. In *PADL*, pages 91–105, 1999.

[29] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP*, pages 50–61, 2006.

[30] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with type equality coercions. In *TLDI*, pages 53–66, 2007.

[31] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsing. In *AFP*, pages 184–207, 1996.

[32] J. Voigtländer. Free theorems involving type constructor classes: functional pearl. In *ICFP*, pages 173–184, 2009.

[33] D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. OutsideIn (X) Modular type inference with local assumptions. *J. Funct. Program.*, 1(1):1–80, 2011.

[34] P. Wadler. Theorems for free! In *FPLCA*, pages 347–359, 1989.

[35] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *ICFP*, pages 249–262, 2003.

[36] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993.