JFP **22** (6): 757–796, 2012. © Cambridge University Press 2012
doi:10.1017/S0956796812000226 First published online 10 July 2012

Finally tagless observable recursion for an abstract grammar model

DOMINIQUE DEVRIESE and FRANK PIESSENS
IBBT-Distrinet, KU Leuven, Belgium
(*e-mail:* `dominique.devriese,frank.piessens}@cs.kuleuven .be`)

Abstract

We define a finally tagless, shallow embedding of a typed grammar language. In order to avoid the limitations of traditional parser combinator libraries (no bottom-up parsing, no full grammar analysis or transformation), we require object-language recursion to be observable in the meta-language. Since existing proposals for recursive constructs are not fully satisfactory, we propose new finally tagless primitive recursive constructs to solve the problem. To do this in a well-typed way, we require considerable infrastructure, for which we reuse techniques from the multirec generic programming library. Our infrastructure allows a precise model of the complex interaction between a grammar, a parsing algorithm and a set of semantic actions. On the flip side, our approach requires the grammar author to provide a type- and value-level encoding of the grammar's domain and we can provide only a limited form of constructs like *many*. We demonstrate five meta-language grammar algorithms exploiting our model, including a grammar pretty-printer, a reachability analysis, a translation of quantified recursive constructs to the standard one and an implementation of the left-corner grammar transform. The work we present forms the basis of the `grammar-combinators` parsing library,[1] which is the first to work with a precise, shallow model of abstract context-free grammars in a classical (not dependently typed) functional language and which supports a wide range of grammar manipulation primitives. From a more general point of view, our work shows a solution to the well-studied problem of observable sharing in shallowly embedded domain-specific languages and specifically in finally tagless domain-specific languages.

1 Introduction

Parser combinator libraries are a prime example of using functional languages to embed a Domain-Specific Language (DSL) in a shallow way, i.e. reusing many facilities from the host language. Nevertheless, despite their advantages, current mainstream purely functional parser combinator libraries are not satisfactory from a parsing theory point of view. While many other parsing tools employ more powerful bottom-up parsing algorithms, parser combinators are naturally restricted to top-down algorithms. Unlike other tools, they do not employ much grammar analysis or precalculate tables; grammar authors are not provided with standard implementations of well-known grammar analysis, transformation or visualization techniques.

[1] `http://projects.haskell.org/grammar-combinators`

In this paper, we work towards the aim of functional parsing libraries, which combine the advantages of parser combinators with the power that is standard in parser generators and associated tooling. In this paper, we focus on a major problem that needs to be solved for this to happen: Lifting the limitations of the grammar model currently used by parser combinators.

It turns out that many of the limitations of purely functional parser combinator libraries are caused by the direct encoding of recursion in the grammar object language using meta-language recursion. We show that we can do better with a new encoding of primitive recursive constructs. Our constructs are parametric in the interpretation of the grammar's recursion.

Our object-language recursive constructs use the finally tagless style as described by Carette *et al.* (2009). This modelling of a parsing DSL allows grammar algorithms to interpret a grammar's production rules in the way they need to and to distinguish regular, context-free and extended context-free grammars (CFGs) in a natural way. We use an alternative to the *fix* construct of Carette *et al.*, which seems better suited for the parsing domain. The main technical challenge we face is ensuring that our constructs remain well-typed, for which we employ techniques from the multirec generic programming library (Rodriguez *et al.*, 2009). We show that our infrastructure allows for a precise and modular modelling of complex interaction between grammar, parsing algorithm and semantic actions.

A limitation of our representation of recursion is that it crucially depends on the grammar author providing a type- and value-level encoding of the grammar's domain. In addition, the encoding of some constructs becomes more difficult: We will discuss how we need to restrict the standard *many* operator (corresponding to Kleene-*) to non-terminal references to allow algorithms to interpret the constructs in the way they need. We have no detailed measurements of parsing performance for our grammars, but with current compilers, our additional indirection and our use of generic programming techniques introduce significant performance costs.

We show that our approach does bring important additional expressivity by demonstrating five well-known grammar algorithms from the parsing literature, including a grammar pretty-printer, a reachability analysis, a translation of quantified recursive constructs to the standard one and an implementation of the left-corner grammar transform. In our `grammar-combinators` library, we provide implementations of a range of other algorithms. These include an implementation of the packrat parsing algorithm (Ford, 2002) and a grammar transformation that induces a bottom-up matching order on the original grammar using a top-down parsing of the transformed grammar.

More generally, the problem of observing recursion and sharing in Embedded DSL (EDSL) terms has triggered a lot of research. Our approach presents a novel solution applied to a parsing DSL. Our solution does not compromise referential transparency, or unnecessarily force the user to resort to models of code with side effects. We keep the validation of our approach in other domains (like the typical example of hardware description DSLs) as future work. Our design also extends the set of known programming patterns for finally tagless models of DSLs.

1.1 Contributions

In this text, we make the following contributions:

- We provide evidence that observably recursive constructs are needed for full power, purely functional parsing DSLs.
- We define primitive recursive constructs enabling a shallow embedding of a purely functional grammar DSL.
- We use techniques from the multirec generic programming library in order to properly type our constructs.
- The deforestation that our infrastructure allows is shown to be a precise model of the complex interaction between grammar, parsing algorithm and semantic actions, independent from the matching order.
- We present five grammar algorithms, including the left-corner grammar transformation, showing that our encoding provides significant and important additional expressivity over traditional parser combinator libraries.

An earlier account of some of the results in this paper was presented at the 2011 Practical Aspects of Declarative Languages Conference (Devriese & Piessens, 2011). The current work has a different presentation of the material, highlighting the relation with finally tagless encodings, uses a more standard notation and type classes for applicative functor operations and presents a rationale for our techniques. The content of Section 5 was not presented at Practical Aspects of Declarative Languages Conference. It is partly new and partly appeared in a technical report (Devriese & Piessens, 2010).

1.2 Outlook

In Section 2, we introduce an example grammar, and a standard encoding of abstract parser combinators in a finally tagless style. We take a brief look at the problem of left-recursion, and then explain the problem with our standard modelling of object-language recursion using direct meta-language recursion.

We introduce our new recursive constructs in Section 3, and we show how these can be properly typed in Section 4. We present this through gradual (initially untyped) transformations of the first definition of our example grammar. This allows us to tackle technical problems one at a time and to show the rationale of our encoding.

In Section 5, we demonstrate the increased power of our grammar model with the definition of five grammar algorithms:

- In Section 5.1, we show that the recursive structure can be observed in our model by implementing a *grammar pretty-printer*.
- In Section 5.2, we show the equivalence between our final recursive constructs and an alternative encountered in Section 3. The resulting algorithm is also a useful technical aid for what follows.
- We show that our recursive constructs permit complex analyses, by implementing a *reachability analysis* in Section 5.3.

- In order to prove that our grammar model supports simple grammar transformations introducing new non-terminals, and provides tight control of the constructs allowed in a grammar, we define a *translation of quantified recursive constructs into the standard one* in Section 5.4.
- Finally, in Section 5.5, we show that our technique supports complex, realistic grammar transformations with the implementation of the standard *left-corner grammar transform.*

We discuss related work in Section 6.

Much of the Haskell code in this text relies on a set of Haskell extensions that is currently only supported by the Glasgow Haskell Compiler (GHC).[2] However, these are all well-accepted extensions that do not make type-checking undecidable. Our library optionally supports the use of Template Haskell (Sheard & Peyton Jones, 2002) for performing grammar transformations at compile-time.

2 Finally tagless parser combinators

2.1 *Arithmetic expressions*

We start our presentation with a standard example from the parser literature: A simple grammar describing arithmetic expressions of the form "$(6 * (4 + 2)) + 6$", in a formalism similar to Extended Backus–Naur Form ((E)BNF) (Aho *et al.*, 2006, Section 2.2).

```
Line    → Expr EOF
Expr    → Expr '+' Term
        → Term
Term    → Term '*' Factor
        → Factor
Factor  → '(' Expr ')'
        → Digit+
Digit   → '0' | '1' | '2' | ... | '8' | '9'
```

The definitions of Expr and Term are such that "$a+b*c$" can only be interpreted as "$a+(b*c)$" and "$a+b+c$" only as "$(a+b)+c$". This modelling of operator precedence and left-associativity is idiomatic for LR-style grammars, but fundamentally relies on left-recursion: one of the productions of non-terminal Expr, for example, refers back to Expr in the first position.

In order to obtain a parser for this grammar (without manually writing it ourselves), parser generators like Yacc (Johnson, 1979) and ANTLR (Parr & Quong, 1995) are typically used to translate the grammar (provided in an ((E)BNF)-like formalism) into source code in the developer's programming language. This technique has proven successful in practice, but suffers from various downsides: little assurance for syntax- and type-correctness of generated code, little reuse of the developer's existing programming environment (editor, type-checker, build

[2] TypeFamilies, GADTs, MultiParamTypeClasses, FunctionalDependencies, FlexibleContexts and RankNTypes.

system etc.), limited support for abstraction, no support for grammars constructed at runtime etc.

2.2 *Finally tagless parser combinators*

Parser combinator libraries provide an elegant alternative, modelling the grammar directly in a general-purpose programming language. These libraries treat parsers as first-class values that can be combined, extended, reused, abstracted from etc. Many of these libraries only provide a single parsing algorithm, even though there is no immediate technical reason for this. In this section, we define a more abstract grammar model, without this coupling. This abstraction also saves us the trouble of actually explaining a concrete parsing algorithm; our abstract model can be used with many of the well-known parser combinator libraries (e.g. uu-parsinglib (Swierstra, 2009) or Parsec (Leijen & Meijer 2001)).

The technique we use to achieve this decoupling has been already used by Swierstra and Duponcheel (1996), and has been described and popularised by Carette *et al.* (2009) as the finally tagless modelling of DSLs. In this style, we define our grammars abstractly over a parsing algorithm with parser types $p\ a$, where p has instances for a set of type classes containing primitive parsing operators. The type constructor p is parameterised by the type of parsing results.

We define the necessary primitive parser operators in a type class: *CharProductionRule*. Since parser combinators were a motivating example for the development of the concept of applicative functors (McBride & Paterson, 2008) and the type classes *Applicative* and *Alternative* (repeated below), it is no coincidence that these map perfectly to our needs.[3] Standard applicative functor laws apply, but they may only be valid *morally* in some of our examples (e.g. for the pretty-printer in Section 5.1: equivalent expressions might be pretty-printed in different but equivalent ways).

```
class (Functor f) ⇒ Applicative f where
  pure :: a → f a
  (⊛) :: f (a → b) → f a → f b
class Applicative f ⇒ Alternative f where
  empty :: f a
  (⦶) :: f a → f a → f a

class (Alternative p) ⇒ CharProductionRule p where
  endOfInput :: p ()
  token      :: Char → p Char
```

In our setting, the *Applicative* operator ⊛ consecutively applies two given parsers. It produces a parsing result by applying the first parser's result to that of the second parser. The *pure* primitive parser matches the empty string, producing the value

[3] In the `grammar-combinators` library, we can unfortunately not use the *Applicative* or *Alternative* type classes due to a technical reason related to an advanced feature that we do not discuss in this text (Template Haskell lifting of grammars). The library also uses a different notation for the applicative operators, for historical reasons.

$$\begin{array}{ll}
(\circledS) :: \mathit{Functor}\ f \Rightarrow (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b & (\mathbin{\circledast\!>}) :: \mathit{Applicative}\ f \Rightarrow f\ a \rightarrow f\ b \rightarrow f\ b \\
(\circledS) = \mathit{fmap} & ma \mathbin{\circledast\!>} mb = \mathit{flip}\ \mathit{const} \circledS ma \circledast mb \\
(\mathbin{<\!\circledast}) :: \mathit{Applicative}\ f \Rightarrow f\ a \rightarrow f\ b \rightarrow f\ a & (\mathbin{<\!\circledS}) :: \mathit{Functor}\ f \Rightarrow a \rightarrow f\ b \rightarrow f\ a \\
ma \mathbin{<\!\circledast} mb = \mathit{const} \circledS ma \circledast mb & f \mathbin{<\!\circledS} m = \mathit{const}\ f \circledS m
\end{array}$$

Fig. 1. Definitions of standard related and derived *Applicative* operators $\circledS$, $\mathbin{<\!\circledast}$, $\mathbin{\circledast\!>}$ and $\mathbin{<\!\circledS}$.

provided as argument as its parsing result. The *Alternative* disjunction operator $\obar$ models a choice between two parsers producing the same result type and returns the result of the parser that matched. The *Alternative empty* primitive parser never matches anything, and can therefore return an arbitrary result type.

We will use only two operators that are specific to the parsing domain: the *endOfInput* and *token* parsers. The first matches the end of the input string, returning a unit result and the latter matches a single, specified character in the input stream and returns it on success.

Figure 1 shows the definitions of standard related and derived applicative operators $\circledS$, $\mathbin{<\!\circledast}$, $\mathbin{\circledast\!>}$ and $\mathbin{<\!\circledS}$, which respectively apply a given function to the result of a rule, ignore a sequenced rule's result and replace a rule's result with a given value. We temporarily omit operators *many* and *some*, which apply a given parser any (resp. any non-zero) number of times, but we come back to them in Section 3.3. Note that $\circledS$ is a synonym for *fmap* in the *Functor* type class, and for *Applicative* functors, it is required to satisfy the (defining) property $\mathit{fmap}\ f\ m = \mathit{pure}\ f \circledast m$. In the rest of the paper we will consistently regard it as a derived operator and not discuss its instances.

The gist of the finally tagless technique is visible in the type signature of the parsing functions[4]:

$$\begin{array}{l}
\mathit{line}, \mathit{expr}, \mathit{term}, \mathit{factor} :: (\mathit{CharProductionRule}\ p) \Rightarrow p\ \mathit{Integer} \\
\mathit{digit} :: (\mathit{CharProductionRule}\ p) \Rightarrow p\ \mathit{Char}
\end{array}$$

The functions are defined abstractly over any type constructor p, which is an instance of the class *CharProductionRule*. Because of this constrained universal quantification over p, parametricity ensures that these functions can only construct values of type $p\ a$ through the primitive operators defined in class *CharProductionRule* and its parent classes. The finally tagless style allows us to define more general or more restricted typeclasses, so we can extend or restrict the primitive constructs which the functions have access to and mix and match as suited. This is a facility we will exploit later on, for example to make the distinction between extended and normal CFGs.

The functions can be defined as follows in terms of the primitive constructs:

$$\begin{array}{ll}
\mathit{line} & = \mathit{expr} \mathbin{<\!\circledast} \mathit{endOfInput} \\
\mathit{expr} & = (+) \circledS \mathit{expr} \mathbin{<\!\circledast} \mathit{token}\ \texttt{'+'} \circledast \mathit{term} \\
 & \obar\ \mathit{term}
\end{array}$$

[4] The parsing functions all return the calculated *Integer* value of matches, except for *digit* which just returns a *Char*. It would be slightly cleaner to make *digit* also return a numeric value, but this would be a bit more verbose throughout the text.

```
term   = (*) <$> term <* token '*' <*> factor
         <|> factor
factor = read <$> some digit
         <|> token '(' *> expr <* token ')'
digit  = token '0' <|> token '1' <|> ... <|> token '9'
```

For every non-terminal, a grammar function is defined directly as a Haskell value using the primitive parsing and combinator operators from the *CharProductionRule* type class and its parents. The definitions look fairly standard for an applicative parser combinator library, even though they are in fact abstract over the parsing algorithm used. The code is fairly concise and reasonably close to the original grammar.

Note that the above definitions of the parser functions incorporate semantic actions; all parsers return the semantic value of the non-terminal they represent: the integer or char value of the matched string. We consider this coupling of grammar and semantics non-ideal and we will come back to this in Section 4.4.

2.3 Left-recursion

Readers familiar with parser combinator libraries will however have noticed an important problem in the above code. With a mainstream applicative parser combinator library like uu-parsinglib, it does not actually work. The problem is caused by the left-recursion in the definition: *expr*, *term* and *factor* all immediately refer to themselves in the leftmost position of one of their alternatives. A simple top-down parsing algorithm asked to parse an *expr*, would at some point try to match the first alternative for *expr*. The first thing it then needs is a parse of *expr* at the location where it just started looking for an *expr*. Less naive parser combinator libraries exist that can handle left recursion to a certain extent during top-down parsing (Frost *et al.*, 2008; Danielsson & Norell 2010; Might *et al.*, 2011). However, other libraries like uu-parsinglib and Parsec require the programmer to manually transform the grammar to a non-left-recursive form:

```
line, expr, term, factor :: (CharProductionRule p) => p Integer
exprTail, termTail :: (CharProductionRule p) => p (Integer -> Integer)
digit :: (CharProductionRule p) => p Char
line     =                   expr <* endOfInput
expr     = foldr ($) <$>     term <*> many exprTail
exprTail = (+) <$            token '+' <*> term
term     = foldr ($) <$>     factor <*> many termTail
termTail = (*) <$            token '*' <*> factor
factor   = read <$>          some digit
           <|>               token '(' *> expr <* token ')'
digit    =                   token '0' <|> token '1' <|> ... <|> token '9'
```

This transformed version of the grammar uses an alternative modelling of operator precedence and associativity that does not rely on left-recursion and can be used with naive top-down parsing algorithms. In fact, standard combinators exist (e.g.

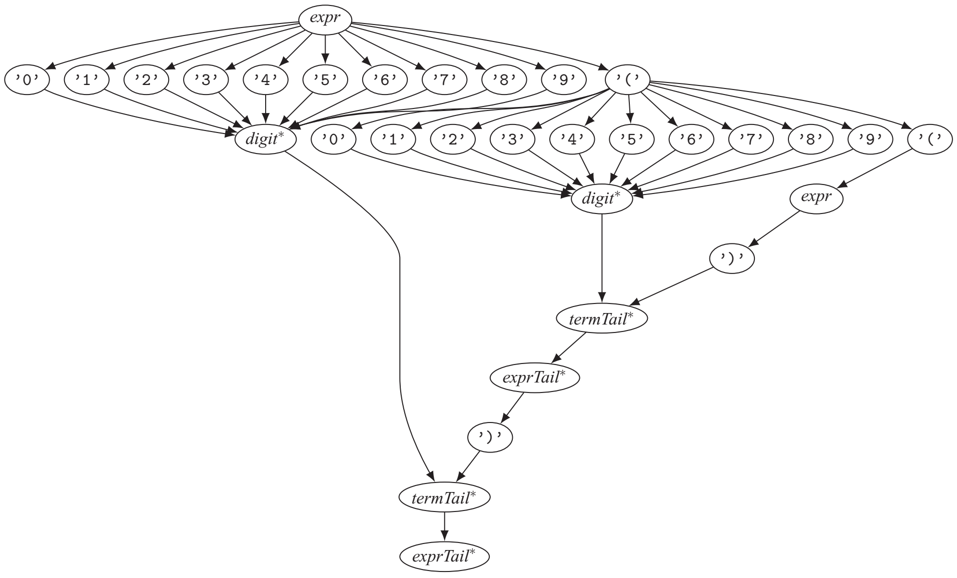


Fig. 2. A graphical representation of the *expr* parser after some expansions of its definition (see Section 2.4). The *expr* node at the right (as well as some of the other nodes) can be expanded further, arbitrarily deep.

pChainL in uu-parsinglib), which implement this pattern generically. But even with these combinators, properly identifying and dealing with left recursion remains the responsibility of the programmer.

There is however a more fundamental problem with the grammar model we have defined so far.

2.4 ω-regular grammars considered harmful

The problem lies in the modelling of recursion between non-terminals using recursively defined Haskell values. Haskell supports this, thanks to its call-by-need (lazy) evaluation strategy. At first sight, it seems that this allows a faithful representation of the recursive structure of the original grammar. However, closer inspection reveals that what the Haskell values represent is in fact not so much a graph as an infinite tree. We can see this by considering, for example, the most recent definition of the *expr* parser function. Because of Haskell's purely functional nature (Sabry, 1998), *expr* is observationally equivalent to what we get if we expand it to its definition, and likewise if we expand subexpressions to their definitions (highlighting the term being expanded in each step):

$$\begin{aligned}
\mathit{expr} &\equiv \mathit{foldr}\ (\$) \circledS\ \boldsymbol{term} && \circledast\ \mathit{many\ exprTail}\\
&\equiv \mathit{foldr}\ (\$) \circledS (\mathit{foldr}\ (\$) \circledS\ \boldsymbol{factor} \circledast \mathit{many\ termTail}) && \circledast\ \mathit{many\ exprTail}\\
&\equiv \mathit{foldr}\ (\$) \circledS (\mathit{foldr}\ (\$) \circledS (\mathit{read} \circledS \mathit{some\ digit} \mathbin{\bigcirc\!\!\!\!\mid}\ \mathit{pSym}\ \texttt{'('} \mathbin{\ast\!\!>} \boldsymbol{expr} \mathbin{<\!\!\ast} \mathit{pSym}\ \texttt{')'})\\
&\qquad\qquad\qquad\qquad\qquad\qquad \circledast\ \mathit{many\ termTail}) && \circledast\ \mathit{many\ exprTail}
\end{aligned}$$

Figure 2 shows a graphical representation of the *expr* parser after some further expansions.

In this way, we find an expansion of the definition of *expr* containing *expr* itself as a subexpression. We can continue expanding forever, obtaining an infinite number of expanded expressions, growing in size, and each indistinguishable from the original definition of *expr*. In fact, for any n, it is even possible to construct a different expression that cannot be distinguished from the original in less than n evaluation steps: take the original definition of *expr*, perform $n + 1$ expansions and then make a change in the result of the final expansion.

This observation has very real practical consequences. A parser library working with our parser definitions (or those in most parser combinator libraries, which model recursion the same way), and respecting referential transparency (see Section 6.3), is fundamentally limited. It cannot, for example, print a representation of the grammar in any finite number of evaluation steps n, because it might be looking at another grammar that can only be distinguished from the original after more than n computation steps. Similarly, no parsing library using this grammar model can calculate parsing tables completely upfront, fully execute a grammar transformation or perform a sanity check for LL(1)-ness.

Because of the similarity of these "infinite-tree" grammar definitions to what one might see as infinite regular grammars, we will refer to this grammar model as ω-regular.[5]

3 A different modelling of recursion

These fundamental limitations are in fact an instance of a more general problem. For many DSLs, object language terms feature a mutually recursive structure and it is often advantageous to be able to observe this structure in the meta-language. For example, Sheard (2005) cites the problem as one of the main reasons to build a special-purpose hardware design language instead of embedding the DSL in a general-purpose programming language.

So, what could be a better way to represent recursion? In this section and the next, we first consider the *fix* construct, which Carette *et al.* (2009) use in their finally tagless modelling of a typed lambda calculus, and show that it is not perfect for our needs. Next, we present our approach, which we introduce step by step. We incrementally transform the parser combinators grammar introduced before, at first postponing well-typedness concerns until we are ready to show the solution for this. We will clearly mark all untyped pseudo-code as such in what follows.

3.1 Fixing recursion?

In their finally tagless model of a typed lambda calculus, Carette *et al.* (2009) use a *fix* construct to model recursion. In our setting, such a construct would resemble the following:

[5] Our usage of the term ω-regular grammars is related to, but not the same as other usages in the literature. For some insights from language theory, we refer to Park (1981), who proves a relation between functions using $*$ and $\dagger$ operators and minimal and maximal fixpoints. In those terms, what we call ω-regular grammars correspond to expressions generated by the regular operators with $*$ replaced by $\dagger$.

```
class FixProductionRule p where
  fix :: (p v → p v) → p v
```

Recursive production rules could then be defined as follows:

```
expr :: (CharProductionRule p, FixProductionRule p) ⇒ p Integer
expr = fix $ λself →     (+) ⓢ self <* token '+' ⊛ term
                       ⦶ term
```

This *fix* operator employs a finally tagless style and effectively makes object language recursion observable in the meta-language, allowing meta-language algorithms to interpret the recursion in the way that they need to. However, if we consider the above definition more closely, it turns out that we missed a recursive occurrence of *expr*. Indeed, the grammar is *mutually* recursive, with *term* referencing *factor*, and *factor* referencing *expr* again. Indeed, what we require is an encoding of this mutual recursion, allowing us to model the combined fixpoint of the following functions. The example uses an omitted primitive *fix3*, which can be defined in terms of *fix*:

```
expr, term, factor :: CharProductionRule p ⇒
  p Integer → p Integer → p Integer → p Integer
digit :: (CharProductionRule p) ⇒ p Char
expr   e t f = (+) ⓢ e <* token '+' ⊛ t
               ⦶ term e t f
term   e t f = (*) ⓢ t <* token '*' ⊛ f
               ⦶ factor e t f
factor e t f = read ⓢ some digit
               ⦶ token '(' *> e <* token ')'
digit        = token '0' ⦶ token '1' ⦶ ... ⦶ token '9'
line' :: (FixProductionRule p, CharProductionRule p) ⇒ p Integer
line' = e <* endOfInput where (e, t, f) = fix₃ expr term factor
```

This approach seems successful, although the syntax is somewhat verbose. In a typical lambda calculus, we could make it more concise by defining a object-language record type containing three fields *expr*, *term* and *factor*. We could then construct a value of that record type as the fixpoint of a single function based on the above functions. However, to do this, we need record types in our object language, and adding them to our parsing DSL purely for this technical reason is not our preferred solution. We conjecture that the above syntax can also be made more concise with a superficial Haskell extension similar to the recursive do notation by Erkök and Launchbury (2002). In fact, we think that our *FixProductionRule* type class can be seen more generally as an analogon to their *MonadRec* (later renamed to *MonadFix*) for applicative functors.

However, we can also choose a different way to define a mutual recursion construct: We model the record suggested above as a function from a finite domain to production rules:

warning: untyped pseudo-code...

```
data Domain = Line | Expr | Term | Factor | Digit
type Grammar = Domain → p ?
class FixGramProductionRule p where
  fixG :: ((dom → p ?) → (dom → p ?)) → dom → p ?
```

It turns out that this idea can be elaborated to a workable solution; there are ways to properly type this *fixG* construct (we will encounter such techniques further on). However, this *fixG* construct deviates from standard practice in grammar definitions, since it can be used multiple times in different locations in the same grammar. In standard CFG formalisms, all the recursion occurs at the top level. Our proposal therefore does not introduce an actual *fix* construct, but instead we model the grammar as the function of which it is the fixed point.

3.2 Towards context-free grammars

This idea corresponds to a classic technique from the functional programmer's bag of tricks: defining the grammar with open recursion. We factor out all recursive calls in the definition by calls to a *self* function that it receives as an argument.

warning: untyped pseudo-code...

```
g_arith :: (CharProductionRule p) ⇒ (Domain → p ?) → Domain → p ?
g_arith self Line  = self Expr ⊛ endOfInput
g_arith self Expr  = self Expr ⊛ token '+' ⊛ self Term
                   ⫶ self Term
  ...
```

Even though it is not clear how to type this solution, this model does effectively solve the problem of unobservable recursion. Algorithms working with the grammar can provide custom interpretations of recursion to suit their needs. However, the *self* parameter obscures the definition while its role is fairly technical and as a model of a primitive object language recursion primitive, it stylistically differs from the other object language primitives which are defined in type classes.

A more *finally tagless* recursion primitive can be defined by replacing the *self* parameter by a primitive ⟨·⟩, defined in an additional type class *RecProductionRule*. Production rule types *p* become linked to the domain for which recursive calls are allowed, which we reflect in the *RecProductionRule* class's parameters and its functional dependencies:

warning: untyped pseudo-code...

```
class (CharProductionRule p) ⇒ RecProductionRule p dom | p → dom where
  ⟨·⟩ :: dom → p ?
g_arith :: (RecProductionRule p Domain) ⇒ Domain → p ?
g_arith Line  = ⟨Expr⟩ ⊛ endOfInput
g_arith Expr  = ⟨Expr⟩ ⊛ token '+' ⊛ ⟨Term⟩
              ⫶ ⟨Term⟩
  ...
```

This modelling is equivalent to the one using open recursion using the *self* parameter. In fact, we will define a translation algorithm in Section 5.2, turning the representation using *RecProductionRule* into the one using a *self* parameter. This algorithm will be useful for technical reasons.

3.3 Extended context-free grammars

There is actually one thing still missing in this definition: We have craftily hidden the use of the *some* operator in the production rule for *Factor* by including it in the ellipsis above:

warning: untyped pseudo-code...

$$
\begin{aligned}
&g_{arith}\ \mathit{Factor} = \mathit{token}\ \text{'('} \circledast \langle \mathit{Expr} \rangle \circledast \mathit{token}\ \text{')'}\\
&\qquad\qquad \oplus\ \mathit{some}\ \langle \mathit{Digit} \rangle
\end{aligned}
$$

The *some* operator is defined as follows (together with its sibling *many*):

$$
\begin{aligned}
&\mathit{many}, \mathit{some} :: (\mathit{CharProductionRule}\ p) \Rightarrow p\ a \rightarrow p\ [a]\\
&\mathit{many}\ p = \mathit{pure}\ [\,]\ \oplus\ \mathit{some}\ p\\
&\mathit{some}\ p\ \ = (:)\ \circledS\ p \circledast \mathit{many}\ p
\end{aligned}
$$

These definitions essentially rely on unobservable meta-language recursion, which we need to replace with an observable form of recursion as well. In addition to the $\langle\cdot\rangle$ primitive recursion operator *RecProductionRule* type class, we define restricted versions of *many* and *some* in the *LoopProductionRule* type class, as follows:

warning: untyped pseudo-code...

$$
\begin{aligned}
&\textbf{class}\ (\mathit{RecProductionRule}\ p\ \mathit{dom}) \Rightarrow \mathit{LoopProductionRule}\ p\ \mathit{dom} \mid p \rightarrow \mathit{dom}\ \textbf{where}\\
&\quad \langle\cdot\rangle^{*} :: \mathit{dom} \rightarrow p\ \textbf{?}\\
&\quad \langle\cdot\rangle^{+} :: \mathit{dom} \rightarrow p\ \textbf{?}
\end{aligned}
$$

The grammar type and the production rules for factor now become

warning: untyped pseudo-code...

$$
\begin{aligned}
&g_{arith} :: (\mathit{LoopProductionRule}\ p\ \mathit{Domain}) \Rightarrow \mathit{Domain} \rightarrow p\ \textbf{?}\\
&\ \ldots\\
&g_{arith}\ \mathit{Factor} = \mathit{token}\ \text{'('} \circledast \langle \mathit{Expr} \rangle \circledast \mathit{token}\ \text{')'}\\
&\qquad\qquad \oplus\ \langle \mathit{Digit} \rangle^{+}\\
&\ \ldots
\end{aligned}
$$

Note that operators $\langle\cdot\rangle^{*}$ and $\langle\cdot\rangle^{+}$ are less powerful than the *many* and *some* operators, which allow any production rule (not just recursive references) to be quantified. This restriction is needed to make it possible for grammar algorithms to interpret these object-language constructs appropriately. However, we think that the new constructs are still general enough for most purposes. Grammar authors may sometimes need to split out a production rule to be quantified into an additional non-terminal.

Note also that we are in fact replacing library algorithms (*many* and *some*) by what are essentially new built-in operators in our object language. This is unfortunate, but it is a part of the cost we pay in our approach to rule out ω-regular grammars. We will show in Section 5.4 that $\langle\cdot\rangle^*$ and $\langle\cdot\rangle^+$ support standard grammar transformations.

3.4 Typing context-free grammars

So this last representation is promising in the sense that algorithms can instantiate the abstract grammar with their own interpretations of recursion and the primitive parser combinator operations. Unfortunately, it is not clear how to properly define the value that each of the production rules produce. In a typical implementation, our example grammar above produces result values of the following Abstract Syntax Tree (AST) type:

```
newtype Line  = SExpr Expr
data Expr     = Sum Expr Term
              | STerm Term
data Term     = Product Term Factor
              | SFactor Factor
data Factor   = Paren Expr
              | Number [Digit]
newtype Digit = MkDigit Char
```

When we now try to define the type of g_{arith}, we run into another problem. It turns out that our modelling of the grammar as a function from the grammar domain to production rules forces all production rules to produce the same type of values:

```
                    warning: untyped pseudo-code...
g_arith :: (CharProductionRule p) ⇒ Domain → p ?
```

Similarly, if we try to define the type of the $\langle\cdot\rangle$ operator, we cannot express that the parser result of $\langle idx\rangle$ should vary based on the value of *idx*.

```
                    warning: untyped pseudo-code...
class (CharProductionRule p) ⇒ RecProductionRule p dom | p → dom where
   ⟨·⟩ :: dom → p ?
g_arith Line = SExpr ⓢ ⟨Expr⟩ ⊛ endOfInput
g_arith Expr = Sum   ⓢ ⟨Expr⟩ ⊛ token '+' ⊛ ⟨Term⟩
             ⦶ STerm ⓢ ⟨Term⟩
 ...
```

The essential problem here is that all our non-terminals are of type *Domain* so that all references $\langle idx\rangle$ must share a single result type (because Haskell is not dependently typed, see Section 6.2.2). Therefore, we cannot express that non-terminal *Line* corresponds to a different type of semantic values than non-terminal *Expr*.

4 Typing our recursion model

It turns out that we can define precise types for the untyped pseudo-code in the previous section by using a representation of non-terminals not sharing a single type.

4.1 Representing non-terminals

We model the set of non-terminals (the *domain*) as a "subkind" with proof terms, using the technique employed by Rodriguez *et al.* (2009) to model indices into a set of mutually recursive data types in multirec. The generalized algebraic data type (GADT) (Peyton Jones *et al.*, 2006) ϕ_{arith} is a "subkind" that represents the domain of our arithmetic expressions grammar. Note that Haskell's separation between type and value name spaces allows the data constructor *Expr* and the type *Expr* to share the same name.

$$
\begin{array}{lll}
\textbf{data}\ \phi_{arith}\ ix\ \textbf{where} & Line & ::\ \phi_{arith}\ Line \\
 & Expr & ::\ \phi_{arith}\ Expr \\
 & Term & ::\ \phi_{arith}\ Term \\
 & Factor & ::\ \phi_{arith}\ Factor \\
 & Digit & ::\ \phi_{arith}\ Digit
\end{array}
$$

We use the previously defined AST types *Line*, *Expr*, *Term*, *Factor* and *Digit* to represent the non-terminals at the type-level. The GADT ϕ_{arith} introduces, for every non-terminal ix, a term of type $\phi_{arith}\ ix$, serving as a proof that ix is part of the domain ϕ_{arith}. With this "subkind" representation, the compiler will guarantee that a function f typed $\forall ix\ .\ \phi\ ix \rightarrow \ldots$ is polymorphic over precisely the five non-terminal types in the domain.

4.2 A first typing of our grammars and the recursion operator $\langle\cdot\rangle$

This representation of our domain as a subkind with proof terms allows us to present a first proper typing of our grammars and the recursion operator $\langle\cdot\rangle$, which we introduced as untyped pseudo-code before.

We first consider the primitive recursion construct $\langle\cdot\rangle$, defined in the *RecProductionRule* type class. Within a grammar with domain ϕ (e.g. type constructor ϕ_{arith} above), we can now declare that $\langle\cdot\rangle$ can be invoked on any value idx of type $\phi\ ix$ for some ix. The expression $\langle idx\rangle$ represents a parser for that non-terminal, and returns a value of type ix: the AST type of the non-terminal. The constructs $\langle\cdot\rangle^{*}$ and $\langle\cdot\rangle^{+}$ are defined analogously in the *LoopProductionRule* type class. We use functional dependencies to couple production rules with the domain for which recursive references can be made.

$$
\begin{array}{l}
\textbf{class}\ (CharProductionRule\ p) \Rightarrow RecProductionRule\ p\ \phi\ |\ p \rightarrow \phi\ \textbf{where} \\
\quad \langle\cdot\rangle\quad ::\ \phi\ ix \rightarrow p\ ix \\
\textbf{class}\ (RecProductionRule\ p\ \phi) \Rightarrow LoopProductionRule\ p\ \phi\ |\ p \rightarrow \phi\ \textbf{where}
\end{array}
$$

```
⟨·⟩*  :: φ ix → p [ix]
⟨·⟩+  :: φ ix → p [ix]

g_arith :: (LoopProductionRule p φ_arith) ⇒ φ_arith ix → p ix
g_arith Line   = SExpr   Ⓢ ⟨Expr⟩ <⊛ endOfInput
g_arith Expr   = Sum     Ⓢ ⟨Expr⟩ <⊛ token '+' ⊛ ⟨Term⟩
               ⦶ STerm   Ⓢ ⟨Term⟩
g_arith Term   = Product Ⓢ ⟨Term⟩ <⊛ token '*' ⊛ ⟨Factor⟩
               ⦶ SFactor Ⓢ ⟨Factor⟩
g_arith Factor = Paren   <Ⓢ token '(' ⊛ ⟨Expr⟩ <⊛ token ')'
               ⦶ Number  Ⓢ ⟨Digit⟩+
g_arith Digit  = MkDigit Ⓢ (token '0' ⦶ token '1' ⦶ ... ⦶ token '9')
```

4.3 Semantic value families

With this typed version of our grammars, we are making good progress, but this representation of recursion in our object language is still not fully satisfactory. The problem is in the result types of the recursive calls. Algorithms are now free to plug in their own interpretation of object-language recursion, but these are still forced to work with the full AST types as result types of the recursive calls. In many cases, we want to be able to plug in different representation types, often a different type for every non-terminal.

We can make this more concrete for the example of our grammar language. There, the AST result types might at first sight seem acceptable, since conceptually the AST is proper to the grammar, and practically we can apply any set of semantic actions once we have the AST by implementing them as a structural fold (a *catamorphism*) over the AST. However, this approach also allows semantics that are not formulated as such catamorphisms. For example, the following semantics negates all literals that are inside an uneven number of parentheses:

```
weirdSem :: φ_arith ix → ix → Int
weirdSem idx v = go False idx v
  where unMkDigit (MkDigit c) = c
        neg :: Bool → Int → Int
        neg True x = −x
        neg False x = x
        go :: Bool → φ_arith ix → ix → Int
        go inv Line (SExpr e) = go inv Expr e
        go inv Expr (STerm t) = go inv Term t
        go inv Expr (Sum e t) = go inv Expr e + go inv Term t
        go inv Term (SFactor f) = go inv Factor f
        go inv Term (Product t f) = go inv Term t * go inv Factor f
        go inv Factor (Paren e)  = go (¬ inv) Expr e
        go inv Factor (Number n) = neg inv $ read $ map unMkDigit n
        go _ Digit d = read (unMkDigit d : [])
```

A disadvantage of semantics like *weirdSem* (which are not formulated as catamorphisms) is that they are inherently coupled to a top-down matching order: the semantics has to be applied to the top AST node once it is available. A bottom-up parser already reduces production rules before it is sure at which depth the production will fit in the final AST and it might want to force the semantics to be already applied at such times during parsing, e.g. for optimization purposes.[6] However, this is inherently not possible for semantics like *weirdSem*, whose behaviour depends on the depth of the match in the final AST. Also for semantic reasons, we find it preferable to define grammar semantics as catamorphisms over ASTs and exclude definitions like *weirdSem*.

We can achieve this by abstracting our model even further, this time over *semantic value families*. These are data families (Schrijvers *et al.*, 2008) indexed by the non-terminal types that we have seen before. A semantic value family r associates each non-terminal type ix with the type of its semantic value $r\ ix$. We define one such family for the ϕ_{arith} domain, written $[\![]\!]^{value}_{\cdot}$. Note that the dot in this notation is a placeholder for the type ix.

$$
\begin{array}{llcl}
\textbf{data family } [\![]\!]^{value}_{\cdot}\ ix & & & \\
\textbf{newtype instance } [\![]\!]^{value}_{\cdot}\ Line & & = & [\![\cdot]\!]^{value}_{Line}\ Integer \\
\textbf{newtype instance } [\![]\!]^{value}_{\cdot}\ Expr & & = & [\![\cdot]\!]^{value}_{Expr}\ Integer \\
\textbf{newtype instance } [\![]\!]^{value}_{\cdot}\ Term & & = & [\![\cdot]\!]^{value}_{Term}\ Integer \\
\textbf{newtype instance } [\![]\!]^{value}_{\cdot}\ Factor & & = & [\![\cdot]\!]^{value}_{Factor}\ Integer \\
\textbf{newtype instance } [\![]\!]^{value}_{\cdot}\ Digit & & = & [\![\cdot]\!]^{value}_{Decimal}\ Char
\end{array}
$$

This semantic value family specifies that all of our non-terminals have *Integer* semantic values (their calculated value) except for *Digit*, which has a character as its semantic value.

We can now redefine the primitive recursion operator to return values of some semantic value family r, which (like the domain ϕ) is required to be the same throughout the grammar by the *RecProductionRule* type class's functional dependencies. Note that we provide default definitions of operators $\langle\cdot\rangle^{*}$ and $\langle\cdot\rangle^{+}$ in terms of each other and $\langle\cdot\rangle$ operator. However, we expect instances of *LoopProductionRule* to provide custom definitions of at least one of the two operators, otherwise they will behave as their ω-regular analogs.

$$
\begin{array}{l}
\textbf{class}\ (CharProductionRule\ p) \Rightarrow RecProductionRule\ p\ \phi\ r \mid p \rightarrow \phi, p \rightarrow r\ \textbf{where} \\
\quad \langle\cdot\rangle :: \phi\ ix \rightarrow p\ (r\ ix) \\
\textbf{class}\ (RecProductionRule\ p\ \phi\ r) \Rightarrow LoopProductionRule\ p\ \phi\ r \mid p \rightarrow \phi, p \rightarrow r\ \textbf{where} \\
\quad \langle\cdot\rangle^{*}\ \ :: \phi\ ix \rightarrow p\ [r\ ix] \\
\quad \langle idx\rangle^{*} = pure\ []\ \obar\ \langle idx\rangle^{+} \\
\quad \langle\cdot\rangle^{+}\ :: \phi\ ix \rightarrow p\ [r\ ix] \\
\quad \langle idx\rangle^{+} = (:)\ \textcircled{\$}\ \langle idx\rangle \circledast \langle idx\rangle^{*}
\end{array}
$$

[6] Such a parser would typically use Haskell's *seq* function to force the semantics to actually be evaluated at such moments.

In the next section, we will show how this definition allows us to decouple the grammar from a semantic value family. Here we can already show how we can use the new definition of $\langle\cdot\rangle$ to make the grammar work for the family $[\![]\!]^{value}_{\cdot}$. Like before, the mixing of semantic values in the grammar hampers the grammar's readability:

$$\begin{array}{lll}
g_{arith} :: (LoopProductionRule\ p\ \phi_{arith}\ [\![]\!]^{value}_{\cdot}) \Rightarrow \phi_{arith}\ ix \rightarrow p\ [\![]\!]^{value}_{ix} & & \\
g_{arith}\ Line & = (\lambda [\![v]\!]^{value}_{Expr} \rightarrow [\![v]\!]^{value}_{Line}) & \circledS\ \langle Expr\rangle \mathbin{<\!*}\ endOfInput \\
g_{arith}\ Expr & = (\lambda [\![v_1]\!]^{value}_{Expr}\ [\![v_2]\!]^{value}_{Term} \rightarrow [\![v_1 + v_2]\!]^{value}_{Expr}) & \circledS\ \langle Expr\rangle \mathbin{<\!*}\ token\ \texttt{'+'} \circledast \langle Term\rangle \\
 & \ \ \varobar\ (\lambda [\![v]\!]^{value}_{Term} \rightarrow [\![v]\!]^{value}_{Expr}) & \circledS\ \langle Term\rangle \\
g_{arith}\ Term & = (\lambda [\![v_1]\!]^{value}_{Term}\ [\![v_2]\!]^{value}_{Factor} \rightarrow [\![v_1 * v_2]\!]^{value}_{Term}) & \circledS\ \langle Term\rangle \mathbin{<\!*}\ token\ \texttt{'*'} \circledast \langle Factor\rangle \\
 & \ \ \varobar\ (\lambda [\![v]\!]^{value}_{Factor} \rightarrow [\![v]\!]^{value}_{Term}) & \circledS\ \langle Factor\rangle \\
g_{arith}\ Factor & = (\lambda [\![v]\!]^{value}_{Expr} \rightarrow [\![v]\!]^{value}_{Factor}) & \mathbin{<\!\$}\ token\ \texttt{'('} \circledast \langle Expr\rangle \mathbin{<\!*}\ token\ \texttt{')'} \\
 & \ \ \varobar\ ([\![\cdot]\!]^{value}_{Factor} \circ read \circ map\ (\lambda [\![c]\!]^{value}_{Decimal} \rightarrow c)) & \circledS\ \langle Digit\rangle^{+} \\
g_{arith}\ Digit & = [\![\cdot]\!]^{value}_{Decimal} & \circledS \\
 & \qquad (token\ \texttt{'0'} \varobar token\ \texttt{'1'} \varobar \ldots \varobar token\ \texttt{'9'}) &
\end{array}$$

4.4 Semantic value family polymorphism

So, the next question is: Can we decouple the grammar from its semantics? Clearly, there are other than aesthetic reasons for doing this. For our arithmetic expressions, we have already seen a grammar producing AST values and a grammar calculating integer values for them. Other useful semantic processors (a set of semantic actions for non-terminals in a grammar) transform the same expressions into reverse polish notation, construct an AST or perform some form of side effects in a *Monad*. It is clear that we can improve the modularity of our grammar language by decoupling a grammar from sets of semantic actions.

Our solution for this decoupling here uses (again) techniques from the multirec generic programming library (Rodriguez *et al.*, 2009), which uses a representation of mutually recursive data types as the fixed point of a *pattern functor* to manipulate them in generic algorithms. The AST data types shown previously are an example of such a family of mutually recursive data types, and the following is its pattern functor:

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathbf{data}\ PF_{arith}\ r\ ix\ \mathbf{where}} \\
\quad SExprF & :: r\ Expr & \rightarrow & & & PF_{arith}\ r\ Line \\
\quad SumF & :: r\ Expr & \rightarrow & r\ Term & \rightarrow & PF_{arith}\ r\ Expr \\
\quad STermF & :: r\ Term & \rightarrow & & & PF_{arith}\ r\ Expr \\
\quad ProductF & :: r\ Term & \rightarrow & r\ Factor & \rightarrow & PF_{arith}\ r\ Term \\
\quad SFactorF & :: r\ Factor & \rightarrow & & & PF_{arith}\ r\ Term \\
\quad ParenF & :: r\ Expr & \rightarrow & & & PF_{arith}\ r\ Factor \\
\quad NumberF & :: [r\ Digit] & \rightarrow & & & PF_{arith}\ r\ Factor \\
\quad MkDigitF & :: Char & \rightarrow & & & PF_{arith}\ r\ Digit
\end{array}$$

This PF_{arith} GADT defines analogons to each of the constructors of our AST data types, but recursive positions of type ix are replaced with values $r\ ix$ of the argument semantic value family r. As such, the semantic value family r plays the role of a *subtree representation functor* (our terminology), defining what values to keep for subtrees of AST nodes. Pattern functor values are tagged with the AST node type they represent.

Note that we do not use the type functor combinators that Rodriguez *et al.* (2009) define to build pattern functors. This is because we do not require the generic operations that can be derived over these combinators and because we think our direct presentation of the pattern functor is clearer.

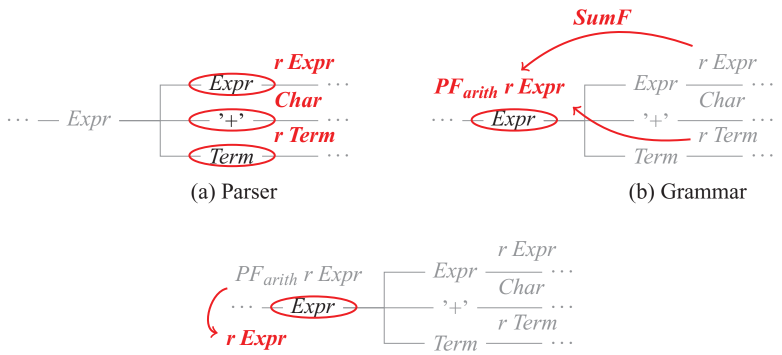
Rodriguez *et al.* (2009) also define a type family PF mapping domains ϕ to their pattern functor $PF\ \phi$. The following type family instance registers PF_{arith} as the pattern functor for domain ϕ_{arith}:

$$\textbf{type instance}\ PF\ \phi_{arith} = PF_{arith}$$

Like for simply recursive types, data types isomorphic to our original AST data types can be recovered from this pattern functor by taking its fixed point using a type-level fixpoint combinator. But the pattern functor allows to do more with the AST values. Rodriguez *et al.* (2009) demonstrate how to go back and forth between a type ix in a domain ϕ and its *one-level unfolding* of type $PF\ \phi\ I_*\ ix$ (with I_* a wrapping identity functor: $I_*\ ix \sim ix$). In this way, a value of the AST type $Expr$ can be converted into an unfolded value of type $PF_{arith}\ I_*\ Expr$, exposing the top-level of its structure (similar to the unfold operation for iso-recursive types (Pierce 2002, pp. 276–277)). Generic operations on instances of the pattern functor can then be used to implement various generic algorithms. All of this gives an impressive, elegant and powerful generic programming machinery, but for our purposes, the pattern functor is useful in another way.

A powerful feature of the pattern functor is that it abstracts over the subtree representation functor r, allowing subtrees to be represented differently than as full subtrees. If we take our semantic value family $[\![]\!]^{value}_{\cdot}$ as this subtree representation functor (instead of the wrapping identity functor I_*), then subtrees in the one-level unfolding of an AST are represented just by their calculated value (instead of a full sub-AST). For example, the value $(SumF\ [\![15]\!]^{value}_{Expr}\ [\![3]\!]^{value}_{Term})$ of type $(PF_{arith}\ [\![]\!]^{value}_{\cdot}\ Expr)$ represents an $Expr$ value, constructed as the sum of another $Expr$ and a $Term$, where we only know that the arithmetic values of the left-hand side $Expr$ and the right-hand side $Term$ are respectively 15 and 3. In general, the pattern functor PF_{arith} allows us to represent an AST where subtrees have already been processed into a semantic value, and this turns out to be precisely the vehicle we need for modelling the collaboration between a grammar, a parsing algorithm and a semantic processor.

Let us consider production rule $\text{Expr} \rightarrow \text{Expr}\ `\text{+}\text{'}\text{Term}$ as an example. Figure 3 shows a graphical illustration of this collaboration (for a semantic processor working with a semantic value family r). In Figure 3(a), the parser has matched the right-hand side elements of the production rule, and has obtained their semantic values, typed $r\ Expr$, $Char$ and $r\ Term$. In Figure 3(b), the grammar specifies how to combine these three values to the single-layer top of an AST, constructing a value of type $PF_{arith}\ r\ Expr$. For this production rule, the $SumF$ constructor is used, throwing



(c) Semantic processor

Fig. 3. (Colour online) A graphical representation of the collaboration between parser, grammar and semantic processor using ϕ_{arith}'s pattern functor over a semantic value family r as an intermediate representation.

away the parse result for the token '+'. Note that the grammar does not make any assumptions about the semantic value family r. In Figure 3(c), the semantic processor accepts the constructed $PF_{arith}\ r\ Expr$ value, calculates the combined semantic value and returns a processed value of type $r\ Expr$ to the parser for use in subsequent matches. Note that nothing here assumes any specific matching order (top-down vs. bottom-up).

For readers who are familiar with the terminology (Meijer *et al.*, 1991), it is interesting to note that the grammar's action on the semantic values is an anamorphism from concrete parsing trees to our mutually recursive data types. Correspondingly, the semantic processor specifies a catamorphism for the mutually recursive data types, and multirec's pattern functor machinery allows the parser to explicitly fuse the two together according to its own matching order.

With this machinery, we can effectively decouple grammars from their semantic processors and *vice versa*. In the next section, we take a look at the resulting code to see how it all fits together.

4.5 So what do we get?

So our finally tagless model of observable recursion is completed; we know how to abstract from the representation of return values of recursive calls, and we can even model the interaction between a grammar and its semantic processors, and abstract the grammar from the processors. We finally show the resulting definition of our running example grammar:

```
type ExtendedCFG φ = ∀ p r ix . (LoopProductionRule p φ r) ⇒ φ ix → p (PF φ r ix)
g_arith :: ExtendedCFG φ_arith
g_arith Line  = SExprF   $ ⟨Expr⟩ <* endOfInput
g_arith Expr  = STermF   $ ⟨Term⟩
              | SumF     $ ⟨Expr⟩ <* token '+' * ⟨Term⟩
g_arith Term  = SFactorF $ ⟨Factor⟩
```

$$\begin{aligned}
& && \obar\ \mathit{ProductF}\ \textcircled{\scriptsize\$}\ \langle \mathit{Term} \rangle \mathbin{<\!\!*}\ \mathit{token}\ \texttt{'*'} \circledast \langle \mathit{Factor} \rangle \\
g_{arith}\ & \mathit{Factor} && = \mathit{NumberF}\ \textcircled{\scriptsize\$}\ \langle \mathit{Digit} \rangle^{+} \\
& && \obar\ \mathit{ParenF}\ \mathbin{<\!\$}\ \mathit{token}\ \texttt{'('} \circledast \langle \mathit{Expr} \rangle \mathbin{<\!\!*}\ \mathit{token}\ \texttt{')'} \\
g_{arith}\ & \mathit{Digit} && = \mathit{MkDigitF}\ \textcircled{\scriptsize\$}\ (\mathit{token}\ \texttt{'0'} \obar \mathit{token}\ \texttt{'1'} \obar \ldots \obar \mathit{token}\ \texttt{'9'})
\end{aligned}$$

We first define a general *ExtendedCFG* type synonym (CFG for context-free grammar), expressing that an extended CFG is a function returning a production rule for every non-terminal. The $\forall \cdot$ quantification expresses that it must be defined for any production rule interpretation type p supporting the CFG operations of type class *LoopProductionRule* (and its parents *Applicative*, *Alternative*, *CharProductionRule*, *RecProductionRule*). It must also work for any semantic value family r, producing values of the pattern functor $PF\ \phi$ with r as the subtree representation type.

Our grammar g_{arith} is an extended CFG for the domain ϕ_{arith}. Its production rules are defined using the combinators we saw before, and values of $PF_{arith}\ r$ are produced using the pattern functor's constructors. Stylistically, the pattern functor constructors end up at the beginning of each production rule, giving a nice visual tagging of the rules, and defining for each production rule what kind of AST node it corresponds to. This final definition of our grammar is not linked to any parsing algorithm, matching order or set of semantic actions. As such, it is about as close as it gets to the formal definition of the grammar in Section 2.1.

Our semantic processors are algebras over the pattern functor. In fact, our type synonym *Processor* is identical to multirec's *Algebra* as defined by Rodriguez *et al.* (2009). Note also that syntactically, they look remarkably similar to syntax-directed definitions traditionally used with parser generators (Aho *et al.*, 2006, pp. 303–323):

$$\begin{aligned}
& \textbf{type}\ \mathit{Processor}\ \phi\ r = \forall\, ix\ .\ \phi\ ix \to PF\ \phi\ r\ ix \to r\ ix \\
& calc_{arith} :: \mathit{Processor}\ \phi_{arith}\ [\![\cdot]\!]^{value} \\
& calc_{arith}\ \mathit{Line}\ \ (\mathit{SExprF}\ [\![e]\!]^{value}_{Expr}) && = [\![e]\!]^{value}_{Line} \\
& calc_{arith}\ \mathit{Expr}\ \ (\mathit{SumF}\ [\![e]\!]^{value}_{Expr}\ [\![t]\!]^{value}_{Term}) && = [\![e + t]\!]^{value}_{Expr} \\
& calc_{arith}\ \mathit{Expr}\ \ (\mathit{STermF}\ [\![t]\!]^{value}_{Term}) && = [\![t]\!]^{value}_{Expr} \\
& calc_{arith}\ \mathit{Term}\ \ (\mathit{ProductF}\ [\![e]\!]^{value}_{Term}\ [\![t]\!]^{value}_{Factor}) && = [\![e * t]\!]^{value}_{Term} \\
& calc_{arith}\ \mathit{Term}\ \ (\mathit{SFactorF}\ [\![t]\!]^{value}_{Factor}) && = [\![t]\!]^{value}_{Term} \\
& calc_{arith}\ \mathit{Factor}\ (\mathit{ParenF}\ [\![e]\!]^{value}_{Expr}) && = [\![e]\!]^{value}_{Factor} \\
& calc_{arith}\ \mathit{Factor}\ (\mathit{NumberF}\ ds) && = [\![read\ \$\ map\ (\lambda [\![d]\!]^{value}_{Decimal} \to d)\ ds]\!]^{value}_{Factor} \\
& calc_{arith}\ \mathit{Digit}\ \ (\mathit{MkDigitF}\ c) && = [\![c]\!]^{value}_{Decimal}
\end{aligned}$$

This processor implements the direct calculation of *Integer* values for subexpressions that we have previously described. Its type expresses that it is a processor for domain ϕ_{arith}, producing semantic values of family $[\![\cdot]\!]^{value}$. Like in traditional parser combinator libraries, a semantic processor can also produce side effects, simply by working with monadic calculations as semantic values instead of simple values.

Another example of a semantic processor, for which we do not need to provide any code, has been defined by Rodriguez *et al.* (2009). Their function $to :: \phi\ ix \to PF\ \phi\ I_{*}\ ix \to ix$ in the *Fam* type class transforms a single-level unfolding of an AST (as described earlier) back into the traditional AST data type. Serendipitously, composing *to* with the I_{*} constructor yields a ready-to-use and important semantic processor for our grammars. The function $(I_* \circ) \circ to$ (applying I_* to the result of applying *to* to two arguments) is precisely the semantic processor that produces a wrapped version of AST as its semantic value. This direct correspondence illustrates that our use of multirec pattern functors to abstract semantic actions is a natural and powerful fit.

A processor and a grammar can be combined using the following function. It takes an extended CFG for domain ϕ, and a processor for domain ϕ and semantic value family r and turns it into an extended CFG that produces values of semantic value family r.

$$\begin{array}{l}\textbf{type}\ \mathit{ProcessingExtendedCFG}\ \phi\ r = \\ \quad \forall p\ ix\ .\ (\mathit{LoopProductionRule}\ p\ \phi\ r) \Rightarrow \phi\ ix \rightarrow p\ (r\ ix) \\ \mathit{applyProcessor} :: \mathit{Processor}\ \phi\ r \rightarrow \mathit{ExtendedCFG}\ \phi \rightarrow \mathit{ProcessingExtendedCFG}\ \phi\ r \\ \mathit{applyProcessor}\ \mathit{proc}\ g\ \mathit{idx} = \mathit{proc}\ \mathit{idx}\ \$\ g\ \mathit{idx}\end{array}$$

Some of the algorithms we define further on in this text will be able to work on grammars of types *ExtendedCFG and ProcessingExtendedCFG*. It is therefore useful to define a more general type of grammars as follows:

$$\begin{array}{l}\textbf{type}\ \mathit{GeneralExtendedCFG}\ \phi\ r\ rr = \\ \quad \forall p\ ix\ .\ (\mathit{LoopProductionRule}\ p\ \phi\ r) \Rightarrow \phi\ ix \rightarrow p\ (rr\ ix)\end{array}$$

Note that *ProcessingExtendedCFG* ϕ r is the same type as *GeneralExtendedCFG* ϕ r r and *ExtendedCFG* ϕ can be written as $\forall r$. *GeneralExtendedCFG* ϕ r $(PF\ \phi\ r)$. For non-extended and regular CFGs, we introduce analogous type synonyms:

$$\begin{array}{l}\textbf{type}\ \mathit{GeneralCFG}\ \phi\ r\ rr = \forall p\ ix\ .\ (\mathit{RecProductionRule}\ p\ \phi\ r) \Rightarrow \phi\ ix \rightarrow p\ (rr\ ix) \\ \textbf{type}\ \mathit{ProcessingCFG}\ \phi\ r = \mathit{GeneralCFG}\ \phi\ r\ r \\ \textbf{type}\ \mathit{CFG}\ \phi = \forall r\ .\ \mathit{GeneralCFG}\ \phi\ r\ (PF\ \phi\ r)\end{array}$$

There are ways to abstract this even further to remove duplication between the extended and the non-extended type synonyms, but we do not go into that here.

4.6 Grammar ingredients

In summary, our approach requires the grammar author to provide five things.

- The standard AST data types from Section 3.4 (the types *Line*, *Expr* etc. for our example).
- The domain subkind with the proof term constructors as in Section 4.1 (ϕ_{arith} for our example), defining the collection of non-terminals for the grammar. Various grammar algorithms require extra information about the domain, which needs to be provided through instances of the type classes *ShowFam*, *FoldFam* and *EqFam* that we will encounter further on.
- The pattern functor from Section 4.4 (PF_{arith} for our example), defining the recursive structure of the relations between the non-terminals, and the corresponding instance of the multirec *PF* type family. As discussed in Section 4.4, it can also be useful to implement multirec's *Fam* type class,

which defines the link between the domain, the pattern functor and the AST types.

- A grammar for the domain (g_{arith} in our example), defining the concrete syntactic structure. Various algorithms allow the programmer to analyse and/or transform the grammar. Multiple grammars can even be defined for the same domain.
- If the programmer wants to create a parser, he probably also requires one or more semantic processors as defined in Section 4.4 (e.g. $calc_{arith}$ for our example). These define how to combine parsed non-terminals to a value needed. Standard processors exist (e.g. a constant processor that leads to a recognizer for the grammar or the AST constructing processor $(I_* \circ) \circ to$ that we encountered in Section 4.4).

However, of these five things, the second and the third consist of boilerplate code, which could be mechanically derived from the definition of the AST data types. In fact, the multirec library provides Template Haskell functions that mechanise this translation. The concepts we defined in addition to multirec (like the instances for the *ShowFam*, *FoldFam* and *EqFam* type classes) could be generated in a similar way.

Finally, we note again that we do not use Rodriguez *et al.*'s (2009) type functor combinators to define the pattern functor. These combinators allow them to derive certain generic operations over it, reducing the amount of the required boilerplate code. We avoid them for presentation reasons: We find that they make pattern functor and semantic processor definitions more difficult to read and we do not need the automatically derived generic operations.

5 The proof of the pudding

In their previously mentioned paper, Carette *et al.* (2009) show how a finally tagless encoding allows them to interpret a DSL for a simple higher order typed object language in different ways. They demonstrate an evaluator, a compiler, a partial evaluator and call-by-name and call-by-value continuation passing style transforms. In Sections 3 and 4, we have extended their approach with a model of recursion in the object language such that it is observable in the meta-language.

We will now demonstrate that we can define different interpretations for the recursive constructs. In fact, these interpretations will work similarly to that of Carette *et al.*'s (2009) different interpretations of object language primitives: A suitable production rule interpretation type is defined, and the behaviour of primitive parsing and recursion constructs supported by the algorithm is defined in the instances of the *Applicative*, *Alternative*, *CharProductionRule*, *RecProductionRule* and/or *LoopProductionRule* type classes. Transformations are possible using a production rule interpretation parametric in an abstract underlying interpretation type p. In this section, we demonstrate this approach with a couple of such algorithms, both analyses and transformations.

The algorithms we discuss will have varying requirements on the grammars they work for (and for transformations: the grammars they produce), either for fundamental reasons (e.g. *foldLoops* and *transformLeftCorner* will be defined only for *processing* grammars and cannot straightforwardly be extended to abstract grammars) or for reasons of conciseness (e.g. *isReachable* and *foldReachable* will be defined for normal grammars only but can trivially be generalised to *extended* CFGs).

5.1 Pretty-printing grammars

A first grammar algorithm that requires a custom interpretation of recursion is pretty-printing. The implementation is not terribly difficult but it is instructive as a first demonstration of how to work with our recursion model. Furthermore, as a first test bed, it will also motivate some further infrastructure we need to put in place. This algorithm is a simplified version of the one in our `grammar-combinators` library.

To compute textual representations, we use a custom production rule interpretation type *PrintRuleInterp*, containing simply a *String* representation of the rule. It needs to carry the domain type ϕ and semantic value family r along in its type because of the functional dependencies of the production rule interpretation type classes.

```
newtype PrintRuleInterp (φ :: * → *) (r :: * → *) v = MkPRI {printRule :: String}
```

We implement the *ProductionRule* operations by simply constructing a proper *String* representation of the rule. Note that this is in fact the first time in this paper that we provide instances for these classes.

```
instance Applicative (PrintRuleInterp φ r) where
  pure _      = MkPRI "pure"
  a ⊛ b       = MkPRI $ printRule a ++ "  " ++ printRule b
instance Alternative (PrintRuleInterp φ r) where
  empty       = MkPRI "empty"
  a ⊕ b       = MkPRI $ "(" ++ printRule a ++ "  |  " ++ printRule b ++ ")"
instance CharProductionRule (PrintRuleInterp φ r) where
  endOfInput  = MkPRI "endOfInput"
  token t     = MkPRI $ show t
```

For the *RecProductionRule* instance, we need to know how to represent a non-terminal as a *String*. We therefore require our domain ϕ to be an instance of a new type class called *ShowFam*, telling us how to convert a domain proof term into a *String*.

```
class ShowFam φ where showIdx :: φ ix → String
instance (ShowFam φ) ⇒ RecProductionRule (PrintRuleInterp φ r) φ r where
  ⟨idx⟩ = MkPRI $ "<" ++ showIdx idx ++ ">"
instance (ShowFam φ) ⇒ LoopProductionRule (PrintRuleInterp φ r) φ r where
  ⟨idx⟩* = MkPRI $ "<" ++ showIdx idx ++ ">*"
  ⟨idx⟩+ = MkPRI $ "<" ++ showIdx idx ++ ">+"
```

ghci> $\mathit{putStr}\ \$\ \mathit{printGrammar}\ g_{arith}$

```
<Line>   ::= <Expr> EOI
<Expr>   ::= <Term> | (<Expr> '+' <Term>)
<Term>   ::= <Factor> | (<Term> '*' <Factor>)
<Factor> ::= <Digit>+ | ('(' <Expr> ')')
<Digit>  ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Fig. 4. Printing out an (E)BNF-like representation of the arithmetic expressions grammar with the library grammar printing algorithm (result manually indented).

Given this interpretation for production rules, we can define how to print the production rules for a single non-terminal:

$$\begin{array}{l} \mathit{printNT} :: (\mathit{ShowFam}\ \phi) \Rightarrow \mathit{GeneralExtendedCFG}\ \phi\ r\ rr \to \phi\ ix \to \mathit{String} \\ \mathit{printNT}\ \mathit{gram}\ \mathit{idx} = \texttt{"<"} +\!\!+ \mathit{showIdx}\ \mathit{idx} +\!\!+ \texttt{">"} +\!\!+ \texttt{" ::= "} +\!\!+ \mathit{printRule}\ (\mathit{gram}\ \mathit{idx}) \end{array}$$

This *printNT* function takes a grammar, a non-terminal proof term, and produces a string representation of the grammar's production rules for that non-terminal. Note that it takes our most general form of grammar *GeneralExtendedCFG*.

To print a full grammar, all that is left to do is to consecutively apply this *printNT* function to all non-terminals in a grammar. To do this, we again need information from the domain, and we define this as another general requirement for domains in the *FoldFam* type class. Since we cannot require that there exists a list of all non-terminals (because all their proof terms have a different type), the *FoldFam* class contains a function *foldFam*, which folds a given function over all non-terminals in the domain.

$$\begin{array}{l} \textbf{class}\ \mathit{FoldFam}\ (\phi :: * \to *)\ \textbf{where}\ \mathit{foldFam} :: (\forall ix\ .\ \phi\ ix \to b \to b) \to b \to b \\ \mathit{printGrammar} :: (\mathit{FoldFam}\ \phi, \mathit{ShowFam}\ \phi) \Rightarrow \mathit{GeneralExtendedCFG}\ \phi\ r\ rr \to \mathit{String} \\ \mathit{printGrammar}\ \mathit{gram} = \mathit{foldFam}\ ((+\!\!+) \circ (+\!\!+\texttt{"\textbackslash n"}) \circ \mathit{printNT}\ \mathit{gram})\ \texttt{""} \end{array}$$

One might have the impression that we are defining ad hoc *XFam* classes for all of our algorithms, but this impression is false. The type classes *FoldFam* and *ShowFam* (and *EqFam*, which we will encounter further on) express general requirements for domains. Only for presentation purposes, we have chosen to define them when we first encountered the need. The instances for our domain ϕ_{arith} are trivial:

$$\begin{array}{l} \textbf{instance}\ \mathit{ShowFam}\ \phi_{arith}\ \textbf{where}\ \mathit{showIdx}\ \mathit{Line} = \texttt{"Line"} \\ \qquad\qquad\qquad\qquad\qquad\quad \mathit{showIdx}\ \mathit{Expr} = \texttt{"Expr"} \\ \qquad\qquad\qquad\qquad\qquad\quad \mathit{showIdx}\ \mathit{Term} = \texttt{"Term"} \\ \qquad\qquad\qquad\qquad\qquad\quad \mathit{showIdx}\ \mathit{Factor} = \texttt{"Factor"} \\ \qquad\qquad\qquad\qquad\qquad\quad \mathit{showIdx}\ \mathit{Digit} = \texttt{"Digit"} \\ \textbf{instance}\ \mathit{FoldFam}\ \phi_{arith}\ \textbf{where} \\ \quad \mathit{foldFam}\ f = f\ \mathit{Line} \circ f\ \mathit{Expr} \circ f\ \mathit{Term} \circ f\ \mathit{Factor} \circ f\ \mathit{Digit} \end{array}$$

A more polished version of this algorithm produces output as given in Figure 4.

5.2 *Open recursion*

In Section 3.2, our first attempt at a better representation of recursion used a form of open recursion, different from $\langle\cdot\rangle$ construct, which we introduced later. It is in fact easy to formalize the equivalence between these two representations of grammars. In this section, we define a function called *openRecursion* that will turn out to be a useful technical aid in the implementation of other algorithms. It has the following type signature:

$$\begin{aligned}&\mathit{openRecursion} :: (\mathit{CharProductionRule}\ p) \Rightarrow\\&\quad \mathit{GeneralCFG}\ \phi\ r\ rr \rightarrow (\forall\, ix\ .\ \phi\ ix \rightarrow p\ (r\ ix)) \rightarrow \phi\ ix \rightarrow p\ (rr\ ix)\end{aligned}$$

This function turns a grammar using $\langle\cdot\rangle$ construct from the *RecProductionRule* type class into a grammar taking a *self* parameter instead. To implement this, we define a production rule type *ORRule*, which wraps a production rule taking a *self* parameter.

$$\textbf{newtype}\ \mathit{ORRuleInterp}\ p\ \phi\ r\ v\ = \mathit{MkORR}\ \{\mathit{unORR} :: (\forall\, ix\ .\ \phi\ ix \rightarrow p\ (r\ ix)) \rightarrow p\ v\}$$

We omit the instances for the classes *Applicative*, *Alternative* and *CharProductionRule* for this type. They simply pass through the *self* parameter to their components (if any). The *RecProductionRule* instance replaces calls $\langle idx\rangle$ with calls to *self idx*.

$$\begin{aligned}&\textbf{instance}\ \mathit{CharProductionRule}\ p \Rightarrow \mathit{RecProductionRule}\ (\mathit{ORRuleInterp}\ p\ \phi\ r)\ \phi\ r\ \textbf{where}\\&\quad \langle idx\rangle = \mathit{MkORR}\ \$\ \lambda \mathit{self} \rightarrow \mathit{self}\ idx\end{aligned}$$

In the implementation of *openRecursion*, we construct the production rule for non-terminal *idx* in the new grammar by interpreting the grammar with our *ORRuleInterp* production rule type and unwrapping the result.

$$\mathit{openRecursion}\ g\ \mathit{self}\ idx = \mathit{unORR}\ (g\ idx)\ \mathit{self}$$

Note by the way that the reverse transformation is even easier to define:

$$\begin{aligned}&\mathit{closeRecursion} :: (\mathit{RecProductionRule}\ p\ \phi\ r) \Rightarrow\\&\quad (\forall\, p\ .\ (\mathit{CharProductionRule}\ p) \Rightarrow (\forall\, ix\ .\ \phi\ ix \rightarrow p\ (r\ ix)) \rightarrow \phi\ ix \rightarrow p\ (rr\ ix)) \rightarrow\\&\quad \phi\ ix \rightarrow p\ (rr\ ix)\\&\mathit{closeRecursion}\ g\ idx = g\ \langle\cdot\rangle\ idx\end{aligned}$$

5.3 *Reachability*

The previous transformation is sometimes a useful technical tool in the implementation of other algorithms. In this section, we implement a simple non-terminal reachability analysis. We perform a depth-first search while keeping track of an environment of non-terminals already encountered, which we represent as a function from non-terminals to *Bool*s. The environment *nothingSeen* represents the empty set:

```
newtype SeenEnv φ = MkSG {seenIdx :: ∀ ix . φ ix → Bool}
nothingSeen :: SeenEnv φ
nothingSeen = MkSG $ \_ → False
```

To mark a non-terminal as seen in an environment, we need to be able to override the wrapped function for a single non-terminal and leave it unmodified for others. In fact, overriding polymorphic functions in this way is another general requirement on domains, which we model in the *EqFam* type class:

```
class EqFam φ where
  overrideIdx :: (∀ ix . φ ix → r ix) → φ oix → r oix → (∀ ix . φ ix → r ix)
```

This type class models a general notion of domains with a decidable equality between non-terminals. However, unlike a simpler equality test (like the derived *eqIdx* below), the *overrideIdx* function allows us to exploit this decidable equality to override a polymorphic function over a domain ϕ for one of the non-terminals $\phi\ oix$.

We need to instantiate the *EqFam* type class for all of our domains:

```
instance EqFam φ_arith where overrideIdx _ Line v Line = v
                             overrideIdx _ Expr v Expr = v
                             overrideIdx _ Term v Term = v
                             overrideIdx _ Factor v Factor = v
                             overrideIdx _ Digit v Digit = v
                             overrideIdx f _ _ idx = f idx
```

Using the general *overrideIdx* function, we can define a specialisation *overrideIdxK* for functions returning values of a constant type. We use a standard constant type functor K_* from multirec. We can also use it to define equality of non-terminal proof terms.

```
overrideIdxK :: (EqFam φ) ⇒ (∀ ix′ . φ ix′ → v) → φ oix → v → φ ix → v
overrideIdxK f idx v = unK* ∘ overrideIdx (K* ∘ f) idx (K* v)
eqIdx :: (EqFam φ) ⇒ φ ix1 → φ ix2 → Bool
eqIdx idx1 = overrideIdxK (const False) idx1 True
```

With this additional infrastructure, we can update our sets of non-terminals as follows:

```
setSeen :: (EqFam φ) ⇒ φ ix → SeenEnv φ → SeenEnv φ
setSeen idx s = MkSG $ overrideIdxK (seenIdx s) idx True
```

We implement our reachability analysis as a *foldReachable* function. Like the *foldFam* that we have seen before, this function folds a function over a set of non-terminals. However, unlike that function, it does not fold the function over all non-terminals in the domain. The folding is restricted to the non-terminals reachable from a given start non-terminal in a given grammar:

```
type Folder φ n = ∀ ix . φ ix → n → n
foldReachable :: ∀ φ ix n r rr . (EqFam φ) ⇒
  GeneralCFG φ r rr → φ ix → Folder φ n → n → n
```

The implementation of this function uses an interpretation type wrapping an algorithm with the set of encountered non-terminals *SeenEnv* ϕ as a mutable state variable. The wrapped algorithm takes the function to be folded and the start value and its return type is the result type of the fold.

```
newtype FoldReachableRuleInterp φ n v = MkFRRI {
  foldRule :: Folder φ n → n → State (SeenEnv φ) n}
putSeen :: (EqFam φ) ⇒ φ ix → State (SeenEnv φ) ()
putSeen idx = modify $ setSeen idx
```

The algorithm is simple. For leaf rules in the grammar, the algorithm does not need to do anything, and for branch nodes, we simply iterate over subnodes. We omit the instances for *Applicative*, *Alternative* and *CharProductionRule*, which simply translate all operations into the following *foldLeaf* or *foldBranch* as appropriate:

```
foldLeaf :: FoldReachableRuleInterp φ n v
foldLeaf = MkFRRI $ λ_ n → return n
foldBranch :: FoldReachableRuleInterp φ n v → FoldReachableRuleInterp φ n v′ →
              FoldReachableRuleInterp φ n v″
foldBranch ra rb = MkFRRI $ λf n → do n′ ← foldRule ra f n
                                      foldRule rb f n′
```

The only magic of the algorithm is in the handling of references to non-terminals. For a reference to the non-terminal *idx*, we need to check if we have encountered the non-terminal *idx* already and if so, terminate the recursive search. If not, we fold the fold function over the non-terminal and subsequently recurse over the production rules of this non-terminal:

```
foldRef :: (EqFam φ) ⇒ φ ix → FoldReachableRuleInterp φ n v →
           FoldReachableRuleInterp φ n v′
foldRef idx r = MkFRRI $ λf n →
  do seen ← gets (λseenSet → seenIdx seenSet idx)
     if seen then return n else putSeen idx ≫ foldRule r f (f idx n)
```

In order to define an instance of *RecProductionRule* for our *FoldReachableRuleInterp* using *foldRef*, we need to modify that type to carry along the rules for the entire grammar. By using the previously introduced *openRecursion* function as a technical aid, we avoid a bit of this verbiage. The *self* parameter we provide to that algorithm is constructed using the *foldRef* function. We then evaluate the fold in the start non-terminal's production rule with an initially empty set of seen non-terminals:

```
foldReachable g idx f n =
  let g′ :: ∀ ix . φ ix → FoldReachableRuleInterp φ n (rr ix)
      g′ = openRecursion g (λidx → foldRef idx (g′ idx))
  in evalState (foldRule (g′ idx) f n) nothingSeen
```

Checking whether a non-terminal *end* is reachable from a non-terminal *start* is easy to implement in terms of *foldReachable*:

$$\begin{array}{l} \mathit{isReachable} :: \forall\, \phi\ r\ rr\ ix\ ix'\ .\ (\mathit{EqFam}\ \phi) \Rightarrow \\ \quad \mathit{GeneralCFG}\ \phi\ r\ rr \to \phi\ ix \to \phi\ ix' \to \mathit{Bool} \\ \mathit{isReachable}\ g\ \mathit{start}\ \mathit{end} = \mathit{foldReachable}\ g\ \mathit{start}\ ((\vee) \circ \mathit{eqIdx}\ \mathit{end})\ \mathit{False} \end{array}$$

5.4 Production rule origami

In Section 3.3, we have introduced $\langle\cdot\rangle^*$ and $\langle\cdot\rangle^+$ operators, and we have defined their types in Section 4.4. Formally, these operators are very similar to the other recursive operator $\langle\cdot\rangle$, and our modelling of them in the *LoopProductionRule* type class allows us to define different interpretations, like for $\langle\cdot\rangle$ operator.

From a parsing point of view, these operators are less fundamental than $\langle\cdot\rangle$ operator. In this section, we implement a standard transformation, known as the recursive interpretation of *regular right part* grammars (Grune & Jacobs 2008, Section 2.3.2.4), which transforms grammars using $\langle\cdot\rangle^*$ and $\langle\cdot\rangle^+$ to grammars which only use $\langle\cdot\rangle$.

The grammar transformation works by replacing calls to $\langle idx\rangle^*$ for non-terminal idx with normal references $\langle idx^*\rangle$ to newly introduced non-terminals idx^*. We define suitable production rules for these new non-terminals in the transformed grammar. This transformation is implemented in our library as the *foldLoops* algorithm.

In a first step, we define the domain of the transformed grammar:

$$\begin{array}{l} \mathbf{data}\ \cdot^1\ \ ix \\ \mathbf{data}\ \cdot^*\ \ ix \\ \mathbf{data}\ \cdot_{fl}\ \ \phi\ ix\ \mathbf{where}\ \ \cdot^1\ :: \phi\ ix \to \phi_{fl}\ ix^1 \\ \qquad\qquad\qquad\qquad\quad \cdot^*\ :: \phi\ ix \to \phi_{fl}\ ix^* \end{array}$$

We introduce new non-terminal types ix^1 and ix^*, parameterised over an underlying non-terminal type ix. The new non-terminal ix^1 represents the base non-terminal ix, and ix^* as its quantified *-variant.[7] The new domain ϕ_{fl}, parameterised over an underlying domain ϕ (the dot is a placeholder for ϕ in the definition), contains proof terms for ix^1 and ix^* given a proof that ix is a non-terminal in the underlying domain ϕ. We use the names $\cdot^1$ and $\cdot^*$ in Haskell's namespace for types and values as respectively the non-terminal types and the constructors of the proof terms.

All necessary type classes (like *FoldFam* and *ShowFam*, and others that we have not encountered yet) can be implemented for this new domain. As an example, we show the *FoldFam* instance, which simply uses the underlying domain ϕ's *foldFam* function to iterate over both types of non-terminals in domain ϕ_{fl}:

$$\begin{array}{l} \mathbf{instance}\ \mathit{FoldFam}\ \phi \Rightarrow \mathit{FoldFam}\ \phi_{fl}\ \mathbf{where} \\ \quad \mathit{foldFam}\ f\ n = \mathit{foldFam}\ (\lambda idx \to f\ idx^*)\ \$\ \mathit{foldFam}\ (\lambda idx \to f\ idx^1)\ n \end{array}$$

For representing semantic values for the new domain, we introduce a semantic value family adapter r_{flv}, parameterised over an underlying semantic value family r.

[7] We do not require values for ix^1 and ix^* types, so we define these using the EmptyDataTypes GHC Haskell extension.

As you might expect, r_{flv} wraps a value of type $r\ ix$ for the new non-terminal ix^1 and a value of type $[r\ ix]$ for the non-terminal ix^*.

$$\begin{array}{l}
\textbf{data family}\ \cdot_{flv}\ (r :: * \to *)\ ix \\
\textbf{newtype instance}\ \cdot_{flv}\ r\ ix^1 = FLBV\ \{unFLBV :: r\ ix\} \\
\textbf{newtype instance}\ \cdot_{flv}\ r\ ix^* = FLMV\ \{unFLMV :: [r\ ix]\} \\
consFLV :: r_{flv}\ ix^1 \to r_{flv}\ ix^* \to r_{flv}\ ix^* \\
consFLV\ (FLBV\ v)\ (FLMV\ vs) = FLMV\ (v : vs)
\end{array}$$

In a second step, we define the *foldLoops* algorithm, which turns an extended CFG over domain ϕ into the equivalent non-extended CFG over the larger domain ϕ_{fl}. The algorithm only supports grammars that have already been combined with a semantic processor, i.e. grammars of type *ProcessingExtendedCFG*. This leads to the following type signature:

$$foldLoops :: ProcessingExtendedCFG\ \phi\ r \to ProcessingCFG\ \phi_{fl}\ r_{flv}$$

The transformed grammar is defined by the production rules for both types of non-terminals in domain ϕ_{fl}. The production rules for a non-terminal idx^* are straightforward. Such a non-terminal must either be the corresponding base non-terminal idx^1 followed by another instance of non-terminal idx^* itself, or it must be empty. In both cases, we make sure to produce the correct semantic value.

$$\begin{array}{ll}
foldLoops\ bgram\ idx^* = & consFLV \circledS \langle idx^1\rangle \circledast \langle idx^*\rangle \\
& \oplus\ pure\ (FLMV\ [\,])
\end{array}$$

The production rules for a base non-terminal idx^1 are obtained by taking the production rules of the unmodified grammar and replacing all references to $\langle idx\rangle^*$ with calls to $\langle idx^*\rangle$. We perform this substitution by instantiating the original grammar's production rules with a special production rule interpretation type *FLW*. The type *FLW* implements a production rule for the original CFG over domain ϕ in terms of an underlying production rule for the transformed CFG over the extended domain ϕ_{fl}. The classes *Applicative*, *Alternative* and *CharProductionRule* are implemented by just passing the call through to the underlying production rules and wrapping/unwrapping the results as appropriate (not shown for brevity). The *RecProductionRule* instance transforms a reference $\langle idx\rangle$ into a reference $\langle idx^1\rangle$ and the *LoopProductionRule* instance transforms a quantified reference $\langle idx\rangle^*$ into the desired normal reference $\langle idx^*\rangle$. The default definition of $\langle\cdot\rangle^+$ transforms $\langle idx\rangle^+$ into $(:) \circledS \langle idx\rangle \circledast \langle idx\rangle^*$, which is perfect for our purposes.

$$\begin{array}{l}
\textbf{data}\ FLRuleInterp\ p\ (\phi :: * \to *)\ (r :: * \to *)\ v = MkFLRI\ \{unFLRI :: p\ v\} \\
\textbf{instance}\ (RecProductionRule\ p\ \phi_{fl}\ r_{flv}) \Rightarrow \\
\quad RecProductionRule\ (FLRuleInterp\ p\ \phi\ r)\ \phi\ r\ \textbf{where} \\
\quad \langle idx\rangle = MkFLRI\ \$\ unFLBV \circledS \langle idx^1\rangle \\
\textbf{instance}\ (RecProductionRule\ p\ \phi_{fl}\ r_{flv}) \Rightarrow \\
\quad LoopProductionRule\ (FLRuleInterp\ p\ \phi\ r)\ \phi\ r\ \textbf{where} \\
\quad \langle idx\rangle^* = MkFLRI\ \$\ unFLMV \circledS \langle idx^*\rangle
\end{array}$$

ghci> *putStr* \$ *printGrammar* (*foldLoops* \$ *applyProcessor* $calc_{arith}$ g_{arith})

```
<Line*> ::= (<Line> <Line*>) | pure
<Expr*> ::= (<Expr> <Expr*>) | pure
<Term*> ::= (<Term> <Term*>) | pure
<Factor*> ::= (<Factor> <Factor*>) | pure
<Digit*> ::= (<Digit> <Digit*>) | pure
...
<Factor> ::= (<Digit> <Digit*>) | '(' <Expr> ')'
...
```

Fig. 5. A printed version of the added production rules for $\cdot^*$ non-terminals added by the *foldLoops* algorithm. We omit the $\cdot^1$ production rules that are identical to the rules in Figure 4.

ghci> *putStr* \$ *printReachableGrammar* (*filterDiesE*
(*transformLeftCornerE calcGrammarArith*)) \$ *LCBase Expr*

```
(...)
<Expr>          ::= ('(' <Expr-'('>) | ('0' <Expr-'0'>) | ('1' <Expr-'1'>) |
                    ('2' <Expr-'2'>) | ('3' <Expr-'3'>) | ('4' <Expr-'4'>) |
                    ('5' <Expr-'5'>) | ('6' <Expr-'6'>) | ('7' <Expr-'7'>) |
                    ('8' <Expr-'8'>) | ('9' <Expr-'9'>)
<Expr-'('>      ::= <Expr> ')' <Expr-Factor>
<Expr-Factor>   ::= <Expr-Term>
<Expr-Term>     ::= (('*' <Factor>) <Expr-Term>) | <Expr-Expr> | pure
<Expr-Expr>     ::= '+' <Term> (<Expr-Expr> | pure) | (EOI <Expr-Line>)
<Expr-Line>     ::= empty
<Expr-'9'>      ::= <Expr-Digit>
<Expr-Digit>    ::= <Digit>* <Expr-Factor>
(...)
```

Fig. 6. Some rules from the printed version of the arithmetic expressions grammar after applying the left-corner transform and dead-branch removal. Output reformatted, reordered and selected.

We can now finish our algorithm with the definition of the transformed grammar's production rules for non-terminals idx^1. These simply unwrap the *FLRuleInterp* production rule interpretation type:

$$foldLoops\ bgram\ idx^1 = FLBV \circledS unFLRI\ (bgram\ idx)$$

In Figure 5, we show the result of applying the *foldLoops* algorithm to the arithmetic expressions grammar. We omit the base rules that are identical to the rules in Figure 4.

5.5 The left-corner transform, declaratively...

As a final demonstration of a special interpretation of recursive constructs, we show by example that our framework allows the definition of non-trivial general grammar transformations. We develop an implementation of the left-corner transform as defined (among others) by Moore (2000). It removes left-recursion from a grammar, solving the problem that we have seen in Section 2.3.

Figure 6 partially shows the result of applying the left-corner transformation to the arithmetic expressions grammar. What happens is that for, for example, the

Expr non-terminal, the transformation has analysed the set of terminals and the set of non-terminals that a match of *Expr* can possibly start with. The second set is called the set of *left corners* of *Expr*. New non-terminals define what remains of the *Expr* non-terminal after one of these terminals (e.g. $Expr - \texttt{'('}$) or non-terminals (e.g. $Expr - Factor$) has been matched. The new rules are not (directly or mutually) left-recursive but they define the same language as the original grammar.

In the literature, the left-corner transform is typically presented in an algorithmic style (e.g. Blum & Koch, 1999; Moore, 2000; Baars *et al.*, 2009): an initial grammar is analysed, and step-by-step new rules are added to obtain a final transformed grammar. We conjecture that such an implementation can be supported in our framework using techniques similar to Baars *et al.* (2009). They define transformation arrows to generate new type-level identifiers as well as keep track of a modifiable typing environment for non-terminal references.

However, we prefer to give a more declarative account of the transformation. It turns out that by analysing the algorithmic description, we can identify the three different forms of production rules that will be generated, and the production rules for all three can be derived from the rules in the original grammar. For any given domain ϕ, we define a new domain ϕ_{lc}, containing three types of non-terminals: for given non-terminals a and b and terminal t, we have non-terminals a_1 (representing the base non-terminal a), $b \setminus_{NT} a$ (matching the remainder of non-terminal a when non-terminal b has already been matched) and $t \setminus_T a$ (matching the remainder of non-terminal a when character t has been matched). Note again the $\cdot$ as placeholder in notations.

$$\begin{array}{llll}
\textbf{data}\ \cdot_1\ ix & & & \\
\textbf{data}\ (\cdot \setminus_{NT} \cdot)\ ix'\ ix & & & \\
\textbf{data}\ (\setminus_T \cdot)\ ix & & & \\
\textbf{data}\ \cdot_{lc}\ \ \phi\ ix\ \textbf{where} & \cdot_1 & :: \phi\ ix \rightarrow \phi_{lc}\ ix_1 \\
 & \cdot \setminus_{NT} \cdot & :: \phi\ ix' \rightarrow \phi\ ix \rightarrow \phi_{lc}\ (ix' \setminus_{NT} ix) \\
 & \cdot \setminus_T \cdot & :: Char \rightarrow \phi\ ix \rightarrow \phi_{lc}\ (\setminus_T ix)
\end{array}$$

For a semantic value family r for the underlying domain ϕ, we define a new semantic value family r_{lc} for our new domain ϕ_{lc}, with appropriate semantic values for the newly introduced non-terminals. For example, since a non-terminal $b \setminus_{NT} a$ represents the remainder of a non-terminal a starting with a non-terminal b that has already been parsed, we define the type of its semantic value as $r\ b \rightarrow r\ a$: a function that returns the semantic value of non-terminal a when given the value of the already parsed non-terminal b.

$$\begin{array}{lll}
\textbf{data family}\ \cdot_{lc}\ \ (r :: * \rightarrow *)\ ix & & \\
\textbf{newtype instance}\ r_{lc}\ ix_1 & = LCV_1 & \{unLCV_1 :: r\ ix\} \\
\textbf{newtype instance}\ r_{lc}\ (ix' \setminus_{NT} ix) & = LCV_{\cdot \setminus_{NT} \cdot} & \{unLCV_{\cdot \setminus_{NT} \cdot} :: r\ ix' \rightarrow r\ ix\} \\
\textbf{newtype instance}\ r_{lc}\ (\setminus_T ix) & = LCV_{\cdot \setminus_T \cdot} & \{unLCV_{\cdot \setminus_T \cdot} :: Char \rightarrow r\ ix\}
\end{array}$$

In order to construct the production rules for these new non-terminals, we need to analyse the existing rules in the grammar. The information we need is collected in the four fields of production rule interpretation type *TLCRuleInterp*:

```
data TLCRuleInterp p φ r v =
  MkTLCRI {tlcEmpty      :: Maybe v,
           tlcFull       :: p v,
           tlcNTMinNT    :: ∀ ix′ . φ ix′ → p (r ix′ → v),
           tlcNTMinT     :: Char → p (Char → v)}
```

The field *tlcEmpty* keeps track of whether the production rule can (directly) match the empty string, and if so, what value that produces. Under *tlcFull*, we keep an unmodified version of the original production rule. Under *tlcNTMinNT*, we keep the original production rule with leading (direct) references to a given base non-terminal removed (or, in the absence of such a leading reference, a never-matching *empty* rule) and *tlcNTMinT* provides the original production rule with leading (direct) references to a given terminal removed.

We do not show the instances for the *Applicative*, *Alternative* and *CharProductionRule* type classes for brevity. In the *Applicative* instance, we need to make sure to properly handle empty and non-empty left-hand sides in the sequencing operator (to make sure we properly detect *leading* tokens and references). In the *CharProductionRule* instance, we interpret a call to *token tt* specially under the *tlcNTMinT* interpretation, replacing it with the *pure id* rule that simply passes through the already matched token.

The *RecProductionRule* instance is the most interesting one. Under the *tlcNTMinNT* interpretation of the base production rule (where the current rule has to consume a given already matched non-terminal), we need to interpret a call to a base non-terminal $\langle idx \rangle$ as a pure rule that simply passes through the already matched semantic value, but only if the already matched non-terminal is the requested non-terminal *idx*. Otherwise, the *tlcNTMinNT* interpretation must fail. To do this in a well-typed way, we use the function *overrideIdx*, defined in Section 5.3:

```
overrideIdx :: (EqFam φ) ⇒ (∀ ix . φ ix → r ix) → φ oix → r oix → (∀ ix . φ ix → r ix)
```

The *RecProductionRule* instance above defines the *tlcNTMinNT* interpretation of an underlying production rule as a function that will fail for all non-terminals except for the requested non-terminal, in which case it is an empty rule passing through the already matched result. A technical problem is that the *overrideIdx* function requires the result type of the overridden function to be directly parametric in the non-terminal type *ix*, requiring us to wrap and unwrap the returned rules in the wrapper type *WrapNTMinNTP*. The other interpretations are straightforward.

```
instance (RecProductionRule p φlc rlc, EqFam φ) ⇒
  RecProductionRule (TLCRuleInterp p φ r) φ r where
  ⟨idx :: φ ix⟩ = MkTLCRI {tlcEmpty = Nothing,
                          tlcFull = unLCV₁ $ ⟨idx₁⟩,
                          tlcNTMinNT = rNTMinNT,
                          tlcNTMinT = const empty}
    where rNTMinNT :: ∀ ix′ . φ ix′ → p (r ix′ → r ix)
          rNTMinNT idxm = unWNMNP $
```

$$\begin{aligned}&\quad\; \mathit{overrideIdx}\ (\backslash\_ \to \mathit{WNMNP}\ \mathit{empty})\ \mathit{idx}\ (\mathit{WNMNP}\ (\mathit{pure}\ \mathit{id}))\ \mathit{idxm}\\ &\textbf{newtype}\ \mathit{WrapNTMinNTP}\ p\ r\ ix\ ix' = \mathit{WNMNP}\ \{\mathit{unWNMNP} :: p\ (r\ ix' \to r\ ix)\}\end{aligned}$$

With these instances, we have the machinery that we need to analyse a grammar's production rules, and we can proceed to the actual transformation of the grammar in the function *transformLeftCorner*. This function is restricted to processing grammars because the left-corner transform inherently mixes transformed versions of rules from the original grammar and new rules of standard forms, making it difficult to work with non-processing grammars.

$$\begin{aligned}&\mathit{transformLeftCorner} ::\\ &\quad (\mathit{FoldFam}\ \phi, \mathit{EqFam}\ \phi) \Rightarrow \mathit{ProcessingCFG}\ \phi\ r \to \mathit{ProcessingCFG}\ \phi_{lc}\ r_{lc}\end{aligned}$$

To define the production rules of the transformed grammar, we need to know the FIRST sets of the non-terminals (Aho *et al.*, 2006 pp. 188–189): the set of terminals that a match of a given non-terminal can start with. To obtain this information, we make use of a general algorithm *calcFirst*, which performs the standard FIRST-set analysis. We omit its implementation, which is relatively straightforward (~70 LOC in the library). With this extra information, we call another function *transformLeftCorner'*, which will generate the actual production rules for our new non-terminals.

$$\mathit{transformLeftCorner}\ \mathit{gram}\ \mathit{idx} = \mathit{transformLeftCorner}'\ \mathit{gram}\ (\mathit{calcFirst}\ \mathit{gram})\ \mathit{idx}$$

The production rules for non-terminals idx_1 are of the following form: They first expect to see one of the tokens of the FIRST set of the non-terminal *idx* and then pass on the work to the non-terminal $t \backslash_T idx$, properly wrapping and unwrapping values along the way:

$$\begin{aligned}&\mathit{transformLeftCorner}'\ \mathit{bgram}\ \mathit{firstSet}\ \mathit{idx}_1 =\\ &\quad \textbf{let}\ \mathit{ruleT}\ \mathit{tt} = \mathit{flip}\ (\$)\ Ⓢ\ \mathit{token}\ \mathit{tt}\ ⊛\ (\mathit{unLCV}_{\cdot\backslash_T\cdot}\ Ⓢ\ \langle \mathit{tt} \backslash_T \mathit{idx}\rangle)\\ &\quad \textbf{in}\ \ \mathit{LCV}_1\ Ⓢ\ \mathit{Set.fold}\ ((\oplus) \circ \mathit{ruleT})\ \mathit{empty}\ (\mathit{firstSet}\ \mathit{idx})\end{aligned}$$

Omitting the production rules for non-terminals $ix' \backslash_{NT} ix$ (which are technically similar to those that follow), all that is still required for the left-corner grammar transformation are the rules for non-terminals $t \backslash_T idx$. These rules come in two forms, because the non-terminal *idx* can start with character *t* in two ways. Either one of the original production rules for the non-terminal *idx* starts with character *t* directly, and in that case the remainder of that production rule becomes the production rule for $t \backslash_T idx$. This remainder of the original production rule is precisely what is represented by its interpretation under *tlcNTMinT t*.

The other possibility is that a production rule of *idx* starts with a (direct or indirect) reference to another non-terminal *idxB*, and that non-terminal directly starts with character *t*. This is captured by a production rule for non-terminal $t \backslash_T idx$ that starts with the remainder of the production rules for non-terminal *idxB* starting with character *t* (which we again get using that production rule's interpretation under *tlcNTMinT*) and then references non-terminal $idxB \backslash_{NT} idx$. Because non-terminal $idxB \backslash_{NT} idx$ represents the remainder of a base non-terminal *idx* after a non-terminal *idxB* has been matched, its production rules will properly match the remainder of non-terminal *idx*.

```
transformLeftCorner′ bgram _ (t \_T (idx :: φ ix1)) =
  let bMinT :: φ ix2 → p (Char → r ix1)
      bMinT idxB = flip (∘) ⓢ tlcNTMinT (bgram idxB) t ⊛
                              (unLCV ·\_NT· ⓢ ⟨idxB \_NT idx⟩)
      bMinTs = foldFam ((⦶) ∘ bMinT) empty
  in    LCV ·\_T· ⓢ bMinTs
      ⦶ LCV ·\_T· ⓢ tlcNTMinT (bgram idx) t
```

Note that we do not actually check whether character *t* is in the FIRST set of non-terminal *idxB*, nor that *idxB* is a left corner of *idx*. These would both be worthwhile optimisations, but they are not necessary because in those cases subsequent parts of the production rule in question become *empty* rules and can be removed using general postprocessing algorithms (dead-branch removal and dead non-terminal unfolding).

5.6 *The grammar-combinators library*

The above algorithms show that our grammar model adds useful expressiveness to parser combinator libraries in a finally tagless style: we can do more grammar analyses and transformations. The limitation of parser combinator libraries to top-down parsing algorithms is also lifted: after the left-corner transform, a top-down matching order in the transformed grammar corresponds to a left-corner matching order for the original grammar (Rosenkrantz & Lewis, 1970). Our semantic processors can be applied during parsing, independent of the matching order.

In addition to what we have shown, we have implemented an elaborate grammar analysis, transformation and parsing Haskell library called `grammar-combinators`. This library is designed as a collection of independently usable grammar algorithms. The library provides a combination of various features that, to the best of our knowledge, are unavailable in any existing parser EDSL library.

Practical features include a powerful transformation library (including the left-corner transform and a uniform version of Paull's left-recursion removal (Aho *et al.*, 2006 p. 177), support for performing grammar transformations at compile time using Template Haskell (Sheard & Peyton Jones, 2002)), a generic packrat parser (Ford, 2002) and basic interfaces to uu-parsinglib (Swierstra, 2009) and Parsec (Leijen & Meijer 2001) as *backend parsers*. The library is an open source and is available online.

5.7 *Limitations*

Notwithstanding its advantages, our typing of recursive constructs entails a certain overhead in defining concepts such as the domain, its proof terms and pattern functor and semantic value families, when compared to standard parser combinators (see Section 4.6). On top of this, some limitations need to be taken into account.

Our recursive constructs are clearly more verbose than the almost trivial recursion in typical parser combinator libraries. However, we believe that a certain verbosity is unavoidable if we wish to support a wide range of standard algorithms from the parsing literature, many of which require observable recursion. Also, supporting additional recursive constructs requires quite a bit of work. In this paper, we can see that supporting quantified recursive constructs on top of normal recursive constructs (which is again almost trivial in normal parser combinators) required an extra type class, a translation algorithm involving a model of the modified domain and semantic value families etc.

A compelling feature of parser combinators that we have not looked at is the ease with which you can combine unrelated parsers into new ones. An example is the definition of grammar patterns like typical comment styles or standard number notations. We require a full view of grammars, and this makes us lose some of the simple compositionality of parser combinators. We are experimenting with a grammar combination primitive that partly recovers this, but it is not ready for inclusion in the library.

Another limitation is that the added abstraction unfortunately has a performance cost. In some initial tests, we have effectively noticed an important performance impact, even though general optimizations for generic code (Magalhães *et al.*, 2010) appear to reduce it considerably. The performance impact could also be reduced by performing grammar transformations at compile-time using Template Haskell. A more detailed performance analysis remains for future work, but we expect that compiler improvements are needed (like better inlining heuristics and more control over partial evaluation) to improve performance of generic code in general and our code in particular.

6 Related work

For background material on CFGs, parsing and grammar transformations, we refer to Aho *et al.* (2006).

6.1 Finally tagless DSLs

The finally tagless style for modelling a typed object language in a meta-language was identified and popularised by Carette *et al.* (2009). They demonstrate a model of a higher order typed lambda calculus in a typed functional meta-language (they use both Haskell and ML). Their model is parameterised by an interpretation of the primitive operations of their object language. It uses meta-language typing to statically ensure type-correctness of the modeled object language terms. Carette *et al.*, demonstrate a set of different interpretations of their lambda calculus: an evaluator, a staged interpreter, a partial evaluator and call-by-name and call-by-value continuation passing style transformations.

In this text, we have described why the standard *fix* operator is not a perfect fit for our requirement for meta-language observable recursion in our parsing DSL and we have defined an alternative model. It is interesting however that for any grammar AST, we can also consider the AST as a representation type for terms of a separate embedded type object language, representing the semantics of the grammar. In Carette *et al.*'s (2009) terminology, the standard AST type from Section 4 is an "initial" embedding of this language, but we could have used a "finally tagless" model for it as well. In this model, the *Sum* constructor would, for example, correspond to a *sum* function in a grammar-specific *ArithSemantics* type class. Such a finally tagless encoding is more extensible than the naive AST representation, but our representation using the multirec pattern functor actually features this extensibility as well; because the pattern functor is parametric in the representation of recursive sub-data, we can apply the same technique as Swierstra (2008). In this sense, the pattern functor offers an alternative "initial" modelling, less naive than the standard GADT representation, and lacking its inextensibility. In addition, it offers some benefits of its own that seem unavailable in a finally tagless style (e.g. it supports generic algorithms using multirec (Rodriguez *et al.*, 2009)). A more detailed study of this correspondence is an interesting future work.

6.2 Parser combinators

Parser combinators have a long history (see Leijen & Meijer (2001) for references), but most work employs an ω-regular representation of grammars with the associated downsides that we have discussed in Section 1. Here we limit ourselves to work that uses a representation of grammars in which recursion is observable. Even then, almost all libraries are tied to a single parsing algorithm.

6.2.1 TTTAS

Baars and Swierstra (2004) and Baars *et al.* (2009) implement the left-corner grammar transform (Moore 2000) using type-level naturals as the representation of non-terminals. They ensure type-safety using a type environment encoded as a list of types. They propose a transformation library based on the arrows abstraction, which they use essentially for the generation of fresh type-level identifiers. Like ours, their grammar representation explicitly represents the grammar's recursion in a well-typed way and allows them to implement the left-corner transform and support left-recursive grammars.

Nevertheless, we believe our work provides advantages over theirs. Our representation of non-terminals as a "subkind with proof terms" (Rodriguez *et al.*, 2009) and type environments as data families is less complex. We provide semantic value family polymorphism, which they do not. They use stateful *Trafo* transformation arrows to allow for generation of fresh non-terminal identifiers. This allows them to implement the standard, imperative-style descriptions of grammar transformations and imperatively extend domains step-by-step during the transformation. Our algorithms work with *fixed* domains, which we found beneficial in the sense that it has forced us to formulate the algorithms in a more functional style. However, there may be algorithms that do not lend themselves well to such a reformulation (although we have not encountered them in the parsing domain), in which case more complex techniques like Baars and Swierstra's (2004) extensible domains are required.

Finally, Baars and Swierstra's (2004) grammars seem designed for compiler-generation in Viera *et al.*'s (2008) alternative for the standard Haskell *read*-function and they are less easily human-readable than our grammars. In the parsing domain, Baars and Swierstra (2004) only discuss an implementation of the left-corner grammar transform, while we show the importance of our approach for a wider parsing library, discussing implementations of a variety of useful algorithms for grammar analysis, transformation and parsing.

6.2.2 Dependently typed parser combinators

Brink *et al.* (2010) describe a dependently typed parser combinator library implemented in the Agda programming language (Norell, 2007). Agda's dependently typed nature strongly simplifies the requirements on the representation of non-terminals (types of production rules can more simply depend on non-terminals). They implement the left-corner transformation in their formalism, and provide a machine-checkable proof of a language-inclusion property for the transformation.

The proof of correctness properties beyond type-safety is out of range in our Haskell implementation. In addition to making such proofs possible, the power of dependent types also lets the authors get away with very simple models of grammars (a list of production rules) and production rules (a left-hand side non-terminal and a list of right-hand side symbols). They simply recalculate types from these simple models when needed instead of going through a lot of trouble to model and carry them around. Our use of Haskell limits us to a more restricted formalism, but this does make our ideas more portable and our approach more disciplined. Our use of a finally tagless model allows us to define different sets of primitives that can be mixed and matched (keeping e.g. extended CFGs separate from normal CFGs), whereas Brink *et al.* (2010) restrict themselves to standard CFGs.

Danielsson & Norell (2010) use Agda to define a provably terminating parser combinator library of *total parser combinators*. They use unobservable (co-) recursion,[8] limiting them to a top-down parser algorithm. They manage to support left-recursion (although their approach does not seem suited for online parsing) with an algorithm based on the Brzozowski derivatives, and they provide a static termination guarantee using dependent types and a mixture of induction and coinduction. It is interesting that in an unpublished draft, seemingly a pre-cursor of their total parser combinators, Danielsson and Norell (2010) investigate a model of grammars with observable recursion, using an operator $\_!$ similar to our $\langle\cdot\rangle$. They discuss it as one of the two alternative modellings of grammars that solves certain technical modularity problems of a more standard parser combinator model. The authors do not discuss the fact that their "grammar-based" model makes recursion observable or the additional power this provides to the model.

[8] What we have been calling recursion throughout this paper is actually corecursion in their terminology.

6.3 Observable recursion

In order to model and work with recursive structures in a pure language like Haskell, several approaches have been explored in the literature. One branch of research has focused on introducing a varying amount of impurity, ranging from observing sharing within the *IO* monad (Gill, 2009) to adding referential identity as a fundamental language feature (Claessen & Sands, 1999). We do not go into these approaches in detail, as it is our goal to model the recursion in the parsing EDSL with a representation that is observable in Haskell without compromising purity. Much of this research focuses on the application domain of hardware description languages and we would be interested to see if our approach can be successfully applied in this field as well.

Carette *et al.* (2009) provide a form of observable recursion through the *fix* primitive, which we have discussed in detail in Section 3, so we do not go into that further. Another interesting proposal is the recursive do notation as proposed by Erkök and Launchbury (2002), who add a primitive recursive operator for monads in a type class *MonadRec* (later renamed to *MonadFix*):

```
class Monad m ⇒ MonadRec m where
  mfix :: (α → m α) → m α
```

Instances are supposed to obey three laws (strictness, purity and left-shrinking). Analogous to the translation of Haskell's **do**-notation to pure code involving the monadic operators, Erkök and Launchbury (2002) define a recursive **mdo**-notation and a translation to pure code involving monadic operators *and* the *mfix* primitive. Erkök and Launchbury's (2002) proposal could be used to provide observable recursion in a monadic parser EDSL, but unfortunately, a monadic parser EDSL is more difficult to analyse for other reasons (see e.g. Swierstra and Duponcheel, 1996, Section 5.2). As discussed in Section 3, we believe that the *FixProductionRule* type class and its *fix* method (based on the *fix* method defined by Carette *et al.*, 2009) are natural analogons of *MonadRec* and *mfix* for applicative functors and we think it is an interesting future work to extend the bracket notation for applicative code by McBride & Paterson, (2008) with a notation for recursion.

The **do**-notation for arrows by Paterson (2001) also translates recursion to a recursion primitive in the *ArrowLoop* class, which seems to support observable recursion. Allowing for the (limited-depth) LL(1) analysis performed by UUParse (Swierstra & Duponcheel, 1996) was a motivation for the development of arrows (Hughes, 2000), so together with the observable recursion primitive provided by the *ArrowLoop* type class and the recursive **do**-notation, arrows may allow for the more elaborate kinds of grammar analysis and transformation that we perform, but we are not aware of any work that investigates this in more detail.

Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State (Belgian Science Policy), the Research Foundation – Flanders (FWO) and the Research Fund K.U. Leuven. Dominique Devriese holds a PhD fellowship of the Research Foundation – Flanders (FWO). We thank Arie Middelkoop, Tom Schrijvers, Adriaan Moors, Doaitse Swierstra, Nils Anders Danielsson and anonymous reviewers for their comments.

References

Aho, Alfred V., Lam, Monica S., Sethi, R. & Ullman, Jeffrey D. (2006) *Compilers: Principles, Techniques and Tools*, 2nd ed. Boston, MA: Addison-Wesley.

Baars, A. I. and Swierstra, S. D. (2004) Type-safe, self-inspecting code. In *Proceedings of the 2004 HASKELL Workshop*, Snowbird, UT, USA, September 22.

Baars A. I., Swierstra, S. D. & Viera, M. (2009) Typed transformations of typed abstract syntax. In *Proceedings of the TLDI'09*, Savannah, GA, USA, January 24, pp. 15–26.

Blum, N. & Koch, R. (1999) Greibach normal form transformation revisited. *Inf. Comput.* **150**(1), 112–118.

Brink, K., Holdermans, S. & Löh, A. (2010) Dependently typed grammars. In *Proceedings of the MPC*, Québec City, Canada, June 21–23.

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

Claessen, K. & Sands, D. (1999) Observable sharing for functional circuit description. In *Proceedings of the ASIAN '99*, Phuket, Thailand.

Danielsson, N. A. & Norell, U. (2010) Total parser combinators. In *Proceedings of the 15th ICFP*, Baltimore, MD, USA, September 27–29.

Devriese, D. & Piessens, F. (2010) *Explicitly Recursive Grammar Combinators – Implemention of Some Grammar Algorithms.* Tech. Rep. CW594, KU Leuven, CS. Available at: `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW594.abs.html`.

Devriese, D. & Piessens, F. (2011) Explicitly recursive grammar combinators. In *Proceedings of the 13th PADL'11*, Austin, TX, USA, January 24–25.

Erkök, L. & Launchbury, J. (2002) A recursive do for haskell. In *2002 Haskell Workshop*, Pittsburgh, PA, USA, October 3.

Ford, B. (2002) Packrat parsing: Simple, powerful, lazy, linear time – functional Pearl. In *Proceedings of the ICFP*, Pittsburgh, PA, USA, October 4–6.

Frost, R. A., Hafiz, R. & Callaghan, P. (2008) Parser combinators for ambiguous left-recursive grammars. In *Proceedings of the 10th PADL*, San Francisco, CA, USA.

Gill, A. (2009) Type-safe observable sharing in Haskell. In *Proceedings of the HASKELL*, Edinburgh, Scotland, September 3, pp. 117–128.

Grune, D. & Jacobs, C. J. H. (2008) *Parsing Techniques: A Practical Guide*, 2nd ed. New York: Springer-Verlag (ISBN 038720248X).

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

Johnson, S. C. (1979) YACC. In *Unix Programmer's Manual*, vol. 2b. Madison, WI: Bell Laboratories.

Leijen, D. & Meijer, E. (2001) *Parsec: Direct Style Monadic Parser Combinators for the Real World.* Tech. Rep. UU-CS-2001-27, Universiteit Utrecht CS.

Magalhães, J. P., Holdermans S., Jeuring J. & Löh, A. (2010) Optimizing generics is easy! In *Proceedings of the PEPM*, Madrid, Spain, January 18–19.

McBride, C. & Paterson, R. (2008) Functional pearl: Applicative programming with effects. *J. Funct. Program.* **18**(1), 1–13.

Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the FPLCA*, August 1991. (Lect. Notes in Comp. Sci., 523). New York: Springer-Verlag.

Might, M., Darais, D. & Spiewak, D. (2011) Parsing with derivatives: A Functional Pearl. *Proceedings of the ICFP*, Tokyo, Japan, September 19–21.

Moore, R. C. (2000) Removing left recursion from context-free grammars. In *Proceedings of the NAACL*, Seattle, WA, USA, May 4.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, Sweden.

Park, D. (1981) Concurrency and automata on infinite sequences. In *Proceedings of the Theoretical Computer Science, 5th GI-Conference*, Karlsruhe, Germany, March 23–25, pp. 167–183.

Parr, T. J. & Quong, R. W. (1995) ANTLR: A predicated-LL(k) parser generator. *Softw. Pract. Exp.* **25**(7), 789–810.

Paterson, R. (2001) A new notation for arrows. In *Proceedings of the ICFP*, Florence, Italy, September 3–5, 240 pp.

Peyton Jones, S., Vytiniotis, D., Weirich, S. & Washburn, G. (2006) Simple unification-based type inference for GADTs. In *Proceedings of the ICFP*, Portland, OR, USA, September 18–20.

Pierce, B. C. (2002) *Types and Programming Languages*. Cambridge, MA: MIT Press.

Rodriguez, A., Holdermans, S., Löh, A. & Jeuring, J. (2009) Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ICFP*, Edinburgh, Scotland, August 31– September 2.

Rosenkrantz, D. J. & Lewis, P. M. (1970) Deterministic left corner parsing. In *Proceedings of the 11th Annual Symposium on Switching and Automata Theory*, October 28–30.

Sabry, A. (1998) What is a purely functional language? *J. Funct. Program.* **8**(1), 1–22.

Schrijvers, T., Peyton Jones, S. , Chakravarty, M. & Sulzmann, M. (2008) Type checking with open type functions. In *Proceedings of the 13th ICFP*, British Columbia, Canada, September 22–24.

Sheard, T. (December 2005) Another look at hardware design languages. Available at: `http://www.cs.pdx.edu/∼sheard/`.

Sheard, T. & Peyton Jones, S. (2002) Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Pittsburgh, USA, 3rd October, pp. 1–16.

Swierstra, W. (July 2008) Data types à la carte. *J. Funct. Program.* **18**, 423–436.

Swierstra, D. (2009) Combinator Parsing: A Short Tutorial. Language Engineering and Rigorous Software Development, Piriapolis, Uruguay, Revised Tutorial Lectures, Lecture Notes in Computer Science, vol. 5520. New York: Springer-Verlag, pp. 252–300.

Swierstra, S. & Duponcheel, L. (1996) Deterministic, error-correcting combinator parsers. *Proceedings of the AFP*, Olympia, WA, USA.

Viera, M., Swierstra, S. D. & Lempsink, E. (2008) Haskell, do you read me? Constructing and composing efficient top-down parsers at runtime. In *Proceedings of the HASKELL 2008*, British Columbia, Canada, September 25.