Graph Parser Combinators

Steffen Mazanek and Mark Minas

Universität der Bundeswehr, München, Germany
{steffen.mazanek,mark.minas}@unibw.de

Abstract. A graph language can be described by a graph grammar in a manner similar to a string grammar known from the theory of formal languages. Unfortunately, graph parsing is known to be computationally expensive in general. There are quite simple graph languages that crush most general-purpose graph parsers.

In this paper we present graph parser combinators, a new approach to graph parsing inspired by the well-known string parser combinators. The basic idea is to define primitive graph parsers for elementary graph components and a set of combinators for the construction of more advanced graph parsers. Using graph parser combinators special-purpose graph parsers can be composed conveniently. Thereby, language-specific performance optimizations can be incorporated in a flexible manner.

Keywords: functional programming, graph parsing, parser combinators, visual languages.

1 Introduction

Graphs are a central data structure in computer science. Among other things, they are heavily used for modeling and specifying. For instance, we have specified visual languages using graph grammars [1]. Such graph grammars are used to define a particular graph language in analogy to string grammars known from formal language theory.

As in the setting of string grammars we are interested in solving the membership problem (checking whether a given graph belongs to a particular graph language) and parsing (finding a corresponding derivation), respectively. However, while string parsing of context-free languages can be performed in $O(n^3)$, e.g., by using the well-known algorithm of Cocke, Kasami and Younger [2], graph parsing is computationally expensive. There are even context-free graph languages the parsing of which is NP-complete [3]. Thus, a general-purpose graph parser cannot be expected to run in polynomial time for arbitrary grammars. The situation can be improved by imposing particular restrictions on the graph languages or grammars. Anyhow, even if a language can be parsed in polynomial time by a general-purpose parser, a special-purpose parser tailored to the language is likely to outperform it.

For this reason we propose a different approach to graph parsing: Graph Parser Combinators. We mainly have been inspired by the work of Hutton and Meijer

[4] who have proposed monadic parser combinators for string parsing. The idea of parser combinators is much older, though. The basic principle of a parser combinator library is that primitive parsers are provided that can be combined into more advanced parsers using a set of powerful combinators.

The most prominent combinators are the sequence and the choice combinator that can be used to make parsers resemble a grammar very closely. However, a wide range of other combinators is also imaginable, e.g., to cover common patterns like repetition or optionality. Thereby partial results can be composed in a flexible way. Further application-specific combinators can be added easily. The non-linear structure of graphs suggests even more interesting patterns worth an abstraction.

Parser combinators are very popular, because they integrate seamlessly with the rest of the program and hence the full power of the host language can be used. Unlike Yacc [5] no extra formalism is needed to specify the grammar. Another benefit is that parsers are first-class values within the language. For example, we can construct lists of parsers or pass them as function parameters. The possibilities are only restricted by the potential of the host language.

Having all these benefits in mind the question arises how parser combinators can be adopted to graphs. The discussion of this idea is the main contribution of this paper. We introduce the theoretical background in Sect. 3 and 4. Thereafter, we propose a framework and a set of graph parser combinators in Sect. 5. Then we go on to demonstrate the practical use of these combinators by applying them to a real-world example, namely the visual language VEX (visual expressions). This example is introduced in Sect. 2 and the corresponding parser is constructed in Sect. 6. Therewith, we demonstrate that efficient special-purpose graph parsers can be implemented straightforwardly.

A First Impression

At this point, we provide a toy example to give an impression of what a parser constructed using our combinators is going to look like.

An important advantage of the combinator approach is that a more operational description of a language can be given. For example, the language of the strings $\{a^k b^k c^k | k \in I\!N\}$ is not context-free. Hence a general-purpose parser for context-free languages cannot be applied at all, although parsing this language actually is very easy: "Take as many a characters as possible, then accept exactly the same number of b characters and finally accept exactly the same number of c characters."

Using PolyParse [6], a well-known and freely-available parser combinator library for strings maintained by Wallace, a parser for this string language can be defined as shown in Fig. 1a. The type of this parser determines that there may be a user-state `s` (not used in this example), that the tokens have to be characters and that the result is a number (namely k). The combinator `many` applies a given parser multiple times, while collecting the results. If the given word is not a member of the language one of the calls of `exactly` fails.

```
abc::Parser s Char Int
abc =
 do as←many (char 'a')
    let k = length as
    exactly k (char 'b')
    exactly k (char 'c')
    return k
```

(a) String parser

```
abcG::NGrappa s Char Int
abcG n =
 do (n',as)←chain (dirEdge 'a') n
    let k = length as
    (n'',_)←exactChain k (dirEdge 'b') n'
    exactChain k (dirEdge 'c') n''
    return k
```

(b) Graph parser

Fig. 1. Parsers for the string and the graph language $a^k b^k c^k$

Note, that the given parser uses the `do`-notation, syntactic sugar Haskell [7] provides for dealing with monads. Monads in turn provide a means to simulate state in Haskell. In the context of parser combinators they are used to hide the list of unconsumed input. Otherwise all parsers in a sequence would have to pass this list as a parameter explicitly.

In order to motivate our approach to graph parser combinators we provide the graph equivalent to the previously introduced string parser `abc`. Strings generally can be represented as directed, edge-labeled graphs straightforwardly. For instance, Fig. 2 provides the graph representation of the string "aabbcc". A graph parser for this graph language can be defined using our combinators in a manner quite similar to the parser discussed above as shown in Fig. 1b. The main difference between the implementations of `abc` and `abcG` is, that we have to pass through the position, i.e., the node, we currently process.

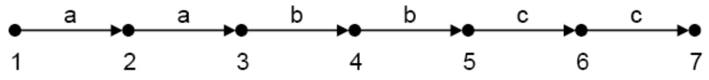
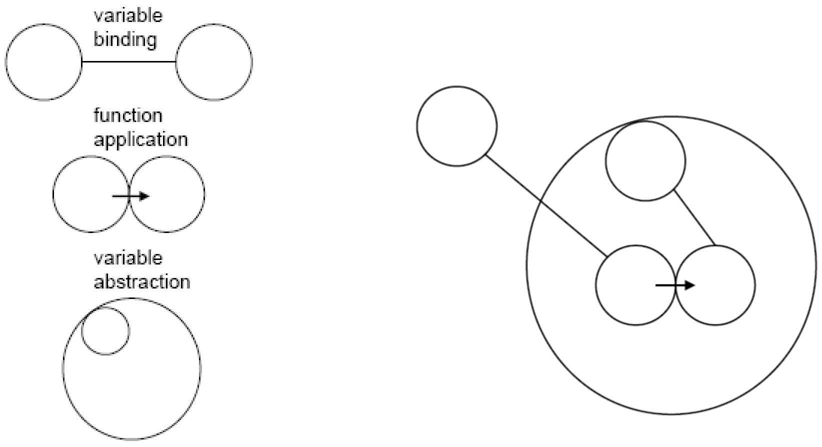


Fig. 2. The string graph "aabbcc"

Although many details have been omitted here, we hope that the idea behind graph parser combinators is understandable already. In Sect. 5 we present the framework and the combinators that can be used to conveniently implement parsers like `abcG`.

2 A Running Example: VEX

In this section we introduce the visual language VEX as our running example. Visual languages and graphs are highly related, because graphs are a very natural means for describing complex situations at an intuitive level. Graphs in particular appear to be well suited as an intermediate data structure in visual



(a) Language elements (b) Example diagram: $\lambda x.(y\ x)$

Fig. 3. The visual language VEX

language editors. For instance, in editors generated using the diagram editor generator DiaGen [1], visual objects and their spatial relations are mapped to graph components and a graph parser is used to check whether a diagram is a member of the particular language. This mapping is described in more detail in the next section.

VEX [8] is a language for the visualization of lambda expressions. Thereby variable bindings are explicitly defined by lines and not implicitly given by their names as in normal lambda calculus. In Fig. 3a the basic elements of the language are depicted. A VEX diagram basically consists of a set of circles, lines and arrows, whose layout determines the represented lambda term.

In VEX each variable identifier is represented by an empty circle (labeling text may be contained, however) that is connected by a line to a so-called root node. A root node is again an empty circle with one or more lines touching it, leading to all identifiers representing the same variable.

A root node may either be internally tangential to another circle, it then represents the parameter of a λ-abstraction, or it is not contained in any other circle, it then denotes a free variable. A circle representing a λ-abstraction contains its parameter circle and a VEX (sub-)diagram as its body.

Function application is expressed by two circles externally tangent to each other. An arrow between these circles indicates the direction of application. As usual application associates to the left. However, a different order of application can be determined by a particular numbering scheme [8].

In Fig. 3b a syntactically correct VEX diagram is given that represents the lambda term $\lambda x.(y\ x)$. We do not want to go into more detail here – [8] provides a full and more precise description of the syntax. In the following, we use the language VEX to clarify the notions of graphs, graph grammars and parsing. Furthermore, the benefits of our actual parser combinator library are demonstrated by constructing a special-purpose parser for VEX graphs in Sect. 6.

3 Graphs

Our framework deals with a generalized notion of graphs where edges are allowed to visit an arbitrary number of nodes. Such graphs are often called hypergraphs. In this section we give a formal definition of these graphs and Haskell types for their representation.

Following [3] a graph consists of a set of edges and a set of nodes. An edge is an atomic item with a fixed number of tentacles, called the type of the edge. It can be embedded into a graph by attaching each of its tentacles to a node. Directed graphs are a special case of this notion, i.e., they are graphs whose edges are distinguished by exactly two tentacles, the first representing the source of the edge and the second the target, respectively.

Graphs are a flexible modeling concept suitable, e.g., for modeling a large number of different visual languages [9]. Using VEX as an example we briefly sketch how syntax analysis for diagrams can be performed. It is usually conducted by a chain of processing steps. For instance, in DiaGen [1] first a so-called spatial relationship graph (SRG) is created. This SRG contains a component edge for every diagram component as well as spatial relationship edges (like *contains* or *touches*) linking these components via their attachment nodes. A simple example is depicted in Fig. 4.

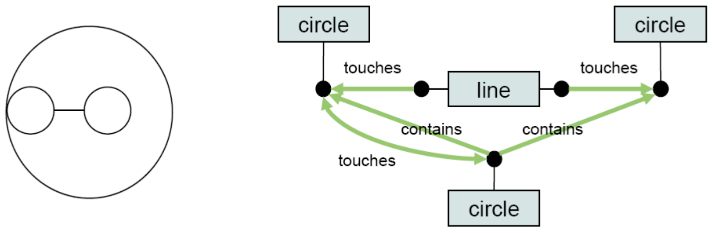


Fig. 4. Spatial relationship graph of a diagram representing the identity function

We represent edges by a rectangular box marked with a particular label and nodes by filled black circles. A line between an edge and a node indicates that the node is visited by that edge. Spatial relationship edges are directed, binary edges, so that we can represent them as colored arrows in the figure for clarity's sake.

In a second step this SRG is transformed to the reduced graph model (or abstract syntax graph) by the so-called reducer. This step is closely related to lexical analysis in a string setting. The reduction of complexity allows more readable grammars and often reasonably efficient parsers.[1] The derivation structure (if any) yielded by the parser can finally be used for attribute evaluation and diagram beautification.

In Fig. 5 the reduced graph model of Fig. 3b is shown. Nodes additionally are labeled with their identifiers. The free variable root node in the upper left corner of Fig. 3b, e.g., is mapped to an edge labeled "freevar". The reference of the variable to this root node is mapped to an edge "bind" that links the edges "freevar" and "var".

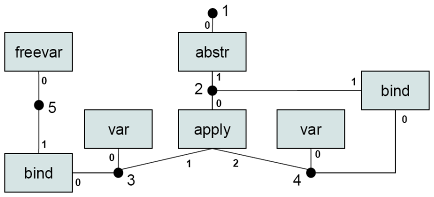


Fig. 5. Graph representation of the VEX expression of Fig. 3b

The small numbers close to the edges are the tentacle numbers. Without these numbers the image may be ambiguous, since different tentacles usually play different roles. For instance, the tentacle with number 1 of "abstr" edges always has to be attached to the body of the abstraction and the tentacles 1 and 2 of "apply" edges to a function and its argument, respectively.

Before we provide the Haskell types representing the graph data structures we have to introduce our graph model more formally. It differs from standard definitions as found in, e.g., [3] that do not introduce the notion of a *context*.

Let C be a set of labels and $type : C \to I\!N$ a typing function for C. In the following, a graph H over C is a finite set of tuples (lab, e, ns), where e is an edge identifier unique in H, $lab \in C$ is an edge label and ns is a sequence of node identifiers such that $type(lab) = |ns|$ (length of sequence). The nodes represented by the node identifiers in ns are called *incident* to edge e. We call a tuple (lab, e, ns) a *context* (in analogy to [10], although our graphs are not inductively defined).

The position of a particular node n in the sequence of nodes within the context of an edge e represents the tentacle of e that n is attached to. Hence the order of nodes matters. The same node identifier also may occur in more than one context indicating that the edges represented by those contexts are connected via this node.

Note, that our notion of graphs is slightly more restrictive than the usual one, because we cannot represent isolated nodes. In particular the nodes of H are implicitly given as the union of all nodes incident to its edges. In fact, in many graph application areas isolated nodes simply do not occur. For example, in our context of visual languages diagram components are represented by edges, and nodes just represent their connection points, i.e., each node is attached to at least one edge. As we will see, our definition is very advantageous for parsing. In particular we can transfer the concept of input consumption straightforwardly by just using contexts as tokens (cf. Sect. 5). So we ignore this issue at the moment in order to keep the technicalities simple.

[1] In case of VEX this reduction is quite complicated. We do not discuss this step here in detail.

The following Haskell code introduces the basic data structures for representing nodes, edges and graphs altogether:

```
type Node = Int
type Edge = Int
type Tentacle = Int
type Context lab = (lab, Edge, [Node])
type Graph lab = Set (Context lab)
```

For the sake of simplicity, we represent nodes and edges by integer numbers. We declare a graph as a set of contexts, where each context represents a labeled edge including its incident nodes. For instance, the VEX graph given in Fig. 5 can be represented as follows:

```
data VexLab = AbstrL | ApplyL | VarL | BindL | FreevarL

vg::Graph VexLab
vg = {(AbstrL,0,[1,2]), (ApplyL,1,[2,3,4]), (VarL,2,[4]),
      (BindL,3,[4,2]), (VarL,4,[3]), (BindL,5,[3,5]), (FreevarL,6,[5])}
```

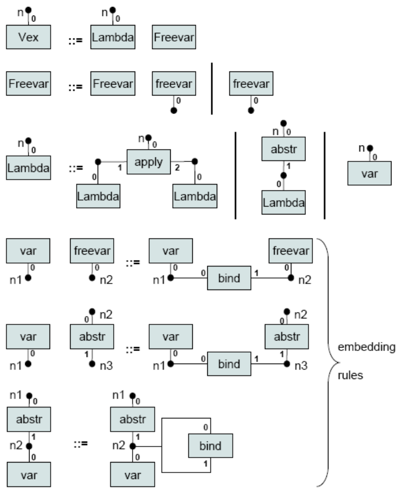
We enumerate the terminal edge labels explicitly as data literals. Using strings also is perfectly possible, but following our approach the type checker already excludes some meaningless input. The node numbers occurring in `vg` correspond directly to the identifiers given in the figure. The edges are numbered uniquely.

4 Graph Grammars and Parsers

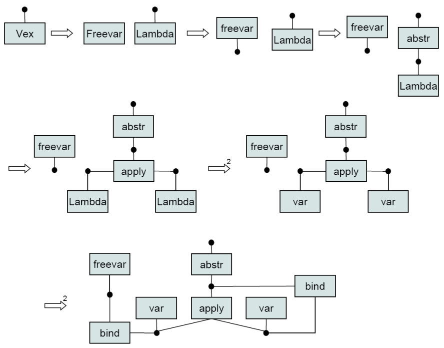
A set of graphs, i.e., a graph language, can be defined using a graph grammar – an extension of formal language theory to graphs. A widely known kind of graph grammar are hyperedge replacement grammars (HRG) as described in [3]. Here, a nonterminal edge of a given graph is replaced by a new graph that is glued to the remaining graph by fusing particular nodes. Formally, such a HRG G is a quadruple $G = (N, T, P, S)$ that consists of a set of nonterminals $N \subset C$ (i.e., labels), a set of terminals $T \subset C$ with $T \cap N = \emptyset$, a finite set of productions P and a start symbol $S \in N$. The productions are context-free, i.e., they describe how a single nonterminal edge can be replaced with a graph.

Unfortunately this notion is not powerful enough to describe VEX; we have to extend it with so-called embedding rules – special productions that insert an additional edge into a particular context. Such context-sensitive embedding rules are needed for the description of many graph languages.

The graph grammar for VEX can be defined as $G_V = (N_V, T_V, P_V, \mathit{Vex})$ where $N_V = \{\mathit{Vex}, \mathit{Lambda}, \mathit{Freevar}\}$, $T_V = \{\mathit{freevar}, \mathit{apply}, \mathit{abstr}, \mathit{var}, \mathit{bind}\}$ and P_V contains the productions given in Fig. 6a. By convention we use lower case letters to label terminal edges; in contrast labels of nonterminal edges start with a capital letter. The productions are written very similar to BNF known from string grammars, i.e., left-hand side *lhs* and right-hand side *rhs* are separated by the symbol `::=` and several *rhs* of one and the same *lhs* can be combined by vertical bars. Node labels are used to identify corresponding nodes of *lhs*



(a) Productions of the grammar



(b) Derivation of the graph given in Fig. 5

Fig. 6. The graph language of VEX

and *rhs*. They act as variables. In order to apply a production they have to be instantiated with nodes actually occurring in the graph.

As usual the language defined by a grammar is given by all terminal graphs derivable in an arbitrary number of steps from the start symbol. Note again, that the grammar of VEX is not context-free, i.e., there are productions with more than one edge at their *lhs*, although the graph language is quite simple. In particular the embedding of the "bind" edge cannot be defined in a context-free manner. Thus, the derivation is not a tree, but a DAG. The given grammar also does not ensure that there has to be exactly one "bind" edge connected to each variable. Such restrictions usually have to be expressed by so-called application conditions (see, e.g., [1]). A derivation of our example VEX graph (Fig. 5) is given in Fig. 6b.

A general-purpose graph parser for HRGs gets passed a particular HRG and a graph as parameters and constructs a derivation tree of this graph according to the grammar. This can be done, for instance, in a way similar to the well-known algorithm of Cocke, Younger and Kasami [2] in the context of strings (all HRGs can be transformed to the graph equivalent of the string notion Chomsky Normal Form). However, such a parser could not deal with our VEX grammar, because of the embedding rules.

Another problem regarding this grammar is caused by the "Freevar" productions. Since the terminal "freevar" edges are not connected in a specific way a lot of different derivations are possible. It is simply not clear, in which order the different "freevar" edges have to be derived (although their actual order does not matter at all). A general-purpose parser usually performs poorly on such highly ambiguous grammars.

Having this in mind, we can state, that a general-purpose parser capable of parsing VEX graphs has to be quite powerful. Hence the language VEX strongly motivates the construction of a special-purpose parser.

5 A Graph Parser Combinator Library

The main contribution of this paper is the proposal of the combinator approach to graph parsing. To validate this concept we provide a prototypical implementation of a respective library in this section. We later demonstrate that therewith the construction of special-purpose parsers can be greatly simplified.

Our design goals have been:

– Intuitive look and feel, i.e., short training period for people already familiar with parser combinators.
– Straightforward translation of a grammar to a parser.
– Simple parsers for simple languages even if the grammar of the language is complicated.[2]
– Sufficient performance for practically relevant applications.

In implementing our parsers we do not start from scratch. Rather we use PolyParse [6] as a base: a light-weight monadic parser combinator library already mentioned in the introduction. PolyParse has in particular appeared to be well-suited for practical applications. Thus, our approach has the additional benefit that many users are already familiar with the usage of the proposed framework.

However, we have to adapt the library a little to deal with sets of tokens rather than lists. The new type `Parser` can be defined as follows:

```
newtype Parser s t a = P (s → Set t → (EitherE Error a, s, Set t))
```

The type parameter `t` defines the type of the tokens. As a witness of success the parser returns a value of type `a`. In addition to the set of tokens, a state of type `s` is carried along. For instance, we will need such a state when parsing VEX to keep track of the root nodes in scope. `EitherE` is a type similar to `Either` that additionally provides support for different gradations of failing.

The existing instance of the type class `Monad` for the type `Parser` can be reused without any changes. Thus, we also

– hide the remaining input,
– keep track of an explicit state, which can be updated while parsing,
– allow further parsers to be parameterized with the results of previous ones (cf. `abcG`),
– support the convenient construction of results with `return`.

However, we cannot reuse several primitive parsers from PolyParse (mainly `next::Parser s t t`) since they depend on the first token of the input list,

[2] A good example is the language of the string graphs $\{a^k b^k c^k | k \in I\!N\}$. In contrast to the string language there is a context-free graph grammar describing this language, however, it is quite complicated despite the simplicity of the language (cf. [3]).

Table 1. Set-specific primitive parsers and combinators

Name	Type	Description
`aSatisfying`	`(t->Bool)-> Parser s t t`	a token satisfying a particular condition, nondeterministic and consuming
`remainingInp`	`Parser s t (Set t)`	the whole remaining input, not consuming
`satisfyingInp`	`(t->Bool)-> Parser s t (Set t)`	all tokens satisfying a particular condition, consuming
`oneOf`	`[Parser s t a]-> Parser s t a`	chooses the first parser in the list that succeeds
`best`	`[Parser s t a]-> Parser s t a`	chooses the parser consuming the largest part of the input successfully
`eoi`	`Parser s t ()`	for complete input consumption, otherwise a correct subgraph is extracted

which is meaningless in the context of sets. Instead we have to add several set-specific primitive parsers listed in Table 1 (we omit their declarations here, they are rather technical). The use of the also listed combinator `best` will become clear shortly.

Note, that up to now we have not defined any graph-specific combinators. Indeed our basic framework can be used to parse all kinds of token sets.

After this preparatory work we can define graph-specific combinators quite conveniently. However, let us first define our basic graph parser type by specializing the type `Parser`. Thereby we choose contexts (as introduced in Sect. 3) as our token type. The name `Grappa` is an acronym for graph parsing:

```
type Grappa s lab a = Parser s (Context lab) a
```

A main difference between graph parsing and string parsing is that in a graph setting we normally do not know where to actually start parsing. There is no such thing as a first token/context. Thus, most of our parsers will need a start node as a parameter explicitly. For convenience, we define an additional type `NGrappa` that hides this node parameter:

```
type NGrappa s lab a = Node→Grappa s lab a
```

Most combinators will return a result of type `NGrappa`. The following function converts such an `NGrappa` to a normal `Grappa` by trying every node of the graph as a starting point successively.

```
nGrappaToGrappa::NGrappa s lab a→Grappa s lab a
nGrappaToGrappa ng = do
                     g←remainingInp
                     best (map ng (nodes g))
```

Using `remainingInp` the current set of tokens, i.e., graph, is queried (but not consumed). Thereafter the combinator `best` is used to find the best starting node, i.e., the one the largest part of the input can be successfully consumed from.[3] We construct the list of possible parsers by mapping `ng` to all nodes of the graph. Of course this function has to be used with care – at least if performance is an issue. However, its declaration demonstrates how flexibly parser combinators can be composed.

We further provide a combinator `oneOfN` for dealing with alternatives based on type `NGrappa`.

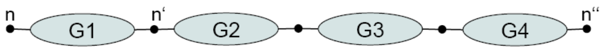
```
oneOfN::[NGrappa s lab a]→NGrappa s lab a
oneOfN ngs n = oneOf (map ($n) ngs)
```

In Table 2 we briefly sketch some of the graph-specific primitive parsers provided by our library. They are all nondeterministic and consume the context they return. The parser `dirEdge` from the introductory example (Fig. 1b) is a specialization of `edge` that uses 0 as the incoming and 1 as the outgoing tentacle number, respectively. Those numbers correspond to source and target of a binary edge.

Table 2. Some graph-specific primitive parsers

Name	Type	Description
`context`	`(Context l->Bool)-> Grappa s l (Context l)`	a context satisfying a particular condition
`labContext`	`Eq l=>l-> Grappa s l (Context l)`	a context with a particular label
`edge`	`Eq l=>l->Tentacle->Tentacle-> NGrappa s l (Node,Context l)`	a labeled context connected to the active node via a particular tentacle also returning its successor (via the other, outgoing tentacle)
`connLab-Context`	`Eq l=>l->[(Tentacle,Node)]-> Grappa s l (Context l)`	a labeled context connected to the given nodes via the given tentacles

We omit the implementation details of the primitive parsers and rather switch the focus to combinators. A first important combinator is `chain`, in a sense the graph equivalent of `many`. It can be used to parse several successive graphs connected via intermediary nodes as shown in the figure below.



Therefore, a given `NGrappa` is applied as long as possible assuring proper connections between the different parses. As a result the list of the partial results

[3] This is an expensive operation. However, the alternative of returning an arbitrary result is often not meaningful in a graph setting. For instance, the parser `abcG` would always succeed immediately returning $k = 0$.

is returned. The connecting nodes are especially important, because they have to be provided as starting points for the parser one at a time. This is realized by passing through the active node, i.e., the parser has to also return the node where it stops.

```
chain::NGrappa s lab (Node,a)→NGrappa s lab (Node,[a])
chain p n = do {
            (n',x)←p n;
            (n'',xs)←chain p n';
            return (n'',x:xs)
            } ‘onFail‘ return (n,[])
```

The string combinator `exactly` used in the introduction can be carried over to graphs quite easily, too:

```
exactChain::Int→NGrappa s lab (Node,a)→NGrappa s lab (Node,[a])
exactChain 0 p n = return (n,[])
exactChain num p n = do
                     (n',x)←p n
                     (n'',xs)←exactChain (num-1) p n'
                     return (n'',x:xs)
```

Note, that with `chain` and `exactChain` all combinators used in the definition of our motivating example `abcG`, i.e., the parser for the string graphs $\{a^k b^k c^k | k \in \mathbb{N}\}$, are introduced already.

In addition to these combinators several parser combinators known from string parsing can be reused for graph parsing straightforwardly, e.g., `oneOf`. Furthermore, there are some combinators that have to be used with care. Their semantics changes in a graph setting, because they do not maintain proper connections (unlike strings the order of tokens, i.e., contexts, does not represent a particular kind of connection anymore). For instance, the combinator `many` can be used in the context of graphs just to parse more or less independent subgraphs or star shapes (when dealing with `NGrappas`):

```
star::NGrappa s lab a→NGrappa s lab [a]
star ng = many ∘ ng
```

Note, that several combinators and associated parsers have to pass through the active node. In particular parsers for languages of string graphs, therefore, include some boilerplate code (cf. Fig. 1b). It is common practice to hide this complexity in an additional state monad. However, practical graph languages usually are more branched. Due to this property the monadic approach results in frequent updates of this state. Thus, we prefer passing the active node as a function parameter explicitly.

There are a lot more combinators imaginable. Our library, e.g., provides further combinators for dealing with trees, separators, etc. However, the ones presented in this section give a good first insight. Furthermore, they are in particular needed to tackle our example VEX.

6 Parsing VEX

In this section we construct a parser for the example language VEX using the combinators defined in the previous section. The purpose of the presentation of this parser is twofold. First, we demonstrate how the combinators have to be used. And second, we intend to show that special-purpose graph parsers can be defined quite straightforwardly.

Our goal is to map a VEX graph to the λ-term it represents. Hence, we define the type `Lambda` as the result type of the parser.

```
data Lambda = Abstr String Lambda |
              Apply Lambda Lambda |
              Var String
```

Note, that `Lambda` is a kind of tree and not a DAG as one might expect, because variable bindings are resolved by proper naming. Here an additional advantage of parser combinators shows up. We can compute a useful result while parsing and need not post-process an intermediate data structure somehow representing the derivation.

VEX is an example where we really need a parser state. In particular we store a number used to construct a name for the next fresh variable to be bound in an abstraction. Moreover, the state contains a lookup table that maps node numbers to variable names. The type `VexState` represents these requirements.

```
type VexState = (Int,[(Node,String)])
```

The top-level parser `vex` first consumes all (`many`) "freevar" edges using the parser `freevar`. Thereby the lookup table is extended properly introducing fresh names for the new variables. We are not interested in the parsing order of the "freevar" edges. Hence, we `commit` the intermediate results to prevent backtracking. Finally, the remaining graph is parsed using the parser `lambda`.

```
vex::Grappa VexState VexLab Lambda
vex = do
      many (commit freevar)
      nGrappaToGrappa lambda

freevar::Grappa VexState VexLab ()
freevar = do
          (_,_,[n])←labContext FreevarL
          stUpdate (λ(nn,vt)→(nn+1,(n,"v"++show nn):vt))
```

The parser `lambda` accepts either an application (`apply`), an abstraction (`abstr`) or a variable (`var`). All of these subordinated parsers have the same type.

```
lambda,apply,abstr,var::NGrappa VexState VexLab Lambda
lambda = oneOfN [apply, abstr, var]
```

At "apply" edges there are two directions to further process the graph. Both have to be traversed and must yield a valid subgraph. This is a main difference to string parsing where the next token is always predetermined. We deal with this issue by parsing an application depth-first beginning with its left side. Since the diagram language VEX does not support sharing of common subexpressions we can consume recognized input unconditionally anyhow.

```
apply n = do
          (_,_,ns)←connLabContext ApplyL [(0, n)]
          l1←lambda (ns!!1)
          l2←lambda (ns!!2)
          return (Apply l1 l2)
```

The nodes visited via particular tentacles (here, 1 and 2) are accessed by using the tentacle number as a parameter to the list index operator (!!). Both subgraphs have to be parseable with lambda again. The overall result then is composed by the application of the data constructor Apply to the results yielded by parsing these subgraphs.

Parsing an abstraction means introducing a new variable in the lookup table that can be released after parsing its body. The state can be queried and changed using stGet::Parser s t s and stUpdate::(s->s)->Parser s t (), respectively. These functions are provided by PolyParse for dealing with the user-state.

```
abstr n = do
          (_,_,ns)←connLabContext AbstrL [(0, n)]
          oldstate@(nn,vt)←stGet
          stUpdate (const (nn+1,(ns!!1,"v"++show nn):vt))
          l←lambda (ns!!1)
          stUpdate (const oldstate)
          return (Abstr ("v"++show nn) l)
```

Edges labeled “var” represent variables. Additionally, there has to be a “bind” edge to a node that can be mapped to a variable name via the lookup table. Note, that it would even be possible to raise the error level, i.e., prevent backtracking, if there is no corresponding “bind” edge. However, we still backtrack, since we are also interested in finding correct subgraphs.

```
var n = do
        connLabContext VarL [(0, n)]
        (_,_,ns)←connLabContext BindL [(0, n)]
        (_,vt)←stGet
        case lookup (ns!!1) vt of
          Nothing→fail $ show (ns!!1) ++
                           " out of scope!"
          Just v→return (Var v)
```

The parser for VEX graphs can be called using the function vexParse that applies the PolyParse function runParser to the initial state (1,[]) and just returns the result or an error message.

```
vexParse::Graph VexLab→Either Error Lambda
vexParse = getResult ∘ runParser vex (1,[])
```

For our example graph vg it succeeds with the result λv2→(v1 v2).

A Brief Remark on Performance

In [3] it is proved that there are (even context-free) graph languages for which parsing is NP-complete. These languages, of course, cannot be parsed with our combinators efficiently. However, there are also languages on which most general-purpose graph parsers perform poorly, although they can be parsed efficiently.

For instance, even the quite simple language VEX causes a general-purpose parser to run in exponential time (at least, if it is not tweaked somehow). The exponent thereby depends on the number of “freevar” edges, since those can be derived in a lot of different orders ($n!$). The parser presented previously, however, is not affected by this issue: We simply have committed to the first result at any one time. Similar language-specific performance optimizations can be applied to many languages.

We have also measured the execution time to parse the string graphs $a^k b^k c^k$ for several k using a general-purpose parser (also implemented in Haskell). As already mentioned it is possible to describe this graph language with a HRG. It turns out that these string graphs can be parsed in polynomial time regarding this grammar, however, performance is worse than one might expect for such a simple language.

In contrast, the special-purpose parser presented as a motivating example in the introduction can be applied to very large graphs. Even if we do not know the starting node and all nodes have to be tried successively we only get a factor of $3*k$. This still outperforms our general-purpose parser. String graphs are a rather artificial example, though. We have to conduct a more elaborate performance comparison in the future.

7 Related Work

To our best knowledge graph parser combinators have not been considered up to now. So in this section we briefly sketch several related approaches to parsing in general and dealing with graphs in functional languages.

Besides PolyParse [6] there are other parser combinator libraries that are also widely-used. For instance, Parsec [11] is well-known for its high performance and good error reporting capabilities. The main difference between Parsec and PolyParse is that Parsec is predictive by default only backtracking where an explicit `try` is inserted whereas in PolyParse backtracking is the default except where explicitly disallowed by a `commit`. We have to gain more experience with graph parser combinators to judge which approach is better suited in our setting.

In the context of strings the complement of the parser combinator approach is parser generation. Thereby a grammar is given, e.g., in EBNF from which a real parser in a particular language can be generated. The tool Happy [12] can be used to generate such a Haskell parser. However, the advantage of a parser generator for strings – namely that efficient parsers can be generated for nearly arbitrary context-free grammars – does not count that much in a graph setting.

At this point we also have to mention the work of Erwig [10], because our declarations are mainly inspired by his approach. Erwig criticized the imperative style of the algorithms described in, e.g., [13] and proposed a new approach: looking at graphs as inductively defined data types. So he presented a graph declaration where nodes are added inductively one at a time. Incident edges are represented as a part of their so-called contexts. Unfortunately his library [14] does not generalize to hypergraphs and also does not support graph rewriting and parsing.[4]

Also highly related are approaches that aim at the combination of functional programming and graph transformation. At the time of writing a textbook is work in progress that provides an implementation of the categorical approach to graph transformation with Haskell [16]. Since graphs are a category a higher level of abstraction is used to implement graph transformation algorithms. An even more general framework is provided in [17]. The benefit of this approach is its generality since it just depends on categories with certain properties. However, up to now parsers are not considered, so we cannot compare usability and performance.

8 Conclusion and Further Work

In this paper we have introduced graph parser combinators, a novel approach to the construction of special-purpose graph parsers. By applying the combinator approach to the domain of graph parsing we further have demonstrated that this well-known technique can be used to parse all kinds of non-linear structures.

Graph parsing is highly relevant for real world applications as we have demonstrated with an example from the domain of visual languages. Here, graph parsing is crucial for the syntactical analysis of diagrams.

Further we have demonstrated that our library can be flexibly used and easily extended. It allows the construction of graph parsers even for context-sensitive graph grammars. Since the implementation of our combinators is quite similar to the more conventional string-based parser combinator libraries Haskell programmers will be familiar with its usage immediately. In combination with the implemented general-purpose graph parser (not discussed in this paper) we have taken an important first step towards a functional graph parsing library.

As we have backed up by examples our library in its present form is perfectly usable already. Nevertheless a lot of work remains to be done. First of all, we have to provide a solid theoretical foundation. We particularly have to investigate how to simplify the process of writing correct parsers using our framework. Therefore, we need to provide mappings from graph grammar formalisms to parsers on top of our framework. This would permit the more precise evaluation of graph parser combinators with respect to power and performance.

Another area of future research is the extension of the set of combinators. Compared to strings graphs can exhibit many more interesting patterns. And finally, it would be interesting to see how capabilities for error recovery could be

[4] In fact, Erwig discussed termgraph rewriting in [15], however, the presented algorithm is tailored to the problem and cannot be generalized straightforwardly.

added (like [18] for strings) or how a breadth-first search strategy would affect the performance.

All in all, we propose the following as an attractive approach to graph parsing: First, try using a general-purpose graph parser. If it is applicable and reasonably efficient everything is ok. However, if additional flexibility is needed or performance is an issue consider the use of graph parser combinators. The library presented in this paper permits the rapid construction of special-purpose graph parsers. Thereby, language-specific performance optimizations can be incorporated easily.

Acknowledgements

We would like to thank Malcolm Wallace for his useful comments regarding PolyParse, Martin Erwig, whose work on graphs in a functional setting inspired this paper, and the anonymous referees for a lot of valuable suggestions.

References

1. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44(2), 157–180 (2002)
2. Kasami, T.: An efficient recognition and syntax analysis algorithm for context free languages. Scientific Report AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachussetts (1965)
3. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. I, pp. 95–162. World Scientific, Singapore (1997)
4. Hutton, G., Meijer, E.: Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996)
5. Johnson, S.C.: Yacc: Yet another compiler compiler. Technical Report 32, Bell Laboratories, Murray Hill, New Jersey (1975)
6. Wallace, M.: PolyParse (2007), `http://www.cs.york.ac.uk/fp/polyparse/`
7. Peyton Jones, S.: Haskell 98 Language and Libraries. The Revised Report. Cambridge University Press (2003)
8. Citrin, W., Hall, R., Zorn, B.: Programming with visual expressions. In: Haarslev, V. (ed.) Proc. 11th IEEE Symp. Vis. Lang, pp. 294–301. IEEE Computer Soc. Press, Los Alamitos (1995)
9. Minas, M.: Hypergraphs as a uniform diagram representation model. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 281–295. Springer, Heidelberg (2000)
10. Erwig, M.: Inductive graphs and functional graph algorithms. J. Funct. Program. 11(5), 467–492 (2001)
11. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Dept. of Comp. Science, Universiteit Utrecht (2001)
12. Gill, A., Marlow, S.: Happy - the parser generator for Haskell, `http://www.haskell.org/happy`
13. King, D.: Functional Programming and Graph Algorithms. PhD thesis, University of Glasgow (1996)

14. Erwig, M.: FGL - A Functional Graph Library, `http://web.engr.oregonstate.edu/~erwig/fgl/haskell/`
15. Erwig, M.: A functional homage to graph reduction. Technical Report 239, FernUniversität Hagen (1998)
16. Schneider, H.J.: Graph transformations - an introduction to the categorical approach (2007), `http://www2.cs.fau.de/~schneide/gtbook/`
17. Kahl, W., Schmidt, G.: Exploring (finite) Relation Algebras using Tools written in Haskell. Technical Report 2000-02, Fakultät für Informatik, Universität der Bundeswehr, München (2000)
18. Swierstra, S.D., Azero Alcocer, P.R.: Fast, error correcting parser combinators: a short tutorial. In: Pavelka, J., Tel, G., Bartosek, M. (eds.) SOFSEM 1999. LNCS, vol. 1725, pp. 111–129. Springer, Heidelberg (1999)