Parsing of Hyperedge Replacement Grammars with Graph Parser Combinators

Steffen Mazanek[1] **and Mark Minas**[2]

[1] steffen.mazanek@unibw.de
[2] mark.minas@unibw.de
Institut für Softwaretechnologie
Universität der Bundeswehr München, Germany

Abstract: Graph parsing is known to be computationally expensive. For this reason the construction of special-purpose parsers may be beneficial for particular graph languages. In the domain of string languages so-called parser combinators are very popular for writing efficient parsers. Inspired by this approach, we have proposed *graph parser combinators* in a recent paper, a framework for the rapid development of special-purpose graph parsers. Our basic idea has been to define primitive graph parsers for elementary graph components and a set of combinators for the flexible construction of more advanced graph parsers. Following this approach, a declarative, but also more operational description of a graph language can be given that is a parser at the same time.

In this paper we address the question how the process of writing correct parsers on top of our framework can be simplified by demonstrating the translation of *hyperedge replacement grammars* into graph parsers. The result are recursive descent parsers as known from string parsing with some additional nondeterminism.

Keywords: graph parsing, functional programming, parser combinators, hyperedge replacement grammars

1 Introduction

Graph languages are widely-used nowadays, e.g., for modeling and specification. For instance, we have specified visual languages using graph grammars [Min02]. In this context we are particularly interested in solving the membership problem, i.e., checking whether a given graph belongs to a particular graph language, and parsing, i.e., finding a corresponding derivation. 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, Younger and Kasami [Kas65], graph parsing is computationally expensive. There are even context-free graph languages the parsing of which is NP-complete [DHK97]. 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.

Unfortunately the development of a special-purpose graph parser is an error-prone and time-consuming task. The parser has to be optimized such that it is as efficient as possible, but still correct. Backtracking, for instance, has to be prevented wherever possible. Therefore, in a recent paper [MM07] we have proposed *graph parser combinators*, a new approach to graph parsing that allows the rapid construction of special-purpose graph parsers. Further we have introduced a Haskell [Pey03] library implementing this approach. It provides the generic parsing framework and a predefined set of frequently needed combinators.

In [MM07] we further have demonstrated the use of this combinator framework by providing an efficient special-purpose graph parser for VEX [CHZ95] as an example. VEX is a graph language for the representation of lambda terms. The performance gains mainly have resulted from the fact, that VEX is context-sensitive and ambiguous – properties many general-purpose graph parsers do not cope well with. The structure of VEX graphs is quite simple though, i.e., they basically are trees closely reflecting the structure of lambda terms. Only variable occurrences are not identified by names as usual; they rather have to refer to their binding explicitly by an edge. Nevertheless, the parser for VEX could be defined quite operationally as a tree traversal.

However, the operational description of languages like, e.g., structured Flowgraphs is much more difficult. Parsers get very complex and hard to read and verify. This brings up the question, whether graph parser combinators actually are powerful enough to express standard graph grammar formalisms. One such formalism are hyperedge replacement grammars [DHK97], which allow such languages to be described in a declarative and natural way.

Therefore, the main contribution of this paper is a method for the straightforward translation of hyperedge replacement grammars [DHK97] to parsers on top of our framework. The resulting parsers are readable and can be customized in a variety of ways. They are quite similar to top-down recursive descent parsers as known from string parsing where nonterminal symbols are mapped to functions. Unfortunately, in a graph setting we have to deal with additional nondeterminism: besides different productions for one and the same nonterminal, we also have to guess particular nodes occurring in the right-hand side of a production. We can use backtracking, but performance naturally suffers.

However, our approach can be used to build an initial, yet less efficient parser. Language-specific performance optimizations can then be used to improve the parser's efficiency step by step. Moreover, the presented approach offers the following benefits:

- Combination of declarative and operational description of the graph language.
- An application-specific result can be computed.[1]
- Context information can be used to describe a much broader range of languages.
- Robust against errors. The largest valid subgraph is identified.

This paper is structured as follows: We discuss the combinator approach to parsing in Sect. 2 and introduce our graph model in Sect. 3. We go on with the presentation of our framework in Sect. 4 and discuss the actual mapping of a hyperedge replacement grammar in Sect. 5. Finally, we discuss related work (Sect. 6) and conclude (Sect. 7).

[1] A general-purpose parser normally returns a derivation sequence or a parse tree, respectively. Several systems, however, provide support for attributed graph grammars.

2 Parser Combinators

Our approach has been inspired by the work of Hutton and Meijer [HM96] who have proposed monadic parser combinators for string parsing (although the idea of parser combinators actually is much older). The basic principle of such a parser combinator library is that primitive parsers are provided that can be combined into more advanced parsers using a set of powerful combinators. For example, there are the sequence and choice combinators that can be used to emulate a grammar. However, a wide range of other combinators are also possible. For instance, parser combinator libraries often include a combinator `many` that applies a given parser multiple times, while collecting the results.

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 [Joh75] no extra formalism is needed to specify the grammar. Functional languages are particularly well-suited for the implementation of combinator libraries. Here, a parser basically is a function (as we will see). A combinator like choice then is a higher-order function. Higher-order functions, i.e., functions whose parameters are functions again, support the convenient reuse of existing concepts [Hug89]. For instance, consider a function `symb` with type `Char->Parser` that constructs a parser accepting a particular symbol. Then we can easily construct a list `pl` of parsers, e.g., by defining `pl=map symb ['a'..'z']` (applies `symb` to each letter). A parser `lcl` that accepts an arbitrary lower-case letter then can be constructed by folding `pl` via the choice operator, i.e., `lcl=foldr choice fail pl`. Thereby, `fail` is the neutral element of `choice`.

At this point, we provide a toy example to give an impression of how a parser constructed with monadic combinators looks like. For that purpose we compare the implementation of a parser for the string language $\{a^k b^k c^k | k > 0\}$ and a graph parser for the corresponding language of string graphs as defined in [DHK97].

An important advantage of the combinator approach is that a more operational description of a language can be given. For instance, our exemplary language of strings $a^k b^k c^k$ 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 the same number of b characters and finally accept the same number of c characters."

Using PolyParse [Wal07], a well-known and freely-available parser combinator library for strings, a parser for this string language can be defined as shown in Fig. 1a. The type of this parser determines that the tokens are characters and the result a number, i.e., k for a string $a^k b^k c^k$.

```
abc::Parser Char Int
abc =
 do
 as<-many1 (char 'a')
 let k=length as
 exactly k (char 'b')
 exactly k (char 'c')
 return k
```

```
abcG::Node->Grappa Int
abcG n =
 do
 (n',as)<-chain1 (dirEdge "a") n
 let k=length as
 (n'',_)<-exactChain k (dirEdge "b") n'
 exactChain k (dirEdge "c") n''
 return k
```

Figure 1: Parsers for a) the string and b) the graph language $a^k b^k c^k$

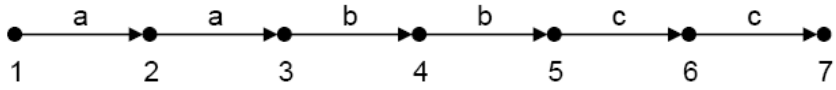

Figure 2: The string graph "aabbcc"

If the given word does not begin with a member of the language one of the calls of `exactly` fails.

The code is written in the functional programming language Haskell [Pey03]. The given parser uses the `do`-notation, syntactic sugar Haskell provides for dealing with monads. Monads in turn provide a means to simulate state in Haskell. In the context of parsers they are used to hide yet unconsumed input. Otherwise, all parsers in a sequence would have to pass this list as a parameter explicitly. Users of the library, however, do not have to know how this works in detail. They rather can use the library like a domain-specific language for parsing nicely embedded into a fully-fledged programming language.

In order to motivate our combinator approach to graph parsing, 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".[2]

A graph parser for this graph language can be defined using our combinators in a manner quite similar to the parser discussed above. It is 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 n we currently process.

3 Graphs

In this section we introduce hypergraphs and the basic Haskell types for their representation. Our graph model differs from standard definitions as found in, e.g., [DHK97], that do not introduce the notion of a *context*.

Let C be a set of labels and $type : C \to \mathbb{N}$ a typing function for C. In the following, a hypergraph H over C is a finite set of tuples (lab, e, ns), where e is a (hyper-)edge[3] identifier unique in H, $lab \in C$ is an edge label and ns is a sequence of node identifiers such that $type(lab) = |ns|$, the length of the 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 [Erw01].

The position of a particular node n in the sequence of nodes within the context of an edge e represents the so-called tentacle of e that n is attached to. Hence the order of nodes matters.

[2] In contrast to the string language there is a context-free hyperedge replacement grammar describing this language. However, it is quite complicated despite the simplicity of the language (cf. [DHK97]). An Earley-style parser for string generating hypergraph grammars like this is discussed in [SF04].

[3] We call hyperedges just edges and hypergraphs just graphs if it is clear from the context that we are talking about hypergraphs.

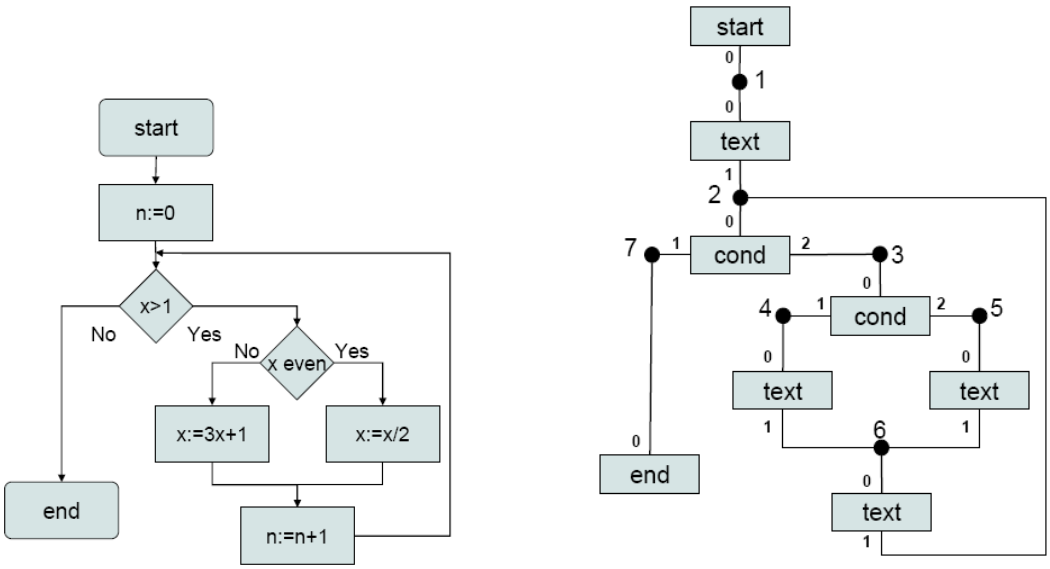

Figure 3: An exemplary Flowchart a) and its hypergraph representation b)

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 hypergraphs 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 hypergraph application areas isolated nodes simply do not occur. For example, in the context of visual languages diagram components can be represented by hyperedges, and nodes just represent their connection points, i.e., each node is attached to at least one edge [Min02].

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 = (String, Edge, [Node])
type Graph = Set Context
```

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.[4]

Throughout this paper we use Flowcharts as a running example. In Fig. 3a a structured Flowchart is given. Syntax analysis of structured Flowcharts means to identify the represented structured program (if any). Therefore, each Flowchart must have a unique entry and a unique exit point.

[4] In the actual implementation these types are parameterized and can be used more flexibly.

Flowcharts can be represented by hypergraphs that we call Flowgraphs in the following. In Fig. 3b the hypergraph representation of the exemplary Flowchart is given. Hyperedges are represented by a rectangular box marked with a particular label. For instance, the statement `n:=0` is mapped to a hyperedge labeled "text". The filled black circles represent nodes that we have additionally marked with numbers. A line between a hyperedge and a node indicates that the node is visited by that hyperedge.

The small numbers close to the hyperedges are the tentacle numbers. Without these numbers the image may be ambiguous. For instance, the tentacle with number 0 of "text" hyperedges always has to be attached to the node the previous statement ends at whereas the tentacle 1 links the statement to its successor. The Flowgraph given in Fig. 3b is represented as follows using the previous declarations:

```
fcg = {("start",0,[1]), ("text",1,[1,2]), ("cond",2,[2,7,3]),
       ("cond",3,[3,4,5]), ("text",4,[4,6]), ("text",5,[5,6]),
       ("text",6,[6,2]), ("end",7,[7])}
```

The language of Flowgraphs can be described using a hyperedge replacement grammar in a straightforward way as we see in the next section. We provide a special-purpose parser for Flowgraphs on top of our framework in Sect. 5.

4 Parsing Graphs with Combinators

In this section we introduce our graph parser combinators. However, first we clarify the notion of parsing in a graph setting.

4.1 Graph Grammars and Parsers

A widely known kind of graph grammar are hyperedge replacement grammars (HRG) as described in [DHK97]. Here, a nonterminal hyperedge of a given hypergraph is replaced by a new hypergraph 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 $N \subset C$, 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 graph grammar for Flowgraphs can be defined as $G_{FC} = (N_{FC}, T_{FC}, P_{FC}, FC)$ where $N_{FC} = \{FC, Stmts, Stmt\}$, $T_{FC} = \{start, end, text, cond\}$ and P_{FC} contains the productions given in Fig. 4a. Left-hand side *lhs* and right-hand side *rhs* of each production are separated by the symbol `::=` and several *rhs* of one and the same *lhs* are separated by vertical bars. Node numbers are used to identify corresponding nodes of *lhs* and *rhs*.

The derivation tree of our exemplary Flowgraph as introduced in Fig. 3b is given in Fig. 4b. Its leaves are the terminal edges occurring in the graph whereas its inner nodes are marked with nonterminal edges indicating the application of a production. The direct descendants of an inner node represent the edges occurring in the *rhs* of the applied production. The numbers in parentheses thereby identify the nodes visited by the particular edge.

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

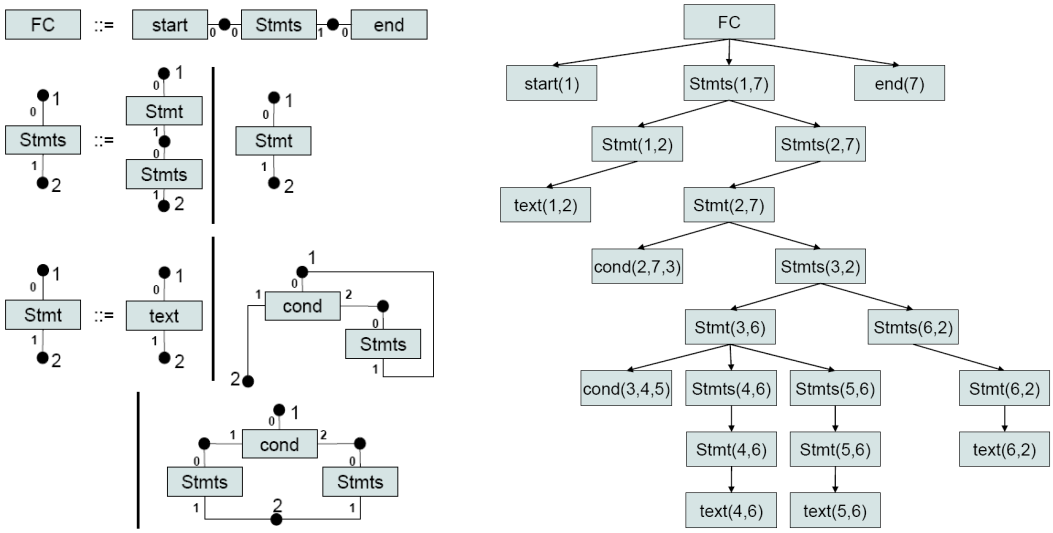

Figure 4: Flowgraphs, a) grammar and b) derivation tree of the example

done, for instance, in a way similar to the well-known algorithm of Cocke, Younger and Kasami [Kas65] known from string parsing (indeed, all HRGs can be transformed to the graph equivalent of the string notion Chomsky Normal Form). This approach has been elaborated theoretically by Lautemann [Lau89] and proven to be useful in practical applications, e.g., in [Min02] for the syntax analysis of diagrams.

Flowgraphs can be parsed with such a general-purpose graph parser in a straightforward way. However, as mentioned in the introduction there are graph languages that are not context-free (and thus cannot be described by a HRG) or that are highly ambiguous (thus causing most general-purpose parsers to perform poorly). Furthermore, here we are not interested in the derivation tree, but rather in the program represented by the graph, i.e., its semantics. For these reasons graph parser combinators are beneficial either way. We now briefly introduce the framework and describe how the HRG of Flowgraphs (and other HRGs similarly) can be translated into a graph parser on top of our framework.

4.2 The Combinator Library

Due to space restrictions in the following we focus on those types and functions that are needed to translate hyperedge replacement grammars schematically. Further information and a more general version of the framework can be found in [MM07]. First we provide the declaration of the type `Grappa` representing a graph parser:

```
newtype Grappa res = P (Graph -> (Either res Error, Graph))
```

This type is parameterized over the type `res` of the result. Graph parsers basically are functions from graphs to pairs consisting of the parsing result (or an error message, respectively) and the graph that remains after successful parser application.

Name	Type	Description
`context`	`(Context->Bool)-> Grappa Context`	A context satisfying a particular condition.
`labContext`	`String->Grappa Context`	A context with a particular label.
`connLabContext`	`String->[(Tentacle,Node)]-> Grappa Context`	A labeled context connected to the given nodes via the given tentacles.
`edge`	`Tentacle->Tentacle->String-> Node->Grappa (Node,Context)`	A labeled context connected to the given node via a particular tentacle also returning its successor (via the other, outgoing tentacle).
`dirEdge`	`String->Node-> Grappa (Node,Context)`	A directed edge, `edge 0 1`.

Table 1: Graph-specific primitive parsers

Name	Type	Description
`oneOf`	`[Grappa res]->Grappa res`	Returns the first successful parser of the input list, corresponds to `|` in grammars.
`chain`	`(Node->Grappa (Node, res))-> Node->Grappa (Node, [res])`	A chain of graphs, a node is passed through.
`bestNode`	`(Node->Grappa res)-> Grappa res`	Identifies the node from which the best continuation is possible, very expensive.
`noDanglEdgeAt`	`Node->Grappa ()`	Succeeds if the given node is not incident to an edge, handy for ensuring dangling edge condition.
`allDifferent`	`[Node]->Grappa ()`	Succeeds if the given nodes are distinct, handy for ensuring identification condition.
`connComp`	`Grappa res->Grappa [res]`	Applies the given parser once per connected component, while collecting the results.

Table 2: Some graph parser combinators

In general, the most primitive parsers are `return` and `fail`. Both do not consume any input. Rather `return` succeeds unconditionally with a particular result whereas `fail` always fails; thereby, backtracking is initiated.

In Table 1 we provide some important graph-specific primitive parsers. They are all nondeterministic, i.e., support backtracking on failure, and consume the context they return. Additionally we provide the primitive parser `aNode::Grappa Node` that returns a node of the remaining graph (with backtracking).

In Table 2 we briefly sketch some of the graph parser combinators provided by our library. Variations of `chain` have already been used in the introductory example, e.g., `chain1` that demands at least one occurrence.

```
fc::Grappa Program
fc = do
     (_,_,[n1])<-labContext "start"
     (_,_,[n2])<-labContext "end"
     stmts (n1,n2)

stmts::(Node, Node)->Grappa Program
stmts (n1, n2) = oneOf [stmts1, stmts2]
  where stmts1 = do
                 s<-stmt (n1, n2)
                 return [s]
        stmts2 = do
                 n'<-aNode
                 s<-stmt (n1, n')
                 p<-stmts (n', n2)
                 return (s:p)

stmt::(Node, Node)->Grappa Stmt
stmt (n1, n2) = oneOf [stmt1, stmt2, stmt3]
  where stmt1 = do
                connLabContext "text" [(0,n1),(1,n2)]
                return Text
        stmt2 = do
                (_,_,ns)<-connLabContext "cond" [(0,n1)]
                p1<-stmts ((ns!!1), n2)
                p2<-stmts ((ns!!2), n2)
                return (IfElse p1 p2)
        stmt3 = do
                (_,_,ns)<-connLabContext "cond" [(0,n1),(1,n2)]
                p<-stmts ((ns!!2), n1)
                return (While p)
```

Figure 5: A parser for Flowgraphs

5 Parsing Flowgraphs

In this section we directly translate the grammar given in Fig. 4a to a parser for the corresponding language using our framework. Our goal is to map a Flowgraph to its underlying program represented by the recursively defined type `Program`:

```
type Program = [Stmt]
data Stmt = Text | IfElse Program Program | While Program
```

In Fig. 5 the parser for Flowgraphs is presented. It is not optimized with respect to performance. Rather it is written in a way that makes the translation of the HRG explicit. For each nonterminal edge label l we have defined a parser function that takes a tuple of nodes $(n_1,...,n_t)$ as a parameter such that $t = type(l)$. Several *rhs* of a production are handled using the `oneOf`

combinator. Terminal edges are matched and consumed using primitive parsers. Thereby their proper embedding has to be ensured. We use the standard list operator `(!!)` to extract the node visited via a particular tentacle from a node list `ns`.

For instance, `stmt3` represents the while-production. First, a "cond"-edge e visiting the nodes n_1 and n_2 via the tentacles 0 and 1, respectively, is matched and consumed. Thereafter the body of the loop is parsed, i.e., the `stmts` starting at the node visited by tentacle 2 of e, i.e., `ns!!2`, ending again at n_1. Finally the result is constructed and returned. If something goes wrong and backtracking becomes necessary, previously consumed input is released automatically.

The parser is quite robust. For instance, redundant components are just ignored and both the dangling and the identification condition are not enforced. These relaxations can be canceled easily – the first one by adding the primitive parser `eoi` (end of input) to the end of the definition of the top-level parser, the others by applying the combinators `noDanglEdgeAt` and `allDifferent`, respectively, to the nodes involved.

Note, that the implementation of `stmts` follows a common pattern, i.e., a chain of graphs between two given nodes. So using a combinator the parser declaration can be further simplified to `stmts=chain1Betw stmt`. Here, `chain1Betw` ensures at least one occurrence as required by the language. Its signature is

```
chain1Betw::((Node,Node)->Grappa a)->(Node,Node)->Grappa [a]
```

and it is defined exactly as `stmts` except from the fact that it abstracts from the actual parser for the partial graphs.

Performance

This parser is not very efficient. A major source of inefficiency is the use of `aNode` that binds a yet unknown node arbitrarily thus causing a lot of backtracking. This expensive operation has to be used only for the translation of those productions, where inner nodes within the *rhs* are not incident to terminal edges visiting an external node, i.e., a node also occurring in the *lhs*.[5] However, even so there are several possibilities for improvement. For instance, we currently try to make this search more targeted by the use of narrowing techniques as known from functional-logic programming languages [Han07]. Performance can be further improved if particular branches of the search space can be cut. For instance, we can prevent backtracking by committing to a (partial) result. In [MM07] we have demonstrated how this can be done in our framework. Finally, we can apply domain-specific techniques to further improve the performance. For instance, a basic improvement would be to first decompose the given graph into connected components and apply the parser to each of them successively. We provide the combinator `connComp` for this task. However, this step can only be applied to certain languages and at the expense of readability.

So we can start with an easy to build and read parser for a broad range of languages. It may be less efficient, however, it can be improved step by step if necessary. Further it can be integrated and reused very flexibly, since it is a first-class object.

[5] The function `aNode` can also be used to identify the start node in our introductory example `abcG`.

6 Related Work

Our parser combinator framework basically is an adaptation of the PolyParse library [Wal07]. The main distinguishing characteristics of PolyParse are that backtracking is the default behavior except where explicitly disallowed and that parsers can be written using monads. There is an abundance of other parser combinator libraries besides PolyParse that we cannot discuss here. However, a particularly interesting one is the UU parser combinator library of Utrecht University [SA99]. It is highly sophisticated and powerful, but harder to learn for a user. Its key benefit is its support for error correction. Hence a parser does not fail, but a sequence of correction steps is constructed instead.

Approaches to parsing of particular, restricted kinds of graph grammar formalisms are also related. For instance, in [SF04] an Earley parser for string generating graph languages has been proposed. The diagram editor generator DiaGen [Min02] incorporates an HRG parser that is an adaptation of the algorithm of Cocke, Younger and Kasami. And the Visual Language Compiler-Compiler VLCC [CLOT97] is based on the methodology of positional grammars that allows to parse restricted kinds of flex grammars (which are essentially HRGs) even in linear time. These approaches have in common that a restricted graph grammar formalism can be parsed efficiently. However, they cannot be generalized straightforwardly to a broader range of languages like our combinators.

We have demonstrated that semantics can be added very flexibly in our framework. The graph transformation system AGG also provides a flexible attribution concept. Here, graphs can be attributed by arbitrary Java objects [Tae03]. Rules can be attributed with Java expressions allowing complex computations during the transformation process. AGG does not deal with hypergraphs. However, it can deal with a broad range of graph grammars. These are given as so-called parse grammars directly deconstructing the input graph. Critical pair analysis is used to organize reverse rule application.

In [RS95] a parsing algorithm for context-sensitive graph grammars with a top-down and a bottom-up phase is discussed. Thereby first a set of eventually useful production applications is constructed bottom-up. Thereafter viable derivations from this set are computed top-down. Parser combinators generally follow a top-down approach, although in a graph setting bottom-up elements are beneficial from a performance point of view.

Finally there are other approaches that aim at the combination of functional programming and graph transformation. Schneider, for instance, currently prepares a textbook that provides an implementation of the categorical approach to graph transformation with Haskell [Sch07]. 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 [KS00]. The benefit of their approach is its generality since it just depends on categories with certain properties. However, up to now parsing is not considered.

7 Concluding Remarks

In this paper we have discussed graph parser combinators, an extensible framework supporting the flexible construction of special-purpose graph parsers even for context-sensitive graph grammars. It already provides combinators for the parsing of several frequently occurring graph patterns. We even may end with a comprehensive collection of reusable parser components.

Parser combinators are best used to describe a language in an operational way. For instance, we have provided a parser for the graph language $a^k b^k c^k$ as a toy example. Similar situations, however, also appear in practical applications as, e.g., discussed in [Kör07]. We further have provided a schema for the straightforward translation of hyperedge replacement grammars into a parser on top of our framework. The resulting parser is not efficient. It is rather a proof of concept. Languages like our exemplary Flowgraphs can be parsed very efficiently using a standard bottom-up parser. However, the main benefit of our framework is that language-specific optimizations can be incorporated easily in existing parsers, e.g., by providing additional information, using special-purpose combinators, heuristics or even a bottom-up pass simplifying the graph.

Parsing generally is known to be an area functional languages excel in. In the context of string parsing a broad range of different approaches have been discussed. However, in particular the popular combinator approach has not been applied to graph parsing yet. With the implementation of our library we have demonstrated that graph parser combinators are possible and beneficial for the rapid development of special-purpose graph parsers.

Future work

Our approach is not restricted to functional languages though. For instance, in [AD01] the translation of string parser combinators to the object-oriented programming language Java is described. We plan to adapt this approach in the future to, e.g., integrate graph parser combinators into the diagram editor generator DiaGen [Min02]. This hopefully will allow the convenient description of even more visual languages.

The parsers presented in this paper suffer from the fact that purely functional languages are not particularly dedicated to deal with incomplete information. For instance, we have discussed why inner nodes occurring in the right-hand sides of productions have to be guessed. Multi-paradigm declarative languages [Han07] like Curry [Han] are well-suited for such kinds of problems. We currently reimplement our library in a functional-logic style to overcome these limitations. This work will also clarify the relation to proof search in linear logic [Gir87]. Here, the edges of a hypergraph can be mapped to facts that can be connected to a parser via so-called linear implication ($\multimap$). During the proof the parser consumes these facts and at the end none of them must be left.

We further plan to investigate error correction-strategies in a graph setting. For instance, in the context of visual language editors based on graph grammars this would allow for powerful content assist. Whereas in a string setting error-correcting parser combinators are well-understood already [SA99], not much has been done with respect to graphs yet. Admittedly, we do not expect to find an efficient solution to this problem.

Bibliography

[AD01] D. S. S. Atze Dijkstra. Lazy Functional Parser Combinators in Java. Technical report UU-CS-2001-18, Department of Information and Computing Sciences, Utrecht University, 2001.

[CHZ95] W. Citrin, R. Hall, B. Zorn. Programming with Visual Expressions. In Haarslev (ed.), *Proc. 11th IEEE Symp. Vis. Lang*. Pp. 294–301. IEEE Computer Soc. Press, 5–9 1995.

[CLOT97] G. Costagliola, A. D. Lucia, S. Orefice, G. Tortora. A Parsing Methodology for the Implementation of Visual Systems. *IEEE Trans. Softw. Eng.* 23(12):777–799, 1997.

[DHK97] F. Drewes, A. Habel, H.-J. Kreowski. Hyperedge Replacement Graph Grammars. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*. Chapter 2, pp. 95–162. World Scientific, 1997.

[Erw01] M. Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.* 11(5):467–492, 2001.

[Gir87] J.-Y. Girard. Linear Logic. *Theoretical Computer Science* 50:1–102, 1987.

[Han] Hanus, M. (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry/`.

[Han07] M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*. Pp. 45–75. Springer LNCS 4670, 2007.

[HM96] G. Hutton, E. Meijer. Monadic Parser Combinators. Technical report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[Hug89] J. Hughes. Why functional programming matters. *Comput. J.* 32(2):98–107, 1989.

[Joh75] S. C. Johnson. Yacc: Yet Another Compiler Compiler. Technical report 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[Kas65] T. Kasami. 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.

[Kör07] A. Körtgen. Modeling Successively Connected Repetitive Subgraphs. In *Applications of Graph Transformations with Industrial Relevance: International Workshop, AGTIVE'07, Kassel, Germany, October 2007. Proceedings*. LNCS, 2007. to appear.

[KS00] W. Kahl, G. Schmidt. Exploring (finite) Relation Algebras using Tools written in Haskell. Technical report 2000-02, Fakultät für Informatik, Universität der Bundeswehr, München, 2000.

[Lau89] C. Lautemann. The Complexity of Graph Languages Generated by Hyperedge Replacement. *Acta Inf.* 27(5):399–421, 1989.

[Min02] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[MM07] S. Mazanek, M. Minas. Graph Parser Combinators. 2007. To appear in Proc. of 19th Internat. Symp. on the Impl. and Appl. of Functional Languages. http://www.unibw.de/steffen.mazanek/dateien/ifl2007

[Pey03] S. Peyton Jones. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge University Press, 2003.

[RS95] J. Rekers, A. Schürr. A parsing algorithm for context sensitive graph grammars. Technical report 95-05, Leiden University, 1995.

[SA99] S. D. Swierstra, P. R. Azero Alcocer. Fast, Error Correcting Parser Combinators: a Short Tutorial. In Pavelka et al. (eds.), *26th Seminar on Current Trends in Theory and Practice of Inform.* LNCS 1725, pp. 111–129. 1999.

[Sch07] H. J. Schneider. Graph Transformations - An Introduction to the Categorical Approach. 2007. `http://www2.cs.fau.de/~schneide/gtbook/`.

[SF04] S. Seifert, I. Fischer. Parsing String Generating Hypergraph Grammars. In Ehrig et al. (eds.), *Graph Transformations*. Lecture Notes In Computer Science 3256, pp. 352–267. Springer, 2004.

[Tae03] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz et al. (eds.), *AGTIVE*. Lecture Notes in Computer Science 3062, pp. 446–453. Springer, 2003.

[Wal07] M. Wallace. PolyParse. 2007. `http://www.cs.york.ac.uk/fp/polyparse/`.