Functional-Logic Graph Parser Combinators

Steffen Mazanek and Mark Minas

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

Abstract. Parser combinators are a popular technique among functional programmers for writing parsers. They allow the definition of parsers for string languages in a manner quite similar to BNF rules. In recent papers we have shown that the combinator approach is also beneficial for graph parsing. However, we have noted as well that certain graph languages are difficult to describe in a purely functional way.

In this paper we demonstrate that functional-logic languages can be used to conveniently implement graph parsers. Therefore, we provide a direct mapping from hyperedge replacement grammars to graph parsers. As in the string setting, our combinators closely reflect the building blocks of this grammar formalism. Finally, we show by example that our framework is strictly more powerful than hyperedge replacement grammars.

We make heavy use of key features of both the functional and the logic programming approach: Higher-order functions allow the treatment of parsers as first class citizens. Non-determinism and logical variables are beneficial for dealing with errors and incomplete information. Parsers can even be applied backwards and thus be used as generators or for graph completion.

1 Introduction

Declarative languages are known to be exceptionally well-suited for building string parsers. Among functional programmers, the probably most popular approach in this domain are parser combinators. Thereby, some primitive parsers are defined that can be combined into more advanced parsers using a set of powerful combinators. These combinators are higher-order functions that can be used to make parsers resemble a grammar very closely [1,2].

Parser combinators integrate seamlessly into the rest of the program, hence the full power of the host language can be used. Unlike parser generators as Yacc, no extra formalism is needed to specify a 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.

Due to these benefits we have started to carry over this approach to the domain of graph parsing recently [3,4]. Graph languages are widely-used nowadays, e.g., for modeling and specification. For instance, we have specified visual languages [5] using so-called hyperedge replacement grammars [6]. There, graphs

A. Voronkov (Ed.): RTA 2008, LNCS 5117, pp. 261–275, 2008.
© Springer-Verlag Berlin Heidelberg 2008

are used as a model for diagrams and a graph parser can be used to check whether a given diagram is syntactically correct.

Hyperedge replacement grammars, HRG for short, are a well-known way of describing languages of hypergraphs, i.e., graphs where edges are allowed to visit an arbitrary number of nodes. Although restricted in power, this formalism comprises several beneficial properties: It is context-free and still quite powerful. Grammars are comprehensible, and reasonably efficient parsers can be defined for practical languages (in general parsing is NP-complete, though). In this context, rewriting means the replacement of a non-terminal hyperedge of a given hypergraph with a new hypergraph that is glued to the remaining graph by fusing particular nodes (cf. [6]).

In [4] we have discussed how HRGs can be translated to parsers using purely functional combinators. The resulting parsers indeed closely resemble the grammar. They are similar to top-down recursive descent parsers known from string parsing where non-terminal symbols are mapped to functions. In addition, the nodes actually visited by a particular non-terminal edge have to be given as function parameters to ensure the proper embedding of the graph the non-terminal is replaced by. However, these parsers suffer from an inherent problem: Inner nodes occurring in the right-hand side of a production are not known in advance, but have to be guessed in order to establish a match. It is a nontrivial task to realize this guessing efficiently in a purely functional language.

In contrast, logic languages excel at dealing with incomplete information. Free variables can be introduced that are instantiated automatically in order to find solutions. Backtracking is the default behavior and does not need to be implemented by hand. Unfortunately, purely logic languages like Prolog do not support the straightforward definition of higher-order functions like our combinators. Thus, the "remaining input" would have to be passed more explicitly resulting in a lot of boilerplate code.[1]

Having this in mind, graph parsing appears to be a domain asking for multi-paradigm declarative programming languages [8]. Those are already known to be well-suited for string parsing [9]. In the domain of graph parsing their benefits stand out even more. The functional-logic framework of graph parser combinators presented in this paper offers the following striking features:

- Straightforward translation of HRGs to reasonably efficient parsers.
- Application-specific results due to a powerful attribution concept.
- Usable context information. This allows the convenient description of several languages that cannot be defined with a HRG.
- Robust against errors. Valid subgraphs can be extracted.
- Bidirectionality. Besides syntax analysis parsers can be used to construct or complete graphs with respect to the language they describe.

[1] Prolog provides Definite Clause Grammars, syntactic sugar to hide the *difference list* mechanism needed to build efficient string parsers in logic languages. However, a graph is not linearly structured, so this notation cannot be used here. Tanaka's Definite Clause Set Grammars [7] are not supported by common Prolog systems.

This paper is organized as follows: In Sect. 2 we introduce HRGs. We continue with the presentation of an excerpt from the actual framework implemented in the functional-logic programming language Curry (Sect. 3). Thereafter, we discuss the parsing of HRGs and provide some examples (Sect. 4). Finally, we sketch the related work (Sect. 5) and conclude (Sect. 6).

2 Hypergraphs and HRGs

In this section we introduce hypergraphs, the notion of graphs our framework is based on, and the HRG formalism [6].

Let C be a set of labels and $type : C \rightarrow I\!N$ a typing function for C. In the following, a hypergraph H over C is a finite multiset of (hyper-)edges[2] $e = (lab, ns)$, where $lab \in C$ is an edge label and ns is a sequence of attachment nodes such that $type(lab) = |ns|$, the length of the sequence. The nodes in ns are called *incident* to (or *visited* by) the edge e. The position of a particular node n in ns represents the so-called tentacle of e that n is attached to. Hence the order of nodes in ns matters.

Note that our notion of hypergraphs is slightly more restrictive than the more common definition given by [6], because we cannot directly represent isolated nodes. Rather 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 visited by at least one edge [5].

Throughout this paper we use structured flowcharts as a running example, i.e., flowcharts that have a unique entry and a unique exit point. In Fig. 1a a structured flowchart is given. Here, syntax analysis means to identify the represented structured program (if any).

Flowcharts can be represented by hypergraphs that we call flowgraphs in the following. In Fig. 1b the hypergraph model of the exemplary flowchart is given. Edges are represented by a rectangular box marked with a particular label. For instance, the statement `n:=0` is mapped to an edge labeled "text". The filled black circles represent nodes that we have additionally marked with numbers. A line between an edge and a node indicates that the node is visited by that edge.

The small numbers close to the edges are the tentacle numbers representing the index of a particular node in ns. Without these numbers the image may be ambiguous. For instance, the tentacle with number 0 of "text" edges always has to be attached to the node the previous statement ends at whereas the tentacle 1 links the statement to its successor.

The language of flowgraphs can be described using a hyperedge replacement grammar in a straightforward way. Formally, such a HRG G is a quadruple $G = (N, T, P, S)$ that consists of a set of non-terminals $N \subset C$, a set of terminals

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

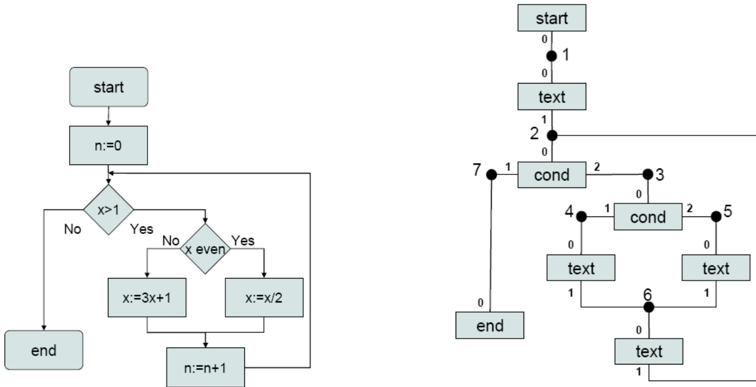


Fig. 1. An exemplary flowchart a) and its hypergraph representation b)

$T \subset C$ with $T \cap N = \emptyset$, a finite set of context-free productions P over N and a start symbol $S \in N$.

The HRG for flowgraphs then 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. 2a. The notation is similar to BNF rules as known from string grammars. Nodes in a production act as variables. In order to apply a production they have to be instantiated with nodes actually occurring in the graph. We use labels to identify corresponding nodes.

As usual, a language defined by a HRG consists of all graphs whose edges are labeled only with terminal labels and that can be derived in an arbitrary number of steps from the start symbol. Given a HRG and a graph, a graph parser constructs a derivation tree of this graph with respect to the grammar. This can be done, for instance, in a way similar to the algorithm of Cocke, Younger and Kasami well-known from string parsing. How this algorithm actually can be adapted to HRGs is discussed in [10,5]. The (unique) derivation tree of the exemplary flowgraph introduced in Fig. 1b is given in Fig. 2b. Its leaves represent the terminal edges occurring in the graph whereas its inner nodes are marked with non-terminal edge labels indicating the application of a production. The direct descendants of an inner node represent the edges the non-terminal is replaced by. Thereby, the numbers in parentheses identify the nodes actually visited by the particular edge.

3 A Basic Combinator-Framework for Graph Parsing

We now introduce the framework as realized in the functional-logic programming language Curry[3]. As we progress, we briefly review some important aspects of Curry to make this paper self-contained.

[3] `http://www.informatik.uni-kiel.de/~curry/report.html`

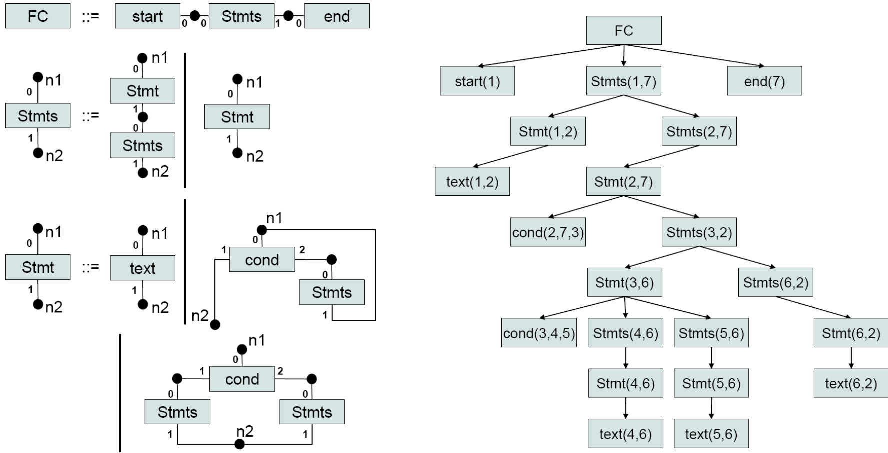


Fig. 2. Flowgraphs, a) grammar and b) sample derivation tree

Curry is a declarative multi-paradigm language combining interesting features from both functional and logic programming [8]. The Curry syntax is very close to Haskell[4]. The main addition are free (logic) variables in conditions and right-hand sides of defining rules. A Curry program consists of definitions of functions and data types on which these functions operate. Functions are defined by conditional equations with constraints in the conditions. They are evaluated lazily and can be called with partially instantiated arguments, a feature we make use of heavily. Function calls with free variables are evaluated by a possibly non-deterministic instantiation of the required arguments, i.e. arguments whose values are necessary to decide the applicability of a rule. This mechanism is called *narrowing* [11].

The following Curry code introduces the basic data structures for representing graphs. For the sake of simplicity, we represent nodes by integer numbers and edge labels by strings (although we do not rely on any particular type at all). Corresponding to the definition in Sect. 2 we declare a graph as a list of labeled edges each with its incident nodes. The actual order of edges does not matter.

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

The flowgraph given in Fig. 1b can be represented as follows using the previous declarations:

```
ex = [("start",[1]),("text",[1,2]),("cond",[2,7,3]),("cond",[3,4,5]),
      ("text",[4,6]),("text",[5,6]),("text",[6,2]),("end",[7])]
```

[4] http://www.haskell.org/onlinereport/

Next, we provide the declaration of the type `Grappa` representing a graph parser. This type is parameterized over the type `res` of the result. Graph parsers are (non-deterministic) functions from graphs to pairs consisting of the parsing result and the graph that remains after successful parser application. In contrast to Haskell, we do not have to deal with parsing errors and backtracking explicitly (no need for "lists of successes"). Instead, similar to [9], we rely on the non-deterministic notion of functions inherent to functional-logic programming languages like Curry.

```
type Grappa res = Graph -> (res, Graph)
```

We proceed by defining some important primitives for the construction of graph parsers. Given an arbitrary value, `pSucceed` always succeeds returning this particular value as a result. In contrast, `eoi` (end of input) only succeeds if the graph is already completely consumed. In this case, as a result we simply return `()`, the only value of the so-called unit type. Note that in Curry it does not need to be stated explicitly that `eoi` fails on non-empty input – the absence of a rule is enough.

```
pSucceed::res -> Grappa res            eoi::Grappa ()
pSucceed v g = (v, g)                  eoi [] = ((), [])
```

An especially important primitive parser is `edge`. It only succeeds if the given edge `e` is part of the particular graph `g`. It is implemented in a logic programming style making use of an equational constraint indicated by `=:=`.

```
edge::Edge -> Grappa ()
edge e g | g=:=(g1++e:g2) = ((), g1++g2)
          where g1, g2 free
```

A constraint e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable terms. Here, this means that the edge `e` indeed is contained in the graph `g`. In this case, the edge has to be consumed. This is realized by returning just `g1++g2` as the remaining graph.[5] Note that, in contrast to Prolog, free variables like `g1` and `g2` need to be declared explicitly (to make their scopes clear).

In Fig. 3 we provide some important parser combinators. They are defined in a fairly standard way (cf., e.g., [2,9,12]). The choice operator `<|>` takes two parsers and succeeds if either the first or the second one succeeds. In fact, it is a special case of the standard Curry operator `(?)::a->a->a`. Two parsers can also be combined via `<*>`, the successive application where the result is constructed by function application (as in [12]).[6] The second parser thereby starts with the input the first parser has left. For convenience we also define `*>` and `<*` that

[5] `(++)` is the standard operator for list concatenation. In contrast, `(:)` is the list constructor that can be used to add a single element to the front of a list.

[6] In previous versions of the framework [3,4] we have composed parsers using monads to make use of context. This does not seem to be necessary with the functional-logic approach as we see later. In fact, type classes are not supported in Curry yet.

```
(<|>)::Grappa res -> Grappa res -> Grappa res
p1 <|> _  = p1
_  <|> p2 = p2

(<*>)::Grappa (res1->res2) -> Grappa res1 -> Grappa res2
(p1 <*> p2) g = case p1 g of
                (pv, g') -> case p2 g' of
                            (qv, g'') -> (pv qv, g'')

(<*)::Grappa res1 -> Grappa res2 -> Grappa res1
p <* q = (\x _ -> x) <$> p <*> q
(*>)::Grappa res1 -> Grappa res2 -> Grappa res2
p *> q = (\_ x -> x) <$> p <*> q

(<$>)::(res1->res2) -> Grappa res1 -> Grappa res2
f <$> p = pSucceed f <*> p
(<$)::res1 -> Grappa res2 -> Grappa res1
f <$ p = const f <$> p
```

Fig. 3. Standard parser combinators

throw away one of the results. Finally, the parser transformers `<$>` and `<$` can be used to either apply a function to the result of a parser or to just replace it by another value.

On top of these basic combinators we can define various other useful combinators. For instance, we provide the combinator `many` to deal with simple repetition (the graph equivalent to the Kleene star) as:

```
many::Grappa a -> Grappa [a]
many p = pSucceed []
many p = (:) <$> p <*> many p
```

This definition can be read as: "`many p` always succeeds returning nothing (`[]`). It may also succeed by applying `p`, and thereafter `many p` again. In this case their results are combined using the list constructor `(:)`." Note, however, that this definition causes a lot of backtracking. In the string setting a combinator for simple repetition normally returns $n + 1$ different results where n is the number of successive occurrences of p **at the beginning of the string**. If a graph contains n occurrences of p, altogether $\sum_{i=0}^{n} \binom{n}{i} i!$ results are possible, since any number of occurrences can be chosen in any order. It is possible to disregard "redundant" results by using encapsulated search, but this way we lose some nice properties of our parsers. The problem with `many` is not inherent to our graph parsing approach, though. In fact, `many` is only needed to parse HR languages, which contain either highly disconnected graphs, or graphs which have vertices with high degree – both properties are known to be indicators for high parsing complexity [10].

We provide another typical combinator that we need later. The combinator `chain1Betw p (n1,n2)` can be used to identify a non-empty chain of graphs that can be parsed with `p`. This chain has to be anchored between the nodes `n1` and `n2`. Later we also need a parser `exactChain1Betw` that forces this chain to be of a particular length. We omit its declaration, since it can be defined very similar to `chain1Betw`:[7]

```
chain1Betw::((Node,Node)->Grappa a) -> (Node,Node) -> Grappa [a]
chain1Betw p (n1,n2) = (:[]) <$> p (n1,n2)
chain1Betw p (n1,n2) = (:)   <$> p (n1,n) <*> chain1Betw p (n,n2)
                            where n free
```

`chain1Betw` can be conveniently defined, because we do not need to know the inner node `n` in advance. We simply define it as a free variable, which can be instantiated according to the Curry narrowing semantics. Representing graph nodes as free variables actually is a functional-logic design pattern [13] that we here exploit in a novel way.

4 Parsing of HRGs

In this section we provide a direct mapping from HRGs to parsers based on the previously introduced framework. We exemplify the translation by means of the grammar given in Fig. 2a. We further provide some additional examples to demonstrate interesting properties of our parsers.

In Fig. 4 the parser for flowgraphs is presented. The type annotations are just for convenience and can also be omitted. For each non-terminal edge label l we have defined a parser function that takes a tuple of nodes $(n_1, ..., n_k)$ as a parameter such that $k = type(l)$. For each production over l we insert a new function body. Each terminal edge in the right-hand side of the production is matched and consumed using the primitive parser `edge`, each non-terminal one is translated to a call of the function representing this non-terminal. A free variable is introduced for each inner node of a production.

In contrast to string parsing the order of parsers in a successive composition via `*>` is not that important as long as left recursion is avoided. Nevertheless, the chosen arrangement might have an impact on the performance. Usually, it is advisable to deal with the terminal edges first.

Parsers defined in such a way are quite robust. For instance, they ignore redundant components, i.e., those just remain at the end. However, complete input consumption can be enforced easily by a subsequent application of `eoi`. Thus, instead of `fc` we can use the extended parser `fc <* eoi`.

[7] Actually, `chain1Betw = exactChain1Betw k where k free`, i.e., we can also define `chain1Betw` in terms of `exactChain1Betw`.

```
fc::Grappa ()
fc = edge ("start", [sn]) *> stmts (sn,en) *> edge ("end", [en])
     where sn, en free

stmts::(Node,Node) -> Grappa ()
stmts (n1,n2) = stmt (n1,n2)
stmts (n1,n2) = stmt (n1,n) *>
                stmts (n,n2)
                where n free

stmt::(Node,Node) -> Grappa ()
stmt (n1,n2) = edge ("text", [n1,n2])
stmt (n1,n2) = edge ("cond", [n1,nno,nyes]) *>
               stmts (nno,n2) *> stmts (nyes,n2)
               where nno, nyes free
stmt (n1,n2) = edge ("cond", [n1,n2,nbody]) *>
               stmts (nbody,n1)
               where nbody free
```

Fig. 4. A parser for flowgraphs

One problem is still left with our flowgraph parser: As it is, it accepts too many graphs. Further conditions have to be enforced to ensure correctness [6]:

- identification condition: matches have to be injective, i.e., involved nodes have to be pairwise distinct.
- dangling edge condition: there must not be other edges in the remaining graph visiting inner nodes of a match.

For instance, the flowgraphs shown in Fig. 5 can also be parsed successfully with the parser given in Fig. 4 although they are no members of the language defined by G_{FC}. In the context of visual languages it often is convenient to relax the dangling edge condition (cf. [5]). This allows for easier specifications. However, from a theoretical point of view, this is not satisfactory. In fact, both conditions can be ensured by additional checks. For instance, we can use inequality constraints on node variables to ensure that they are pairwise distinct, i.e., that a particular match is injective. However, these constraints cannot be globally set, but rather have to be added to the parsers for every single production making them less readable.

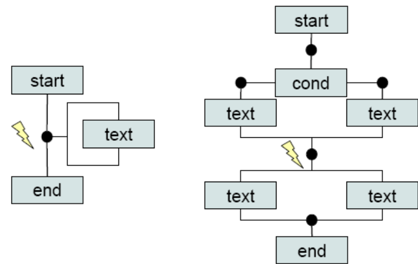


Fig. 5. Violation of identification condition and dangling edge condition

Semantics

So far we only have checked if the given graph is, or at least contains, a valid flowgraph. However, a major benefit of the combinator approach is that

language-specific results can be computed in a flexible way [2,14]. Say, we want 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
```

We do not provide the complete mapping here. Rather we use the translation of the branching production as an example to show how easily parsers can be enriched with attribution.

```
stmt::(Node,Node) -> Grappa Stmt
stmt (n1,n2) = edge ("cond", [n1,nno,nyes]) *>
               IfElse <$> stmts (nno,n2) <*> stmts (nyes,n2)
               where nno, nyes free
```

The result type has to be changed to `Stmt`. Further, we can use the combinator `<$>` to directly construct a statement from the two subprograms. Note that in Curry (as in Haskell) the data constructor `IfElse` is implicitly typed `Program->Program->Stmt`. In this situation we can make the parser definition even more concise, because `stmts` now really is just `chain1Betw stmt`.

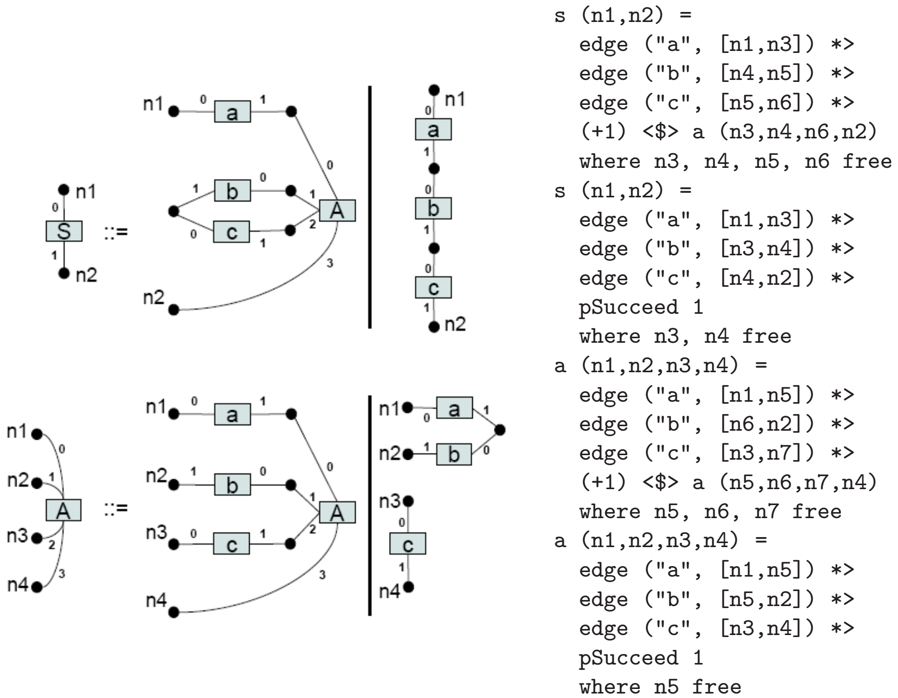
Not just a Parser

Parsing is not the only thing we can do with these functions. We can also apply them backwards to construct graphs of the language. For instance, we can enumerate all graphs in the language up to a particular size. As a result we know that there are only 2 flowgraphs (up to isomorphism) of size 4, 6 of size 5 and 21 of size 6.

We can further use the parser to perform a kind of auto-completion. Say, the edge `text(1,2)` in the graph given in Fig. 1b is missing, such that the flowgraph is not a member of the language anymore. We can try inserting an edge `e` as a free variable and see how `e` is instantiated by the parser. For our example we get several possible completions. For instance, we could add an edge `start(2)`. However, there is only one completion that consumes the whole input: `text(1,2)`, the one we deleted.

This approach could be the starting point for the realization of advanced error correction for graphs. For the error correction of strings a sophisticated and powerful Haskell parser combinator framework has already been proposed [12]. However, a functional-logic approach may be more understandable and easier to adapt to graphs. Such graph completion could be very useful in order to realize powerful content assist for graph grammar based diagram editors like the ones generated with DiaGen [5].

In certain cases we can also perform the mapping of semantics back to a graph, e.g., given a particular program we can construct a corresponding flowgraph. Thereby, nodes are not instantiated, but left as free variables. Indeed the particular node numbers do not matter as long as equal nodes can be identified.



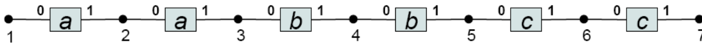
```
s (n1,n2) =
  edge ("a", [n1,n3]) *>
  edge ("b", [n4,n5]) *>
  edge ("c", [n5,n6]) *>
  (+1) <$> a (n3,n4,n6,n2)
  where n3, n4, n5, n6 free
s (n1,n2) =
  edge ("a", [n1,n3]) *>
  edge ("b", [n3,n4]) *>
  edge ("c", [n4,n2]) *>
  pSucceed 1
  where n3, n4 free
a (n1,n2,n3,n4) =
  edge ("a", [n1,n5]) *>
  edge ("b", [n6,n2]) *>
  edge ("c", [n3,n7]) *>
  (+1) <$> a (n5,n6,n7,n4)
  where n5, n6, n7 free
a (n1,n2,n3,n4) =
  edge ("a", [n1,n5]) *>
  edge ("b", [n5,n2]) *>
  edge ("c", [n3,n4]) *>
  pSucceed 1
  where n5 free
```

Fig. 6. Graph grammar a) and corresponding parser b) for the graph language $a^k b^k c^k$

Another Example: $a^k b^k c^k$

We give another example to demonstrate that the readability of a language description can also be improved by using graph parser combinators.

In a string setting the language $\{a^k b^k c^k | k > 0\}$ is not context-free. In contrast, there is a context-free string generating hypergraph grammar that defines a corresponding graph language [6,15]. A hypergraph grammar is string generating, if all graphs in its language have a linear structure, i.e., are a chain of directed edges; below we provide the graph representation of the string “aabbcc” as an example.



The grammar for the graph language $a^k b^k c^k$ as introduced in [6] is given in Fig. 6a. It is quite complex and hard to grasp despite the structural simplicity of the language. In Fig. 6b a nearly straightforward translation of this grammar to a parser is given. We only have added some attribution to compute the particular value of k.

With our framework much more readable descriptions are possible. One of them is given below. Basically it states that there have to be two nodes `n3` and `n4` and a number $k > 0$ such that there is a chain of k “a”-edges between `n1` and

n3, a chain of k "b"-edges between n3 and n4 and finally a chain of k "c"-edges between n4 and n2.[8]

```
abc (n1,n2) = exactChain1Betw k (dirEdge "a") (n1,n3) *>
              exactChain1Betw k (dirEdge "b") (n3,n4) *>
              exactChain1Betw k (dirEdge "c") (n4,n2) *>
              pSucceed k
              where k, n3, n4 free
```

The crucial point of this solution is that logic variables like k can be used to share results across parsers. This way context information can be exploited. This approach enables us to not only describe languages more conveniently, but also to describe languages that cannot be defined with a HRG at all. For instance, there is no HRG for Sierpinski triangles that are regular, i.e. equally deep unfolded.[9] In contrast, this can easily be done in our system just by introducing an additional parameter `depth`.

Performance

The operational semantics of Curry is based on an optimal evaluation strategy. As an extension to lazy functional programming its behavior is demand-driven. Thus, it ensures optimal evaluation on well-defined classes of programs [11]. However, in [6] it is proved that there are (even context-free) graph languages where parsing is NP-complete. These languages, of course, cannot be parsed with our combinators efficiently. Most practically relevant languages, however, are quite efficient to parse.

To give an impression we provide some performance data for the language $a^k b^k c^k$ in Fig. 7. The measurement has been executed on standard hardware using the Münster Curry Compiler[10]. We see that the more readable parser `abc` is even more efficient. Both parsers have a polynomial runtime behavior (mainly because the first node of the string graph is not known in advance). Since our parsers follow a top-down approach with backtracking we cannot completely avoid that partial results are computed more than once. Bottom-up parsers, which exploit dynamic programming techniques are usually more efficient.

Note, however, that the presented framework has not been optimized with respect to performance. For instance, a more efficient graph representations could be used, e.g., as a mapping `String->[[Node]]` so that all edges with a particular label can be queried much faster.

A good thing with parser combinators is that performance optimizations specific to the particular graph language can be incorporated easily. For instance, a basic improvement for flowgraphs would be to first decompose the given graph

[8] We make use of the primitive `dirEdge lab (n1,n2) = edge (lab, [n1,n2])` to make the combinator `exactChain1Betw` directly applicable.

[9] However, this can be realized with a special kind of parallel replacement mechanism as described in [16].

[10] http://danae.uni-muenster.de/~lux/curry/

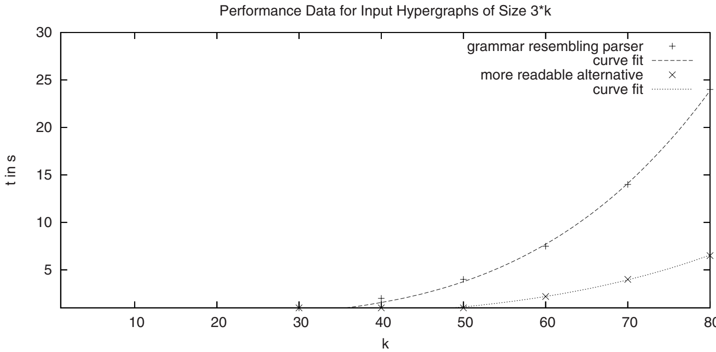


Fig. 7. Performance comparison of both parsers for the language $a^k b^k c^k$

into connected components and apply the parser to each of them successively. We provide the combinator `connComp::Grappa a->Grappa [a]` for this task. So, for a broad range of languages, we can start with a parser, which is easy to build and read, and which can be further improved if necessary. Providing additional information also can boost performance, e.g., if we know the first node in our language of string graphs, both parsers need less than a second for $k = 80$.

And even backwardly applied as generators our parsers are reasonably efficient. Compared to other graph transformation tools [17] they seem to be in the center-field. For instance, we have generated a Sierpinski triangle of generation 11 with nearly 200.000 edges in about a minute.

5 Related Work

In principle our graph parser combinators are quite similar to conventional string parser combinators like [2,14,9] to name just a few. Many ideas can be carried over straightforwardly. The main differences emerge from the non-linear structure of graphs and the appearance of nodes as connection points between tokens.

From all parser combinator approaches the UU library [12] is special in the sense that it probably provides the most powerful mechanisms to correct all kinds of errors in strings. There, a parser does never fail, but rather constructs a minimal sequence of correction steps. We have shown how our library can be used for restricted kinds of error handling. Redundant edges, for instance, may just remain at the end. This is already quite powerful, since in contrast to strings graphs are sets of components, i.e., there is no particular order imposed. Thus, it does not matter where the redundant components are placed. Furthermore, due to its logic nature, we can conveniently deal with errors that are fixable by embedding additional edges. However, other correction actions like edge relabeling or the gluing of distinct nodes cannot be computed in such a convenient manner.

Another interesting related observation is that parsing of visual languages can be modeled (and even executed) in linear logic [18], a resource-oriented refinement of classical logic. For instance, in [19] the embedding of constraint multiset

grammars into linear logic is discussed. However, it seems that hypergraph parsing can be modeled even more straightforwardly. Here, the edges of a hypergraph can be mapped to facts that can be fed into 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. For instance, to parse a simple flowgraph using Lolli [20] we can write `(start 1, text 1 2, end 2) -o fc`. The combinators presented in this paper also hide the remaining resources from the user. But their major benefit is the flexibility they can be applied with.

Also related are approaches to parsing of particular, restricted kinds of graph grammar formalisms. For instance, in [15] an Earley parser for string generating graph languages has been proposed. The diagram editor generator DiaGen [5] incorporates an HRG parser that is an adaptation of the algorithm of Cocke, Younger and Kasami. And the Visual Language Compiler-Compiler VLCC [21] 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.

6 Conclusion

In this paper we have discussed functional-logic graph parser combinators, an extensible framework supporting the flexible construction of special-purpose graph parsers even for (some) context-sensitive graph languages. It has turned out that functional-logic languages are exceptionally well-suited for graph parsing.

In particular we have demonstrated that hyperedge replacement grammars can be mapped to parsers straightforwardly. We have also noted that these grammars are sometimes not the most readable means to describe graph languages; several graph languages cannot even be defined with a HRG at all. In contrast, using our framework we can easily define readable parsers for languages like the string graphs $a^k b^k c^k$ or regular Sierpinski triangles. The resulting parsers are sufficiently efficient for many practical graph languages.

Functional-logic parsers can also be applied backwards. This way they can be used to enumerate a graph language or for graph-completion. Since graphs are well-suited as a model for visual languages, such graph-completion can be very beneficial in the context of diagram editors. We plan to connect our framework with the diagram editor generator DiaGen [5] to provide powerful content assist to the user.

References

1. Hutton, G.: Higher-order functions for parsing. Journal of Functional Programming 2(3), 323–343 (1992)
2. Fokker, J.: Functional parsers. In: Advanced Functional Programming, First Intl. Spring School on Advanced Functional Programming Techniques-Tutorial Text, London, UK, pp. 1–23. Springer, Heidelberg (1995)

3. Mazanek, S., Minas, M.: Graph parser combinators. In: Proc. of 19th Intl. Symp. on the Impl. and Appl. of Functional Languages. Springer, Heidelberg (2008)
4. Mazanek, S., Minas, M.: Parsing of hyperedge replacement grammars with graph parser combinators. In: Proc. of 7th Intl. Workshop on Graph Transf. and Visual Modeling Techniques. Electronic Communications of the EASST (to appear, 2008)
5. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44(2), 157–180 (2002)
6. 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)
7. Tanaka, T.: Definite-clause set grammars: a formalism for problem solving. J. Log. Program. 10(1), 1–17 (1991)
8. Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
9. Caballero, R., López-Fraguas, F.J.: A functional-logic perspective of parsing. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 85–99. Springer, Heidelberg (1999)
10. Lautemann, C.: The complexity of graph languages generated by hyperedge replacement. Acta Inf. 27(5), 399–421 (1989)
11. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. J. ACM 47(4), 776–822 (2000)
12. 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)
13. Antoy, S., Hanus, M.: Functional logic design patterns. In: Proc. of the 6th Intl. Symposium on Functional and Logic Programming. LNCS, vol. 2441, pp. 67–87. Springer, Heidelberg (2002)
14. Hutton, G., Meijer, E.: Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996)
15. Seifert, S., Fischer, I.: Parsing string generating hypergraph grammars. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 352–367. Springer, Heidelberg (2004)
16. Habel, A., Kreowski, H.J.: Pretty patterns produced by hyperedge replacement. In: Göttler, H., Schneider, H.-J. (eds.) WG 1987. LNCS, vol. 314, pp. 32–45. Springer, Heidelberg (1988)
17. Taentzer, G., et al.: Generation of sierpinski triangles: A case study for graph transformation tools. In: Proc. of AGTIVE 2007. LNCS. Springer, Heidelberg (2008)
18. Girard, J.Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
19. Bottoni, P., Meyer, B., Marriott, K., Parisi-Presicce, F.: Deductive parsing of visual languages. In: Proc. of the 4th Intl. Conference on Logical Aspects of Computational Linguistics, London, UK, pp. 79–94. Springer, Heidelberg (2001)
20. Hodas, J.S., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Inf. Comput. 110(2), 327–365 (1994)
21. Costagliola, G., Lucia, A.D., Orefice, S., Tortora, G.: A parsing methodology for the implementation of visual systems. IEEE Trans. Softw. Eng. 23(12), 777–799 (1997)