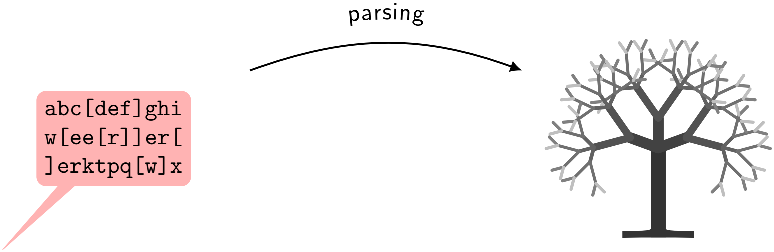
A functorial presentation of parsers

G. Feierabend, Stellenbosch University

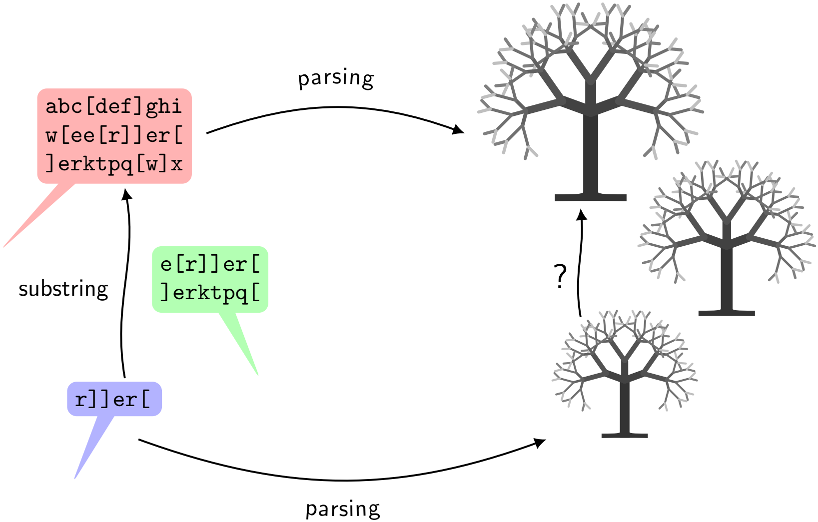
Introduction

Introduction

Translating unformatted input into some meaningful data structure:



Introduction



The SOFiA Proof Assistant Project

A SOFiA proof:

```
┌[X][Y][[X]=[Y]] /L1: assumption.
│[[X]=[X]] /L2: self-equate from L1(1).
└[[Y]=[X]] /L3: right substitution, L1(3) in L2(1).
[[X][Y][[X]=[Y]]:[[Y]=[X]]] /L4: synapsis (L1-3).
```

Non-functorial parser for SOFiA

```haskell
option p = option1 p <|> return []
option1 p =
    do vs1 <- p
       vs2 <- option p
       return (vs1 ++ vs2)

legalSymbolChars = ['a'..'z'] ++ ['A'..'Z'] ++ ['0'..'9'] ++ ['%',' ','+', '!',
                    '\'']

isValidSymbol :: String -> Bool
isValidSymbol cs = [c | c <- cs, not $ elem c legalSymbolChars] == []

sCharacter :: Parser Char
sCharacter = sat (\x -> elem x legalSymbolChars)

specialChar :: Char -> Parser Char
specialChar x = sat (== x)

sSymbol :: Parser String
sSymbol =
    do x <- sCharacter
       xs <- many sCharacter
       return [y | y <- (x:xs), y /= ' ']

sFormulator :: Parser SofiaTree
sFormulator =
    do many $ specialChar ' '
       specialChar ':'
       many $ specialChar ' '
       return (newSofiaTree [] Implication [])
      <|> do many $ specialChar ' '
             specialChar '='
             many $ specialChar ' '
             return (newSofiaTree [] Equality [])
            <|> do x <- sSymbol
                   return (newSofiaTree x Symbol [])

sAtom :: Parser SofiaTree
sAtom =
    do many (specialChar ' ')
       specialChar '['
       many (specialChar ' ')
       x <- sFormula
       many (specialChar ' ')
       specialChar ']'
       many (specialChar ' ')
       return (newSofiaTree "" Atom [x])
      <|> do many (specialChar ' ')
             specialChar '['
             many (specialChar ' ')
             x <- sStatement
             many (specialChar ' ')
```

(a) lines 1–52

```haskell
             specialChar ']'
             many (specialChar ' ')
             return (newSofiaTree "" Atom [x])
      <|> do many (specialChar ' ')
             specialChar '['
             many (specialChar ' ')
             specialChar ']'
             many (specialChar ' ')
             return (newSofiaTree "" Atom [])

sStatement :: Parser SofiaTree
sStatement = do x <- sAtom
                xs <- many sAtom
                return (newSofiaTree "" Statement (x:xs))

sFormula :: Parser SofiaTree
sFormula =
    do x <- sFormulator;
       do y <- sStatement;
          zs <- option
                    (do z1 <- sFormulator
                        z2 <- sStatement
                        return [z1, z2]
                    )
          do f <- sFormulator
             return (newSofiaTree "" Formula ([x, y] ++ zs ++ [f]))
            <|> return (newSofiaTree "" Formula ([x, y] ++ zs))
         <|> return (newSofiaTree "" Formula [x])
      <|> do x <- sStatement
             y <- sFormulator
             zs <- option
                    (do z1 <- sStatement
                        z2 <- sFormulator
                        return [z1, z2]
                    )
             do f <- sStatement
                return (newSofiaTree "" Formula ([x, y] ++ zs ++ [f]))
               <|> return (newSofiaTree "" Formula ([x, y] ++ zs))

sExpression :: Parser SofiaTree
sExpression =
    do x <- sFormula
       return x
      <|> do x <- sStatement
             return x

treeParse :: String -> SofiaTree
treeParse x = case parsed of
                []          -> newSofiaTree "" Error []
                [(_, x:xs)] -> newSofiaTree "" Error []
                _           ->  fst $ head $ parsed
                where parsed = parse sExpression x
```

(b) lines 53–104

Figure: recursive descend parser for SOFiA

Labelled Charged Graphs

Finite sequences

Definition

A **finite sequence** of length n over a set Σ is **a function from** $\{1, \ldots, n\}$ **to** Σ. We denote the **length** of a finite sequence s by $||s||$. The unique sequence of length 0 is denoted by ε. Given any set Σ, we **denote the set of all finite sequences over** Σ **by** Σ^*.

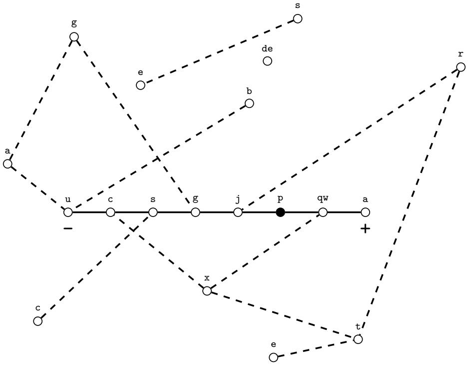
Definition

Given finite sequences $s_1, s_2 \in \Sigma^*$ over some set (*alphabet*) Σ, we define $s_1 \oplus s_2$ to be the **concatenation** of s_1 and s_2.

Definition

A *labelled charged graph (LCG)* is a tuple $(V, E, \Sigma, \lambda, c, \rho)$:

- (V, E): graph
- Σ: *alphabet*
- $\lambda : V \to \Sigma^*$: *labelling function*
- c: *charge*
- ρ: *root*



LCG-isomorphisms

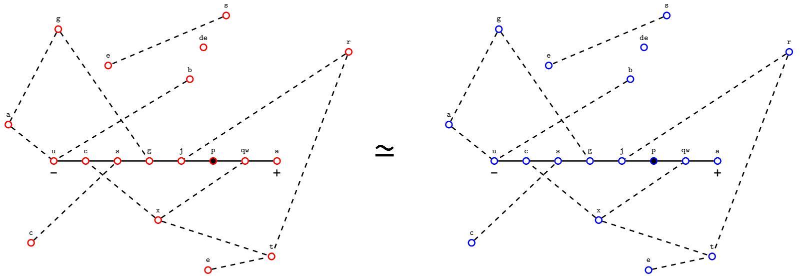


Figure: two LCGs with different underlying vertex-sets but the same structure

LCG-isomorphisms

Definition

$G_1 = (V_1, E_1, \Sigma, \lambda_1, c_1, \rho_1)$ and $G_2 = (V_2, E_2, \Sigma, \lambda_2, c_2, \rho_2)$ are *isomorphic* to each other, if there exists a bijection $\varphi : V_1 \to V_2$ such that:

(I1) φ *preserves edges*: $\forall_{v_a, v_b \in V_1} [(v_a, v_b) \in E_1 \Leftrightarrow (\varphi(v_a), \varphi(v_b)) \in E_2]$.

(I2) φ *preserves the length of charges*: $||c_1|| = ||c_2||$.

(I3) φ *preserves charges*: For all $i \in \{1, 2, \dots, ||c_1||\}$, $\varphi(c_1(i)) = c_2(i)$.

(I4) φ *preserves labels*: For all $v \in V_1$, $\lambda_1(v) = \lambda_2(\varphi(v))$.

(I5) φ *preserves roots*: $\varphi(\rho_1) = \rho_2$.

We then write $G_1 \simeq G_2$.

Concatenation of LCGs

The coincidence relation ~ is an equivalence relation on the disjoint union of the underlying vertex sets of two LCGs.

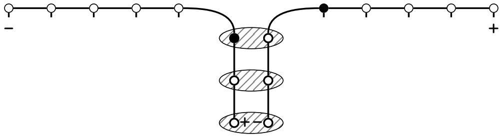


Figure: coinciding vertices

Concatenation of LCGs

Given two LCGs G_1 and G_2, the vertex set of the concatenation $G_1 \uplus G_2$ is the quotient set $(V_1 \sqcup V_2)/\sim$.

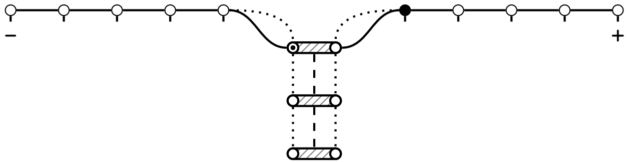


Figure: concatenating two LCGs

Monoids over LCGs

Monoid "up to isomorphism"

Definition

Let M be a set and let $\cong$ be an equivalence relation on M and let $\otimes$ be a binary operation on M. We say that $(M, \otimes, \cong)$ is a *monoid up to isomorphism*, if all of the following conditions hold:

(1) For all $a, b, c \in M$, $(a \otimes b) \otimes c \cong a \otimes (b \otimes c)$.

(2) There exists $e \in M$ such that for all $a \in M$, $e \otimes a \cong a \cong a \otimes e$.

Theorem

LCGs over a fixed alphabet Σ together with concatenation and LCG-isomorphisms form a monoid up to isomorphism.

Proof idea

Let $G_1 = (V_1, E_1, \Sigma, \lambda_1, c_1, \rho_1)$, $G_2 = (V_2, E_2, \Sigma, \lambda_2, c_2, \rho_2)$ and $G_3 = (V_3, E_3, \Sigma, \lambda_3, c_3, \rho_3)$ be LCGs over some fixed alphabet Σ and let

- $(G_1 \uplus G_2) \uplus G_3 = (V_L, E_L, \Sigma, \lambda_L, c_L, \rho_L)$
- $G_1 \uplus (G_2 \uplus G_3) = (V_R, E_R, \Sigma, \lambda_R, c_R, \rho_R)$

Let $\varphi : V_L \to V_R$ be defined as follows:

$$\varphi([(a, v)])$$

$$= \begin{cases} [(1, v_1)] & \text{if } \exists_{v_1 \in V_1} [(a, v)] = [(1, [(1, v_1)])] \\ [(2, [(1, v_2)])] & \text{if } \exists_{v_2 \in V_2} [[(a, v)] = [(1, [(2, v_2)])] \text{ and } |[(2, v_2)]| = 1] \\ [(2, [(2, v_3)])] & \text{if } \exists_{v_3 \in V_3} [|[(a, v)]| = 1 \text{ and } [(a, v)] = [(2, v_3)]] \end{cases}$$

Then:

- Show that φ is well-defined and an LCG-isomorphism.
- Show that any single node LCG with a label of length 0 is an identity.

Extending the definitions

We can extend LCG-concatenation to equivalence classes of LCGs:

Lemma

Let G_1, $G_1^{\prime}$, G_2 be LCGs such that there exists an isomorphism φ between G_1 and $G_1^{\prime}$ and let 1_G be the identity isomorphism on G_2. Then we can define isomorphisms $1_G \uplus \varphi$ and $\varphi \uplus 1_G$ between $G_2 \uplus G_1$ and $G_2 \uplus G_1^{\prime}$, and $G_1 \uplus G_2$ and $G_1^{\prime} \uplus G_2$ respectively.

Theorem

Let $[G], [H]$ be equivalence classes of LCGs over some fixed alphabet Σ and suppose $G_1, G_2 \in [G]$ and $H_1, H_2 \in [H]$. Then $[G_1 \uplus H_1] = [G_2 \uplus H_2]$.

Corollary

Equivalence classes of isomorphic LCGs together with their concatenation form a monoid.

Words

We represent words as equivalence classes of single-vertex LCGs.

Theorem

Words together with their concatenation form a sub-monoid of the monoid of equivalence classes of LCGs.

Definition

A word $[W]$ is *embedded* into another word $[W']$ if there exist two words $[L]$ and $[R]$ such that $[L] \uplus [W] \uplus [R] = [W']$.

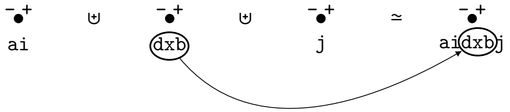


Figure: a word embedded into another word

The category of embedded LCGs and the parser

Embeddings of LCGs

We can extend the notion of embeddings to arbitrary LCGs:

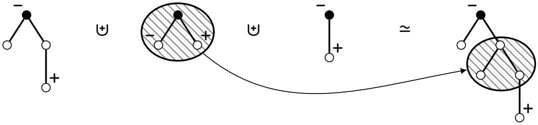


Figure: a general LCG embedded into another LCG

Definition

An LCG G is *embedded* into another LCG G' if there exist two LCGs L and R such that $[L] \uplus [G] \uplus [R] = [G']$.

Theorem

Equivalence classes of isomorphic LCGs together with embeddings form a category denoted by $\mathbf{eLCG(\Sigma)}$.

Theorem

Words give rise to a subcategory of $\mathbf{eLCG(\Sigma)}$, *denoted by* $\mathbf{wLCG(\Sigma)}$.

The free monoid

Theorem (universal property of the *free monoid*)

in **Mon**: $M(A) \overset{\hat{f}}{\dashrightarrow} N$

in **Set**:

$$\begin{array}{ccc} |M(A)| & \xrightarrow{|\hat{f}|} & |N| \\ \uparrow i & \nearrow f & \\ A & & \end{array}$$

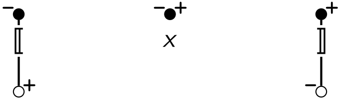
Theorem

$(\Sigma^*, \oplus, \varepsilon)$ *is the* **free monoid** *over* Σ.

Theorem

The monoid $(wLCG(\Sigma)_0, \uplus)$ *is isomorphic to* $(\Sigma^*, \oplus)$. *We denote this isomorphism by* ϕ.

We can now define a function f from an alphabet Σ with two distinct special elements $[\![$ and $]\!]$ to respectively map $[\![$, $x \in \Sigma \setminus \{[\![,]\!]\}$ and $]\!]$ to the equivalences classes of the following LCGs:



Theorem

Let Σ be an alphabet with two distinct special elements denoted $[\![$ and $]\!]$ and let $f : \Sigma \to eLCG(\Sigma)_0$ be the function defined as follows:

$$f(x) = \begin{cases} [\bullet\!\!-\!\!\circ] & \text{if } x = [\![\\ [\circ\!\!-\!\!\bullet] & \text{if } x =]\!] \\ \ulcorner x \urcorner & \text{otherwise} \end{cases}$$

Moreover, let $\hat{f}$ be the unique monoid homomorphism from $(\Sigma^*, \oplus, \varepsilon) \to eLCG(\Sigma)_0$ such that $f = \hat{f} \circ i$ and let $p = \hat{f} \circ \phi$.

We can define a functor $P : wLCG(\Sigma) \to eLCG(\Sigma)$ as follows:

$$P_0([W]) = p([W])$$
$$P_1([W_1], [L_1], [R_1], [W_2]) = (p([W_1]), p([L_1]), p([R_1]), p([W_2]))$$

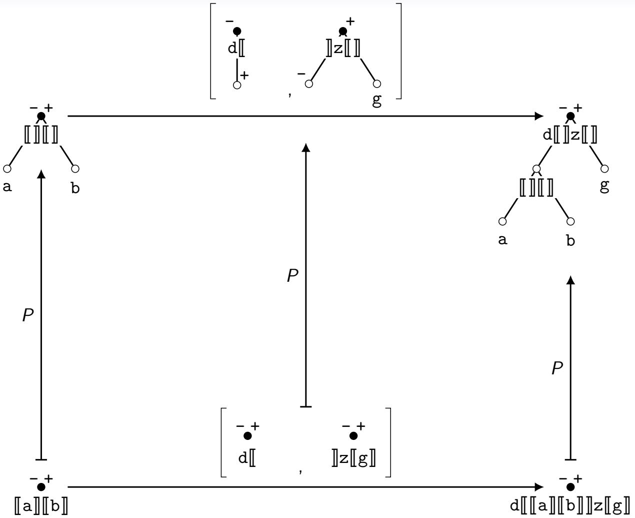


Figure: the parser functor “in action”

Functorial parser for SOFiA

```
data Tree = Vertex [Char] [Tree] deriving Show
data LCT = LCT Int Int Tree deriving Show

(\+/) :: LCT -> LCT -> LCT
(\+/) (LCT ca1 ca2 ta) (LCT cb1 cb2 tb) = LCT c1 c2 (merge c d ta tb)
  where  c  = ca2 - cb1
         d  = min ca2 cb1
         c1 = (max 0 (cb1 - ca2)) + ca1
         c2 = (max 0 (ca2 - cb1)) + cb2

merge :: Int -> Int -> Tree -> Tree -> Tree
merge c d (Vertex l1 ts1) (Vertex l2 ts2) =
  if c < 0 then Vertex l2 ((merge (c + 1) d (Vertex l1 ts1) (head ts2)):(tail ts2))
  else if c == 0 && d == 0 then Vertex (l1 ++ l2) (ts1 ++ ts2)
  else if c == 0 && d /= 0 then Vertex (l1 ++ l2)
    ((init ts1) ++ ((merge 0 (d - 1) (last ts1) (head ts2)):(tail ts2)))
  else Vertex l1 ((init ts1) ++ [merge (c - 1) d (last ts1) (Vertex l2 ts2)])

parse :: [Char] -> LCT
parse xs = foldl (\+/) (LCT 0 0 (Vertex [] [])) (map f xs)
  where f x = case x of
              '[' -> LCT 0 1 (Vertex "[" [Vertex "" []])
              ']' -> LCT 1 0 (Vertex "]" [Vertex "" []])
              _   -> LCT 0 0 (Vertex [x] [])
```

Thank you very much for listening!

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley series in computer science and information processing. Addison-Wesley Publishing Company, 1986. ISBN: 9780201100884.

[2] S. Awodey. *Category Theory*. Oxford Logic Guides. OUP Oxford, 2010. ISBN: 9780199587360.

[3] G. Chartrand. *Introductory Graph Theory*. Dover Books on Mathematics Series. Dover, 1977. ISBN: 9780486247755.

[4] G. Feierabend. *The SOFiA Proof Assistant*. 2022. URL: http://81.7.3.57:3000/.

[5] G. Hutton. *Programming in Haskell*. 2005. URL: http://www.cs.nott.ac.uk/~pszgmh/pih.html.

[6] Z. Janelidze. *The SOFiA Proof Assistant Project*. 2022. URL: https://www.zurab.online/2022/08/the-sofia-proof-assistant-project.html.

[7] B. Laing. "Sketching SOFiA". MA thesis. Stellenbosch University, 2020.

[8] S. MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1971. ISBN: 9781461298397.

[9] M. Sipser. *Introduction to the theory of computation*. 3rd ed. Florence, AL: Course Technology, Jan. 2021.