Modular Edit Lenses

Martin Hofmann
Joint work with Benjamin Pierce and Daniel Wagner (2009-2014)

25th-29th July 2016
Lady Margaret Hall, Oxford, UK

Introduction

- Three chapters:
  - symmetric lenses,
  - symmetric edit lenses (symmetric lenses plus edit language)
  - edit languages for XML-like trees
- Focus on combinators for the modular construction of lenses:
  - Symmetric monoidal structure (sequential and parallel composition)
  - Inductive data structures, "folds" & "hylomorphisms"
  - Container datatypes, generic edit operations for container types.
  - Pattern for lens construction based on list partitioning
- Loose ends and suggestions for projects.
- Literature:
  - Symmetric Lenses (POPL2011) `http://dmwit.com/papers/201009SL_full.pdf`, long version in JACM 2015.
  - Edit Lenses (POPL2012) `http://dmwit.com/papers/201107EL.pdf`
  - PhD thesis D. Wagner: `http://dmwit.com/papers/201407SEL_ANFfBL.pdf`
  - Edit Languages for Information Trees (BX2013) `http://dmwit.com/papers/201107ELfIT_full.pdf`

Bidirectional transformation

Synchronising data in different representations:

- File systems
- Web data
- Software models
- Data formats

Issues:

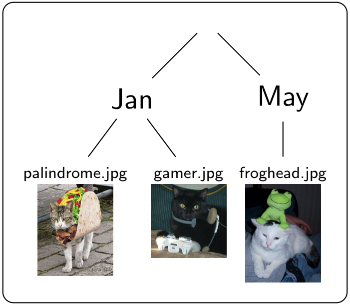
- Must specify the translation / correspondence between the representations
- Both representations will contain parts that the other one does not have
- Want to have combinators / DSL to write such translations reliably
- Reasoning about translations

Setup

Daniel shares cat pictures with his coworkers, but prefers a different organization scheme than they do.

- At home: tree-structured file-system
- On the web: flat-list picture gallery with tags

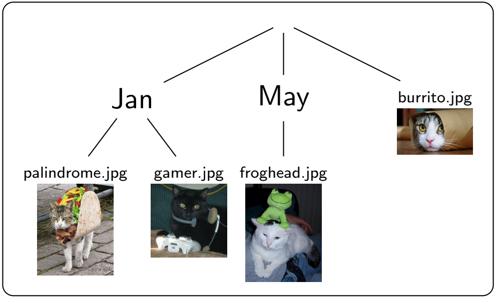
Two Structures



Goal, Part 1: Adding to the Web Gallery

	[costume,food]
	[costume]
	[costume]
	[onlyface]

Goal, Part 2: Fixing the Filename



Jan

May

burrito.jpg

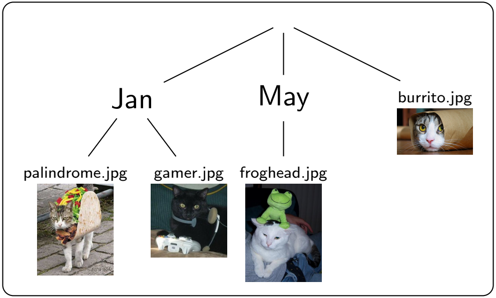
palindrome.jpg

gamer.jpg

froghead.jpg

	[costume,food]
	[costume]
	[costume]
	[onlyface]

Goal, Part 3: Changing Tags



Jan

May

burrito.jpg

palindrome.jpg

gamer.jpg

froghead.jpg

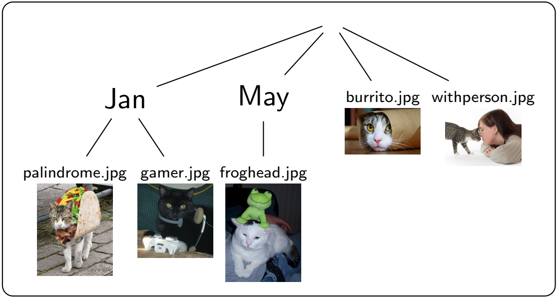
[costume,food]

[costume]

[costume]

[food,onlyface]

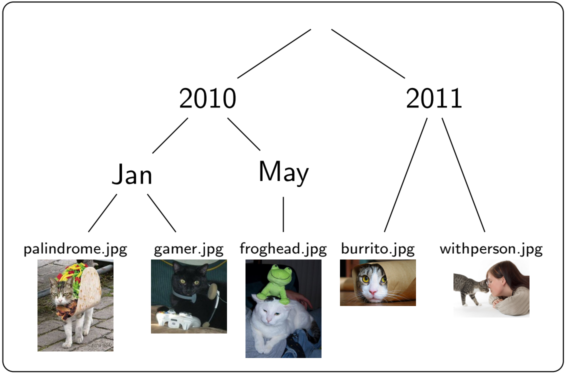
Goal, Part 4: Adding to the File System



- Jan
  - palindrome.jpg
  - gamer.jpg
- May
  - froghead.jpg
- burrito.jpg
- withperson.jpg

	[costume,food]
	[costume]
	[costume]
	[food,onlyface]
	[ ]

Goal, Part 5: Restructuring


2010
2011
Jan
May
palindrome.jpg
gamer.jpg
froghead.jpg
burrito.jpg
withperson.jpg


[costume,food]

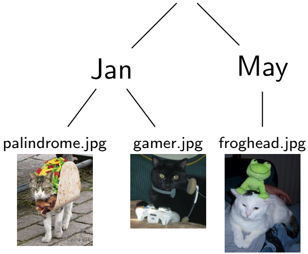

[costume]


[costume]


[food,onlyface]


[ ]

Typing "get"




[costume,food]


[costume]

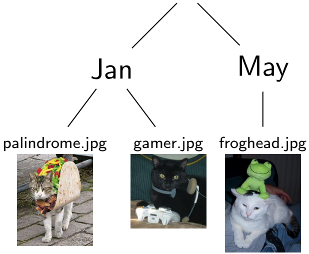

[costume]

data FS = Directory Name [FS]
| File Name Picture

type Web =
[(Picture, [Tag])]

$$\ell : \mathsf{FS} \overset{?}{\leftrightarrow} \mathsf{Web}$$

Typing "get"



$$\begin{array}{lcl} \text{data FS} & = & \text{Directory Name [FS]} \\ & | & \text{File Name Picture} \end{array}$$

$$\begin{array}{l} \text{type Web} = \\ \quad \text{[(Picture, [Tag])]} \end{array}$$

$$\ell : \text{Web} \overset{?}{\leftrightarrow} \text{FS}$$

Formalizing the Oddity: Roundtrip Laws

$$put(get(a), a) = a$$

$$get(put(b, a)) = b$$

Either possibility forbidden!

Towards symmetrization

- Symmetric Constraint Maintainers
  (Meertens; 1998)
- Towards an Algebraic Theory of Bidirectional Transformations
  (Stevens; ICGT 2008)
- Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions
  (Stevens; MoDELS 2007)
- Algebraic Models for Bidirectional Model Synchronization
  (Diskin; MoDELS 2008)
- Supporting Parallel Updates with Bidirectional Model Transformations
  (Xiong, Song, Hu, and Takeichi; ICMT 2009)

Yet no composition.

Symmetric lenses

A lens framework with

1. symmetry
2. composition

Symmetric lenses

A lens framework with

1. symmetry
2. composition
3. . . . and other nice combinators

Starting Point: Asymmetric Lenses

$$\ell : A \overset{a}{\leftrightarrow} B$$

$$\begin{array}{lcl} \text{get} & : & A \rightarrow B \\ \text{put} & : & B \times A \rightarrow A \end{array}$$

$$get(put(b, a)) = b$$

$$put(get(a), a) = a$$

L/R Symmetry

$$\ell : A \overset{a}{\leftrightarrow} B$$

$$\begin{aligned} \text{putr} &: A \times B \to B \\ \text{putl} &: B \times A \to A \end{aligned}$$

$$get(put(b, a)) = b$$

$$put(get(a), a) = a$$

Complements

$$\ell : A \overset{a}{\leftrightarrow} B$$

$$\begin{array}{lcl} \text{putr} & : & A \times S_B \to B \\ \text{putl} & : & B \times S_A \to A \end{array}$$

$$get(put(b, a)) = b$$

$$put(get(a), a) = a$$

I/O Symmetry

$$\ell : A \overset{a}{\leftrightarrow} B$$

$$\begin{array}{lcl} \text{putr} & : & A \times S_B \rightarrow B \times S_A \\ \text{putl} & : & B \times S_A \rightarrow A \times S_B \end{array}$$

$$get(put(b, a)) = b$$

$$put(get(a), a) = a$$

Unifying Complements

$$\ell : A \leftrightarrow B$$

$$\begin{array}{lcl} \text{putr} & : & A \times S \rightarrow B \times S \\ \text{putl} & : & B \times S \rightarrow A \times S \end{array}$$

$$get(put(b, a)) = b$$

$$put(get(a), a) = a$$

Need to initialize

Unlike in asymmetric case now need a special element

$$init \in S$$

to initiate a “synchronization dialogue”.

Updated Lens Laws

$$\ell : A \leftrightarrow B$$

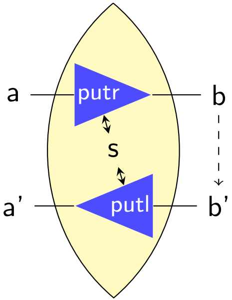
$$\begin{array}{lcl} \text{putr} & : & A \times S \to B \times S \\ \text{putl} & : & B \times S \to A \times S \end{array}$$

$$\frac{putr(a, s) = (b, s')}{putl(b, s') = (a, s')}$$

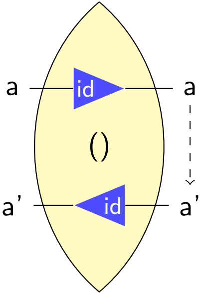
$$\frac{putl(b, s) = (a, s')}{putr(a, s') = (b, s')}$$

No law needed for *init* (yet!)

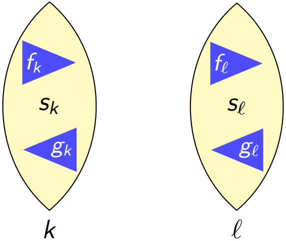
Updated Wiring Diagram

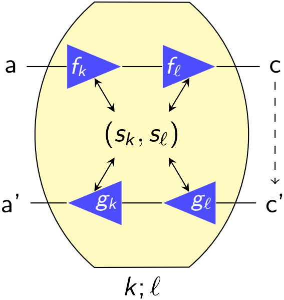


Warm-up: Identity Lens

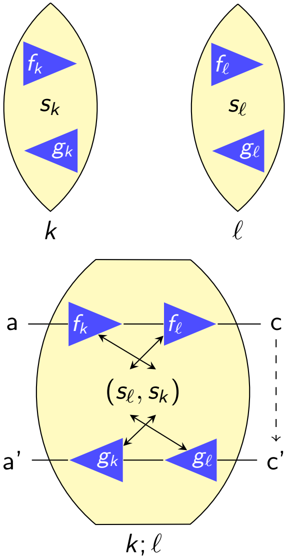


Composition





Another Composition



Lens Equivalence

$k \equiv \ell$ when there is a relation $R \subset k.S \times \ell.S$ and:

$$\frac{s_k \; R \; s_\ell \qquad k.putr(a, s_k) = (b_k, s'_k) \qquad \ell.putr(a, s_\ell) = (b_\ell, s'_\ell)}{b_k = b_\ell \wedge s'_k \; R \; s'_\ell}$$

$$\frac{s_k \; R \; s_\ell \qquad k.putl(b, s_k) = (a_k, s'_k) \qquad \ell.putl(b, s_\ell) = (a_\ell, s'_\ell)}{a_k = a_\ell \wedge s'_k \; R \; s'_\ell}$$

$$\ell.init \; R \; k.init$$

Symmetric Lenses as a Category

- $\equiv$ is an equivalence relation (prove that!)
- cf. bisimulation / coinduction.
- composition is associative up to $\equiv$
- We obtain a category whose objects are sets and morphisms are $\equiv$-equivalence classes of symmetric lenses.

Observational equivalence

Put object

Given a lens $\ell \in X \leftrightarrow Y$, define a put object for ℓ to be a member of $X + Y$. Define a function *apply* taking a lens, an element of that lens' complement set, and a list of put objects, by pushing the list's elements through the lens beginning with the given element.

Observational equivalence

Lenses $k, \ell \in X \leftrightarrow Y$ are observationally equivalent (written $k \approx \ell$) if, for every sequence of put objects $P \in (X + Y)^\star$ we have

$$apply(k, k.init, P) = apply(\ell, \ell.init, P).$$

Theorem

$k \approx \ell$ iff $k \equiv \ell$.

Bijective lenses

Every bijective function gives rise to a lens:

$$\frac{f \in X \to Y \qquad f \text{ bijective}}{iso_f \in X \leftrightarrow Y}$$

$$\begin{array}{lcl} C & = & Unit \\ init & = & () \\ putr(x, ()) & = & (f(x), ()) \\ putl(y, ()) & = & (f^{-1}(y), ()) \end{array}$$

Terminal lens

$$\frac{x \in X}{\mathit{term}_x \in X \leftrightarrow \mathit{Unit}}$$

$$\begin{array}{lcl}
C & = & X \\
\mathit{init} & = & x \\
\mathit{putr}(x', c) & = & ((), x') \\
\mathit{putl}((), c) & = & (c, c)
\end{array}$$

Opposite lens

$$\frac{\ell \in X \leftrightarrow Y}{\ell^{op} \in Y \leftrightarrow X}$$

$$\begin{array}{lcl} C & = & \ell.C \\ init & = & \ell.init \\ putr(y, c) & = & \ell.putl(y, c) \\ putl(x, c) & = & \ell.putr(x, c) \end{array}$$

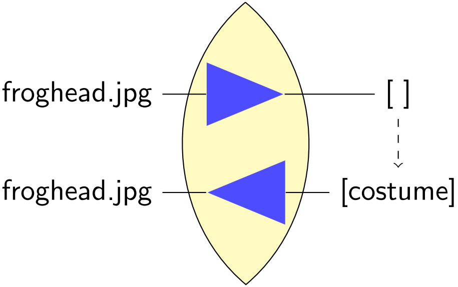
Disconnect lens

$$\frac{x \in X \qquad y \in Y}{disconnect_{xy} \in X \leftrightarrow Y}$$

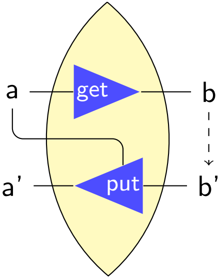
$$disconnect_{xy} \;=\; term_x ; term_y^{op}$$

The disconnect lens does not synchronize its two sides at all. The complement, $disconnect.C$, is $X \times Y$; inputs are squirreled away into one side of the complement, and outputs are retrieved from the other side of the complement.

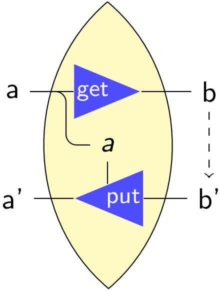
Why disconnect?



Lifting Asymmetric Lenses

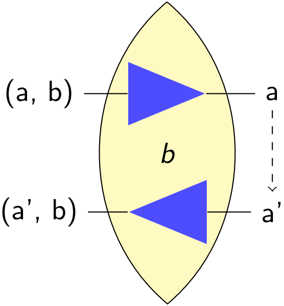


$\ell : A \overset{a}{\leftrightarrow} B$



$\ell^{sym} : A \leftrightarrow B$

Projection



$\pi_1 : A \times B \leftrightarrow A$

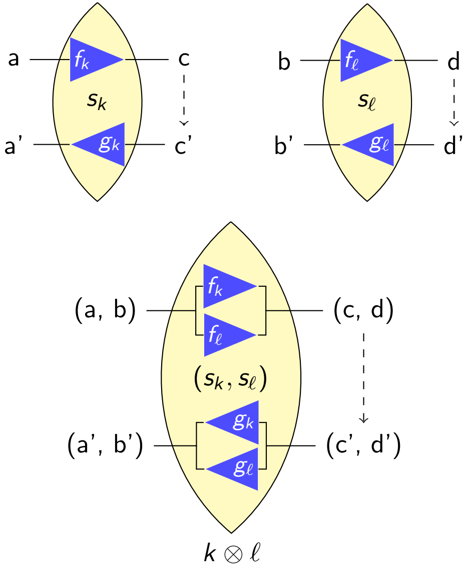
(π_2 is similar)

Tensor Product

$$\frac{k \in X \leftrightarrow Z \qquad \ell \in Y \leftrightarrow W}{k \otimes \ell \in X \times Y \leftrightarrow Z \times W}$$

$$\begin{array}{lcl}
C & = & k.C \times \ell.C \\
init & = & (k.init, \ell.init) \\
putr((x,y),(c_k,c_\ell)) & = & \text{let } (z, c_k') = k.putr(x, c_k) \text{ in} \\
 & & \text{let } (w, c_\ell') = \ell.putr(y, c_\ell) \text{ in} \\
 & & ((z,w),(c_k', c_\ell')) \\
putl((z,w),(c_k,c_\ell)) & = & \text{let } (x, c_k') = k.putl(z, c_k) \text{ in} \\
 & & \text{let } (y, c_\ell') = \ell.putl(w, c_\ell) \text{ in} \\
 & & ((x,y),(c_k', c_\ell'))
\end{array}$$

Tensor product, pictorially



Naturality

Projections are natural in the following sense.

$$\begin{tikzcd}
X_k \times X_\ell \arrow[r, "k \otimes \ell"] \arrow[d, "id_{X_k} \otimes term_{x_i}"'] & Y_k \times Y_\ell \arrow[d, "id_{Y_k} \otimes term_{y_i}"] \\
X_k \times Unit \arrow[r, "k \otimes id_{Unit}"] \arrow[d, "\rho_{X_k}"'] & Y_k \times Unit \arrow[d, "\rho_{Y_k}"] \\
X_k \arrow[r, "k"] & Y_k
\end{tikzcd}$$

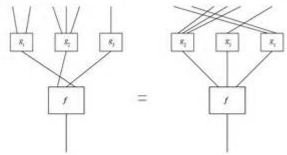
Symmetric monoidal category

Theorem

Symmetric lenses with their tensor product form a symmetric monoidal category.

This means, we can regard lenses as wirings

- composition corresponds to chaining
- tensor product corresponds to juxtaposition



Open question

Does the category have a trace ?

For this, we would need to construct from a lens $\ell : X \times Y \leftrightarrow X \times Z$ a trace $\mathrm{tr}(\ell) : Y \leftrightarrow Z$.

Graphically, this corresponds to joining the two X-ends of ℓ with a “feedback” wire.

This trace operation should validate all equations that hold “graphically” in this sense.

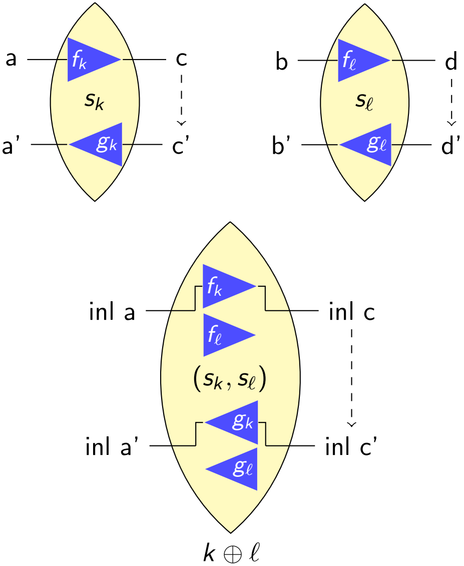
Sum lens

$$\frac{k \in X \leftrightarrow Z \qquad \ell \in Y \leftrightarrow W}{k \oplus \ell \in X + Y \leftrightarrow Z + W}$$

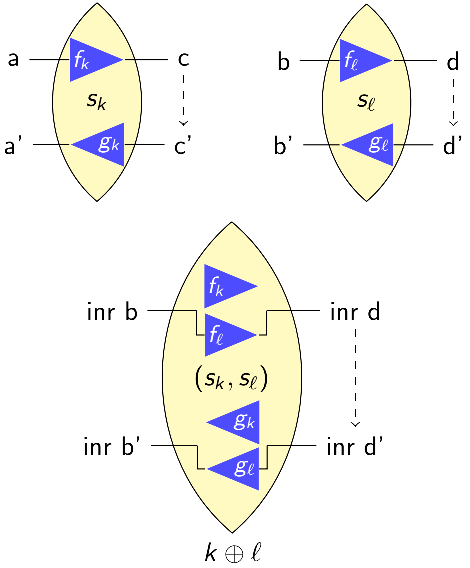
$$\begin{array}{lcl}
C & = & k.C \times \ell.C \\
\mathit{init} & = & (k.\mathit{init}, \ell.\mathit{init}) \\
\mathit{putr}(\mathrm{inl}(x), (c_k, c_\ell)) & = & \mathrm{let}\ (z, c_k') = k.\mathit{putr}(x, c_k)\ \mathrm{in} \\
 & & (\mathrm{inl}(z), (c_k', c_\ell)) \\
\mathit{putr}(\mathrm{inr}(y), (c_k, c_\ell)) & = & \mathrm{let}\ (w, c_\ell') = \ell.\mathit{putr}(y, c_\ell)\ \mathrm{in} \\
 & & (\mathrm{inr}(w), (c_k, c_\ell')) \\
\mathit{putl}(\mathrm{inl}(z), (c_k, c_\ell)) & = & \mathrm{let}\ (x, c_k') = k.\mathit{putl}(z, c_k)\ \mathrm{in} \\
 & & (\mathrm{inl}(x), (c_k', c_\ell)) \\
\mathit{putl}(\mathrm{inr}(w), (c_k, c_\ell)) & = & \mathrm{let}\ (y, c_\ell') = \ell.\mathit{putl}(y, c_\ell)\ \mathrm{in} \\
 & & (\mathrm{inr}(y), (c_k, c_\ell'))
\end{array}$$

This yields another symmetric monoidal structure.

Tensor Sum


a
f_k
c
s_k
a'
g_k
c'
b
f_ℓ
d
s_ℓ
b'
g_ℓ
d'
inl a
f_k
inl c
f_ℓ
(s_k, s_ℓ)
inl a'
g_k
inl c'
g_ℓ
$k \oplus \ell$

Tensor Sum


a
f_k
c
s_k
a'
g_k
c'
b
f_ℓ
d
s_ℓ
b'
g_ℓ
d'
f_k
inr b
f_ℓ
inr d
(s_k, s_ℓ)
g_k
inr b'
g_ℓ
inr d'
$k \oplus \ell$

Injection lens

Exercise: define a reasonable lens:

$$inl_x \in X \leftrightarrow X + Y$$

Note: $\times$ is not a categorical product; $+$ is not a categorical coproduct.

Injections are funny. Cannot be made natural with respect to $\oplus$.

Using sums for Anthony's special cases

- Suppose we have two lenses $\ell, k : X \leftrightarrow Y$ and would like to use ℓ most of the time, but for those $x \in U \subseteq X$ use k.
- What extra data / axioms / properties do we need ?
- We certainly can use the sum: $\ell \oplus k : X + X \leftrightarrow Y + Y$.
- How to synch between X, Y and $X + X, Y + Y$?

List mapping

$$\frac{\ell \in X \leftrightarrow Y}{map(\ell) \in X^\star \leftrightarrow Y^\star}$$

$$\begin{array}{lcl}
C & = & (\ell.C)^\omega \\
init & = & (\ell.init)^\omega \\
putr(x, c) & = & \text{let } \langle x_1, \ldots, x_m \rangle = x \text{ in} \\
 & & \text{let } \langle c_1, \ldots \rangle = c \text{ in} \\
 & & \text{let } (y_i, c'_i) = \ell.putr(x_i, c_i) \text{ in} \\
 & & (\langle y_1, \ldots, y_m \rangle, \langle c'_1, \ldots, c'_m, c_{m+1}, \ldots \rangle) \\
putl & & \text{(similar)}
\end{array}$$

Folding

Given $\ell : \mathit{Unit} + X \times Z \leftrightarrow Z$ can define $\mathrm{fold}(\ell) : X^{*} \leftrightarrow Z$ such that

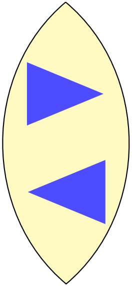
$$\begin{tikzcd}
\mathit{Unit} + X \times X^{\star} \arrow[r, "iso"] \arrow[d, "id_{\mathit{Unit}} \oplus (id_X \otimes \mathrm{fold}(\ell))"'] & X^{\star} \arrow[d, "\mathrm{fold}(\ell)"] \\
\mathit{Unit} + X \times Z \arrow[r, "\ell"'] & Z
\end{tikzcd}$$

…provided that we have some kind of weight function on Z that goes down by doing ℓ.

Exercise: complete this.
Exercise: show that $\mathrm{fold}(\ell)$ is even unique.

Synchronizing Tree Leaves/List Elements

froghead.jpg



disconnect $\otimes$ id

[costume]

Useful Folds

- leaves : $TreeA \leftrightarrow [A]$
- concat : $[[A]] \leftrightarrow [A]$
- partition : $[A \uplus B] \leftrightarrow [A] \times [B]$
- map : $(A \leftrightarrow B) \rightarrow ([A] \leftrightarrow [B])$
- pictures : $\mathsf{FS} \leftrightarrow [\mathsf{Name} \times \mathsf{Picture}]$

Exercise: define those!

Final Lens

$$\begin{tikzcd}
\text{FS} \arrow[d, leftrightarrow, "\text{pictures}"] \\
{[\text{Filename} \times \text{Picture}]} \arrow[d, leftrightarrow, "\text{map}(\text{disconnect} \otimes \text{id})"] \\
{[[\text{Tag}] \times \text{Picture}]}
\end{tikzcd}$$

Forgetful sums and lenses

- The version of sums and lists described is called retentive
- When we change sides or extend list length we use the “retained” values from the last time we were on that side / had that length.
- There is also a forgetful version where upon shortening or changing sides we throw data away.

Further things on datatypes

- Since lenses are self-dual can easily define hylomorphisms: from $k : Z \leftrightarrow \mathit{Unit} + X \times Z$ and $\ell : \mathit{Unit} + X \times W \leftrightarrow W$ obtain $\mathit{Hy}(\ell, k) : Z \leftrightarrow W$ such that $\dots$. Namely, we define $\mathit{Hy}(\ell, k) = \dots$. Exercise: fill in the $\dots$.
- Can define iterators over more than one list.
- Can generalize from lists to other inductive datatypes like binary trees etc.

Containers

- A general framework for datastructures with positions holding data is given by Containers (Joyal, Cockett, Altenkirch, Hasegawa, Ghani, ...):
- Inductive types like lists or trees are also containers, but not vice versa, e.g. labelled graphs are containers but not inductive types.
- A container consists of
  - a set I of shapes, e.g. $I = \mathbb{N}$ for lists
  - for each shape $i \in I$ a set $B(i)$ of positions, e.g. $B(i) = \{0, \ldots, i-1\}$ for lists
- A container (I, B) defines a functor on Sets: $F(X) = \sum_{i \in I} X^{B(i)}$. An element of $F(X)$ consists of a shape i and for each position $p \in B(i)$ an element of X.
- Graphs with X-labelled nodes: I=unlabelled graphs, $B(i)$ nodes of i.
- If $f : X \to Y$ we get a function $F(f) : F(X) \to F(Y)$. Generalizes "map" on lists and trees.

Container lens?

- Would like to generalize $F(f)$ to $F(\ell)$ with ℓ a lens.
- If the shape doesn't change just apply the lens position-wise.
- What if the shape changes (from i to i')?

Ordered containers

- we require a partial ordering with binary meets on shapes.
- $i \leq i'$ means "subshape", e.g., subtree or shorter list.
- If $i \leq i'$ need $B(i) \hookrightarrow B(i')$.
- If $p \in B(i)$ and $p' \in B(i')$ are equal in $B(j)$, thus, $i \leq j, i' \leq j$ then there must exist unique $q \in B(i \wedge i')$ such that ... I.e. B is a pullback preserving functor from I to Sets.
- E.g. meet of two trees = largest common subtree.

Container lens!

$$\frac{\ell \in X \leftrightarrow Y}{F_{I,B}(\ell) \in F_{I,B}(X) \leftrightarrow F_{I,B}(Y)}$$

$$\begin{array}{lcl}
C = & & \\
\quad \{t \in \prod_{i \in I} B(i) \to \ell.(C) \mid & & \\
\qquad \forall i, i'.\ i \leq i' \supset \forall b {\in} B(i).\ t(i')(b|i') = t(i)(b)\} & & \\
\mathit{init}(i)(b) & = & \ell.\mathit{init} \\
\mathit{putr}((i, f), t) & = & \\
\quad \text{let } f'(b) & = & \text{fst}(\ell.\mathit{putr}(f(b), t(i)(b))) \text{ in} \\
\quad \text{let } t'(j)(b) & = & \\
\qquad \text{if } \exists b_0 \in B(i \wedge j).\ b_0|j = b & & \\
\qquad \text{then snd}(\ell.\mathit{putr}(f(b_0|i), t(j)(b))) & & \text{in} \\
\qquad \text{else } t(j)(b) & & \\
((i, f'), t') & & \\
\mathit{putl} & & \text{(similar)}
\end{array}$$

Asymmetric to symmetric

Every asymmetric lens, i.e., a classical lens in the sense of Foster et al., gives rise to a symmetric lens.

$$\frac{\ell \in X \overset{a}{\leftrightarrow} Y}{\ell^{sym} \in X \leftrightarrow Y}$$

$$\begin{array}{lcl} C & = & \{f \in Y \to X \mid \forall y \in Y.\ \ell.get(f(y)) = y\} \\ init & = & \ell.create \\ putr(x, f) & = & (\ell.get(x), f_x) \\ putl(y, f) & = & \text{let } x = f(y) \text{ in } (x, f_x) \end{array}$$

But not all lenses are of that form.

Spans of lenses

However, for any lens ℓ we can find asymmetric lenses k_1, k_2 such that

$$(k_1^{sym})^{op}; k_2^{sym} = \ell$$

Intermediate "type": set of consistent triples:

$$S_\ell = \{(x, y, c) \in X \times Y \times \ell.C \mid \ell.putr(x, c) = (y, c)\}$$

If $\ell : X \leftrightarrow Y$ then $k_1 : S_\ell \to X$ and $k_2 : S_\ell \to Y$.

Exercise: complete this.

Summary “symmetric lenses”

- Generalize asymmetric lenses to become truly bidirectional
- Can be seen as stateful back-and-forth functions
- best understood modulo bisimulation
- bisimulation coincides with observational equivalence
- began to explore the type and combinator structure of the category of lenses
- sums, product, lists, trees, iterators, hylomorphisms, containers.
- Can alternatively be presented as spans of asymmetric lenses.

Further work, open problems

- Integration with programming / frameworks
- Definition of lenses by recursion
- Higher-order functions

Edit lenses

- Add monoid action to sets (monoid elements = edit operation)
- Lens transports edit operations preserving composition and identity (stateful homomorphism).
- State-based lenses arise as special case
- Fold combinators don't work; replaced with powerful mapping and plumbing combinators for containers
- Advantages of edit-based vs. state-based:
  - bandwidth
  - better alignment

First-class edits

Edits are a monoid M:

$$\mathbf{1}_M \cdot m = m \cdot \mathbf{1}_M = m$$

$$m_1 \cdot (m_2 \cdot m_3) = (m_1 \cdot m_2) \cdot m_3$$

With a partial monoid action $\odot \in M \times X \to X$:

$$\mathbf{1}_M \odot x = x$$

$$(m_1 \cdot m_2) \odot x = m_1 \odot (m_2 \odot x)$$

Editing lists

Set ∂X are edits for X.
Define atomic edits E for X^*:

- modify(p,$\mathrm{d}x$) where $p \in \mathcal{N}, \mathrm{d}x \in \partial X$
- resize(i,j,x) where $i, j \in \mathcal{N}, x \in X$
- reorder(i,f) where f permutes $\{0, \ldots, i\}$

Take E^* (words of atomic edits) for list edits $\partial(X^*)$.

Edits = Monoids

- We model edits as a monoid: set M, binary associative operation $\cdot\cdot\cdot_M$, neutral element $\mathbf{1}_M$,
- $m \cdot_M m'$ represents the combined edit comprising first m' then m,
- $\mathbf{1}_M$ is the neutral edit that does nothing,
- Often we use the free monoid over a set of primitive edits, i.e., an edit is just a list of primitive edits to be executed in sequence,
- Sometimes, however, we may want to optimize concatenations of edits $\rightsquigarrow$ non-free monoids.
- Two sequences that are equal in the monoid must behave the same and may be represented identically
- Simple examples: overwrite monoid (state-based lenses), product monoid.

Modules

Module

A module is a tuple $\langle X, \mathit{init}_X, \partial X, \odot_X \rangle$ comprising a set X, an element $\mathit{init}_X \in X$, a monoid ∂X, and a monoid action $\odot_X$ of ∂X on X.

If X is a module, we refer to its first component by either $|X|$ or just X, and to its last component by $\odot$ or simple juxtaposition.

Product module

Consider modules X and Y.

A primitive edit to a pair in $|X| \times |Y|$ is either an edit to the X part or an edit to the Y part.

$$G^{\otimes}_{X,Y} = \{\text{left}(\mathrm{d}x) \mid \mathrm{d}x \in \partial X\} \cup \{\text{right}(\mathrm{d}y) \mid \mathrm{d}y \in \partial Y\}$$

Define $|X \otimes Y| = |X| \times |Y|$ and $\partial(X \otimes Y) = (G^{\otimes}_{X,Y})^*$.

Questions:

- Define the action
- what about imposing equations on $\partial(X \otimes Y)$?

Sum module

Primitive edits to elements of $|X \oplus Y| = |X| + |Y|$:

$$\begin{aligned} G^{\oplus}_{X,Y} &= \{\mathsf{switch}_{iL}(\mathrm{d}x) \mid i \in \{L, R\}, \mathrm{d}x \in \partial X\} \\ &\cup \{\mathsf{switch}_{iR}(\mathrm{d}y) \mid i \in \{L, R\}, \mathrm{d}y \in \partial Y\} \\ &\cup \{\mathsf{stay}_{L}(\mathrm{d}x) \mid \mathrm{d}x \in \partial X\} \cup \{\mathsf{stay}_{R}(\mathrm{d}y) \mid \mathrm{d}y \in \partial Y\} \\ &\cup \{\mathsf{fail}\} \end{aligned}$$

Define $|X \otimes Y| = |X| \times |Y|$ and $\partial(X \otimes Y) = (G^{\otimes}_{X,Y})^*$.

Again, we leave the action as an exercise.
Question: what about imposing equations on $\partial(X \otimes Y)$?
Odd phenomenon: no matter how you do it, you don't seem to get $X \oplus (Y \oplus Z) \simeq (X \oplus Y) \oplus Z$.

List module

Primitive edits to elements of $|X^*| = |X|^*$:

$$\begin{array}{rcl} G_X^{\mathrm{list}} & = & \{\mathsf{mod}(p, \mathrm{d}x) \mid p \in \mathbb{N}^+, \mathrm{d}x \in \partial X\} \\ & \cup & \{\mathsf{ins}(i) \mid i \in \mathbb{N}\} \quad \cup \quad \{\mathsf{del}(i) \mid i \in \mathbb{N}\} \\ & \cup & \{\mathsf{reorder}(f) \mid \forall i \in \mathbb{N}. f(i) \text{ permutes } \{1, \ldots, i\}\} \\ & \cup & \{\mathsf{fail}\} \end{array}$$

Define $\partial(X^*) = (G_X^{\mathrm{list}})^*$.

Question: can we get this automatically from the initial algebra definition of X^*?

Points for discussion

- Why should application of edits be partial?
- Why distinguish between monoid element and the induced function?
- Lawful vs. free monoid

Stateful monoid homomorphisms

Definition

Given monoids M and N and a complement set C, a stateful monoid homomorphism from M to N over C is a function $h \in M \times C \to N \times C$ satisfying two laws:

$$\frac{}{h(\mathbf{1}_M, c) = (\mathbf{1}_N, c)}$$

$$\frac{h(m, c) = (n, c') \qquad h(m', c') = (n', c'')}{h(m' \cdot_M m, c) = (n' \cdot_N n, c'')}$$

Exercise / question: try to reformulate that as a standard homomorphism between a different kind of monoids.

Lens definition

Definition

Edit lens $\ell : \langle M, X \rangle \leftrightarrow \langle N, Y \rangle$ has:

- a complement set C of private data
- consistency relation $K \in X \times C \times Y$
- stateful monoid homomorphisms

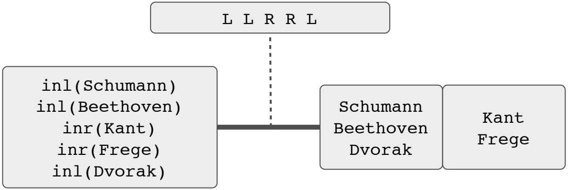
$$\Rrightarrow : M \times C \to N \times C$$

$$\Lleftarrow : N \times C \to M \times C$$

that preserve consistency

Exercise: what does “preserve consistency” mean?

A glimpse at a later slide



`L L R R L`

```
inl(Schumann)
inl(Beethoven)
inr(Kant)
inr(Frege)
inl(Dvorak)
```

```
Schumann
Beethoven
Dvorak
```

```
Kant
Frege
```

Consistency vs. round-trip laws

Round-trip laws:
"<u>There exists an invariant restored by the lens</u>."

Consistency relations:
"There exists an invariant restored by the lens,
<u>and that invariant is K</u>."

Behavioural equivalence

As before, we may consider lenses up to equivalence thus obtaining a category.

Definition (Lens equivalence)

Two lenses $k, \ell : X \leftrightarrow Y$ are equivalent (written $k \equiv \ell$) if, there exists a relation $S \subseteq X \times k.C \times \ell.C \times Y$ such that

- $(init_X, k.init, \ell.init, init_Y) \in S$;
- if $(x, c, d, y) \in S$ and $\mathrm{d}x\ x$ is defined, then if $(\mathrm{d}y_1, c') = k.\Rrightarrow(\mathrm{d}x, c)$ and $(\mathrm{d}y_2, d') = \ell.\Rrightarrow(\mathrm{d}x, d)$, then $\mathrm{d}y_1 = \mathrm{d}y_2$ and $(\mathrm{d}x\ x, c', d', \mathrm{d}y_1\ y) \in S$; and
- analogously for $\Lleftarrow$.

Again, there is an equivalent definition with dialogues (exercise!)

Product lens

$$\frac{k \in X \leftrightarrow Z \qquad \ell \in Y \leftrightarrow W}{k \otimes \ell \in X \otimes Y \leftrightarrow Z \otimes W}$$

$$\begin{array}{lcl} C & = & k.C \times \ell.C \\ \mathit{init} & = & (k.\mathit{init}, \ell.\mathit{init}) \\ K & = & \{\, ((x,z),(c_k,c_\ell),(y,w)) \mid \\ & & \qquad\qquad (x,c_k,y) \in k.K \\ & & \qquad\quad \wedge\, (z,c_\ell,w) \in \ell.K \,\} \end{array}$$

Sum lens

$$\frac{k \in X \leftrightarrow Y \qquad \ell \in Z \leftrightarrow W}{k \oplus \ell \in X \oplus Z \leftrightarrow Y \oplus W}$$

$$\begin{array}{lcl}
C & = & k.C + \ell.C \\
init & = & \mathrm{inl}(k.init) \\
K & = & \{(\mathrm{inl}(x), \mathrm{inl}(c), \mathrm{inl}(y)) \\
 & & \quad \mid (x, c, y) \in k.K\} \\
 & \cup & \{(\mathrm{inr}(z), \mathrm{inr}(c), \mathrm{inr}(w)) \\
 & & \quad \mid (z, c, w) \in \ell.K\} \\
c_k & = & k.init \\
c_\ell & = & \ell.init \\
\ldots & &
\end{array}$$

Sum lens, cont'd

$$\ldots$$

$$
\begin{array}{lcl}
\ldots & & \\
\Rrightarrow_g(\mathsf{switch}_{LL}(\mathrm{d}x), \mathsf{inl}(c)) & = & \mathsf{let}\ (\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c_k) \\
 & & \mathsf{in}\ (\mathsf{switch}_{LL}(\mathrm{d}y), \mathsf{inl}(c')) \\
\Rrightarrow_g(\mathsf{switch}_{RL}(\mathrm{d}x), \mathsf{inr}(c)) & = & \mathsf{let}\ (\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c_k) \\
 & & \mathsf{in}\ (\mathsf{switch}_{RL}(\mathrm{d}y), \mathsf{inl}(c')) \\
\Rrightarrow_g(\mathsf{switch}_{LR}(\mathrm{d}z), \mathsf{inl}(c)) & = & \mathsf{let}\ (\mathrm{d}w, c') = \ell.\Rrightarrow(\mathrm{d}z, c_\ell) \\
 & & \mathsf{in}\ (\mathsf{switch}_{LR}(\mathrm{d}w), \mathsf{inr}(c')) \\
\Rrightarrow_g(\mathsf{switch}_{RR}(\mathrm{d}z), \mathsf{inr}(c)) & = & \mathsf{let}\ (\mathrm{d}w, c') = \ell.\Rrightarrow(\mathrm{d}z, c_\ell) \\
 & & \mathsf{in}\ (\mathsf{switch}_{RR}(\mathrm{d}w), \mathsf{inr}(c')) \\
\Rrightarrow_g(\mathsf{stay}_{L}(\mathrm{d}x), \mathsf{inl}(c)) & = & \mathsf{let}\ (\mathrm{d}y, c') = k.\Rrightarrow(\mathrm{d}x, c) \\
 & & \mathsf{in}\ (\mathsf{stay}_{L}(\mathrm{d}y), \mathsf{inl}(c')) \\
\Rrightarrow_g(\mathsf{stay}_{R}(\mathrm{d}z), \mathsf{inr}(c)) & = & \mathsf{let}\ (\mathrm{d}w, c') = \ell.\Rrightarrow(\mathrm{d}z, c) \\
 & & \mathsf{in}\ (\mathsf{stay}_{R}(\mathrm{d}w), \mathsf{inr}(c')) \\
\Rrightarrow_g(e, c) & = & (\mathsf{fail}, c)\ \text{in all other cases}
\end{array}
$$

List mapping lens

$$\frac{\ell \in X \leftrightarrow Y}{\ell^* \in X^* \leftrightarrow Y^*}$$

$$\begin{array}{lcl}
C & = & \ell.C^* \\
\mathit{init} & = & \varepsilon \\
K & = & \{(x, c, y) \mid |x| = |c| = |y| \wedge \\
 & & \qquad \forall 1 \le p \le |x|.\ (x_p, c_p, y_p) \in \ell.K\} \\
\Rrightarrow_g(\mathrm{mod}(p, \mathrm{d}x), c) & = & \text{let } (\mathrm{d}y, c'_p) = \ell.\Rrightarrow(\mathrm{d}x, c_p) \text{ in} \\
 & & (\mathrm{mod}(p, \mathrm{d}y), c[p \mapsto c'_p])) \\
 & & \ \text{when } p \le n \\
\Rrightarrow_g(\mathrm{mod}(p, \mathrm{d}x), c) & = & (\mathrm{fail}, c) \text{ when } p > n \\
\Rrightarrow_g(\mathrm{fail}, c) & = & (\mathrm{fail}, c) \\
\Rrightarrow_g(\mathrm{d}x, c) & = & (\mathrm{d}x, \mathrm{d}x\, c) \text{ in all other cases} \\
\Lleftarrow & & \text{similar}
\end{array}$$

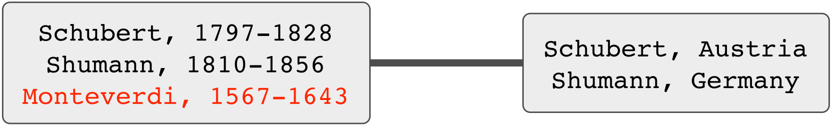
Synchronising composers

Schubert, 1797–1828
Shumann, 1810–1856

Schubert, Austria
Shumann, Germany

(a) initial replicas

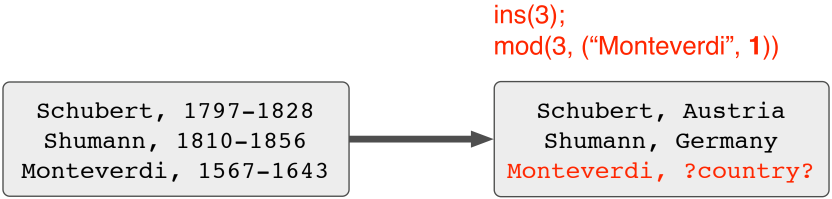
ins(3);
mod(3, (“Monteverdi”, “1567-1643”))



Schubert, 1797–1828
Shumann, 1810–1856
Monteverdi, 1567–1643

Schubert, Austria
Shumann, Germany

(b) a new composer is added to one replica

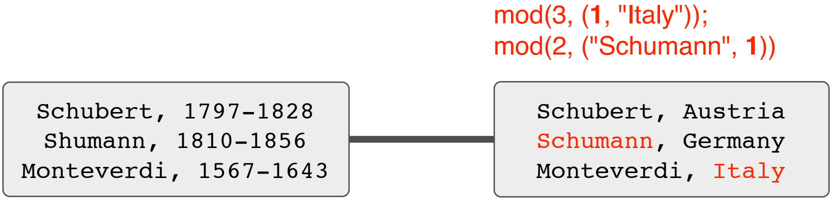


ins(3);
mod(3, (“Monteverdi”, **1**))

Schubert, 1797–1828
Shumann, 1810–1856
Monteverdi, 1567–1643

Schubert, Austria
Shumann, Germany
Monteverdi, ?country?

(c) the lens adds the new composer to the other replica



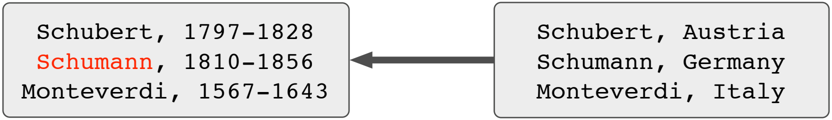
mod(3, (**1**, "Italy"));
mod(2, ("Schumann", **1**))

Schubert, 1797–1828
Shumann, 1810–1856
Monteverdi, 1567–1643

Schubert, Austria
Schumann, Germany
Monteverdi, Italy

(d) the curator makes some corrections

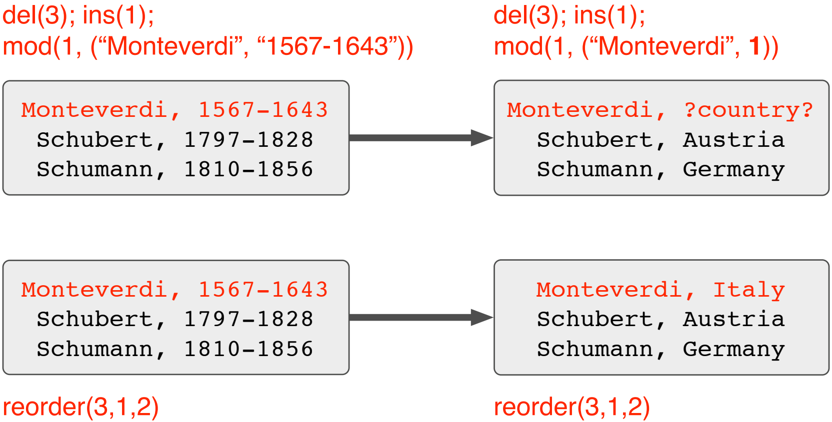
1;
mod(2, ("Schumann", **1**))



Schubert, 1797–1828
Schumann, 1810–1856
Monteverdi, 1567–1643

Schubert, Austria
Schumann, Germany
Monteverdi, Italy

(e) the lens transports a small edit



del(3); ins(1);
mod(1, (“Monteverdi”, “1567-1643”))

del(3); ins(1);
mod(1, (“Monteverdi”, **1**))

```
Monteverdi, 1567-1643
  Schubert, 1797-1828
  Schumann, 1810-1856
```

```
Monteverdi, ?country?
  Schubert, Austria
  Schumann, Germany
```

```
Monteverdi, 1567-1643
  Schubert, 1797-1828
  Schumann, 1810-1856
```

```
  Monteverdi, Italy
  Schubert, Austria
  Schumann, Germany
```

reorder(3,1,2)

reorder(3,1,2)

(f) two different edits with the same effect on the left

Partition lens

- We seek a lens of the form

$$\boxed{\mathit{partition} \in (X \oplus Y)^* \leftrightarrow X^* \otimes Y^*}$$

$$\boxed{\quad\dots\quad}$$

- Once we have it, we can compose many important lenses on lists from it:
  - Use mapping to go from Z^* to $(X \oplus Y)^*$,
  - Transform to $X^* \otimes Y^*$ by partitioning,
  - Work on both parts separately using tensor lens,
  - Go back to $Z*$.

Partition: the code view

$$\mathit{partition} \in (X \oplus Y)^* \leftrightarrow X^* \otimes Y^*$$

$$
\begin{aligned}
C &= \{L, R\}^* \\
\mathit{init} &= \varepsilon \\
K &= \{(z, \mathsf{map}_{\mathsf{tagof}}(z), (\mathsf{lefts}(z), \mathsf{rights}(z))) \mid z \in (|X| + |Y|)^*\}
\end{aligned}
$$

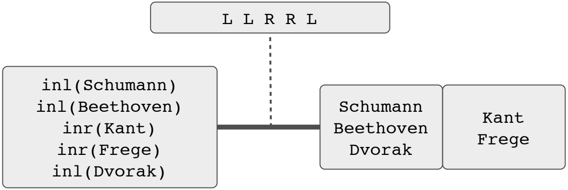
$$
\begin{aligned}
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}v), c) &= (\mathsf{fail}, c) \text{ when } p > |c| && (1)\\
\Rrightarrow_g(\mathsf{mod}(p, \varepsilon), c) &= (\varepsilon, c) \text{ when } 1 \le p \le |c| && (2)\\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}v\mathrm{d}vs), c) &= (d'\,d, c'') \text{ where } 1 < n \quad (d, c') = \Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}vs), c) && (3)\\
&\qquad 1 \le p \le |c| \quad (d', c'') = \Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}v), c') \\
\Rrightarrow_g(\mathsf{mod}(p, \mathsf{switch}_{jk}(\mathrm{d}v)), c) &= (d_2 d_1 d_0, c[p \mapsto k]),\ \text{where } (p_L, p_R) = \mathsf{count}(p, c) \quad d_0 = \mathsf{map}_{\lambda d.\,\mathsf{tag}(j,d)}(\mathsf{del}'(p_j)) && (4)\\
&\qquad d_2 = \mathsf{tag}(k, \mathsf{mod}(p_k, \mathrm{d}v)) \quad d_1 = \mathsf{map}_{\lambda d.\,\mathsf{tag}(k,d)}(\mathsf{ins}'(p_k)) \\
\Rrightarrow_g(\mathsf{mod}(p, \mathsf{stay}_j(\mathrm{d}v)), c) &= (\mathsf{tag}(j, \mathsf{mod}(p_j, \mathrm{d}v)), c),\ \text{where } (p_L, p_R) = \mathsf{count}(p, c) && (5)\\
\Rrightarrow_g(\mathsf{mod}(p, \mathsf{fail}), c) &= (\mathsf{fail}, c) && (6)\\
\Rrightarrow_g(\mathsf{ins}(i), c) &= (\mathsf{left}(\mathsf{ins}(i)), \mathsf{ins}(i)\ c) && (7)\\
\Rrightarrow_g(\mathsf{del}(i), c) &= (d_1 d_0, \mathsf{del}(i)\ c),\ \text{where} \quad c' = \mathsf{reverse}(c) \quad d_0 = \mathsf{left}(\mathsf{del}(n_L{-}1)) && (8)\\
&\qquad (n_L, n_R) = \mathsf{count}(i{+}1, c') \quad d_1 = \mathsf{right}(\mathsf{del}(n_R{-}1)) \\
\Rrightarrow_g(\mathsf{reorder}(f), c) &= (d_L d_R, c'),\ \text{where} \quad h = \mathsf{iso}(c) \quad c' = \mathsf{reorder}(f)\ c && (9)\\
&\qquad h' = \mathsf{iso}(c') \quad (n_L, n_R) = \mathsf{count}(|c|, c) \\
&\qquad h'' = h'^{-1}; f(|c|); h \quad f_k(n \neq n_k) = \lambda p.\,p \\
&\qquad d_L = \mathsf{left}(\mathsf{reorder}(f_L)) \quad f_L(n_L) = \mathsf{inl}; h''; \mathsf{out} \\
&\qquad d_R = \mathsf{right}(\mathsf{reorder}(f_R)) \quad f_R(n_R) = \mathsf{inr}; h''; \mathsf{out} \\
\Rrightarrow_g(\mathsf{fail}, c) &= (\mathsf{fail}, c) && (10)
\end{aligned}
$$

$$
\begin{aligned}
\Lleftarrow_g(\varepsilon, c) &= (\varepsilon, c) && (11)\\
\Lleftarrow_g(\mathrm{d}v\mathrm{d}vs, c) &= (d'd, c'') \text{ when } n > 1,\ \text{where } (d, c') = \Lleftarrow_g(\mathrm{d}vs, c) \quad (d', c'') = \Lleftarrow_g(\mathrm{d}v, c') && (12)\\
\Lleftarrow_g(\mathsf{left}(\mathsf{mod}(p, \mathrm{d}x)), c) &= (\mathsf{stay}_L(\mathsf{mod}(p', \mathrm{d}x)), c),\ \text{where } p' = \mathsf{iso}(c)^{-1}(\mathsf{inl}(p)) && (13)\\
\Lleftarrow_g(\mathsf{left}(\mathsf{reorder}(f)), c) &= (\mathsf{reorder}(f'), c),\ \text{where } g(\mathsf{inr}(p)) = \mathsf{inr}(p) \quad f'(n \neq |c|) = \lambda p.\,p && (14)\\
&\qquad g(\mathsf{inl}(p)) = \mathsf{inl}(f(n_L)(p)) \quad f'(|c|) = h; g; h^{-1} \\
&\qquad (n_L, n_R) = \mathsf{count}(|c|, c) \quad h = \mathsf{iso}(c) \\
\Lleftarrow_g(\mathsf{left}(\mathsf{ins}(i)), c) &= (\mathsf{ins}(i), \mathsf{ins}(i)\ c) && (15)\\
\Lleftarrow_g(\mathsf{left}(\mathsf{del}(0)), c) &= (\varepsilon, c) && (16)\\
\Lleftarrow_g(\mathsf{left}(\mathsf{del}(i)), c) &= (d''\ \mathsf{del}'(p), c''),\ \text{where } h = \mathsf{iso}(c) \quad (n_L, n_R) = \mathsf{count}(|c|, c) && (17)\\
&\qquad p = h^{-1}(\mathsf{inl}(n_L)) \quad (d'', c'') = \Lleftarrow_g(d', c') \\
&\qquad c' = \mathsf{del}'(p)\ c \quad d' = \mathsf{left}(\mathsf{del}(i{-}1)) \\
&\quad \text{when } 0 < i \le n_L{+}1 \\
\Lleftarrow_g(\mathsf{left}(\mathsf{del}(i)), c) &= (\mathsf{fail}, c) \text{ otherwise} && (18)\\
\Lleftarrow_g(\mathsf{left}(\mathsf{fail}), c) &= (\mathsf{fail}, c) && (19)\\
\Lleftarrow_g(\mathsf{right}(\mathrm{d}y), c) &\quad \text{similar}
\end{aligned}
$$

High-level view

Complement: $C = \{L, R\}^*$. Tells where the positions of the LHS belong.

A consistent triple:



L L R R L

inl(Schumann)
inl(Beethoven)
inr(Kant)
inr(Frege)
inl(Dvorak)

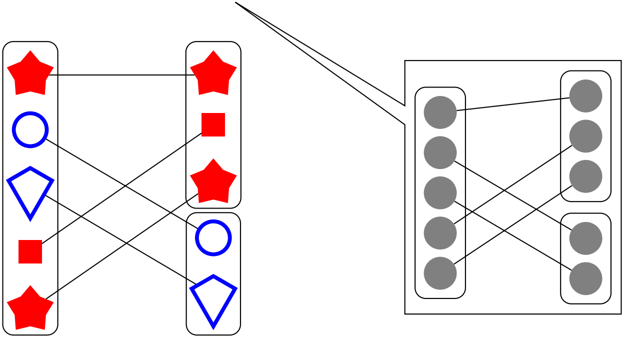
Schumann
Beethoven
Dvorak

Kant
Frege

Partition: the consistency view

$$K \subset (A+B)^* \times C \times (A^* \times B^*)$$

$$K = \{(z, \cdots, (\mathrm{lefts}(z), \mathrm{rights}(z)))\}$$

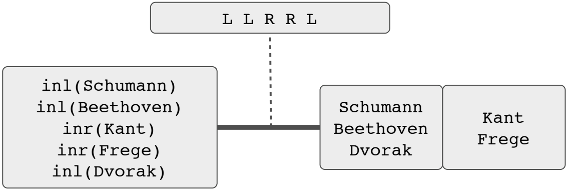


Propagating edits

Figure out how to propagate these edits from left to right:

- $\mathrm{mod}(5, \mathrm{stay}_L(\mathrm{d}n))$, i.e., change “Dvorak” to “Dvořák”.
- Insert or delete a person on the left,
- Reorderings
- Switching sides, e.g. replace Beethoven with Plato $\mathrm{mod}(2, \mathrm{switch}_{LR}(\mathrm{d}n))$

What about similar edits from right to left?



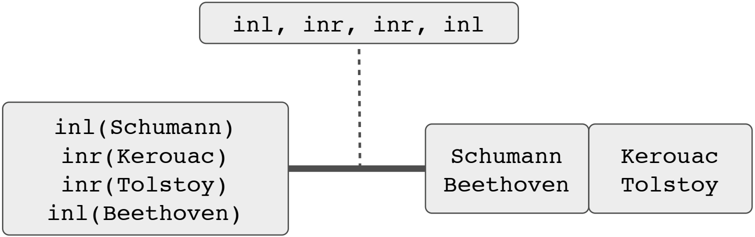
```
L L R R L
```

```
inl(Schumann)
inl(Beethoven)
inr(Kant)
inr(Frege)
inl(Dvorak)
```

```
Schumann
Beethoven
Dvorak
```

```
Kant
Frege
```

Example



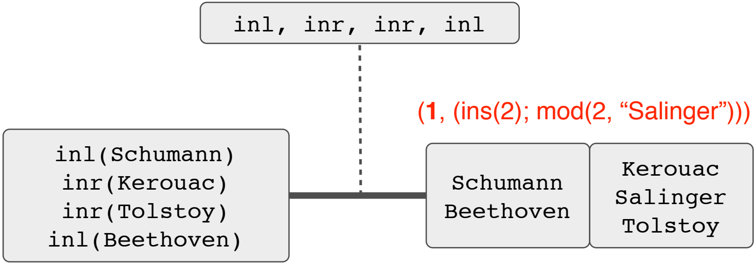
`inl, inr, inr, inl`

`inl(Schumann)`
`inr(Kerouac)`
`inr(Tolstoy)`
`inl(Beethoven)`

`Schumann`
`Beethoven`

`Kerouac`
`Tolstoy`

Notice L↔ inl and R↔ inr



inl, inr, inr, inl

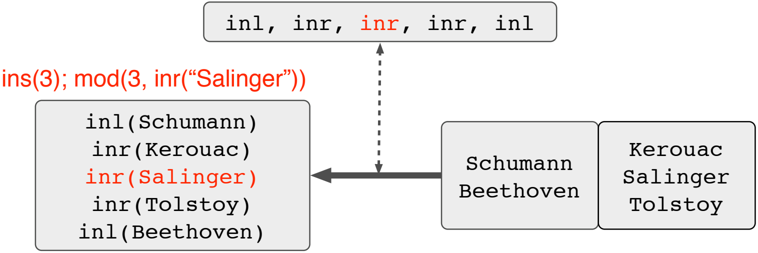
(1, (ins(2); mod(2, “Salinger”)))

inl(Schumann)
inr(Kerouac)
inr(Tolstoy)
inl(Beethoven)

Schumann
Beethoven

Kerouac
Salinger
Tolstoy

(b) an element is added to one of the partitions



inl, inr, inr, inr, inl

ins(3); mod(3, inr(“Salinger”))

inl(Schumann)
inr(Kerouac)
inr(Salinger)
inr(Tolstoy)
inl(Beethoven)

Schumann
Beethoven

Kerouac
Salinger
Tolstoy

(c) the complement tells how to translate the index

Container type

- A module I of shapes (can edit the shapes!) additionally endowed with a partial order (as before).
- A fixed set P of positions
- For each shape i a subset $\mathrm{live}(i) \subseteq P$.
- Can recover $B(i) = \{p \mid p \in \mathrm{live}(i)\}$.

Position edits

Allowed edits

Let $T = \langle I, P, \mathsf{live}\rangle$ be a container type. An edit $\mathrm{d}i \in \partial I$ is an insertion if $\mathrm{d}i\ i \geq i$ whenever defined. It is a deletion if $\mathrm{d}i\ i \leq i$ whenever defined. It is a rearrangement if $|\mathsf{live}(\mathrm{d}i\ i)| = |\mathsf{live}(i)|$ (same cardinality) whenever defined.

- We only employ edits from these three categories as ingredients of container edits; any other edits in the module will remain unused.
- This division of container edits into "pure" insertions, deletions, and rearrangements facilitates the later definition of lenses operating on such edits.
- Q: Should we allow more edits?

Container edits

We define the monoid of edits for a container type $\langle I, P, \mathsf{live}\rangle$ as the free (for now!) monoid generated by

- Modifications: $\mathsf{mod}(p, \mathrm{d}x)$ where $p \in P$ and $\mathrm{d}x \in \partial X$,
- Insertions: $\mathsf{ins}(\mathrm{d}i)$ with $\mathrm{d}i$ an insertion,
- Deletions: $\mathsf{del}(\mathrm{d}i)$ with $\mathrm{d}i$ a deletion,
- Rearrangements: $\mathsf{rearr}(\mathrm{d}i, f)$ with $\mathrm{d}i$ a rearrangement and $f : \mathsf{live}(i) \simeq \mathsf{live}(\mathrm{d}i\, i)$.
- Fail: fail :-)

Action of container edits

fail (i, f) is always undefined

mod$(p, \mathrm{d}x)$ $(i, f) = (i, f[p \mapsto \mathrm{d}x\ f(p)])$ when $p \in$ live(i)

ins$(\mathrm{d}i)$ $(i, f) = (\mathrm{d}i\ i, f')$

where $f'(p) =$ if $p \in$ live(i) then $f(p)$ else init_X

del$(\mathrm{d}i)$ $(i, f) = (\mathrm{d}i\ i, f{\restriction}\text{live}(\mathrm{d}i\ i))$

rearr$(\mathrm{d}i, f)$ $(i, g) = (\mathrm{d}i\ i, g')$

where $g'(p) = g(f(i)(p))$

Container mapping lens

$$\frac{\ell \in X \leftrightarrow Y \qquad T = \langle I, P, \mathsf{live}\rangle \ \text{ a container type}}{T(\ell) \in T(X) \leftrightarrow T(Y)}$$

$$\begin{array}{lcl}
C & = & T(\ell.C) \\
\mathit{init} & = & (\mathit{init}_I, \lambda p.\ \ell.\mathit{init}) \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), (i, f)) & = & (\mathsf{mod}(p, \mathrm{d}y), (i, f')) \\
\multicolumn{3}{c}{\text{when } p \in \mathsf{live}(i) \text{ and where}} \\
\multicolumn{3}{c}{f' = f[p \mapsto c'], (\mathrm{d}y, c') = \ell.\Rrightarrow(\mathrm{d}x, f(p))} \\
\Rrightarrow_g(\mathsf{mod}(p, \mathrm{d}x), (i, f)) & = & (\mathsf{fail}, (i, f)) \text{ if } p \notin \mathsf{live}(i) \\
\Rrightarrow_g(\mathsf{ins}(\mathrm{d}i), (i, g)) & = & (\mathsf{ins}(\mathrm{d}i), \\
 & & (\mathrm{d}i\ i, g[p \mapsto \ell.\mathit{init}])) \\
 & & \text{ when } \mathrm{d}i\ i \text{ is defined} \\
\Rrightarrow_g(\mathsf{del}(\mathrm{d}i), (i, g)) & = & (\mathsf{del}(\mathrm{d}i), (\mathrm{d}i\ i, g{\restriction}\mathsf{live}(\mathrm{d}i\ i))) \\
 & & \text{ when } \mathrm{d}i\ i \text{ is defined} \\
\Rrightarrow_g(\mathsf{rearr}(\mathrm{d}i, h), (i, g)) & = & (\mathsf{rearr}(\mathrm{d}i, h), \\
 & & (\mathrm{d}i\ i, \lambda p.g(h(i)(p))))
\end{array}$$

Container restructuring lens

$$\frac{\begin{array}{c} T = \langle I, P, \mathrm{live} \rangle \ \text{ a container type} \\ T' = \left\langle I', P', \mathrm{live}' \right\rangle \ \text{ a container type} \\ \ell \in I \leftrightarrow I' \end{array}}{[T, T'](\ell) \in T(X) \leftrightarrow T'(X)}$$

$$\ldots$$

Container restructuring lens, cont'd

$$\ldots$$

$$
\begin{aligned}
C &= \ell.K \\
init &= (init_I, \ell.init, init_{I'}) \\
K = \{&((i,f),(i,c,i'),(i',f')) \\
&\mid (i,c,i') \in \ell.K \wedge \forall p \in \mathrm{live}'(i').f(f_{i,c,i'}(p)) = f'(p)\} \\
\Rrightarrow_g(\mathrm{mod}(p,\mathrm{d}x),(i,c,i')) &= (\mathrm{mod}(f^{-1}_{i,c,i'}(p),\mathrm{d}x),(i,c,i') \\
&\quad \text{when } p \in \mathrm{live}(i) \\
\Rrightarrow_g(\mathrm{ins}(\mathrm{d}i),(i,c,i')) &= (\mathrm{rearr}(\mathbf{1},f_i)\mathrm{ins}(\mathrm{d}i'), \\
&\quad (\mathrm{d}i\ \ i, c', \mathrm{d}i'\ \ i')) \\
\Rrightarrow_g(\mathrm{del}(\mathrm{d}i),(i,c,i')) &= (\mathrm{rearr}(\mathbf{1},f_d)\mathrm{del}(\mathrm{d}i'), \\
&\quad (\mathrm{d}i\ \ i, c', \mathrm{d}i'\ \ i')) \\
\Rrightarrow_g(\mathrm{rearr}(\mathrm{d}i,f),(i,c,i')) &= (\mathrm{rearr}(\mathrm{d}i',f_r), \\
&\quad (\mathrm{d}i\ \ i, c', \mathrm{d}i'\ \ i'))
\end{aligned}
$$

Container restructuring lens, cont'd

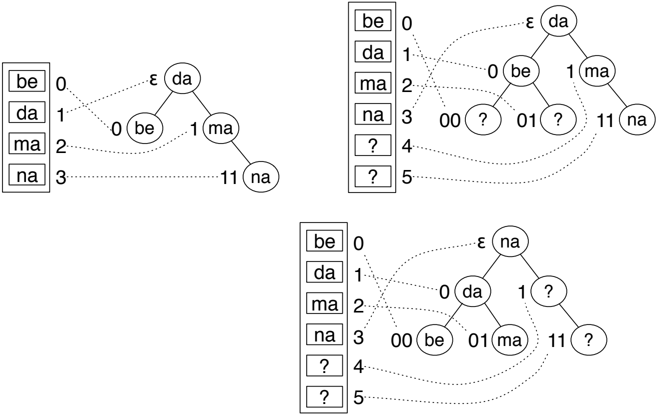
Three families of bijections f_i, f_d, f_r.
must be chosen in such a way that the container edits in which they appear are well-formed (this is possible since $\mathrm{d}i'$ is an insertion, deletion, or restructuring as appropriate) and such that the following three constraints are satisfied: in each case i, i', etc., refer to the current values from above and $p \in \mathsf{live}'(\mathrm{d}i'\ i')$ is an arbitrary position.

$$\begin{aligned}
f_i(\mathrm{d}i'\ i')(p) &= f^{-1}_{i,c,i'}(f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p)) \\
&\qquad \text{when } f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p) \in \mathsf{live}(i) \\
f_d(\mathrm{d}i'\ i')(p) &= f^{-1}_{i,c,i'}(f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p)) \\
f_r(\mathrm{d}i'\ i')(p) &= f^{-1}_{i,c,i'}(f(i)(f_{\mathrm{d}i\ i,c',\mathrm{d}i'\ i'}(p)))
\end{aligned}$$

Exercise / open question: give a more uniform treatment of restructuring.

Container plumbing in action

Restructuring lens models “in-order” flattening.



Inserting two fresh nodes at the end of the list, propagation and restoration of consistency.

Loose end: Typed edit language

An typed edit language (tentative!) comprises

- a set T of "types"
- for each $t \in T$ a set $X(t)$ with distinguished element $\mathit{init}_{X(t)} \in X(t)$
- for any two types t, t' a set of edits $\partial X(t, t')$ with composition and identities, i.e. a category!
- an action of ∂X on X: if $e \in \partial X(t, t')$ and $x \in X(t)$ then $e.x \in X(t')$. I.e. $X(-)$ becomes a set-valued functor (presheaf).

Example: $T =$ list lengths or abstraction thereof, e.g. $= 0, > 0$. Removes partiality of hd, tl.

Idea: Use types to distinguish inl's and inr's in a sum. Solution to associativity conundrum.

Sets to modules

Let X be a set. The free monoid X^* acts on X by

$$(x_n \dots x_1)x = x_n$$

For $x \in X$ define module X_x as $X_x = (X, x, X^*)$.

State-based to edit-based

Let $\ell : X \leftrightarrow Y$ be a state-based symmetric lens and $\ell.\mathit{putr}(x, \ell.\mathit{missing}) = (y, \ell.\mathit{missing})$ be a consistent triple for ℓ.

Exercise: Define an edit based lens between X_x and Y_y.

Edit-based to state-based

Let X be a module. A differ for X is a binary operation $\mathit{dif} \in X \times X \to \partial X$ satisfying $\mathit{dif}(x, x')x = x'$ and $\mathit{dif}(x, x) = \mathbf{1}$.

Thus, a differ finds, for given states x, x', an edit operation $\mathrm{d}x$ such that $\mathrm{d}x\ x = x'$ and $\mathrm{d}x$ is “reasonable” at least in the sense that if $x = x'$ then the produced edit is minimal, namely $\mathbf{1}$.

Exercise: discuss possible differs for X^*.
Exercise: explain how to obtain a state-based lens from an edit-based lens with differ.

Summary edit lenses

- Modelled editing as sets with a partial monoid action. Idea: aaply a apatch to a state.
- Edit lenses are (total!) back-and-forth functions translating edits. As before stateful. Stateful monoid homomorphism. In addition, a consistency relation to be preserved.
- Folding replaced by container mapping, container restructuring, and list partitioning.

What’s missing?

- We seek edit lens primitives for trees and (later on) graphs with unordered children as in XML or WWW.
- Various options beyond hand-crafting from the definitions: Containers “modulo”, a.k.a. combinatorial species (Joyal).
- Use tree automata to describe well-formed unordered trees
- Use wp-calculation to delineate edit operations preserving well-formedness.
- Natural instance of typed edit languages

Information trees

- ... are unordered trees whose edges are labeled by Σ^* with Σ a finite alphabet.
- use braces $\{\!|\,|\!\}$ and $\mapsto$ to denote trees.

$$\{\!|\mathtt{name} \mapsto \{\!|\mathtt{John} \mapsto \{\!|\,|\!\}|\!\}, \mathtt{email} \mapsto \{\!|\mathtt{john@example.com} \mapsto \{\!|\,|\!\}|\!\}$$

- same in abbreviated form:

$$\{\!|\mathtt{name} \mapsto \mathtt{John}, \mathtt{email} \mapsto \mathtt{john@example.com}|\!\}$$

Edit language for trees

$$
\begin{array}{rcl}
e & ::= & \mathit{insert}(t) \;\mid \\
 & & \mathit{hoist}(m, n) \;\mid \\
 & & \mathit{delete}(m) \;\mid \\
 & & \mathit{rename}(m, n) \;\mid \\
 & & \mathit{at}(n, e)
\end{array}
$$

where m, n are names, t is a tree.
Action on trees omitted (guess from names of edits :-)

Sheaves automata (Lugiez et al)

Define tree types (document types) by a special kind of automata (sheaves automata).

Intuitively, a sheaves automaton has a set of states Q and for each $q \in Q$ a sheaves formula which partitions the allowed subtrees into disjoint classes (recursively using states) and specifies an arithmetic constraint between the numbers of subtrees falling into each class.

E.g. two states "person", "address". A "person" has one "address" labelled `address` and many "persons" labeled `friend`. An "address" has subtrees labelled `Street`, `Town`, etc. some of them optional

More examples

- File systems

$$\begin{aligned} \mathsf{FS} &::= (.^* \to \mathsf{F} \mid \mathsf{D})^* \\ \mathsf{F} &::= \texttt{f} \to .^* \\ \mathsf{D} &::= \texttt{d} \to \mathsf{FS} \end{aligned}$$

- Special naming conventions, filenames starting with dot or ending with bin, . . .)
- Tree-structured representation of program text
- Tree representation of game states (SGF)

Known results about sheaves automata

Inclusion and nonemptiness of sheaves automata is decidable; boolean operations are computable.

Presentation of sheaves automata as type system by Pierce & Foster.

Weakest preconditions

Our result (BX2013):
For sheaves automaton A and tree edit e can compute sheaves automaton $e.A$ such that

$$t \in L(e.A) \iff e.t \text{ fails} \vee e.t \in L(A)$$

Write $e : A \to B$ to mean that $\forall t \in L(A).\ e.t$ defined $\Rightarrow e.t \in L(B)$.

We have $e : A \to B \iff L(A) \subseteq L(e.B)$ (decidable!)

Weakest preconditions

Our result (BX2013):
For sheaves automaton A and tree edit e can compute sheaves automaton $e.A$ such that

$$t \in L(e.A) \iff e.t \text{ fails} \vee e.t \in L(A)$$

Write $e : A \to B$ to mean that $\forall t \in L(A).\ e.t$ defined $\Rightarrow e.t \in L(B)$.

We have $e : A \to B \iff L(A) \subseteq L(e.B)$ (decidable!)

$e : A \to B \iff L(A) \subseteq L(e.B)$

"$\Rightarrow$": Suppose $e : A \to B$ and $t \in L(A)$. If $e.t$ is undefined then $e.t \in L(e.B)$ by definition of $e.B$. So, assume $e.t$ defined. By assumption $e.t \in L(B)$ and, again by definition of $e.B$, we have $t \in L(e.B)$.

"$\Leftarrow$": Suppose $L(A) \subseteq L(e.B)$ and $t \in L(A)$ and $e.t$ defined. Then, $t \in L(e.B)$ and, since $e.t$ defined, $e.t \in L(B)$, QED.

Construction of $e.B$

- For every edit e define (by induction on e) a sheaves automaton D_e such that $L(D_e) = \{t \mid e.t \text{ undefined}\}$.
- For every edit e define (by induction on e) a sheaves automaton $e \star B$ such that whenever $e.t$ is defined then $t \in L(e \star B) \iff e.t \in L(B)$ (by anticipating the action of e). If $e.t$ is undefined then t may or may not be in $e \star B$.
- Then put $e.B = D_e \vee e \star B$ with $\vee$ denoting union construction for sheaves automata.
- Unfortunately, union requires product construction ($\rightsquigarrow$ blowup). Should consider nondeterministic automata.

Example $e = \mathit{insert}(t')$

- Recall: inserts t' at the root assuming toplevel labels of t' are not present.
- Thus, D_e checks that one of the toplevel labels of t' is present (cardinality ≥ 1).
- $e \star B$: Add a new initial state s_0'. Label s_0' just like s_0 (initial state of B but "as if t' is present". E.g. if t' has an a label and s has an expression matching a then replace count variable x by $x + 1$. ($\rightsquigarrow$ example for the need for arithmetic constraints).

Edit languages for information trees

- If A is a sheaves automaton can define an edit language A' with $|A'| = L(A)$ and $\partial A' = \{e \mid e : A \to A\}$.
- Can check (using WP) that $e \in \partial A'$.
- Lends itself naturally to a typed generalisation: Types = sheaves automata (finite subset thereof), $\partial(A, B) = \{e \mid e : A \to B\}$.
- Cf. "sum conundrum".

Conclusion and next steps

- First steps towards editing and synchronising unordered trees defined by tree automata.
- Integrates smoothly with existing edit lenses framework and combinators.
- Typed edit lenses can be seen as synthesis with Diskin et al sd/delta lenses.
- WP-calculus for basic edits and sheaves automata.

Possible projects

- Further investigate categorical structure of lenses and edit lenses
- Explore equations and optimizations, e.g., “deforestation”
- Further develop lenses based on information trees
- Study connections with logic. Can we transport formulas across a lens?
- Make connections to recent work from the databases community, e.g. by R. Rodriguez