BiFluX: A Bidirectional Functional Update Language for XML

Hugo Pacheco
Joint work with Tao Zan and Zhenjiang Hu

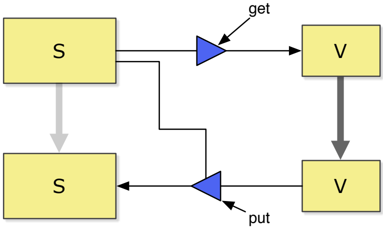
National Institute of Informatics, Tokyo, Japan

BiG Camp

Karuizawa — September 3rd, 2013

BXs and Lenses

- lenses are one of the most popular BX frameworks



Framework

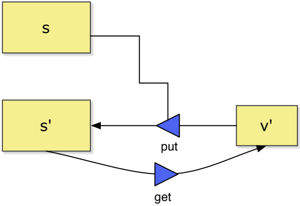
$$\textbf{data}\ s \Rightarrow v = \mathit{Lens}\ \{\mathit{get} :: s \rightarrow v$$
$$,\ \mathit{put} :: s \rightarrow v \rightarrow s\}$$

(Partial) Lens laws

- PUTGET law

  put must translate view updates exactly.

  get defined for updated sources.

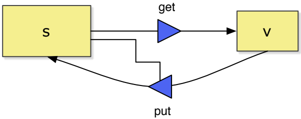


$$s' \in put\ s\ v' \Rightarrow v' = get\ s'$$

- GETPUT law

  put must preserve empty view updates.

  put defined for empty view updates.

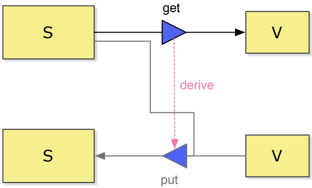


$$v \in get\ s \Rightarrow s = put\ s\ v$$

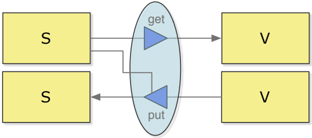
Get-based lens programming

- BX applications vary on the bidirectionalization approach
- write a single program that denotes both transformations

- bidirectionalization: write get in a familiar (unidirectional) programming language and derive a suitable put through particular techniques

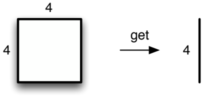


- bidirectional programming languages: programs can be interpreted both as a *get* function and a *put* function

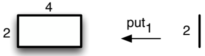


Get-based lens programming

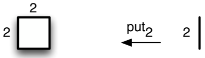
- common trait: write *get* and derive *put* automatically
- easier to maintain
- inherent ambiguity problem: many *put*s for a *get*; which one to choose?
    - *get* the *height* of a box with width and height



    - shall put_{height} preserve the width? (rectangle)



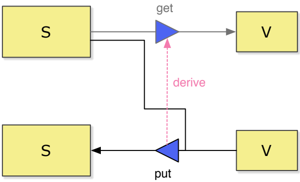
    - shall put_{height} update the width? (square)



- current solutions: only one *put* assumption

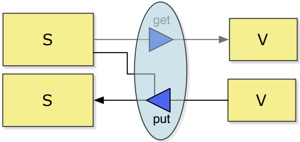
Put-based lens programming

- new alternative approach: write *put* and derive *get*
- only one *get* per *put*: $get\ s = v \Leftrightarrow s = put\ s\ v$
- *put* fully describes a BX



- Constraint solving

  S. Fischer, Z. Hu and H. Pacheco
  "Putback" is the Essence of Bidirectional Programming
  GRACE-TR 2012-08, GRACE Center, National Institute of Informatics, December 2012.

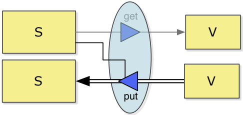


- Put programming language

  H. Pacheco, Z. Hu and S. Fischer
  Combinators for "Putback" Style Bidirectional Programming
  Technical report, July 2013, Submitted.

Putlenses (put programming language)

- normally, users write a $get : S \rightarrow V$ transformation
- but writing a $put : S \rightarrow V \rightarrow S$ update strategy is evidently harder
- putlenses: language of injective $put\ s : V \rightarrow S$ transformations, for any source s



Framework

$$\textbf{data}\ s \Leftarrow v = Putlens\ \{ put :: s \rightarrow v \rightarrow s \\ , get :: s \rightarrow v \}$$

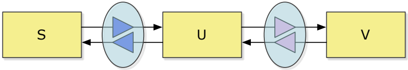
Putlenses language (Overview)

Language of point-free putlens combinators over ADTs

$$
\begin{array}{lcll}
\mathit{Put} & ::= & \mathsf{id} \mid \mathit{Put} \circ\!\!< \mathit{Put} & \text{-- basic combinators} \\
 & \mid & \Phi\ p \mid \mathsf{bot}\ p & \text{-- partial combinators} \\
 & \mid & \mathsf{effect}\ f\ \ \mathit{Put} & \text{-- monadic effects} \\
 & \mid & \mathit{Prod} \mid \mathit{Sum} \mid \mathit{Cond} \mid \mathit{Iso} \mid \mathit{Rec} & \\
\mathit{Prod} & ::= & \mathsf{addfst}\ f \mid \mathsf{addsnd}\ f \mid \mathsf{keepfstOr} \mid \mathsf{keepsndOr} \mid \mathsf{copy} & \text{-- create pairs} \\
 & \mid & \mathsf{remfst}\ f \mid \mathsf{remsnd}\ f & \text{-- destroy pairs} \\
 & \mid & \mathit{Put} \otimes \mathit{Put} & \text{-- product} \\
\mathit{Sum} & ::= & \mathsf{inj}\ p \mid \mathsf{injsOr} \mid \mathsf{injl} \mid \mathsf{injr} & \text{-- create sums} \\
 & \mid & \mathit{Put} \nabla \mathit{Put} \mid \mathit{Put}\ \nabla_p\ \mathit{Put} \mid \mathit{Put} \bullet\!\!\nabla \mathit{Put} \mid \mathit{Put} \bullet\!\!\nabla \mathit{Put} & \text{-- destroy sums} \\
 & \mid & \mathsf{uninjl} \mid \mathsf{uninjr} & \text{-- destroy sums} \\
 & \mid & \mathit{Put} + \mathit{Put} & \text{-- sum} \\
\mathit{Cond} & ::= & \mathsf{ifthenelse} \mid \mathsf{ifVthenelse} \mid \mathsf{ifSthenelse} & \text{-- conditional put app.} \\
\mathit{Iso} & ::= & \mathsf{swap} \mid \mathsf{assocl} \mid \mathsf{assocr} & \text{-- rearrange pairs} \\
 & \mid & \mathsf{coswap} \mid \mathsf{coassocl} \mid \mathsf{coassocr} & \text{-- rearrange sums} \\
 & \mid & \mathsf{distl} \mid \mathsf{distr} & \text{-- distr. sums over pairs} \\
\mathit{Rec} & ::= & \mathsf{in} \mid \mathsf{out} & \text{-- algebraic data types}
\end{array}
$$

Motivation: Bidirectional programming languages

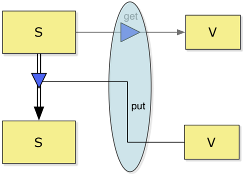
- combinatorial: build complex transformations by composing smaller ones



- require describing the concrete steps that connect source/view
- for instance, putlenses are very flexible but they are:
  - low-level (canonical set of combinators)
  - bad at updating a small part of a source while leaving the rest unchanged
- impractical for larger databases: painful to traverse the source document and explicitly ignore unrelated parts

Idea: Bidirectional update language

- Bidirectional transformation language: programmers write type-changing transformations
  - that abstract a source into a view ($get : S \to V$)
  - that refine a view into a source using the original database as oracle ($put\ s : V \to S$)
- Bidirectional update language: programmers write type-preserving updates
  - that modify a source database by embedding some view information ($put\ v : S \to S$)



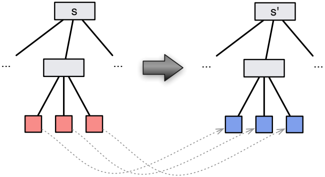
XML update languages

- XML query and transformation languages (XPath, XQuery, XSLT, XDuce) are bad for specifying small updates
- dedicated languages for in-place XML updates:
  - XQuery Update Facility [W3C]:
    - imperative language
    - ill-understood semantics semantics (aliasing, side-effects, depends on traversal order)
  - Flux (Functional Lightweight Updates for XML) [Cheney, ICFP 2008]:
    - functional language
    - clear semantics
    - straightforward type-checking
  - XUpdate, XQuery!, etc...

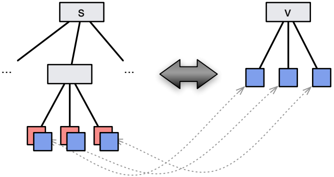
Proposal: BiFluX

- we propose BiFluX, a bidirectional variant of Flux
- modest syntactic extension
  - notion of view (feat. pattern matching, view-source alignment)
  - static restrictions to ensure well-behavedness

- Flux: fixed input schema & new output schema
- unidirectional in-place semantics



- BiFluX: fixed source and view schemas
- bidirectional semantics as putlenses



A BiFluX example (1)

Is this a *put* function?

```
UPDATE $source/books/book BY
        INSERT BEFORE title
        VALUE <author>$view</author>
WHERE title = "Through the Looking-Glass"
```

$$\begin{aligned} &S = \mathit{books}\,[\mathit{book}\,[\mathit{author}\,[\mathit{String}]+, \mathit{title}\,[\mathit{String}]]*] \\ &V = \mathit{String} \end{aligned}$$

A BiFluX example (1)

Is this a *put* function?

```
UPDATE $source/books/book BY
        INSERT BEFORE title
        VALUE <author>$view</author>
WHERE title = "Through the Looking-Glass"
```

$$S = \mathit{books}\,[\mathit{book}\,[\mathit{author}\,[\mathit{String}]+, \mathit{title}\,[\mathit{String}]]*]$$
$$V = \mathit{String}$$

- adds the view as the last author to the source authors
- violates GetPut!

A BiFluX example (2)

Is this a *put* function?

```
UPDATE $source/books/book BY
        REPLACE author[last()]
        WITH <author>$view</author>
WHERE title = "Through the Looking-Glass"
```

$$S = \mathit{books}\,[\mathit{book}\,[\mathit{author}\,[\mathit{String}]+, \mathit{title}\,[\mathit{String}]]*]$$
$$V = \mathit{String}$$

A BiFluX example (2)

Is this a *put* function?

```
UPDATE $source/books/book BY
        REPLACE author[last()]
        WITH <author>$view</author>
WHERE title = "Through the Looking-Glass"
```

$$S = \mathit{books}\,[\mathit{book}\,[\mathit{author}\,[\mathit{String}]+, \mathit{title}\,[\mathit{String}]]*]$$
$$V = \mathit{String}$$

- replaces the last author in the source with the view author
- well-behaved *put* function

Static types and lenses

- XDuce-style regular expression types [Hosoya et al., ICFP 2000, TOPLAS 2005] (with *n*-guarded recursion)

$$\tau \ ::= \ Bool \mid String \mid n[\tau] \mid () \mid \tau|\tau' \mid \tau,\tau' \mid \tau * \ \mid X$$

- Flux: values as sequences of trees

$$\gamma; x \ \vdash \ s \ \Rightarrow x'$$

- typing judgment

$$\Gamma \ \vdash \ \{\tau\}\, s\, \{\tau'\}$$

- BiFluX: strongly-typed implementation as ADTs
- bidirectional semantics

$$\gamma; \Gamma \ \vdash \ \{\tau_S\}\, s\, \{\tau_V\} \ \Rightarrow \ lens$$

- statically generated lenses

Subtyping as lenses

- Flux: type-checking with inclusion-based subtyping

$$\tau <: \tau' \;\; \mathit{iff} \;\; [\![\tau]\!] \subseteq [\![\tau']\!]$$

- we use regular expression subtyping as a “black box”
- we reuse an algorithm with additional witness functions among underlying ADT values [Lu and Sulzmann, APLAS 2004]

$$\begin{tikzcd}
\tau \arrow[r, bend left, "\mathit{ucast}"] \arrow[r, phantom, "<:"] & \tau' \arrow[l, bend left, "\mathit{dcast}"]
\end{tikzcd}$$

$\mathit{dcast}\,(\mathit{ucast}\,x) = x$

ucast total

dcast partial

- but... we implement the witness functions as putlenses

$$\tau <:_{\mathit{lens}} \tau'$$

Core language

- BiFluX → core language → lenses
- we consider two different semantics
  - default bidirectional semantics as lenses
  - Flux “standard” in-place semantics (insert, delete)
- we introduce pattern matching support (to decompose views)
- core BiFluX language:

e ::= “core XQuery expressions”

p ::= “simple XPath expressions”

pat ::= “linear, sequence-based XDuce patterns”

u ::= “Flux in-place updates”

s ::= “BiFluX lens updates”

Core language: Expressions and Paths

- like Flux, we reuse μXQ expressions (core XQuery) as a “black box” [Colazzo et al., JFP 2005]

$$
\begin{array}{lrl}
\textit{Expressions} & e ::= & () \mid e, e' \mid n[e] \mid \mathsf{let}\; x = e \;\mathsf{in}\; e' \\
 & & \mid \mathsf{if}\; e \;\mathsf{then}\; e' \;\mathsf{else}\; e'' \mid e \approx e' \\
 & & \mid \mathsf{for}\; x \in e \;\mathsf{return}\; e' \mid p \\
\textit{Paths} & p ::= & a \mid p :: t \mid p/p' \mid p[e] \mid \$x \\
 & & \mid w \mid \mathsf{true} \mid \mathsf{false} \mid \textit{snapshot pat} \;\mathsf{in}\; p \\
\textit{Axes} & a ::= & \textit{self} \mid \textit{child} \mid \textit{dos} \\
\textit{Tests} & \phi ::= & n \mid * \mid \mathsf{string} \mid \mathsf{bool}
\end{array}
$$

- Expressions: create trees, variables, value comparison, paths
- Paths: navigate a tree
- Axes: change the current focus
- Tests: examine the structure of the tree

Core language: Patterns

- pattern matching is very useful for XML transformations (XDuce, CDuce)
- not as important for typical XML updates (XQuery!, Flux)
- Flux relies on paths to navigate source documents
- but... lossy paths are not suitable for decomposing views (injectivity = union of paths?)
- BiFluX supports pattern matching to decompose views

$$\begin{array}{lcll} \mathit{pat} & ::= & \$x \mid \$x \text{ as } \tau \mid \tau & \text{-- variables, types} \\ & \mid & () \mid n[\mathit{pat}] \mid \mathit{pat}, \mathit{pat}' & \text{-- empty, label, sequence} \end{array}$$

- syntactic restriction: linear patterns (no choice – $\$x \mid ()$, no star – $(\$x)*$)

Core language: In-place updates

- in-place updates (Flux) modify specific parts of the source and leave the remaining data unchanged, producing:
  - a target tree & a target type

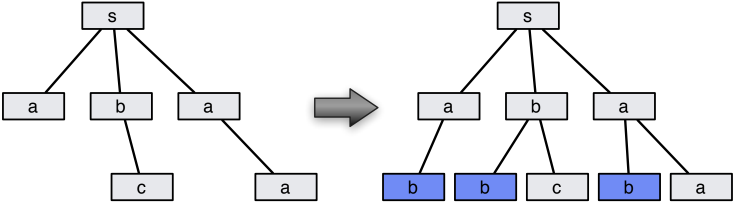
$$\begin{array}{lcl} u & ::= & \mathsf{skip} \mid u; u' \mid \mathsf{if}\, e\, \mathsf{then}\, u \mid \mathsf{let}\, pat = e\, \mathsf{in}\, u \\ & \mid & \mathsf{insert}\, e \mid \mathsf{delete} \mid d[u] \\ d & ::= & p \mid \mathsf{left} \mid \mathsf{right} \mid \mathsf{children} \mid \mathsf{iter} \end{array}$$

- Updates: combination of updates, add variables to the environment, insert expression at current position, delete current position, navigate in a direction and apply an update
- Directions: navigate the tree (path, beginning, end, child sequence, iterate over each element)

In-place update example

Insert a *b* as the first child of each child of the root node

children [iter [children [left [insert *b*]]]]



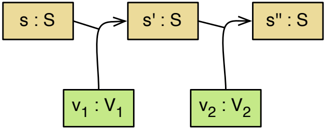
Core language: Bidirectional updates

- bidirectional updates (BiFluX) take source and view types, producing:
  - a *put* function that modifies specific parts of the source to embed all view information
  - a *get* function that computes a view of a given source

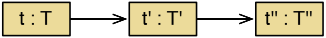
$$
\begin{array}{lcl}
s & ::= & \mathsf{fail} \mid s; s' \mid ds[s] \mid [s]dv \mid \mathsf{replace}\ e \mid \mathsf{upd}\ u \\
 & \mid & \mathsf{let}\ pat = e\ \mathsf{in}\ s \mid \mathsf{letS}\ pat = e\ \mathsf{in}\ s \mid \mathsf{letV}\ pat = e\ \mathsf{in}\ s \\
 & \mid & \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s' \mid \mathsf{ifS}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s' \mid \mathsf{ifV}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s' \\
 & \mid & \mathsf{alignpos}\ e_S\ s\ r \mid \mathsf{align}\ e_S\ p_S\ p_V\ s\ r \\
ds & ::= & p \mid \mathsf{children} \mid \mathsf{iter} \\
dv & ::= & \$x/p \\
r & ::= & \mathsf{if}\ e\ \mathsf{then}\ r\ \mathsf{else}\ r' \mid \mathsf{let}\ pat = e\ \mathsf{in}\ r \\
 & \mid & \mathsf{delete} \mid \mathsf{keepl}\ r \mid \mathsf{keepr}\ r
\end{array}
$$

Core language: Bidirectional updates (basic combinators)

- if we do not embed view information, we must *fail*
- bidirectional composition $(s; s')$ embeds different view information into the source in two steps s and s'



- not lens composition!
- formally, well-behavedness requires “source disjointness” (XPath intersection has been studied by the XML community)
- remember... it is different from in-place composition

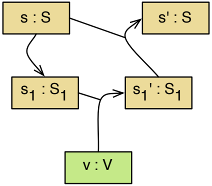


Core language: Bidirectional updates (environment)

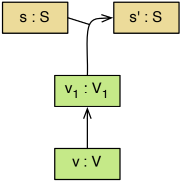
- environment contains three kinds of variable bindings:
  - source variables: lenses from the current source focus
  - view variables: sequence that constitutes the current view
  - normal variables: independent of the current source/view
- three kinds of let expressions:
  - **letS pat = e in s**: adds a new source (and environment) variable from an expression using only source variables
  - **letV pat = e in s**: adds a new view (and environment) variable from an expression using only view variables
  - **let pat = e in s**: adds a new environment variable from any expression
- three kinds of if-then-else combinators:
  - **ifS e then s else $\mathbf{s'}$**: source condition
  - **ifV e then s else $\mathbf{s'}$**: view condition
  - **if e then s else $\mathbf{s'}$**: arbitrary condition

Core language: Bidirectional updates (directions)

- source directions (**ds**[**s**]):
  - apply a lens to the current focus, yielding a new focus
  - lens composition
  - *iter* embeds the same view into each element in the current focus

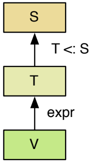


- view directions ([**s**]**dv**)
  - unfolds structure of the view
  - variable-rooted paths ($\$x/p$)
  - no relative view paths
  - only injective paths

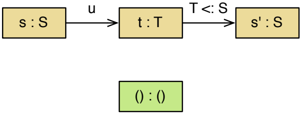


Core language: Bidirectional updates (embedding)

- replace the current source with by some expression (**replace e**):
  - evaluate the expression as lens
  - must use the whole view
  - subtyping as a lens

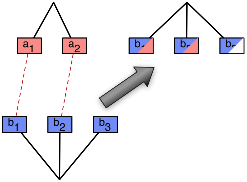


- run an in-place update as a lens (**upd u**):
  - apply an in-place update to the source
  - view must be empty
  - subtyping upcast function
  - wait a minute... is this a valid lens? GetPut? ...putlens semantics



Core language: Bidirectional updates (alignment)

- all our updates this far can only iterate over source sequences
- we introduce constructors for alignment two source and view sequences:
  - alignpos $e_S\ s\ r$: matching by position
  - align $e_S\ p_S\ p_V\ s\ r$: matching according to two paths p_S and p_V



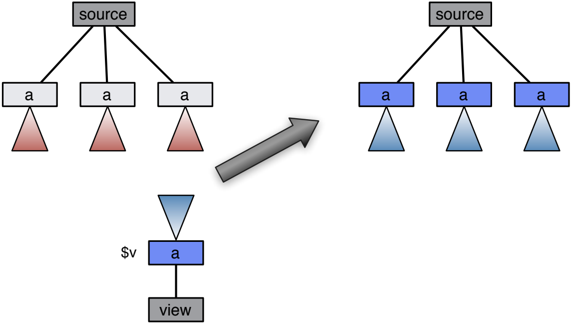
- e_S is a filtering condition on source values
- r allows to recover source elements that satisfied e_S but have no match in the view, but updating them so that $\neg e_S$

$$\begin{array}{rcl} r & ::= & \text{if } e \text{ then } r \text{ else } r' \mid \text{let } pat = e \text{ in } r \\ & \mid & \text{delete} \mid \text{keepl } r \mid \text{keepr } r \end{array}$$

Bidirectional update example

Embed the view to each children of the source

children [iter [letV $\$v = \$view$/child::a in replace $\$v$]]



- the view is put back in duplicated to the source
- the view **a** type must be a subtype of the source **a** type
- the derived *get* function tests for equality of all children

BiFluX language

- BiFluX high-level language (changes to Flux in red):

$$
\begin{array}{lcl}
\mathit{Stmt} & ::= & \mathit{Upd}\ [\textbf{WHERE}\ \mathit{Expr}] \mid \textbf{IF}\ \mathit{Expr}\ \textbf{THEN}\ \mathit{Stmt}\ \textbf{ELSE}\ \mathit{Stmt} \\
 & \mid & \mid \mathit{Stmt}\ ;\ \mathit{Stmt} \mid \{\ \mathit{Stmt}\ \} \mid \textbf{LET}\ \mathit{Pat} = \mathit{Expr}\ \textbf{IN}\ \mathit{Stmt} \\
 & \mid & \textbf{CASE}\ \mathit{Expr}\ \textbf{OF}\ \{\ \mathit{Cases}\ \} \\
\mathit{Upd} & ::= & \textbf{INSERT}\ (\textbf{BEFORE} \mid \textbf{AFTER})\ \mathit{Path}\ \textbf{VALUE}\ \mathit{Expr} \\
 & \mid & \textbf{INSERT AS}\ (\textbf{FIRST} \mid \textbf{LAST})\ \textbf{INTO}\ \mathit{Path}\ \textbf{VALUE}\ \mathit{Expr} \\
 & \mid & \textbf{DELETE}\ [\textbf{FROM}]\ \mathit{Path} \mid \textbf{REPLACE}\ [\textbf{IN}]\ \mathit{Path}\ \textbf{WITH}\ \mathit{Expr} \\
 & \mid & \textbf{UPDATE}\ \mathit{Path}\ \textbf{BY}\ \mathit{Stmt} \\
 & \mid & \textbf{UPDATE}\ \mathit{Path}\ \textbf{BY}\ \mathit{VStmt}\ \textbf{FOR VIEW}\ \mathit{Path}\ [\textbf{Match}] \\
 & \mid & \textbf{KEEP}\ \mathit{Path}\ \textbf{AS}\ (\textbf{FIRST} \mid \mathrm{LAST}) \mid \textbf{CREATE VALUE}\ \mathit{Expr} \\
\mathit{Cases} & ::= & \mathit{Pat} \rightarrow \mathit{Stmt} \mid \mathit{Cases}\ '|'\ \mathit{Cases} \\
\mathit{VStmt} & ::= & \mathit{VUpd}\ '|'\ \mathit{VUpd} \mid \mathit{VUpd} \\
\mathit{VUpd} & ::= & \textbf{MATCH} \rightarrow \mathit{Stmt} \\
 & \mid & \textbf{UNMATCHS} \rightarrow \mathit{Stmt} \\
 & \mid & \textbf{UNMATCHV} \rightarrow \mathit{Stmt} \\
\mathit{Match} & ::= & \textbf{MATCHING BY}\ \mathit{Path} \\
 & \mid & \textbf{MATCHING SOURCE BY}\ \mathit{Path}\ \textbf{VIEW BY}\ \mathit{Path} \\
\mathit{Path} & ::= & \ldots \\
\mathit{Pat} & ::= & \ldots \\
\mathit{Expr} & ::= & \ldots
\end{array}
$$

Conclusions

- reviewed concepts on bidirectional transformation languages
- introduced a novel idea of bidirectional update language
- presented the BiFluX bidirectional XML update language
- unveiled the details of a in-place/bidirectional core language

- BiFluX is work in progress
- our current prototype already supports typical BX examples (not shown in this presentation)

Future work

- finish the implementation of the prototype (with examples)
- at the moment no type inference for patterns and no path intersection (not crucial... we could reuse existing algorithms)
- provide more static guarantees (totality, etc)
- optimization of underlying putlenses