ISSN 1884-0760

GRACE TECHNICAL REPORTS

BiFluX: A Bidirectional Functional Update Language for XML

Hugo Pacheco Tao Zan Zhenjiang Hu

GRACE-TR 2014-02 August 2014

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

BiFluX: A Bidirectional Functional Update Language for XML

Hugo PACHECO
Cornell University, USA
hpacheco@cs.cornell.edu

Tao ZAN
The Graduate University for Advanced Studies, Japan
zantao@nii.ac.jp

Zhenjiang HU
National Institute of Informatics, Japan
hu@nii.ac.jp

August 7th, 2014

Abstract

Different XML formats are widely used for data exchange and processing, being often necessary to mutually convert between them. Standard XML transformation languages, like XSLT or XQuery, are unsatisfactory for this purpose since they require writing a separate transformation for each direction. Existing *bidirectional transformation* languages mean to cover this gap, by allowing programmers to write a single program that denotes both transformations. However, they often 1) induce a more cumbersome programming style than their traditionally unidirectional relatives, to establish the link between source and target formats, and 2) offer limited configurability, by making implicit assumptions about *how* modifications to both formats should be translated that may not be easy to predict.

This paper proposes a bidirectional XML update language called BiFluX (BIdirectional FunctionaL Updates for XML), inspired by the Flux XML update language. Our language adopts a novel *bidirectional programming by update* paradigm, where a program succinctly and precisely describes *how* to update a source document with a target document, in an intuitive way, such that there is a unique "inverse" source query for each update program. BiFluX extends Flux with bidirectional actions that describe the connection between source and target formats. We introduce a core BiFluX language, with a clear and well-behaved bidirectional semantics and a decidable static type system based on regular expression types.

1 Introduction

Nowadays, various XML formats are widely used for data exchange and processing. Since data evolves naturally over time and is often replicated among different applications, it becomes frequently necessary to mutually convert between such formats. However, traditional XML transformation languages, like the XSLT and XQuery standards of the World Wide Web Consortium (W3C), are unsatisfactory for this purpose as they require writing a separate transformation for each direction.

Bidirectional transformation (BX) languages [11] mean to cover this gap, by allowing users to write a single program that can be executed both forwards and backwards, so that consistency between two formats can be maintained for free. A variety of bidirectional languages have emerged over the last 10 years to support bidirectional applications in the most diverse computer science disciplines [11], including functional programming, software engineering and databases. These languages come in different flavors, including many focused on the transformation of tree-structured data with a particular application to XML documents [21, 7, 22, 12, 26, 13, 19], and can be classified into three main paradigms. The first *relational* paradigm [21, 7] prescribes writing a declarative (non-deterministic) consistency relation between two formats, from which a suitable BX is automatically derived. The second *bidirectionalization* paradigm [22, 12, 25] asks users to write a transformation in a traditional unidirectional language, that plays the role of a functional consistency relation. The last *combinatorial* paradigm [26, 13, 19] encompasses the design of a domain-specific bidirectional language in which each combinator denotes a well-behaved BX, allowing users to write correct-by-construction programs by composition.

As most interesting examples of BXs are not bijective, there may be multiple ways to synchronize two documents into a consistent state, introducing ambiguity. Despite of this fact, bidirectional languages are typically designed to satisfy fundamental consistency principles, and support only a fixed set of synchronization strategies (out of a myriad possible) to translate a (non-deterministic) bidirectional specification —the syntactic description of a BX— into an executable BX procedure. This latent ambiguity often leads to unpredictable behavior, as users have limited power to configure and understand what a BX does from its specification. Even for combinatorial languages, that have the theoretical potential to fully specify the behavior of a BX [14, 28], their lower-level programming style requires significant effort and expertise from users to write intricate BXs via the composition of simple, concrete steps; they also scale badly for large formats, since one must explicitly describe how a BX transforms whole documents, including unrelated parts.

Intuitively, the goal of a BX is to translate updates on a target model into updates on a source model (and vice-versa) so that the updated models are kept consistent. As SQL stands for relational databases, a few high-level XML update languages [30, 15, 8] facilitate common modification operations over XML documents. Contrarily to XML transformation languages, XML update languages are well-suited for specifying small *in-place* changes to in a concise way, leaving all the remaining parts of a document unchanged.

In this paper, we propose a novel *bidirectional programming by update* paradigm, in which the programmer writes an update program that describes how to update a source model to embed information from a target model, and the system derives a query from source to target that evinces the consistency between both models. Such a *bidirectional update* allows to express the relationship between source and target models in a simple way —as in the relational paradigm— by saying *which* related source parts are to be updated, but combined with additional actions that supply the missing pieces to tame the ambiguity in *how* target modifications are reflected —as in the combinatorial paradigm. For a wide class of BXs usually known as *lenses* [14], that have a data flow from *source* to *view*, this paradigm opens a new axis in the BX design space that enjoys a unique tradeoff between the declarative style of relational approaches and the stepwise style of combinatorial approaches. This paper demonstrates that a new family of *bidirectional update languages*, featuring an hybrid programming style, can render bidirectional programming more user friendly.

From a linguistic perspective, the main contribution of this paper is conceptual: we propose the idea of extending an update language with bidirectional features to write, directly and at a nice level of abstraction, a view update translation strategy which bundles all the necessary pieces to build a BX. Concretely, we design BiFluX, a type-safe, declarative and expressive language for the bidirectional updating of XML documents that is deeply inspired by Flux [8], a simple and well-designed functional XML update language. We lift unidirectional Flux updates to bidirectional BiFluX updates by imbuing them with an additional notion of view. Reading updates as BXs will motivate a few language extensions to original Flux, and require a suitable bidirectional semantics and extra static conditions on BiFluX programs to ensure that they build well-behaved BXs.

We demonstrate the usefulness of BiFluX by illustrating typical examples of BXs written as bidirectional update programs. These help clarifying the stylistic differences to traditional bidirectional programming approaches, and substantiate that bidirectional update languages can combine a declarative language notation with a flexible and clear semantics. BiFluX has been fully implemented and tested with many examples including those in this paper.

The rest of the paper is organized as follows. After briefly explaining the novel features of BiFluX in Section 2, we show typical examples of BiFluX programs in Section 3. Section 4 presents the core BiFluX language that can be used to interpret high-level BiFluX programs either as unidirectional or bidirectional updates, and Section 5 discusses the static typing and semantics of core BiFluX. Section 6 formalizes the translation from high-level to core BiFluX, Section 7 compares our approach with related work on bidirectional and XML programming, and Section 8 concludes with a synthesis of the main ideas and directions for future work.

2 A Bidirectional Update Language

Before giving concrete examples in Section 3, we start with a brief explanation of the features of BiFluX and its informal semantics to show the big picture of our general framework.

2.1 BiFluX syntax

We define the high-level syntax of BiFluX in Figure 1 as a modest syntactic extension to Flux; the new features that are the focus of this paper are highlighted in green. Flux [8] is a high-level, purely functional language for writing XML updates, with a clear semantics and syntactic typechecking. Similarly, we typecheck BiFluX programs by translating them into a canonical core language with a clear bidirectional semantics. Our examples assume an informal familiarity with commonplace XML technologies like XQuery expressions, XPath paths and XDuce-style regular expression types.

To have a taste of BiFluX, imagine that we want to update the last author of a particular book with title 'Querying XML' in a database of books with type

$$books[book[title[\mathtt{string}], author[\mathtt{string}]^{+}]^{*}]$$

using a view of type $author[\mathtt{string}]$. We can accomplish this by writing an update (with source $\$source$ and view $\$view$):[1]

```
UPDATE $source/books/book BY {
  REPLACE author[last()] WITH $view
} WHERE SOURCE title = "Querying XML"
```

Flux-like syntax At first glance, bidirectional BiFluX programs look just like regular Flux programs. We omit the syntactic definitions of expressions $Expr$, paths $Path$, and patterns Pat. Variables Var are written $\$x, \$y$, etc. Statements $Stmt$ include conditionals, composition, let-binding, case expressions or updates, which may be guarded by a `WHERE` clause that defines a set of conditions. Statements can be empty {} or parenthesized using braces $\{Stmt\}$. As in Flux, *in-place* updates Upd can be *singular*, to update

[1] The position-dependent $last()$ XPath function is actually not supported in our path expressions, and is desugared in BiFluX using pattern matching.

single trees, or *plural*, to update the children of each selected tree. Single insertions (INSERT BEFORE/AFTER) insert a value before or after each node selected by a path, while plural insertions (INSERT AS FIRST/LAST INTO) insert a value at the first or last position of the child-list of each selected node. Singular deletions (DELETE) delete each selected node, while plural deletions (DELETE FROM) delete their content. Single replacements (REPLACE WITH) replace each node selected by a path, while plural replacements (REPLACE IN) replace their content. Single updates UPDATE BY apply a statement to each tree in the result of a path.

Source and view matching The main difference in BiFluX is that updates on sources carry an additional notion of view, what becomes syntactically evident with a new *non-in-place* UPDATE FOR VIEW operation that synchronizes a source sequence with a view sequence. Such synchronization can be configured by the programmer via a matching condition that aligns source and view nodes, and a triple of matching/unmatching clauses that describe the actions for individual source-view nodes. When two source and view nodes MATCH, a bidirectional statement is executed to update the source using the view; an unmatched view node (UNMATCHV) creates a new node in the source, either as a default or according to a unidirectional CREATE statement that provides a fresh source node to be normally updated with the existing view node; an unmatched source node (UNMATCHS) is DELETEd by default, but we may KEEP it by providing a unidirectional statement describing how to invalidate the given WHERE SOURCE selection criteria. It is worth noting that while UPDATE FOR VIEW statements are intrinsically bidirectional, the same BiFluX syntax (e.g., DELETE) may be overloaded and denote either a bidirectional or a unidirectional update depending on the context. The rule is that all BiFluX statements are bidirectional, except inside UNMATCHS or UNMATCHV clauses. An example that puts all these features to use is illustrated later in Figure 3.

Pattern matching Another significant difference to Flux is the support for pattern matching. This is a very useful feature of XML transformation languages like XDuce [17] or CDuce [3], that allow matching tree patterns against the input data to transform it into an output of different shape. Typical XML update languages like XQuery! [15] or Flux [8] do not support pattern matching, since it is not essential and may be more difficult to optimize, and they use solely paths to navigate to the portions of the input documents that are to be updated in-place. In BiFluX, pattern matching can be used to guide the update based on the structure of the data (via LET and CASE statements), and is mostly useful for non-in-place updates that match source and view formats of different shapes.

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

Figure 1: Concrete syntax of BiFluX updates.

Source/view/normal expressions Our language considers three kinds of source, view or normal variables. Expressions in `IF`, `LET`, `CASE` or `WHERE` clauses support additional tags to disambiguate if they refer to only the `SOURCE`, to only the `VIEW`, or to the global environment of an update. These tags can be ignored at first glance and will in fact be omitted in our examples, as they are inferable from context information for each update. Update procedures are omitted but can be easily added to the language.

2.2 Informal semantics and general framework

In general, a BiFluX update is executed for a particular source and view as follows: a source path is evaluated over the current source, yielding a *source focus selection*, to be recursively updated using a *view focus selection* computed by evaluating a view path over the current view, until all the view information is embedded into the source. View and source focus selections denote the mutable parts of the source and view trees that can be updated and used by the update, and subexpressions of the update may restrict the focuses.

Despite the emphasis is on writing updates, BiFluX programs have a bidirectional interpretation. They can be read as 1) an *update function*

$U(s, v') = s'$ that updates a source s into a new source s' which contains a given view v', or 2) a *query function* $Q(s) = v$ that computes a view v from a given source s; these functions may be partial[2]. For the example at the beginning of this section, (assuming that books are uniquely identified by their titles) the corresponding query function is semantically equivalent to the XPath expression that returns the last author of the respective source book:

```
$source/books/book[title="Querying XML"]/author[last()]
```

Our language is carefully designed to ensure that the inferred relationship between sources and views is deterministic, so that capturing it by a query function is appropriate. In other words, there exists a unique query function for each update program written in our language. Moreover, its bidirectional semantics satisfies two basilar synchronization properties; that an update U consistently embeds view information to the source, without view side-effects:

$$U(s, v') = s' \;\Rightarrow\; Q(s') = v' \qquad \textsc{UpdateQuery}$$

and that it does not update already consistent sources:

$$Q(s) = v \;\Rightarrow\; U(s, v) = s \qquad \textsc{QueryUpdate}$$

These two properties are commonly known as the well-behavedness laws of lenses in the bidirectional programming community [11].

For an example of a Flux update that is not (syntactically) valid as a BiFluX update, imagine that we had written instead:

```
UPDATE $source/books/book BY {
  INSERT AS LAST INTO author VALUE $view
} WHERE SOURCE title = "Querying XML"
```

This update function is not idempotent on sources, since it always inserts the view as an extra author of the source book, violating QueryUpdate: even when the view is already an author of the source book, a new duplicated author is inserted. Such class of valid BiFluX updates can be statically checked, namely as the programs for which update normalization and typechecking succeed.

The general architecture of our bidirectional updating framework is illustrated in Figure 2. A BiFluX program is evaluated in two stages. First, it is statically compiled against a source and a view schema (represented as DTDs), producing a bidirectional executable. The generated executable can then be evaluated bidirectionally for particular XML documents conforming to the DTDs: in forward mode as a query Q, or in backward mode as an update U.

[2]We represent the undefined value as $\perp$, failure of a partial function f as $f\,v = \perp$ and partial inclusion ($v \sqsubseteq v'$) as $v \neq \perp \Rightarrow v = v'$.

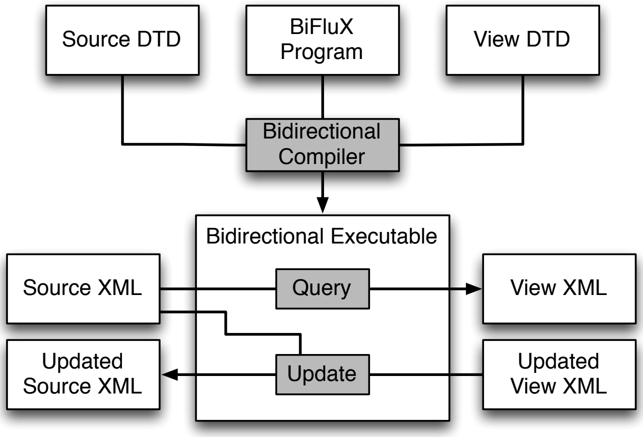

Figure 2: Architecture of the BiFluX framework.

```
PROCEDURE niibook(SOURCE $source AS s:addrbook, VIEW $view AS v:niibook) =
UPDATE $source/addrbook/person BY {
  MATCH -> REPLACE email[ends-with(text(),'nii.ac.jp')][1] WITH $email'
| UNMATCHV -> CREATE VALUE <person> <name/> <tel>+81-3-4212-2000</tel> </person>
| UNMATCHS -> KEEP . ; DELETE email[ends-with(text(),'nii.ac.jp')]
} FOR VIEW employee[$name AS v:name, $email' AS v:email] IN $view/niibook/employee
MATCHING BY name WHERE email[ends-with(text(),'nii.ac.jp')]
```

Figure 3: BiFluX update for the institutional address book example.

3 Examples

This section illustrates how writing a bidirectional update feels like to a programmer, through a series of BX examples using BiFluX:

Example 3.1 demonstrates how the new bidirectional notation, in combination with ordinary unidirectional Flux updates, can intuitively describe a bidirectional update;

Example 3.2 illustrates nested bidirectional updates and flexible non-local matching behavior;

Example 3.3 showcases pattern matching and recursive procedures.

3.1 Institutional address book example

Consider a typical address book represented by the DTD from Figure 4. An address book contains a list of persons, each possessing a name, a list of emails and an optional telephone number. Let us start with the following XML address book with three persons:

```
<!DOCTYPE addrbook [
<!ELEMENT addrbook(person*)>
<!ELEMENT person(name,email*,tel?)>
<!ELEMENT name(#PCDATA)>
<!ELEMENT email(#PCDATA)>
<!ELEMENT tel(#PCDATA)> ]>
```

Figure 4: A simple address book DTD.

```
<addrbook><person><name>Hugo Pacheco</name>
    <email>hpacheco@nii.ac.jp</email>
    <email>hpacheco@gmail.com</email></person>
  <person><name>John Doe</name>
    <email>doe@domain.com</email></person>
  <person><name>Zhenjiang Hu</name><email>zh@nii.ac.jp
    </email><tel>+81-3-4212-2530</tel></person>
</addrbook>
```

On the other hand, the NII's administrative services may keep only a view with the name and institutional address of employees (people with an email at "nii.ac.jp"), as shown in the DTD from Figure 5.

We can easily write a bidirectional update in BiFluX to synchronize these two formats, as illustrated in Figure 3[3]. The root procedure *niibook* takes as arguments the root source and view variables. It focuses on a sequence of source persons by traversing down the path $\$source/addrbook/person$, selecting only those that have NII emails, and focuses on a sequence of view employees by traversing up the (injective) path $\$view/niibook/employee$. Elements in the two sequences are matched by their names. For matching person-employee pairs, the person's first NII email is updated with the employee's email. If a new unmatched employee exists in the view, a new person with a default telephone is created in the source (inheriting its view name and email). If an old unmatched person exists in the source, all its NII emails are deleted.

The unique query for this example simply keeps the first institutional email of each person working at the NII:

```
<niibook><employee><name>Hugo Pacheco</name>
    <email>hpacheco@nii.ac.jp</email></employee>
 <employee><name>Zhenjiang Hu</name>
  <email>zh@nii.ac.jp</email></employee>
</niibook>
```

The update function is more interesting. For instance, if we add Tao (in alphabetical order) as a new NII employee, fix Zhenjiang's email and delete Hugo, we get an updated source where John is left unchanged, Tao is created

[3]The names `s:elem` and `v:elem` are BiFluX type variables that refer to the types of source and view `elem`ents declared in the respective DTDs.

```
<!DOCTYPE niibook [ <!ELEMENT niibook (employee*)>
<!ELEMENT employee (name,email)>
<!ELEMENT name (#PCDATA)> <!ELEMENT email (#PCDATA)> ]>
```

Figure 5: A NII address book DTD.

```
PROCEDURE socialbook(SOURCE $source AS s:addrbook, VIEW $view v:socialbook)
= UPDATE $source/addrbook/group BY { MATCH ->
  UPDATE $person IN $persons BY { MATCH -> {}
  | UNMATCHV -> LET $old = $source/addrbook/group/person IN
      LET $oldperson = $old[name/text() = $person'/name/text()][1] IN
      IF $oldperson THEN CREATE VALUE $oldperson ELSE {}
  } FOR VIEW $person' IN $persons' MATCHING BY name
} FOR VIEW group[$name AS v:name, $persons AS v:person*]
  IN $view/socialbook/group MATCHING BY name
```

Figure 6: BiFluX update for the social address book example.

with a default telephone number (as his name does not match any name in the original source), Zhenjiang's NII email is updated and the NII email of Hugo is deleted:

```
<addrbook><person><name>Hugo Pacheco</name>
     <email>hpacheco@gmail.com</email></person>
  <person><name>John Doe</name>
    <email>doe@domain.com</email></person>
  <person><name>Tao Zan</name><email>zantao@nii.ac.jp
    </email><tel>+81-3-4212-2000</tel></person>
  <person><name>Zhenjiang Hu</name><email>hu@nii.ac.jp
    </email><tel>+81-3-4212-2530</tel></person>
</addrbook>
```

Note that if we queried the updated address book again, we would get a view with the names and NII emails of only Tao and Zhenjiang.

As a BX side note, this precise behavior can not be achieved using the existing typical declarative BX languages, which are not designed with fine user control in mind; alignment-aware combinatorial BX languages could be tailored to produce similar behavior, but getting it right requires much higher effort and expertise.

3.2 Social address book example

In response to the Web 2.0 movement, consider that our address book now supports groups of people according to their social relationships. The new DTD corresponds to the following regular expression type:

```
<!DOCTYPE html[<!ELEMENT html(head,body)>
<!ELEMENT head (#PCDATA)>
<!ELEMENT body (h1,dl)>
<!ELEMENT h1 (#PCDATA)>
<!ELEMENT dl ((dt|dd)*)>
<!ELEMENT dt (a)>
<!ELEMENT a (href,#PCDATA)>
<!ELEMENT dd (h3,dl)>
<!ELEMENT h3 (#PCDATA)> ]>
```

Figure 7: A Netscape bookmark format DTD.

$$\mathit{addrbook}[\mathit{group}[\mathit{name}[\mathtt{string}], \mathit{person}[\ldots]^*]^*]$$

Consider our address book with people now classified into groups:

```
<addrbook><group><name>coworkers</name>
    <person><name>Hugo Pacheco</name>...</person>
    <person><name>Zhenjiang Hu</name>...</person></group>
  <group><name>friends</name><person>
    <name>John Doe</name>...</person></group>
</addrbook>
```

This time, a social media application may only be interested in the groups and names of people, according to a view schema:

$$\mathit{socialbook}[\mathit{group}[\mathit{name}[\mathtt{string}], \mathit{person}[\mathit{name}[\mathtt{string}]]^*]^*]$$

A bidirectional update that synchronizes address and social books is written in Figure 6. It starts by matching groups, proceeding recursively for persons within groups (with default behavior for unmatched groups). Inside, for source-view persons matching on their name, no update is necessary. For each unmatched view person, we attempt to retrieve its address book information from any other group in the original source, or otherwise create a default person.

The corresponding query function produces a view with the same structure but showing only names of groups and persons. For the update function, imagine that we modify the view by reordering the two groups, changing Hugo's group and creating a new group for family members:

```
<addrbook><group><name>friends</name>
    <person><name>Hugo Pacheco</name></person></group>
  <group><name>coworkers</name><person>
    <name>Zhenjiang Hu</name></person></group>
  <group><name>family</name></group>
</addrbook>
```

The correspondingly updated source is as follows:

```
PROCEDURE top(SOURCE $html AS s:html,VIEW $xbel AS v:xbel) =
UPDATE html[head[String], body[$h1 AS s:h1, dl[$nc AS (s:dt|s:dd)*]]] IN $html BY
 { REPLACE IN $h1 WITH $t ; contents($nc,$xc) }
FOR VIEW xbel[title[$t AS String], $xc AS (v:bookmark|v:folder)*] IN $xbel

PROCEDURE contents(SOURCE $nc AS (s:dt|s:dd)*,VIEW $xc AS (v:bookmark|v:folder)*) =
UPDATE $nc BY { CASE $v OF {
   bookmark[href[$url AS String], title[$title AS String]]
-> REPLACE . WITH <dt><a><href>{$url}</href>{$title}</a></dt>
 | folder[title[$title AS String], $fxc AS (v:bookmark|v:folder)*]
-> REPLACE IN h3 WITH $title ; contents(dl/*,$fxc)
 } } FOR VIEW $v IN $xc
```

Figure 8: BiFluX update for the bookmark example.

```
<addrbook><group><name>friends</name>
    <person><name>Hugo Pacheco</name>...</person>
    <person><name>John Doe</name>...</person></group>
  <group><name>coworkers</name><person>
    <name>Zhenjiang Hu</name>...</person></group>
  <group><name>family</name></group>
</addrbook>
```

Since Hugo changed from group coworkers to friends, he is considered an unmatched view person under his new group. Our bidirectional update avoids his original address details to be lost, by looking them up in all groups, instead of only in Hugo's original group (what would be the default behavior). An analogous example motivates an extension to the alignment-aware language of [1].

3.3 Bookmark example

For a different bidirectional updating example, consider the conversion between two popular browser bookmark formats studied in [21]. Netscape stores its bookmarks in an HTML format (Figure 7), while the XBEL open XML bookmark exchange format opts for a loosely equivalent representation (Figure 9). Both formats contain a general title (`h1` or `title`) and a sequence of bookmarks (`dt` or `bookmark`) or folders (`dd` or `folder`), where folders may recursively contain sequences of bookmarks or folders.

The original biXid transformation [21] relies on pattern matching to decompose the source and target formats and can be replicated in BiFluX as shown in Figure 8. The *top* procedure decomposes the source into `head` and `body` (with a title $\$h1$ and a sequence $\$nc$ of `dt`s or `dd`s) and the view into a title $\$t$ and a sequence $\$xc$ of `bookmark`s or `folder`s. Then *top* replaces the source $\$h1$ with $\$t$ and invokes *contents* to update the remaining sequences. The *contents* procedure makes use of a case expression to match

```
<!DOCTYPE xbel[<!ELEMENT xbel(title,(bookmark|folder)*)>
<!ELEMENT title(#PCDATA)><!ELEMENT bookmark(href,title)>
<!ELEMENT folder(title,(bookmark|folder)*)> ]>
```

Figure 9: A XBEL bookmark format DTD.

source and view bookmarks and folders: for a view `bookmark`, we generate a source `dt` element with the bookmark's `href` and `title`; for a view `folder`, we generate a source `dd` element with the folder's `title` and a `dl` with recursively computed contents.

Finally, we can run our BiFluX program as a query that converts from Netscape to XBEL, or as an update that converts from XBEL to Netscape. This is not the best showcase example of BiFluX, since the source and view bookmarks are almost in bijective correspondence and there is small ambiguity to mitigate in the update. Nonetheless, note that our BiFluX program will preserve the original Netscape header, while the analogous biXid program would simply generate default data for unrelated parts.

4 Core Language

The high-level language presented in the previous sections follows a verbose and natural syntax that is convenient for users, but its operations are overlapping, complex and hard to typecheck. Following the design of Flux and as standard for many other languages, we introduce a core update language of canonical operations whose semantics and typing rules are easier to define and manipulate.

4.1 XML values and regular expression types

As several other XML processing languages [17, 8, 9], we consider a type system of regular expression types with structural subtyping[4]:

$$\begin{array}{ll} \text{Atomic types} & \alpha ::= \texttt{bool} \parallel \texttt{string} \parallel n[\tau] \\ \text{Sequence types} & \tau ::= \alpha \parallel () \parallel \tau \mid \tau' \parallel \tau, \tau' \parallel \tau^* \parallel X \end{array}$$

Atomic types $\alpha \in$ *Atom* are primitive booleans, strings or labeled sequences $n[\tau]$. *Sequence types* $\tau \in$ *Type* are defined using regular expressions, including empty sequence $()$, alternative choice $\tau \mid \tau'$, sequential composition τ, τ', iteration τ^* or type variables X; choice and composition are right-nested. We define the usual $\tau^+ = \tau, \tau^*$ and $\tau^? = \tau \mid ()$. Types can also be recursively defined:

[4] We use $\parallel$ for syntax alternatives in the type grammar to prevent confusion.

Type definitions $\tau_D ::= \alpha \parallel () \parallel \tau_D \mid \tau'_D \parallel \tau_D, \tau'_D \parallel {\tau_D}^*$
Type signatures $E ::= \cdot \parallel E, \texttt{type}\ X = \tau_D$

Type definitions τ_D are sequences with no top-level variables (to avoid non-label-guarded recursion [9]). A *type signature* E is a set of named type definitions of the form $X = \tau_D$, and is well-formed if no two types have the same name and all type variables in definitions are declared in E. We write $E(X)$ for the type bound to X in E. Hereafter, we will assume the signature E to be fixed.

In traditional XML-centric approaches [17, 9], values are encoded using a uniform representation that does not record the structure that types impose on values. This "flat" representation is economical and simplifies subtyping, but makes it harder to realize that a value belongs to a type and therefore to integrate regular expression features into functional languages with non-structural type equivalence, such as Haskell or ML. In this paper, we instead consider a structured representation of values (in line with values of algebraic data types) that keep explicit annotations which, in a way, witness how to parse a flat value as an instance of a type [23]:

Atomic values $t ::= \texttt{true} \mid \texttt{false} \mid w \mid n[v]$
Forest values $v ::= t \mid () \mid L\ v \mid R\ v \mid (v, v) \mid [\,v_0, \ldots, v_n\,]$

Atomic values $t \in \mathit{Tree}$ can be $\texttt{true}, \texttt{false} \in \mathit{Bool}$, strings $w \in \Sigma^*$ (for some alphabet Σ), or singleton trees $n[v]$ with a node label n. *Forest values* $v \in \mathit{Val}$ include the empty sequence (), left- $L\ v$ or right- $R\ v$ tagged choices, binary sequences (v, v) and lists of arbitrary length $[\,v_0, \ldots, v_n\,]$. The semantics of a type τ denotes a set of values $[\![\tau]\!]$ that is defined as the minimal solution (formally the least fixed point [17]) of the following set of equations:

$$[\![\texttt{bool}]\!] \triangleq \{\texttt{true}, \texttt{false}\} \qquad [\![n[\tau]]\!] \triangleq \{\, n[v] \mid v \in [\![\tau]\!] \,\}$$
$$[\![\texttt{string}]\!] \triangleq \Sigma^* \qquad [\![X]\!] \triangleq [\![E\ (X)]\!]$$
$$[\![\tau, \tau']\!] \triangleq \{\, (v, v') \mid v \in [\![\tau]\!], v' \in [\![\tau']\!] \,\} \qquad [\![()]\!] \triangleq \{\, () \,\}$$
$$[\![\tau \mid \tau']\!] \triangleq \{\, L\ v \mid v \in [\![\tau]\!] \,\} \cup \{\, R\ v \mid v \in [\![\tau']\!] \,\}$$
$$[\![\tau^*]\!] \triangleq \{\, [\,v_0, \ldots, v_n\,] \mid v_0, \ldots, v_n \in [\![\tau]\!], n \geqslant 0 \,\}$$

In our context, values in the type semantics preserve the type structure. We will denote flat values $\mathit{ft} \in \mathit{FTree}$ and $\mathit{fv} \in \mathit{FVal}$ (dropping left/right tags, parenthesis and list brackets) by:

Flat atomic values $\mathit{ft} ::= \texttt{true} \mid \texttt{false} \mid w \mid n[\mathit{fv}]$
Flat forest values $\mathit{fv} ::= () \mid \mathit{ft}, \mathit{fv}$

and introduce a function $\mathit{flat} : \mathit{Val} \to \mathit{FVal}$ that ignores markup. We denote the usual flat semantics of a type τ as $[\![\tau]\!]_{\mathit{flat}}$.

4.2 Core expression, path and pattern language

In BiFluX, updates instrumentally use XQuery expressions, XPath paths and XDuce patterns to manipulate XML data. This subsection succinctly describes their syntax in our core language, but is not essential for our design and may be skipped on a first reading.

We write *expressions* e in a minimal XQuery-like language that is a variant of the μXQ core language proposed in [9]:

$$
\begin{aligned}
e ::= &\ () \mid e, e' \mid n[e] \mid \texttt{let}\ pat = e\ \texttt{in}\ e' \mid p \mid e \approx e' \\
\mid &\ \ \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \texttt{for}\ x\ \texttt{in}\ e\ \texttt{return}\ e'
\end{aligned}
$$

Despite expressions can be used in updates rather indiscriminately, in BiFluX only a particular subset of the expression language is suitable for denoting foci. Therefore, we differentiate *paths* p in a core path language that represents a minimal dialect of XPath:

$$
\begin{aligned}
p\ &::= \texttt{self} \mid \texttt{child} \mid \texttt{dos} \mid\ ::nt \mid \texttt{where}\ e \mid p\ /\ p' \\
&\ \ \mid\ \ x \mid w \mid \texttt{true} \mid \texttt{false} \mid F(\vec{e}) \\
nt &::= n \mid \texttt{text}() \mid \texttt{node}()
\end{aligned}
$$

To simplify the formal treatment, we consider nodetests $::nt$ that apply to atomic values and where clauses $\texttt{where}\ e$ that filter values satisfying an expression e. We write the syntactic sugar $p :: nt \triangleq p\ /\ ::nt$ and $p[e] \triangleq p\,/\,\texttt{where}\ e$. XPath-like traversals can be defined as $. = \texttt{self}::\texttt{node}()$, $p\ /\ n \triangleq p\ /\ \texttt{child} :: a$ and $p\ //\ n = p\ /\ \texttt{dos} :: \texttt{node}()\ /\ \texttt{child} :: a$.

In contrast to μXQ, our core expression language supports pattern expressions in let bindings: $\texttt{let}\ pat = e\ \texttt{in}\ e'$ matches a pattern pat against the result of an expression e and then executes an expression e' that may refer to the variables bound by pat. We consider a language of XDuce-style *patterns pat* [17]:

$$
pat ::= x\ \texttt{as}\ \tau \mid \tau \mid () \mid n[pat] \mid pat, pat'
$$

Note that we impose a simple but strong syntactic *linearity* restriction on patterns (no alternative choice, no Kleene star) to ensure that matching a value against a pattern binds each variable exactly once. Less severe linearity restrictions are actually known [16], but these simple patterns suffice for our practical needs. Also worth noting is that we require every variable to be annotated with a type. This simplifies our design, but will in turn increase the number of (often unnecessary) annotations in our bidirectional update programs. We see it as an orthogonal problem that can be mitigated using existing tree-based type inference algorithms [32].

4.3 Tripartite environments

A *type environment* Γ consists of a set of bindings $x : \tau$ of variables to types. An *environment* is a function $\gamma : \mathit{Var} \to \mathit{Val}$ from variables to values. As usual, we assume variable names in an environment to be distinct. We write $\Gamma(x)$ and $\gamma(x)$ for the type and value of a variable; $\Gamma[x:\tau]$ and $\gamma[x:=v]$ add a new variable to an environment. We say that γ has type Γ, written $\gamma:\Gamma$, if $\gamma(x):\Gamma(x)$ for all $x\in\mathit{dom}(\Gamma)$. Fresh environments are denoted by the empty set $\emptyset$.

Since our language is bidirectional, we will consider three kinds of variables (and environments): *source variables*, that are accessible from the current source; *view variables*, that are accessible from the current view; and *normal variables*, that are accessible from the global environment of the update and independent from the source or the view. We will talk about source/view/normal paths or expressions, that may only refer to source variables, view variables or any variable, respectively; *non-source* expressions may refer to view and normal variables simultaneously.

To describe source or view environments, we introduce *record types* ν that denote sequences of variable-annotated types:

$$\nu ::= x:\tau,\nu \mid ()$$

As an abuse of notation, we see a record type ν as an ordinary type (by forgetting variable names) or as a type environment; we cast a conforming value v into an environment $\gamma_{v:\nu}$.

4.4 Core update language

Unlike conventional XML update languages, our core update language comprises two kinds of update statements: unidirectional Flux updates, interpreted as *arrows* [20] that modify a document in-place, changing its schema; and bidirectional BiFluX updates, interpreted as BXs [28] that update a source document given a view document or query a source document to compute its view fragment, for fixed source and view schemas. Our core *unidirectional updates* u are adapted from the core Flux update language [8]:

$$\begin{array}{rl} u ::= & \texttt{skip} \mid u;u' \mid \texttt{insert}\ e \mid \texttt{delete} \\ \mid & \texttt{if}\ e\ \texttt{then}\ u\ \texttt{else}\ u' \mid \texttt{case}\ e\ \texttt{of}\ \vec{pat}\to\vec{u} \\ \mid & p[u] \mid \texttt{left}[u] \mid \texttt{right}[u] \mid \texttt{children}[u] \end{array}$$

These include standard operations such as the no-op `skip`, sequential composition, conditionals or case expressions. The basic operations are `insert` e, that inserts a value given an empty sequence as focus; and `delete`, that replaces any value with the empty sequence. We can also apply an update in a specific *direction* (that traverses down a path p, moves to the `left` or `right` of a value, or focuses on the `children` of a labeled node).

Core *bidirectional updates* b are denoted by the grammar:

$$\begin{array}{rl} b ::= & \texttt{skip} \mid \texttt{fail} \mid b_1;b_2 \mid \texttt{view}\ x:=e\ \texttt{in}\ b \\ \mid & P(\vec{p_s},\vec{e_v},\vec{e}) \mid p[b] \mid [b]e \mid \texttt{replace} \mid \texttt{iter}\ b \end{array}$$

$$
\begin{array}{ll}
| & \texttt{ifS}\ e\ \texttt{then}\ b\ \texttt{else}\ b' \mid \texttt{caseS}\ p\ \texttt{of}\ \vec{pat} \to \vec{b} \\
| & \texttt{ifV}\ e\ \texttt{then}\ b\ \texttt{else}\ b' \mid \texttt{caseV}\ e\ \texttt{of}\ \vec{pat} \to \vec{b} \\
| & \texttt{if}\ e\ \texttt{then}\ b\ \texttt{else}\ b' \mid \texttt{case}\ e\ \texttt{of}\ \vec{pat} \to \vec{b} \\
| & \texttt{alignpos}\ e_s\ b\ c\ r \mid \texttt{alignkey}\ e_s\ p_s\ p_v\ b\ c\ r
\end{array}
$$

Here, P is the name of a BiFluX *procedure*. A procedure is defined as a declaration $P(\vec{x} : \vec{\tau}) : \nu_s \Leftrightarrow \nu_v \triangleq s$, meaning that P takes a vector of parameters $\vec{x}$ of types $\vec{\tau}$ and builds a BX between a source of type ν_s and a view of type ν_v. Accordingly, a procedure call $P(\vec{p_s}, \vec{e_v}, \vec{e})$ takes three kinds of arguments: source paths $\vec{p_s}$; non-source expressions $\vec{e_v}$; and normal expressions $\vec{e}$. We collect procedure declarations into a set Δ, that we will assume to be fixed. Procedures may also be recursive.

The bidirectional operation `skip` keeps the source unchanged for an empty view; for a non-empty view, we must `fail` to update the source (as UpdateQuery precludes that all view information must be used to update the source). Composition $b_1; b_2$ updates the source with b_1, and then runs b_2 over the updated source. The statement `view` $x := e$ `in` b models a view dependency, by stating that a view variable x can be computed using the view expression e (written $\$x := e$ in the high-level BiFluX language), and then runs b using the remaining view. Bidirectional statements may also change the current source or view, by focusing down a source path and updating the resultant source ($p[b]$), or by evaluating a non-source expression "backwards" from the view and updating the source with the resultant view ($[b]e$). The basic `replace` operation embeds the view into the source, while `iter` b embeds the same view into each tree in a source forest. As before, we consider three kinds of bidirectional conditionals and case expressions, whose expressions or paths are respectively source, view/non-source or normal variables.

The two special alignment statements update a source sequence using a view sequence. They receive a filtering source expression e_s, and match source elements satisfying e_s with view elements by position (`alignpos`) or by keys (`alignkey`), defined by two source (p_s) and view paths (p_v). Then, a bidirectional statement b processes matched source-view elements, a create statement c instructs how to create a suitable source to match with an unmatched view, and a recover statement r denotes how to process an unmatched source. Create statements c are simply optional unidirectional updates mu[5]. A *recover statement* r is a unidirectional update of the form:

$$
\begin{array}{rl}
r ::= & \texttt{if}\ e\ \texttt{then}\ r\ \texttt{else}\ r' \mid \texttt{delete} \mid \texttt{keep}\ u \\
| & \texttt{case}\ e\ \texttt{of}\ \vec{pat} \to \vec{r}
\end{array}
$$

It supports conditionals and case expressions like regular updates, and two primitive operations: `delete`, to delete an unmatched source; and `keep` u,

[5] We often refer to optional updates mu, optional values mv, etc by defining grammars $mu ::= \cdot \mid u$, $mv ::= \cdot \mid v$ and so on.

$$
\begin{array}{ll}
(\circ\!<) & : (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to (\tau_2 \Leftrightarrow_\Gamma \tau_3) \to (\tau_1 \Leftrightarrow_\Gamma \tau_3) \\
\mathsf{ifSthenelse} & : (\tau_1 \to \texttt{bool}) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \\
\mathsf{unfork} & : (\tau_1 \Leftrightarrow_\Gamma \tau_3) \to (\tau_2 \Leftrightarrow_\Gamma \tau_3) \to (\tau_1, \tau_2 \Leftrightarrow_\Gamma \tau_3) \\
\mathsf{ifVthenelse} & : (\tau_2 \to \texttt{bool}) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \\
\mathsf{remfst} & : (\tau_2 \to \tau_1) \to (\tau_2 \Leftrightarrow_\Gamma (\tau_1, \tau_2)) \\
\mathsf{withEnv} & : (\mathit{Maybe}\ \tau_1 \to \tau_2 \to \Gamma \to \Gamma') \to (\tau_1 \Leftrightarrow_{\Gamma'} \tau_2) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \\
\mathsf{map} & : (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to ({\tau_1}^* \Leftrightarrow_\Gamma {\tau_2}^*) \\
\mathsf{keep} & : \tau \Leftrightarrow_\Gamma () \\
\mathsf{keepfst} & : (\tau_1, \tau_2) \Leftrightarrow_\Gamma \tau_2 \\
\mathsf{keepsnd} & : (\tau_1, \tau_2) \Leftrightarrow_\Gamma \tau_1 \\
\texttt{alignpos} & : (\tau_1 \to \texttt{bool}) \to \mathit{Maybe}\ (\Gamma \to \tau_2 \to \tau_1) \to (\Gamma \to \tau_1 \to \mathit{Maybe}\ \tau_1) \\
 & \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to ({\tau_1}^* \Leftrightarrow_\Gamma {\tau_2}^*) \\
\mathsf{bot} & : \tau_1 \Leftrightarrow_\Gamma \tau_2 \\
\texttt{alignkey} & : (\Gamma \to \tau_1 \to \tau_{k_1}) \to (\Gamma \to \tau_2 \to \tau_{k_2}) \to (\tau_{k_1} \to \tau_{k_2} \to \texttt{bool}) \\
\mathsf{id} & : \tau \Leftrightarrow_\Gamma \tau \to (\tau_1 \to \texttt{bool}) \to \mathit{Maybe}\ (\Gamma \to \tau_2 \to \tau_1) \to (\Gamma \to \tau_1 \to \mathit{Maybe}\ \tau_1) \\
 & \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to ({\tau_1}^* \Leftrightarrow_\Gamma {\tau_2}^*) \\
\mathsf{in} & : n[\tau] \Leftrightarrow_\Gamma \tau \\
\mathsf{ifthenelse} & : (\Gamma \to \mathit{Maybe}\ \tau_1 \to \tau_2 \to \texttt{bool}) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \\
\mathsf{listeq} & : \tau^* \Leftrightarrow_\Gamma \tau
\end{array}
$$

Figure 10: Language of point-free lenses for translating core bidirectional updates (Complete in Appendix D).

to keep an unmatched source modified with u so that it does not satisfy the above e_s filtering expression.

4.5 Core bidirectional lens language

To interpret our core language bidirectionally, we translate core bidirectional updates into a core language of "putback"-style lenses over generalized tree structures [28], supporting regular expression types. For the context of this paper, a *lens* is a BX $l : \tau_s \Leftrightarrow_\Gamma \tau_v$ between a source type τ_s and a view type τ_v under an environment of type Γ, defined using the combinators from Figure 10. The concrete syntax for lenses is not essential in our design, and can be skipped unless for understanding the bidirectional update semantics discussed in Section 5. The intuition for each combinator (read as a transformation from view to source) should be understandable from its type signature, and more details regarding their concrete bidirectional semantics can be found in [28][6].

Each lens in the above language comprises two partial functions $U : \Gamma \to$

[6]An Haskell implementation of these (and more) combinators can be found at `http://hackage.haskell.org/package/putlenses`.

$Maybe\ \tau_s \to \tau_v \to \tau_s$ and $Q : \tau_s \to \tau_v$, satisfying laws similar to UPDATE-QUERY and QUERYUPDATE. Since these functions are partial, updating or querying may fail at runtime. This is sometimes inevitable, for instance, whenever a view value does not satisfy a view condition or a view dependency written in a `WHERE VIEW` clause. Remark that the update function U receives an additional environment of type Γ and an optional source of type $Maybe\ \tau_s$ (as in Haskell, but with short-hand notation $Nothing = .$ and $Just\ v = v$ for values), to account for cases (like `UNMATCHV` clauses) when a new source must be reconstructed from the view without updating an existing source [5, 27]. For a lens polymorphic over its environment type, we often write just $l : \tau_s \Leftrightarrow \tau_v$.

4.6 XML subtyping and ambiguity

The notion of subtyping plays a crucial role in XML approaches with regular expression types. A type τ_1 is said to be a *subtype* of τ_2, written $\tau_1 <: \tau_2$, if the flat values belonging to τ_1 are also values of τ_2, i.e., $[\![\tau_1]\!]_{\mathit{flat}} \subseteq [\![\tau_2]\!]_{\mathit{flat}}$. Under a flat value representation, a value of a type naturally belongs simultaneously to all its supertypes. This motivates a notion of structural equality between types: two types τ_1 and τ_2 are equivalent ($\tau_1 =: \tau_2$), meaning that they accept the same set of flat values, if both $\tau_1 <: \tau_2$ and $\tau_2 <: \tau_1$, e.g., $\tau =: \tau \mid \tau$. It also induces an equivalence relation $\sim$ that ignores structure and relates values parsing the same data using different markup, formally, $v \sim v' \triangleq \mathit{flat}(v) = \mathit{flat}(v')$, e.g., $L\ v \sim R\ v$.

A type τ is said to be *unambiguous* if the equivalence relation for structured values of that type ($\sim_\tau$) is the equality relation ($=_\tau$), intuitively meaning that there is only one way to parse a flat value of type τ into a structured value of type τ. Unambiguous regular expression types have a direct correspondence to algebraic data types [29], and standard automata algorithms exist for deciding unambiguity of regular expression types in polynomial time [10, 31].

Since we retain a structured representation of values, upcasting a value v_1 of type τ_1 into a supertype τ_2 requires more than a proof of subtyping: we must also change v_1 into a value v_2 that contains the same flat information as v_1 but conforms to the structure of τ_2. This problem has been considered in [23], that introduces a subtyping algorithm as a proof system with judgments of the form $\vdash \tau_1 <: \tau_2 \Rightarrow c$, that we treat as a "black box". In BX terms, $c : \tau_2 \twoheadleftrightarrow \tau_1$ is called a *canonizer* [13], which is a bit like a lens from τ_2 to τ_1 that comprises a total upcast function $ucast : \tau_1 \to \tau_2$, and a partial downcast function $dcast : \tau_2 \to \tau_1$. In our sense, canonizers satisfy two properties stating that they only handle structure:

$$ucast\ v_1 \sim v_1 \qquad \text{UP}_\sim$$

$$dcast\ v_2 = v_1 \Rightarrow v_1 \sim v_2 \qquad \text{DOWN}_\sim$$

For an unambiguous type τ_2, we can lift a canonizer $c : \tau_1 \mathrel{\leftrightarrow\!\!\!\!\triangleright\!\!\!\rightarrow} \tau_2$ into a well-behaved lens $\mathsf{lift}\ c : \tau_1 \Leftrightarrow \tau_2$. Similarly, for an unambiguous type τ_1, the inverse c^{-1} of a canonizer $c : \tau_1 \mathrel{\leftrightarrow\!\!\!\!\triangleright\!\!\!\rightarrow} \tau_2$ —even though not a canonizer itself due to partiality— can be lifted into a well-behaved lens $\mathsf{lift}\ c^{-1} : \tau_2 \Leftrightarrow \tau_1$.

5 Type System and Staged Semantics

In traditional XML-based languages with regular expression types [17, 8, 9], the loose separation between types and values lends itself to a Curry-style interpretation: terms in the language are given semantics regardless of typing, and typing rules assess the well-formedness of terms. In BiFluX, however, types come prior to semantics, as an update program describes a BX between two known source and view types on which its meaning may naturally depend. This leads to a Church-style interpretation, where semantics is given to type derivations instead of independent terms. Also, as we have seen before, updates in our core language are given semantics in two stages: they are first interpreted as lens expressions (typed between given types), and lenses have themselves a bidirectional semantics.

5.1 Interpreting expressions, paths and patterns

The type system and semantics for expressions and paths is similar to that of μXQ [9] and Flux [8], adapted to our context. We merge typing and semantics into a single judgment $\Gamma; mx \vdash e : \tau \Rightarrow \lambda\gamma.v$, meaning "in type environment Γ with optional root variable mx, expression e has type τ, and given environment γ it evaluates to value v". Type soundness is guaranteed by construction, such that the argument environment γ is of type Γ and the produced v is of type τ. The concrete rules can be found in Appendix A. We also define a judgment $\Gamma; \tau \vdash e : \tau' \Rightarrow \lambda\gamma\ v.v'$, meaning "in type environment Γ with root type τ, expression e has type τ', and given environment $\gamma : \Gamma$ and root value $v : \tau$ it evaluates to value $v' : \tau'$". The root type and underlying value are added to the (type and value) environments under a fresh root variable:

$$\frac{\Gamma[x : \tau]; x \vdash e : \tau' \Rightarrow \lambda v\ \gamma[x := v].v' \quad x \notin dom(\Gamma)}{\Gamma; \tau \vdash e : \tau' \Rightarrow \lambda\gamma\ v.v'}$$

In BiFluX, we follow a simple approach to pattern matching. We first define pattern type inference as a judgment $\vdash\ pat : \tau \Rightarrow \Pi_\tau$ that reads "pattern *pat* has type τ and yields a lens environment Π_τ". This is straightforward (Appendix B) since we require all pattern variables to be annotated with a type; many others, including [16, 32], have studied more advanced XML-based pattern type inference techniques. A *lens environment* Π_τ is a set of bindings $x := (l, \tau')$ of variables to lenses from a source type τ, such that $l : \tau \Leftrightarrow \tau'$. As an abuse of notation, we see Π_τ as a type environment;

we cast a source value v of type τ into an environment of source variables $\gamma_{v,\Pi_\tau} = \{ x_i := Q_i(v) \mid x_i := (l_i, \tau_i) \in \Pi_\tau \}$.

The second step for matching a pattern against an input value is to simply check for subtyping between the input type and the inferred pattern type [16], as the witness functions of our subtyping algorithm already give us a way to convert values in both directions. For ambiguous patterns that may match an input value in multiple ways, such functions are responsible for resolving the ambiguity. Different matching policies can be modelled by adapting the subtyping proof system, as noted in [23], despite not being essential for our approach.

5.2 Interpreting unidirectional updates

The typechecking and operational semantics of core unidirectional updates mimic those of the core FLUX language. The judgment $\Gamma \vdash \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'$ means that "in type environment Γ, a unidirectional update maps values of type τ to values of type τ', and given environment γ and input value v (of type τ) it produces the updated value v' (of type τ')". The corresponding rules do not present any significant novelty and are relegated to Appendix C.

5.3 Interpreting bidirectional updates

Contrarily to unidirectional updates, that modify values from an input to an output type, bidirectional updates are evaluated against two given source and view types and return a lens between those types. The judgment $\Gamma; \Pi_\tau \vdash \{\tau\}\ b\ \{\nu\} \Rightarrow l$ indicates that "in type environment Γ and lens environment Π_τ, a statement b produces a lens l (between source type τ and view type ν under type environment Γ)". The type environment Γ denotes normal variables (that are in scope of the update function of the generated lens); the lens environment Π_τ denotes source variables from the source τ; and the view type ν defines a view environment (also referred to as ν) that denotes view variables from the view ν. The dichotomy between our representations of source and view environments is justified by the need to keep a special account of view information: declared view variables must be used at least once in the update, while source variables might not be used at all to update the source. Most rules for typechecking and evaluating bidirectional updates as lenses are shown in Figure 11 and Figure 12. (The *listifyS* and *listifyV* functions and the judgment $\vdash\ \tau_1\ \leqslant\ \tau_2 \Rightarrow l$ will be introduced in Section 5.4.)

Source/view expressions are interpreted under only a lens/view environment: "in lens environment Π_τ (and source type τ), a source expression e has type τ', and given a value $v : \tau$ produces a value $v' : \tau'$"; "in view environment ν (and view type ν), a view expression e has type τ', and given a value $v : \nu$ produces a value $v' : \tau'$".

$$\frac{\Gamma; x \vdash e : \tau' \Rightarrow \lambda\gamma_{v,\Pi_\tau}[x := v].v' \quad \Gamma = \Pi_\tau[x : \tau] \quad x \notin vars(\Pi_\tau)}{\Pi_\tau \vdash_S e : \tau' \Rightarrow \lambda v.v'} \qquad \frac{\Gamma_\nu; \cdot \vdash e : \tau' \Rightarrow \lambda\gamma_{v:\nu}.v'}{\nu \vdash_V e : \tau' \Rightarrow \lambda v.v'}$$

Basic combinators The `skip` combinator returns the lens that does not update the source if the view is the empty sequence, while `fail` yields the bottom lens that always fails to update the source; `replace` completely replaces the source with a view that is "smaller", such that the view type is a subtype of the source type.

Composition Bidirectional composition $b_1; b_2$ first splits the view into two parts ν_1 and ν_2, evaluating b_1 with view ν_1 followed by s_2 with view ν_2 over the same source. The auxiliary function $split(\nu, \theta_1, \theta_2) = (\nu_1, \nu_2, l_{12})$ splits ν such that $dom(\nu_1) = dom\ (\nu) \cap \theta_1$ and $dom(\nu_2) = dom\ (\nu) \cap \theta_2$, returning a lens $l_{12} : (\nu_1, \nu_2) \Leftrightarrow \nu$. Parallel lens composition (unfork $l_1\ l_2$) is different from sequential lens composition ($l_1 \circ\!\!< l_2$); for it to be well-behaved, the second update can not affect the result of the first query, i.e., $Q_1(U_2(v, v_2)) \sqsubseteq Q_1(v)$. This property is currently checked dynamically by unfork, that fails to update the source otherwise. We believe that this check could be done at static time by combining recent work on XML query-update independence [2, 4, 6].

Changing source focus The bidirectional update $p[b]$ changes the source focus by traversing down the source path p, and then evaluates b with a fresh lens environment. Note that it is intrinsically different from the unidirectional update $p[u]$, as the source type does not change and the source data is not updated in-place, but modified to embed the view data. Precisely, this requires interpreting the source path as a lens via a judgment $\Pi_\tau \vdash_S \{\tau\}\ p\ \{\tau'\} \Rightarrow l$, that reads "in lens environment Π_τ, path p changes the source focus from type τ to type τ', and produces a lens $l : \tau \Leftrightarrow \tau'$" (Appendix F). Like in FLUX, arbitrary paths can not be used to change the source focus —as prescribed by the rules of Appendix F. For example, only the `self` and `child` axes (and no absolute paths) are supported; this ensures that only descendants of the source focus can be selected as the new source focus and that a selection contains no overlapping elements. The `iter` b operator shifts the source focus to all tree values in a source forest, and runs b for each tree. Its corresponding BX will duplicate the view value for each source tree during updates, and enforce all selected source trees to be the same during queries.

Changing view focus The operation $[b]e$ changes the view focus according to the non-source expression e, and then evaluates b for the intermediate view[7]. The judgment $\Gamma \vdash_V \{\tau\}\ e\ \{\nu\} \Rightarrow l$ indicates that "in type environment Γ, expression e changes the view focus from type ν to type τ, and produces a lens $l : \tau \Leftrightarrow_\Gamma \nu$" (Appendix G). Note that, to be successfully interpreted as a "backward" lens, the expression e may only add information to the view, since all view information must be used to update the source; this implies that only a very restrictive class of injective paths —statically checked by the rules of Appendix G— can be used to change the view focus. The operation `view` $x := e$ `in` b removes a view variable x from the current view environment without loss of information (with the view expression e evidencing how x can be computed from the other view variables), followed by running b.

Conditionals The conditional operations `ifS`, `ifV` and `if` choose between two statements b_1 or b_2 according to a boolean expression e, and differ subtly on the bidirectional behavior of the underlying lenses (see [28]), given that e is a source, view or normal expression. The same rationale is applied to case expressions (Appendix E).

Source-view alignment The alignment operations `alignpos` and `alignkey` synchronize source and view forests. They 1) uniformize the source and view types into lists using *listifyS* and *listifyV*, 2) align source and view elements using matching algorithms [1] on lists by position or by keys —calculated as paths on the current source/view— and 3) run the statement b for matching source/view pairs, the create statement c for unmatched source elements or the recover statement r for unmatched source elements. Create statements are interpreted unidirectionally using a judgment $\Gamma \vdash_{\texttt{create}} \{\tau_2\}\ c\ \{\tau_1\} \Rightarrow \lambda_m \gamma\ v_2 . v_1$, saying that "in type environment Γ, a create statement c between source type τ_1 and view type τ_2 returns an optional function (denoted by $\lambda_m \cdot . \cdot$) that given an environment $\gamma : \Gamma$ and a view value v_2 creates a source value v_1". In reverse direction, the judgment $\Gamma \vdash_{\texttt{recover}} r\ \{\tau_1\} \Rightarrow \lambda\gamma\ v_1 . mv_1$ states that "in type environment Γ, a recover statement r for a source type τ_1 returns a function that given an environment $\gamma : \Gamma$ and a source value $v_1 : \tau_1$ returns an optional recovered source value $mv_1 : \mathit{Maybe}\ \tau_1$". Note that our alignment operations receive a source predicate e denoting which source values have a correspondence to view values; the underlying lenses enforce that newly created source values must satisfy e, while recovered source values do not originate from the view and must not satisfy e.

Procedures Procedure calls $P(\vec{p_s}, \vec{e_v}, \vec{e})$ are interpreted under a new type environment computed from the argument expressions $\vec{e}$. The new lens envi-

[7] We use x_V as a special internal view variable to accommodate the intermediate view as a record type.

ronment is computed from the source paths $\vec{p_s}$, via a judgment $\Pi_\tau \vdash_{\texttt{procS}} \{\tau\}\ \vec{p}\ \{\nu_s\} \Rightarrow l$, and the new view environment is computed from the non-source expressions $\vec{e_v}$ via a judgment $\Gamma \vdash_{\texttt{procV}} \{\nu_v\}\ \vec{e}\ \{\nu\} \Rightarrow l$. The semantics of a BiFluX program is given by converting a set Δ of procedure declarations $P(\vec{x} : \vec{\tau}) : \nu_s \Leftrightarrow \nu_v \triangleq s$ into a set $\Delta_\Leftrightarrow$ of lenses $P = l : \nu_s \Leftrightarrow_{\{x_1:\tau_1,\ldots,x_n:\tau_n\}} \nu_v$, according to the following derivation:

$$\frac{\begin{array}{c}\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \cup \nu_s \cup \nu_v ; \Pi_{\nu_s} \vdash \{\nu_s\}\ s\ \{\nu_v\} \Rightarrow l \\ f\ v_s\ v_v\ \gamma = \gamma \cup \gamma_{v_s:\nu_s} \cup \gamma_{v_v:\nu_v}\end{array}}{\vdash P(\vec{x} : \vec{\tau}) : \nu_s \Leftrightarrow \nu_v \triangleq s \Rightarrow P = \mathsf{withEnv}\ f\ l}$$

5.4 Type normalization

The semantics of our core language relies instrumentally on subtyping to match source and view types. The simplest example is the `replace` bidirectional update, that requires the view type τ_2 to be "smaller" than the source type τ_1 (Figure 11, Figure 12), according to a judgment $\vdash \tau_2 \leqslant \tau_1 \Rightarrow l$ that returns as evidence a lens $l : \tau_1 \Leftrightarrow \tau_2$. As the reader may guess, we could compute $\vdash \tau_2 \leqslant \tau_1 \Rightarrow l$ by checking for subtyping between τ_2 and τ_1 ($\vdash \tau_2 <: \tau_1 \Rightarrow c$) and lifting the resulting canonizer into a lens. But, before doing so, we must verify that: the view type τ_2 is unambiguous, a requirement to lift c into a lens $\mathsf{lift}\ c$; and the source type τ_1 is unambiguous, since the *ucast* function of the canonizer does not consider the original source and a naive implementation of an update would potentially discard source information projected away by the lens.

As an example, recapitulate the source database of books used in Section 2 and imagine how we could evaluate a simple update:

```
REPLACE $source/books/book/title WITH $view
```

using a list $\mathit{title}[\texttt{string}]^*$ as the view type. Consistently with the unidirectional semantics for paths (Appendix A), the evaluation of the source path as a lens traverses down the source tree and keeps only titles, destroying element labels and replacing all authors for the empty sequence, and modifies the source focus to the intermediate type $(\mathit{title}[\texttt{string}], (), ()^*)^*$. Then `replace` checks for subtyping between the view type and the intermediate type, producing a mediating canonizer. Since the intermediate type is ambiguous, the upcasting function of the canonizer would translate a view sequence $[\,\mathit{title}[\texttt{"mybook"}]]$ into an intermediate value $[(\mathit{title}[\texttt{"mybook"}], ((), [\,]))]$, that would in turn lead to an updated source $\mathit{books}[[\mathit{book}[(\mathit{title}[\texttt{"mybook"}], (\mathit{aut}_1, [\,]))]]]$. This is clearly unsatisfactory as it would discard the entire book database except the first author aut_1 of the first book!

Source and view normalization To avoid these problems, we introduce *type normalization* procedures that simplify regular expression types into unambiguous ones while carefully preserving the original markup information that keeps trace of hidden information. Namely, we define a *source normalization* procedure $normS(\tau) = (\tau', l)$, that normalizes a source type into an unambiguous subtype τ' together with a lens $l : \tau \Leftrightarrow \tau'$, and a *view normalization* procedure $normV(\tau) = (\tau', l)$, that normalizes a view type into an unambiguous subtype type τ' together with a lens $l : \tau \Leftrightarrow \tau'$. Their main difference is that they compute lenses in opposite directions, suggesting that source normalization may abstract ambiguous information (like redundant choices) while view normalization may not. Their definitions are shown in Appendix H. We do not claim that our normalization procedures are complete, in the sense that they can disambiguate any ambiguous type, but when they (statically) do succeed the normalized types are unambiguous. In a nutshell, we try to normalize a source type using automata reduction techniques [10, 31], and derive a lens between the ambiguous and unambiguous types; we only normalize view types into isomorphic types. Furthermore, we evaluate source paths in a two-phased semantics: a path is interpreted as a completely information-preserving lens, that just marks unselected source types τ with a special tag $\boxed{\tau}$ instead of replacing them for the empty sequence (Appendix F); and tagged types are to be removed during source normalization, that already requires special handling anyway.

Normalized subtyping and listification We are now ready to safely define $\tau_1 \leqslant \tau_2$ as the following judgment:

$$\frac{normS(\tau_2) = (\tau_2', l_2) \quad normV(\tau_1) = (\tau_1', l_1) \quad \vdash \tau_1' <: \tau_2' \Rightarrow c}{\vdash \tau_1 \leqslant \tau_2 \Rightarrow l_1 \circ\!< \mathsf{lift}\ c \circ\!< l_2}$$

Unlike iteration for unidirectional updates, that changes the focus to all the atomic elements in a forest and updates their values (and types!) independently, bidirectional iteration is type-preserving and involves updating a source sequence with view information and somehow fitting it back into a shape that conforms to the source type. Since *shape alignment* is an intractable problem for arbitrary source and view types [27], we uniformize source and view types into lists of (choices of) atomic types according to two functions:

$$\frac{\begin{array}{c} elems(\tau') = \{\alpha_1, \ldots, \alpha_n\} \quad (\alpha_0 \mid \ldots \mid \alpha_n)^* \text{ unambiguous} \\ normS(\tau) = (\tau', l) \quad \vdash \tau' <: (\alpha_0 \mid \ldots \mid \alpha_n)^* \Rightarrow c \end{array}}{listifyS(\tau) = ((\alpha_0 \mid \ldots \mid \alpha_n), l \circ\!< \mathsf{lift}\ c^{-1})}$$

$$\frac{\begin{array}{c} elems(\tau') = \{\alpha_0, \ldots, \alpha_n\} \quad (\alpha_0 \mid \ldots \mid \alpha_n)^* \text{ unambiguous} \\ normV(\tau) = (\tau', l) \quad \vdash \tau' <: (\alpha_0 \mid \ldots \mid \alpha_n)^* \Rightarrow c \end{array}}{listifyV(\tau) = ((\alpha_0 \mid \ldots \mid \alpha_n), \mathsf{lift}\ c \circ\!< l)}$$

Note that the uniformized list types may be more flexible supertypes, e.g. $(\alpha_1, \alpha_2)^* <: (\alpha_1 \mid \alpha_2)^*$, allowing more values than those that fit the real type. While this does not pose a problem with the for-each semantics of plural high-level updates, it may lead to runtime errors for `UPDATE FOR VIEW` updates over intricate structures. An alternative is to statically check for type equivalence in *listifyS*, supporting only pure source lists. The function $elems : Type \rightarrow \{Atom\}$ returns the set of atomic types in a sequence type.

6 BiFluX to Core Update Normalization

In this section, we formalize the translation from the high-level BiFluX language to the core language presented in Section 4, highlighting the significant gap between them. This process is usually referred to as *normalization* in languages like XQuery and Flux. We define two main normalization functions that interpret statements as bidirectional $[\![-]\!]^b_{Stmt}$ and unidirectional updates $[\![-]\!]^u_{Stmt}$. Most translation rules are straightforward and a few interesting ones are shown in Figure 13; the complete set can be found in Appendix I. Simple bidirectional updates are translated by a function $[\![-]\!]^b_{Upd}(e_s, e_v, \vec{x} = \vec{e})$, where the extra parameters group the `WHERE` clauses of the update into a source selection expression e_s, a view selection expression e_v and a sequence of view bindings $\vec{x} = \vec{e}$; these triples are parsed from a set of conditions, according to their source/view tags, by a function $[\![-]\!]_{Conds}$. Simple unidirectional updates are translated by a function $[\![-]\!]^u_{Upd}(e)$, where e is the conjunction of all the `WHERE` clauses of the update.

For special `UPDATE FOR VIEW` statements, the *splitVStmt* function parses a *VStmt* into a matching statement and two optional unmatched-view and unmatched-source statements. Optional unmatched-view statements are translated using a function $[\![-]\!]^c_{MStmt}(mpat)$ that takes an extra optional view pattern and returns a core create update; if no `UNMATCHV` clause is defined, the U function of the underlying lens will be evaluated without an original source. Optional unmatched-source statements are translated using a function $[\![-]\!]^r_{MStmt}(mpat)$ that takes an extra optional source pattern and returns a core recover update; if no `UNMATCHS` clause is defined, all unmatched source elements are deleted by default.

The translation denotes a partial function from high-level BiFluX to core BiFluX. For example, `INSERT` is not supported for bidirectional updates, `UPDATE FOR VIEW` is not supported for unidirectional updates, and `CREATE` or `KEEP` are only supported under `UNMATCHV` or `UNMATCHS` clauses, respectively. We assume that paths and expressions are expressed in terms of our core languages; this is standard practice as normalization of XQuery expressions or XPath paths can be done independently. To simplify the presentation, we also assume explicit `SOURCE` and `VIEW` tags, though our implementation is elaborated to implicitly distinguish between source/view/normal expressions, using the additional environment information available at the time of typechecking the core language.

7 Related Work

XML update languages Several XML update languages have been proposed, including (among many others) XQuery! [15], Flux [8] and the standard W3C XQuery Update Facility [30]. Even though the specification style, expressiveness and semantics of the XML updates that can be written may vary significantly, they all focus on updating XML documents in-place, i.e., updating selected parts of an XML document, keeping the remaining parts of the document unchanged. This means that update programs can be seen as unidirectional transformations that insert, delete or replace elements in a source document and produce an updated document conforming to a new target type. XML Updates in BiFluX are different in that they determine how to update a source document (using some view information) while preserving its source type. This poses different (BX-related) challenges on how to deal with non-in-place updates (like `UPDATE FOR VIEW` statements that may change the cardinality of a sequence instead of updating each element), and therefore how to modify the remaining parts of the source document (e.g., by changing branching decisions) to accommodate the new data so that the updated source fits into the same type.

XML view updating In [12], the author studies the problem of updating XML views of relational databases by translating view updates written in the XQuery Update Facility into embedded SQL updates. The work of [22] supports updatable views of XML data by giving a bidirectional semantics to the XQuery Core language. The semantic bidirectionalization technique of [25] interprets various XQuery use cases as BXs by encoding them as polymorphic Haskell functions. The Multifocal language [26] allows writing high-level generic XML views that can be applied to multiple XML schemas, producing a view schema and a lens conforming to the schemas. In the four approaches, the programmer writes a view function and the system derives a suitable view update translation strategy using built-in techniques that he can not configure. In BiFluX, he writes an update translation strategy directly as an update (over the source) and the system derives the uniquely related query.

XML bidirectional languages Many bidirectional programming languages support tree-structured or XML data formats. Two popular XML bidirectional languages are XSugar [7] and biXid [21], that describe XML-to-ASCII and XML-to-XML mappings as pairs of intertwined grammars. While XSugar restricts itself to bijective grammars, biXid considers nondeterministic specifications and BXs are inherently ambiguous. Most functional bidirectional programming languages are based on lenses [14, 27, 28, 19], and follow a combinatorial style that puts special emphasis on building complex lenses by composition of smaller combinators. Depending on the choice of combinators, lens languages can become very powerful at specifying application-specific behavior [28, 1, 27]. However, their lower-level nature also induces a more cumbersome programming style that makes it impractical and often unintuitive for users to build non-trivial BXs by piping together several small, surgical steps.

BiFluX features a new programming by update paradigm, that enables the high-level syntax of relational languages such as XSugar and biXid while providing a handful of intuitive update strategies. Remember the huge gap between our high-level BiFluX language (variables, procedures) and the core lens language that gives it semantics (canonical "point-free" combinators). In [18], we have proposed a simple treeless functional language for writing total *put* (or update) functions, such that existence of a well-behaved lens can be checked statically, and corresponding total *get* (or query) functions can be derived automatically. The most significant and innovative difference in BiFluX is again the declarative surface language used to specify BXs as bidirectional update programs, at a notably higher-level of abstraction than native *put* functions. BiFluX programs may however be partial, due to the added expressiveness of our language.

Quotient lenses [13] propose loosening lenses modulo equivalence relations, for easing the processing of ad-hoc data formats. To enable compositional reasoning, most interesting quotient lens combinators (concatenation, union, iteration) require the equivalence relations to be decomposable. At first glance, we could lift our core lens language into quotient lenses to allow seamless composition with (subtyping) canonizers. However, since our notion of equivalence is not decomposable for ambiguous types, this would not overcome the need for our type normalization procedures.

8 Conclusions

In this paper, we propose a novel *bidirectional programming by update* paradigm that comes to light from the idea of extending a traditional update language with bidirectional features. Under the new paradigm, programmers write bidirectional updates that specify how to update a source document to reflect additional view information. We substantiate with examples that this enjoys a better tradeoff between the expressiveness and declarativeness of the written bidirectional programs, by allowing users to write directly, in a friendly notation and at a nice level of abstraction, a view update translation strategy that bundles all the pieces to build a BX.

To demonstrate the potential of this paradigm, we designed BiFluX, a type-safe high-level bidirectional XML update language. We have fully implemented BiFluX in Haskell and the code, together with additional examples, is available from the project's website[8]. Our tool translates source

[8] `http://www.prg.nii.ac.jp/projects/BiFluX`

and view DTDs into Haskell type declarations using the `HaXml` package[9], and interprets bidirectional updates as bidirectional lens transformations between the given schemas in a robust manner: the implementation of the core-to-lens translation is strongly-typed, serving as a proof of soundness that helped us catching early programming errors at compile-time; and the bidirectional semantics is completely guided by an underlying lens language, whose correctness has been shown separately in [28]. To support a flexible design, with arbitrary conditionals, case statements and expressions, the statically generated lenses may undergo runtime checks (for particular source and view documents) to ensure that the underlying update and query functions are well-behaved.

As future work, we plan to provide more static guarantees to BiFluX by incorporating existing path-query static analyses, implement more powerful pattern type inference algorithms to avoid excessive annotations, and extend the class of bidirectional updates that can be written by integrating user-defined lenses for defining source and view foci. We also plan to improve the efficiency of our prototype for large XML databases by exploring optimizations to the underlying lens language, including incremental update translation. To empirically study the practical impact of BiFluX, we are currently undergoing a larger model-based code testing use case.

References

[1] D. M. J. Barbosa, J. Cretin, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *ICFP 2010*, pages 193–204. ACM, 2010.

[2] M. Benedikt and J. Cheney. Destabilizers and independence of XML updates. *Proc. VLDB Endow.*, 3(1-2):906–917, 2010.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP 2003*, pages 51–63. ACM, 2003.

[4] N. Bidoit, D. Colazzo, and F. Ulliana. Type-based detection of XML query-update independence. *Proc. VLDB Endow.*, 5(9):872–883, 2012.

[5] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *POPL 2008*, pages 407–419. ACM, 2008.

[6] S. Bottcher. Testing intersection of XPath expressions under DTDs. In *IDEAS 2004*, pages 401–406. IEEE, 2004.

[9] `http://hackage.haskell.org/package/HaXml`

[7] C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4-5):385–406, 2008.

[8] J. Cheney. FLUX: functional updates for XML. In *ICFP 2008*, pages 3–14. ACM, 2008.

[9] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Static analysis for path correctness of XML queries. *Journal of Functional Programming*, 16(4-5):621–661, 2006.

[10] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree automata techniques and applications*. Available at `http://www.grappa.univ-lille3.fr/tata`, 2007.

[11] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT 2009*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

[12] L. Fegaras. Propagating updates through XML views using lineage tracing. In *ICDE 2010*, pages 309 –320. IEEE, 2010.

[13] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *ICFP 2008*, pages 383–396. ACM, 2008.

[14] J. Nathan Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM TOPLAS*, 29(3):17, 2007.

[15] G. Ghelli, C. Ré, and J. Siméon. XQuery!: An XML query language with side effects. In *EDBT 2006*, volume 4254 of *LNCS*, pages 178–191. Springer, 2006.

[16] H. Hosoya and B. Pierce. Regular Expression Pattern Matching for XML. In *POPL 2001*, pages 67–80. ACM, 2001.

[17] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *ICFP 2000*, pages 11–22. ACM, 2000.

[18] Z. Hu, H. Pacheco, and S. Fischer. Validity checking of putback transformations in bidirectional programming (invited paper). In *FM 2014*, pages 1–15. Springer, 2014.

[19] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1-2):89–118, 2008.

[20] John Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1):67–111, 2000.

[21] S. Kawanaka and H. Hosoya. biXid: a bidirectional transformation language for XML. In *ICFP 2006*, pages 201–214. ACM, 2006.

[22] D. Liu, Z. Hu, and M. Takeichi. An expressive bidirectional transformation language for XQuery view update. *Progress in Informatics*, 10:89–130, 2013.

[23] K. Z. M. Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *APLAS 2004*, volume 3302 of *LNCS*, pages 57–73. Springer, 2004.

[24] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ICFP 2007*, pages 47–58. ACM, 2007.

[25] K. Matsuda and M. Wang. Bidirectionalization for Free with Runtime Recording: Or, a Light-weight Approach to the View-update Problem. In *PPDP 2013*, pages 297–308. ACM, 2013.

[26] H. Pacheco and A. Cunha. Multifocal: A strategic bidirectional transformation language for XML schemas. In *ICMT 2012*, volume 7307 of *LNCS*, pages 89–104. Springer, 2012.

[27] H. Pacheco, A. Cunha, and Z. Hu. Delta lenses over inductive types. In *BX 2012*, volume 49 of *Electronic Comms. of the EASST*, 2012.

[28] H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In *PEPM 2014*, pages 39–50. ACM, 2014.

[29] V. Preoteasa. A relation between unambiguous regular expressions and abstract data types. *Fundamentae Informatica*, 40(1):53–77, 1999.

[30] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon. Xquery update facility 1.0. W3C Recommendation. `http://www.w3.org/TR/xquery-update-10/`, March 2011.

[31] Helmut Seidl. On the finite degree of ambiguity of finite tree automata. *Acta Informatica*, 26(6):527–542, 1989.

[32] S. Vansummeren. Type inference for unique pattern matching. *ACM TOPLAS*, 28(3):389–428, 2006.

A Type system and semantics for queries

Following FLUX, we use a variant of a μXQ, a statically typed XQuery-like core language introduced in [9]. The main syntactic distinction is that we divide expressions into XQuery-like expressions and XPath-like paths. We do not consider attributes, but they can be added to our type system and to our query language, by assuming that every labeled tree has a special sorted product of attribute-tagged atomic values that comes before regular tree content, adapting the semantics to preserve this sortedness and evaluating the `child`, `dos` and the `attribute` axes appropriately.

We interpret expressions using a judgment $\Gamma; mx \vdash e : \tau \Rightarrow \lambda\gamma.v$ that receives an optional variable mx which indicates the current root value of the expression. This is useful for being able to interpret local paths like `self` using the same notation. We also define a matching relation between atomic types and node tests, that is decidable using the following rules:

$$\frac{}{\texttt{string} <: \texttt{text}()} \quad \frac{}{n[t] <: n} \quad \frac{}{\alpha <: \texttt{node}()}$$

Figure 14 and 15 shows the rules for interpreting expressions and paths. The auxiliary judgment $\Gamma; mx \vdash_{\texttt{for}} x \texttt{ in } \tau \to e : \tau' \Rightarrow \lambda\gamma\ v.v'$ iterates over a sequence, binding the variable x to each atomic value. It reads "in type environment Γ with optional root variable mx, iterating over type τ by applying expression e to each atomic value labelled x yields type τ', and given environment γ it evaluates source sequence v to view sequence v'". Note that for-iteration does not change the current focus (i.e., the optional root variable), consistently with the semantics of XQuery.

We consider only one binary operation —expression equality— and the XPath *boolean* function, but more built-in operations can be added without disrupting the semantics. The more intricate evaluation of `dos` is explained in [9]. The type-level function $boolean_\tau : \tau \to \texttt{bool}$ generically converts any value into a boolean value, mimicking the semantics of XPath:

$$
\begin{aligned}
&boolean_{\texttt{bool}}(v) = v\\
&boolean_{\texttt{string}}(w) = w \neq \texttt{""}\\
&boolean_{n[\tau]}(n[v]) = \texttt{true}\\
&boolean_{()}(v) = \texttt{false}\\
&boolean_{\boxed{\tau}}(v) = \texttt{false}\\
&boolean_{\tau_1|\tau_2}(L\ v_1) = boolean_{\tau_1}(v_1)\\
&boolean_{\tau_1|\tau_2}(R\ v_2) = boolean_{\tau_2}(v_2)\\
&boolean_{\tau_1,\tau_2}(v_1, v_2) = boolean_{\tau_1}(v_1) \vee boolean_{\tau_2}(v_2)\\
&boolean_{\tau^*}([\,]) = \texttt{false}\\
&boolean_{\tau^*}([v_0, v_1, \ldots, v_n]) = \texttt{true}\\
&boolean_X(v) = boolean_{E(X)}(v)
\end{aligned}
$$

We restate the results for expression soundness proven in [8]:

Theorem A.1 (Expression soundness)**.**

1. *If* $\Gamma; mx \vdash e : \tau \Rightarrow \lambda\gamma.v$ *and* $\gamma : \Gamma$, *then* $v : \tau$
2. *If* $\Gamma; mx \vdash_{\texttt{for}} x \texttt{ in } \tau \rightarrow e : \tau' \Rightarrow \lambda\gamma\ v.v'$, $\gamma : \Gamma$ *and* $v : \tau$, *then* $v' : \tau'$.

B Type system and semantics for patterns

Our pattern type inference algorithm, given by judgments of the form $\vdash\ pat : \tau \Rightarrow \Pi_\tau$, is described in Figure 16. It is straightforward since we require all variables to be annotated with a type. In the last rule, we pre-compose a lens $l : \tau' \Leftrightarrow \tau$ with a lens environment Π_τ as $l \circ\!< \Pi_\tau = \{ x_i := (l \circ\!< l_i, \tau_i) \mid x_i := (l_i, \tau_i) \in \Pi_\tau \}$.

C Type system and semantics for unidirectional updates

The rules for interpreting in-place unidirectional updates are shown in Figure 17 and 18. They satisfy the following soundness property.

Theorem C.1 (Unidirectional update soundness)**.**

1. *If* $\Gamma \vdash \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'$, $\gamma : \Gamma$ *and* $v : \tau$, *then* $v' : \tau'$.
2. *If* $\Gamma; \Pi_{\tau_1}, \ldots, \Pi_{\tau_n} \vdash_{\texttt{case}} \{\tau\}\ \{\tau_1 \mid \ldots \mid \tau_n\}\ \vec{u}\ \{\tau'\} \Rightarrow \lambda\gamma\ v\ v_{1n}.v'$, $\gamma : \Gamma$, $v : \tau$ *and* $v_{1n} : \tau_1 \mid \ldots \mid \tau_n$, *then* $v' : \tau'$.
3. *If* $\Gamma \vdash_{\texttt{iter}} \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'$, $\gamma : \Gamma$ *and* $v : \tau$, *then* $v' : \tau'$.

D Core bidirectional lens language

The complete language of lenses used in the translation of our core bidirectional language into bidirectional transformations is given in Figures 10 and 19. These combinators are part of the language from [28], with the exception of our specific recursive combinators over lists that are defined as follows:

$$\begin{aligned}
\mathsf{foldlist}\ l &= \mathsf{in} \circ\!< (\mathsf{id} \oplus \mathsf{id} \otimes \mathsf{foldlist}\ l) \circ\!< l \\
\mathsf{mergelist}\ l &= (\mathsf{in} \otimes \mathsf{id}) \circ\!< \mathsf{undistl} \circ\!< (\mathsf{keepfst} \oplus \mathsf{assocl}) \\
&\quad \circ\!< \mathsf{coswap} \circ\!< (\mathsf{id} \otimes \mathsf{mergelist}\ l \oplus \mathsf{id}) \circ\!< l
\end{aligned}$$

Although lists are not defined as recursive types in our type system, some of our type normalization rules inductively decompose lists. We overload $\mathsf{in} : \tau^* \Leftrightarrow (() \mid (\tau, \tau^*))$ and $\mathsf{out} : (() \mid (\tau, \tau^*)) \Leftrightarrow \tau^*$ for lists.

The more interesting combinator is $l_1 \uplus l_2$, that returns the union of two lenses. For instance, for lists we have $\mathsf{unnil} \uplus \mathsf{uncons} = \mathsf{out}$.

Constant complement lenses In bidirectional programming, keeping hidden source parts unaffected by updates is known as view-update translation under a *constant complement* [24], i.e., for a lens $l : \tau_1 \Leftrightarrow \tau_2$ there is a complement function $C : \tau_1 \to \tau_3$ that captures all the hidden source information (in the sense that the tupled C and Q are injective and sufficient to restore any instance of τ_1) satisfying a law $C(U(v_1, v_2)) \sqsubseteq C(v_1)$. However, forcing a constant complement is often too restrictive in practice, as update functions must disallow any update that changes the complement of the original source. For our example, any update changing the cardinality of view titles would not be translatable to the source bookstore; only updating the titles of existing books in-place would be allowed.

Weak constant complement lenses In BiFluX, we encounter many situations where we want to perform non-in-place updates that reshape the source data to accommodate the updated view information. For instance, even if the source database is empty, running our example with a single view title should be able to create a default source book; or running it with an empty view for a non-empty database should necessarily drop the hidden source authors of original books. Nevertheless, we still want to keep hidden source data intact whenever possible. For this purpose, we formulate a weaker notion of constant complement that accepts complement-changing updates only when there is no valid source with the same complement. The intuition is that in-place updates will preserve the complement, while non-in-place updates may not. For our example, the authors associated with titles in the view should be kept unchanged, whereas non-associated authors could be lost.

For each lens $l : \tau_1 \Leftrightarrow_\Gamma \tau_2$ used in our source normalization procedure, we can define a complement type τ_3 and a complement function $C_l : \tau_1 \to \tau_3$ by induction over the lens expressions, as shown in Figure 20. The complement functions for $\mathsf{foldlist}\ l$ and $\mathsf{mergelist}\ l$ follow the same recursive structure as their argument lenses, and terminate whenever the corresponding recursive query functions terminate.

Theorem D.1. *For a source normalization $normS(\tau_1) = (\tau_2, l)$ the lens l weakly preserves a constant complement function $C : \tau_1 \to \tau_3$, defined by induction over l in Figure 20, in the following sense:*

$$(\exists v_1'.\ C\ v_1' = C\ v_1 \wedge Q\ v_1' = v_2) \Rightarrow C\ (U\ (v_1, v_2)) \sqsubseteq C\ v_1$$

E Type system and semantics for bidirectional updates

The complete set of rules for interpreting bidirectional updates is shown in Figures 11, 12, 21 and 22. Note that source arguments of procedures make use of unfork to apply distinct paths in parallel to the current source. We believe that the dynamic checking performed by unfork could be prevented by applying existing techniques for checking XPath-like path disjointness at static time [6]. Bidirectional updates obey the following soudness property.

Theorem E.1 (Bidirectional update soundness)**.**

1. *If* $\Gamma;\Pi \vdash \{\tau\}\ s\ \{\nu\} \Rightarrow l$*, then* $l : \tau \Leftrightarrow_\Gamma \nu$.
2. *If* $\Gamma;\Pi_\tau;\Pi_{\tau_1},\ldots,\Pi_{\tau_n};f \vdash_{\texttt{caseS}} \{\tau\}\ \{\tau_1 \mid \ldots \mid \tau_n\}\ \vec{s}\ \{\nu\} \Rightarrow l$ *and* $f : \tau \rightarrow \tau_1 \mid \ldots \mid \tau_n$*, then* $l : \tau \Leftrightarrow_\Gamma \nu$.
3. *If* $\Gamma;\Pi_\tau \vdash_{\texttt{caseV}} \{\tau\}\ \vec{s}\ \{\nu\}\ \{\nu_1 \mid \ldots \mid \nu_n\} \Rightarrow l$*, then* $l{:}\tau \Leftrightarrow_\Gamma (\nu_1 \mid \ldots \mid \nu_n)$.
4. *If* $\Gamma;\Pi_\tau;\Pi_{\tau_1},\ldots,\Pi_{\tau_n};f \vdash_{\texttt{case}} \{\tau\}\ \{\tau_1 \mid \ldots \mid \tau_n\}\ \vec{s}\ \{\nu\} \Rightarrow l$ *and* $f : \tau \rightarrow \tau_1 \mid \ldots \mid \tau_n$*, then* $l : \tau \Leftrightarrow_\Gamma \nu$.
5. *If* $\Pi_\tau \vdash_{\texttt{procS}} \{\tau\}\ \vec{p}\ \{\nu_s\} \Rightarrow l$*, then* $l : \tau \Leftrightarrow \nu_s$.
6. *If* $\Gamma \vdash_{\texttt{procV}} \{\nu_v\}\ \vec{e}\ \{\nu\} \Rightarrow l$*, then* $l : \nu_v \Leftrightarrow_\Gamma \nu$.

Create and recover statements can be interpreted unidirectionally as shown in Figures 11, 12 and 23. They satisfy similar soudness properties as those for unidirectional updates.

Theorem E.2 (Create and recover statement soudness)**.**

1. *If* $\Gamma \vdash_{\texttt{create}} \{\tau_2\}\ c\ \{\tau_1'\} \Rightarrow \lambda_m\gamma\ v_2.v_1$, $\gamma : \Gamma$ *and* $v_2 : \tau_2$*, then* $v_1 : \tau_1$.
2. *If* $\Gamma \vdash_{\texttt{recover}} r\ \{\tau\} \Rightarrow \lambda\gamma\ v.mv$, $\gamma : \Gamma$ *and* $v : \tau$*, then* $mv : Maybe\ \tau$.
3. *If* $\Gamma;\Pi_{\tau_1},\ldots,\Pi_{\tau_n} \vdash^{\texttt{case}}_{\texttt{recover}} \{\tau_1 \mid \ldots \mid \tau_n\}\ \vec{r}\ \{\tau\} \Rightarrow \lambda\gamma\ v\ v_{1n}.mv$, $\gamma : \Gamma$, $v : \tau$ *and* $v_{1n} : \tau_1 \mid \ldots \mid \tau_n$*, then* $mv : Maybe\ \tau$.

F Type system and semantics for source paths as lenses

In our core bidirectional language, source paths that change the current source focus of a bidirectional update need to be evaluated as lenses (Figure 24), so that they can propagate arbitrary view updates to the source. Not every source path can be interpreted as a lens; for example, we exclude the descendant-or-self axis, for which it is hard to statically provide a precise view type [9]. We also do not consider here the conversion of built-in functions to lenses. We evaluate source paths as lenses using a judgment that receives only the lens environment of source variables:

$$\frac{\Pi_\tau \vdash_S \{\tau\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l_1 \quad f_1\ v = \gamma_{v,\Pi_\tau} \quad f_2\ \gamma = l_1}{\Pi_\tau \vdash_S \{\tau\}\ p\ \{\tau'\} \Rightarrow \mathsf{param}\ f_1\ f_2}$$

and initializes a more complicated judgment $\Gamma \vdash_S \{\tau\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l$ with the source environment.

Soundness can be proved in the usual way:

Theorem F.1 (Source path as lens soundness)**.**

1. *If* $\Pi_\tau \vdash_S \{\tau\}\ p\ \{\tau'\} \Rightarrow l$*, then* $l : \tau \Leftrightarrow \tau'$*.*
2. *If* $\Gamma \vdash_S \{\tau\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l$ *and* $\gamma : \Gamma$*, then* $l : \tau \Leftrightarrow \tau'$*.*
3. *If* $\Gamma \vdash_S^{\texttt{iter}} \{\tau\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l$ *and* $\gamma : \Gamma$*, then* $l : \tau \Leftrightarrow \tau'$*.*

G Type system and semantics for non-source expressions as lenses

In the opposite direction to source paths, in our core bidirectional language we may apply non-source expressions (expressions that may use regular variables or view variables) to change the current view focus. These expressions are useful for building tree structures that match the source types using view variables, and need to be interpreted as lenses from their result type to the current view. To ensure that no view information is lost, not every non-source expression can be evaluated as lenses. For instance, it needs to use all view variables and shall only contain injective paths that follow the structure of the view type; note how there is no rule for $\Gamma \vdash_V \{\alpha\} :: nt\ \{\alpha\} \Rightarrow l$ whenever the nodetest nt does not match the view type α.

, and must match the structure of the view (note that). The corresponding rules to interpret non-source expressions as lenses are given in Figure 25 and 26. When performing pattern matching on views, for instance when evaluating non-source expressions, we need to guarantee that, unlike source patterns that may not bind variables to all the source values, all view information must be present in the patterns variables. We also need to interpret view patterns as "reversed" lenses (Figure 27).

The interpretation of non-source expressions can be summarized by the following theorem:

Theorem G.1 (Non-source expression as lens soundness)**.**

1. *If* $\Gamma \vdash_V \{\tau\}\ e\ \{\nu\} \Rightarrow l$*, then* $l : \tau \Leftrightarrow_\Gamma \nu$*.*
2. *If* $\Gamma \vdash_V \{\tau'\}\ p\ \{\tau\} \Rightarrow l$*, then* $l : \tau' \Leftrightarrow_\Gamma \tau$*.*
3. *If* $\Gamma; x \vdash_V^{\texttt{for}} \{\tau'\}\ e\ \{\tau\} \Rightarrow l$*, then* $l : \tau' \Leftrightarrow_\Gamma \tau$*.*
4. *If* $\vdash_V pat : \tau \Rightarrow \nu; l$*, then* $l : \tau \Leftrightarrow_\nu \cdot$*.*

H Source and view type normalization procedures

In this section we describe in more detail our source and view normalization procedures that attempt to simplify ambiguous regular expression types into unambiguous ones. Simplifying regular expressions and regular expression types into equivalent unambiguous expressions is often done by translating them into string and tree automata, respectively, applying standard automata algorithms to minize their numbers of transitions and states of the automata, and converting them back into equivalent regular language representations [10, 31]. The added difficulty in our case is that we not only want to normalize the types, but to produce a lens between the original and the normalized type that can convert between both, while preserving the additional markup present in values of the original ambiguous types.

We define our source normalization procedure *normS* in two phases. We first simplify the source type into an equivalent unambiguous type using miscellaneous automata techniques, check that the simplified type is unambiguous and then infer a corresponding lens between both types, as follows:

$$\frac{\text{simplify } \tau \text{ into } \tau_0 \quad \tau_0 \text{ unambiguous} \quad \tau \rightsquigarrow \tau_0 = l}{\mathit{normS}(\tau) = (l, \tau_0)}$$

We shall focus our attention on the new $\tau \rightsquigarrow \tau' = l$ step, that is the main emphasis of our approach. The algorithm expressed by the derivations from Figure 28 proceeds by attempting to identify leftmost atomic values in the original type and then invokes a "division" judgment $\tau/\beta = (l, \tau_R)$ that decomposes a type τ into a product β, τ_R.

To simplify our approach, in our algorithms we treat type variables as atomic types by defining $\beta ::= \alpha \mid X$ and assume that the simplification does not expand type variables (but still expect that the type is unambiguous in the usual sense). We also treat marked types $\boxed{\tau}$ in exactly the same way as the empty type. During derivations of the form $\tau \rightsquigarrow \tau'$, we maintain the invariants that τ' is unambiguous and that $\tau <: \tau'$ holds.

As a technical detail, in the derivation of $(\tau_1 \mid \tau_2), \tau_3 \rightsquigarrow \tau'$ we use the function *splitEmpty* to restructure a choice type $\tau_1 \mid \tau_2$ into an equivalent type $\tau_{ne} \mid \tau_e$ that decomposes the original type into a sum of non-empty choices τ_{ne} to the left and empty choices τ_e to the right. This does not affect the forward transformation of the lens, but ensures that the backward transformation of the lens always gives priority to consuming part of a target value when given a choice (since we know that for rules of the kind $\tau_1, \tau_2 \rightsquigarrow \tau_3$ the type τ_1 corresponds to the leftmost part of the view τ_3), instead of generating redundant markup.

Our view normalization procedure *normV* is much simpler and only attempts to remove some empty tags from the view (Figure 29), because the normalized type needs to preserve all the information in the original type, including additional markup for ambiguous types. In fact, when *normV* succeeds the normalized type is isomorphic (in terms of plain algebraic data types) to the original type. In practice, this is not overly restrictive for our examples, given that the view is always a product of variables and assuming that the type inferred from the view DTD is normalized.

Theorem H.1 (Source and update normalization soundness)**.**

1. *If* $normS(\tau) = (l, \tau')$*, then* $\tau =: \tau'$*,* τ' *is unambiguous and* $l : \tau \Leftrightarrow \tau'$*.*
2. *If* $\tau \rightsquigarrow \tau' = l$*,* τ' *is unambiguous and* $\tau =: \tau'$*, then* $l : \tau \Leftrightarrow \tau'$*.*
3. *If* $\tau/\beta = (l, \tau_R)$*, then* $l : (\beta, \tau_R) \Leftrightarrow \tau$ *and* $(\beta, \tau_R) <: \tau$*.*
4. *If* $normV(\tau) = (l, \tau')$*, then* $\tau := \tau'$*,* τ' *is unambiguous and* $l : \tau' \Leftrightarrow \tau$*.*

$$\boxed{\Gamma; \Pi \vdash \{\tau\}\ s\ \{\nu\} \Rightarrow l}$$

$$\frac{}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{skip}\ \{()\} \Rightarrow \mathsf{keep}}$$

$$\frac{}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{fail}\ \{\nu\} \Rightarrow \mathsf{bot}}$$

$$\frac{\begin{array}{c}\Gamma; \Pi_\tau \vdash \{\tau\}\ b_1\ \{\nu_1\} \Rightarrow l_1 \quad \Gamma; \Pi_\tau \vdash \{\tau\}\ b_2\ \{\nu_2\} \Rightarrow l_2 \\ \mathit{split}(\nu, \mathit{vars}(b_1), \mathit{vars}(b_2)) = (\nu_1, \nu_2, l_{12})\end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ b_1; b_2\ \{\nu\} \Rightarrow \mathsf{unfork}\ l_1\ l_2 \mathbin{\circ\!\!<} l_{12}}$$

$$\frac{\begin{array}{c}\Gamma; \Pi_\tau \vdash \{\tau\}\ b_1\ \{\nu\} \Rightarrow l_1 \quad \Gamma; \Pi_\tau \vdash \{\tau\}\ b_2\ \{\nu\} \Rightarrow l_2 \\ \Pi_\tau \vdash_S e : \mathtt{bool} \Rightarrow f\end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{ifS}\ e\ \mathtt{then}\ b_1\ \mathtt{else}\ b_2\ \{\nu\} \Rightarrow \mathsf{ifSthenelse}\ f\ l_1\ l_2}$$

$$\frac{\Pi_\tau \vdash_S \{\tau\}\ p\ \{\tau'\} \Rightarrow l_1 \quad \Gamma; \emptyset \vdash \{\tau'\}\ b\ \{\nu\} \Rightarrow l_2}{\Gamma; \Pi_\tau \vdash \{\tau\}\ p[b]\ \{\nu\} \Rightarrow l_1 \mathbin{\circ\!\!<} l_2}$$

$$\frac{\mathit{listifyS}(\tau) = (\tau_1, l_1) \quad \Gamma; \emptyset \vdash \{\tau_1\}\ b\ \{\nu\} \Rightarrow l_2}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{iter}\ b\ \{\nu\} \Rightarrow l_1 \mathbin{\circ\!\!<} \mathsf{map}\ l_2 \mathbin{\circ\!\!<} \mathsf{listeq}}$$

$$\frac{\begin{array}{c}\Gamma[x_V : \tau']; \Pi_\tau \vdash \{\tau\}\ b\ \{x_V : \tau'\} \Rightarrow l_1 \quad \Gamma \vdash_V \{\tau'\}\ e\ \{\nu\} \Rightarrow l_2 \\ f_1\ mv'_s\ v'_v\ \gamma' = \gamma[x_V := v'_v]\end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ [b]e\ \{\nu\} \Rightarrow \mathsf{withEnv}\ f_1\ l_1 \mathbin{\circ\!\!<} l_2}$$

$$\frac{\begin{array}{c}\Gamma; \Pi_\tau \vdash \{\tau\}\ b\ \{\nu_2\} \Rightarrow l_2 \quad \nu_2 \vdash_V e : \nu_1 \Rightarrow f_{21} \\ x \in \mathit{dom}(\nu) \quad \mathit{split}(\nu, \{x\}, \mathit{dom}(\nu) \backslash \{x\}) = (\nu_1, \nu_2, l_{12})\end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{view}\ x := e\ \mathtt{in}\ b\ \{\nu\} \Rightarrow l_2 \mathbin{\circ\!\!<} \mathsf{remfst}\ f_{21} \mathbin{\circ\!\!<} l_{12}}$$

$$\frac{\begin{array}{c}\mathit{listifyS}(\tau) = (\tau_1, l_1) \quad \mathit{listifyV}(\tau') = (\tau_2, l_2) \quad \Gamma[x_V : \tau_2]; \emptyset \vdash \{\tau_1\}\ b\ \{x_V : \tau_2\} \Rightarrow l \\ \emptyset \vdash_S e : \mathtt{bool} \Rightarrow f_e \quad \Gamma \vdash_{\mathtt{create}} \{\tau_2\}\ c\ \{\tau_1\} \Rightarrow \mathit{mf}_c \quad \Gamma \\ \vdash_{\mathtt{recover}} r\ \{\tau\} \Rightarrow f_r \quad f\ mv_s\ v_v\ \gamma = \gamma[x_V := v_v]\end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{alignpos}\ e\ b\ c\ r\ \{x_V : \tau'\} \Rightarrow l_1 \mathbin{\circ\!\!<} \mathtt{alignpos}\ f_e\ \mathit{mf}_c\ f_r\ (\mathsf{withEnv}\ f\ l) \mathbin{\circ\!\!<} l_2}$$

$$\frac{\Gamma; \Pi_\tau \vdash \{\tau\}\ b_1\ \{\nu\} \Rightarrow l_1 \quad \Gamma; \Pi_\tau \vdash \{\tau\}\ b_2\ \{\nu\} \Rightarrow l_2 \quad \nu \vdash_V e : \mathtt{bool} \Rightarrow f}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{ifV}\ e\ \mathtt{then}\ b_1\ \mathtt{else}\ b_2\ \{\nu\} \Rightarrow \mathsf{ifVthenelse}\ f\ l_1\ l_2}$$

$$\frac{\vdash \nu \leqslant \tau \Rightarrow l}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{replace}\ \{\nu\} \Rightarrow l}$$

$$\frac{\begin{array}{c}\Gamma; \Pi_\tau \vdash \{\tau\}\ b_1\ \{\nu\} \Rightarrow l_1 \quad \Gamma; \Pi_\tau \vdash \{\tau\}\ b_2\ \{\nu\} \Rightarrow l_2 \\ \Gamma; \tau \vdash e : \mathtt{bool} \Rightarrow \lambda\gamma\ v_s.b \quad f\ \cdot\ v_v = \mathtt{true} \quad f\ v_s\ v_v\ \gamma = b\end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{if}\ e\ \mathtt{then}\ b_1\ \mathtt{else}\ b_2\ \{\nu\} \Rightarrow \mathsf{ifthenelse}\ f\ l_1\ l_2}$$

$$\frac{\begin{array}{c}P(\vec{x} : \vec{\tau}) : \nu_s \Leftrightarrow \nu_v \in \Delta \quad \Pi_\tau \vdash_{\mathtt{procS}} \{\tau\}\ \vec{p_s}\ \{\nu_s\} \Rightarrow l_s \quad \Gamma \vdash_{\mathtt{procV}} \{\nu_v\}\ \vec{e_v}\ \{\nu\} \Rightarrow l_v \\ \Gamma; \tau \vdash e : \tau_1 \Rightarrow \lambda\gamma\ v.v_1 \quad \ldots \quad \Gamma; \tau \vdash e : \tau_n \Rightarrow \lambda\gamma\ v.v_n \quad f\ v_s\ v_v\ \gamma = \{x_1 := v_1, \ldots, x_n := v_n\}\end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ P(\vec{p_s}, \vec{e_v}, \vec{e})\ \{\nu\} \Rightarrow l_s \mathbin{\circ\!\!<} \mathsf{withEnv}\ f\ P \mathbin{\circ\!\!<} l_v}$$

Figure 11: Bidirectional update well-formedness and semantics I (Complete definitions in Appendix E).

$$\boxed{\Gamma \vdash_{\texttt{create}} \{\tau_2\}\ c\ \{\tau_1'\} \Rightarrow \lambda_m \gamma\ v_2.v_1} \qquad \boxed{\Gamma \vdash_{\texttt{recover}} r\ \{\tau\} \Rightarrow \lambda\gamma\ v.mv}$$

$$\frac{\Gamma[x_V : \tau] \vdash \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma[x_V := v]\ v.v'}{\Gamma \vdash_{\texttt{create}} \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}$$

$$\frac{\Gamma \vdash \{\tau\}\ u\ \{\tau\} \Rightarrow \lambda\gamma\ v.v'}{\Gamma \vdash_{\texttt{recover}} \texttt{keep}\ u\ \{\tau\} \Rightarrow \lambda\gamma\ v.v'}$$

$$\frac{}{\Gamma \vdash_{\texttt{recover}} \texttt{delete}\ u\ \{\tau\} \Rightarrow \lambda\gamma\ v.\cdot}$$

$$\frac{}{\Gamma \vdash_{\texttt{create}} \{\tau\}\ \cdot\ \{\tau'\} \Rightarrow \cdot}$$

$$\frac{\Gamma[x : \tau]; x \vdash e : \texttt{bool} \Rightarrow \lambda\gamma[x := v].\texttt{true} \qquad \Gamma \vdash_{\texttt{recover}} r\ \{\tau\} \Rightarrow \lambda\gamma\ v.mv}{\Gamma \vdash_{\texttt{recover}} \texttt{if}\ e\ \texttt{then}\ r\ \texttt{else}\ r'\ \{\tau\} \Rightarrow \lambda\gamma\ v.mv}$$

$$\frac{\Gamma[x : \tau]; x \vdash e : \texttt{bool} \Rightarrow \lambda\gamma[x := v].\texttt{false} \qquad \Gamma \vdash_{\texttt{recover}} r'\ \{\tau\} \Rightarrow \lambda\gamma\ v.mv}{\Gamma \vdash_{\texttt{recover}} \texttt{if}\ e\ \texttt{then}\ r\ \texttt{else}\ r'\ \{\tau\} \Rightarrow \lambda\gamma\ v.mv}$$

Figure 12: Bidirectional update well-formedness and semantics II (Complete definitions in Appendix E).

$$\llbracket u \;\texttt{WHERE}\; cs \rrbracket^b_{Stmt} = \llbracket u \rrbracket^b_{Upd}(\llbracket cs \rrbracket_{Conds})$$
$$\llbracket pat \;\texttt{IN}\; p \rrbracket^{\texttt{iter}}_{PatPath}(e_S, b) = (p \mathbin{/} \texttt{where}\; (\texttt{let}\; pat = \texttt{self}\; \texttt{in}\; e_S))[\texttt{iter}\; (\texttt{caseS}\; \texttt{self}\; \texttt{of}\; pat \to b)]$$
$$\llbracket s; s' \rrbracket^b_{Stmt} = \llbracket s \rrbracket^b_{Stmt}; \llbracket s' \rrbracket^b_{Stmt}$$
$$\llbracket pat \;\texttt{IN}\; p \rrbracket^{S}_{PatPath}(e_S, b) = (p \mathbin{/} \texttt{where}\; (\texttt{let}\; pat = \texttt{self}\; \texttt{in}\; e_S))[\texttt{caseS}\; \texttt{self}\; \texttt{of}\; pat \to b]$$

$$\begin{aligned}
\llbracket \texttt{DELETE}\; patp \rrbracket^b_{Upd}(e_S, \texttt{true}, \cdot) &= \llbracket patp \rrbracket^S_{PatPath}(e_S, [\texttt{replace}]())\\
\llbracket \texttt{DELETE FROM}\; patp \rrbracket^b_{Upd}(e_S, \texttt{true}, \cdot) &= \llbracket patp \rrbracket^S_{PatPath}(e_S, \texttt{child}[[\texttt{replace}]()])\\
\llbracket \texttt{REPLACE}\; patp \;\texttt{IN}\; p \;\texttt{WITH}\; e' \rrbracket^b_{Upd}(e_S, e_V, \vec{x} = \vec{e}) &= \llbracket patp \rrbracket^{\texttt{iter}}_{PatPath}(e_S, \texttt{view}\; \vec{x} := \vec{e}\; \texttt{in}\\
&\qquad \texttt{ifV}\; e_V\; \texttt{then}\; [\texttt{replace}]e'\; \texttt{else}\; \texttt{fail})\\
\llbracket \texttt{REPLACE IN}\; patp \;\texttt{WITH}\; e' \rrbracket^b_{Upd}(e_S, e_V, \vec{x} = \vec{e}) &= \llbracket patp \rrbracket^{\texttt{iter}}_{PatPath}(e_S, \texttt{view}\; \vec{x} := \vec{e}\; \texttt{in}\\
&\qquad \texttt{ifV}\; e_V\; \texttt{then}\; \texttt{child}[[\texttt{replace}]e']\; \texttt{else}\; \texttt{fail})\\
\llbracket \texttt{UPDATE}\; patp \;\texttt{BY}\; s \rrbracket^b_{Upd}(e_S, e_V, \vec{x} = \vec{e}) &= \llbracket patp \rrbracket^{\texttt{iter}}_{PatPath}(e_S, \texttt{view}\; \vec{x} := \vec{e}\; \texttt{in}\\
&\qquad \texttt{ifV}\; e_V\; \texttt{then}\; \llbracket s \rrbracket^b_{Stmt}\; \texttt{else}\; \texttt{fail})
\end{aligned}$$

$$\llbracket \texttt{UPDATE}\; pat \;\texttt{IN}\; p \;\texttt{BY}\; vs \;\texttt{FOR VIEW}\; pat' \;\texttt{IN}\; p' \;\texttt{MATCHING SOURCE BY}\; p_s \;\texttt{VIEW BY}\; p_v \rrbracket^b_{Upd}($$
$$e_S, e_V, \vec{x} = \vec{e}) = p[[b]p'] \;\textbf{where}$$

$$((s_{SV}, ms_V, ms_S), {p_s}', {p_v}') = (splitVStmt(vs), \texttt{case self of}\; pat \to p_s, \texttt{case self of}\; pat' \to p_v)$$
$$b = \texttt{alignkey}\; (\texttt{case self of}\; pat \to e_S)\; p'_s\; p'_v\; \llbracket s_{SV} \rrbracket^b_{Stmt}\; \llbracket ms_V \rrbracket^c_{MStmt}(pat')\; \llbracket ms_S \rrbracket^r_{MStmt}(pat)$$

Figure 13: BiFluX bidirectional statement normalization.

$$\boxed{\Gamma; mx \vdash\ e : \tau \Rightarrow \lambda\gamma.v}$$

$$\frac{\Gamma; mx \vdash\ e : \tau \Rightarrow \lambda\gamma.v}{\Gamma; mx \vdash\ e : \texttt{bool} \Rightarrow \lambda\gamma.boolean_{\tau}(v)} \qquad \frac{\Gamma; mx \vdash\ e : \tau \Rightarrow \lambda\gamma.v \quad \vdash \tau <: \tau' \Rightarrow c}{\Gamma; mx \vdash\ e : \tau' \Rightarrow \lambda\gamma.ucast\ c\ v}$$

$$\frac{\Gamma; mx \vdash\ e : \tau \Rightarrow \lambda\gamma.v}{\Gamma; mx \vdash\ n[e] : n[\tau] \Rightarrow \lambda\gamma.n[v]} \qquad \frac{\Gamma; mx \vdash\ e_1 : \tau_1 \Rightarrow \lambda\gamma.v_1 \quad \Gamma; mx \vdash\ e_2 : \tau_2 \Rightarrow \lambda\gamma.v_2}{\Gamma; mx \vdash\ e_1, e_2 : \tau_1, \tau_2 \Rightarrow \lambda\gamma.(v_1, v_2)}$$

$$\frac{\begin{array}{c}\Gamma; mx \vdash\ e_1 : \tau_1 \quad \vdash\ pat : \tau_{pat} \Rightarrow \Pi_{\tau_{pat}} \quad \vdash\ \tau_1 <: \tau_{pat} \Rightarrow c \\ \Gamma \cup \Pi_{pat} \vdash\ e_2 : \tau_2 \Rightarrow \lambda\gamma \cup \gamma_{ucast_c\ v_1, \Pi_{\tau_{pat}}}.v_2\end{array}}{\Gamma; mx \vdash \texttt{let}\ pat = e_1\ \texttt{in}\ e_2 \colon \Rightarrow \lambda\gamma.v_2}$$

$$\frac{\Gamma; mx \vdash\ e_1 : \tau_1 \Rightarrow \lambda\gamma.v_1 \quad \Gamma; mx \vdash\ e_2 : \tau_2 \Rightarrow \lambda\gamma.v_2 \quad \tau_1 =: \tau_2}{\Gamma; mx \vdash\ e_1 = e_2 : \texttt{bool} \Rightarrow \lambda\gamma.v_1 \sim v_2}$$

$$\frac{\Gamma; mx \vdash\ e : \tau \Rightarrow \lambda\gamma.v}{\Gamma; mx \vdash\ boolean(e) : \texttt{bool} \Rightarrow \lambda\gamma.boolean_{\tau}(v)} \qquad \frac{\Gamma; mx \vdash\ e : () \Rightarrow \lambda\gamma.()}{\Gamma; mx \vdash \texttt{where}\ e : () \Rightarrow \lambda\gamma.()}$$

$$\frac{\Gamma; mx \vdash\ e : \texttt{bool} \Rightarrow \lambda\gamma.v \quad \Gamma; mx \vdash\ e_1 : \tau_1 \Rightarrow \lambda\gamma.v_1 \quad \Gamma; mx \vdash\ e_2 : \tau_2 \Rightarrow \lambda\gamma.v_2}{\Gamma; mx \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau_1 \mid \tau_2 \Rightarrow \lambda\gamma.\ \textbf{if}\ v\ \textbf{then}\ L\ v_1\ \textbf{else}\ R\ v_2}$$

$$\frac{\Gamma; mx \vdash\ e : \tau \Rightarrow \lambda\gamma.v \quad \Gamma; mx \vdash_{\texttt{for}} x\ \texttt{in}\ \tau \rightarrow e : \tau' \Rightarrow \lambda\gamma\ v.v'}{\Gamma; mx \vdash\ \texttt{for}\ x\ \texttt{in}\ e\ \texttt{return}\ e' : \tau' \Rightarrow \lambda\gamma.v'}$$

$$\frac{\Gamma; mx \vdash\ \texttt{if}\ e\ \texttt{then}\ \texttt{self}\ \texttt{else}\ () : \tau \Rightarrow \lambda\gamma.v}{\Gamma; mx \vdash \texttt{where}\ e : () \Rightarrow \lambda\gamma.v}$$

$$\frac{}{\Gamma; mx \vdash\ () : () \Rightarrow \lambda\gamma.()} \qquad \frac{}{\Gamma; mx \vdash\ w : \texttt{string} \Rightarrow \lambda\gamma.w} \qquad \frac{}{\Gamma; mx \vdash \texttt{true} : \texttt{bool} \Rightarrow \lambda\gamma.\texttt{true}}$$

$$\frac{}{\Gamma; mx \vdash \texttt{false} : \texttt{bool} \Rightarrow \lambda\gamma.\texttt{false}} \qquad \frac{x \in dom(\Gamma)}{\Gamma; mx \vdash\ x : \Gamma(x) \Rightarrow \lambda\gamma.\gamma(x)} \qquad \frac{\Gamma; x \vdash\ x : \tau \Rightarrow \lambda\gamma.v}{\Gamma; x \vdash\ \texttt{self} : \tau \Rightarrow \lambda\gamma.v}$$

$$\frac{\Gamma; x \vdash\ x : n[\tau] \Rightarrow \lambda\gamma.n[v]}{\Gamma; x \vdash\ \texttt{child} : \tau \Rightarrow \lambda\gamma.v} \qquad \frac{\Gamma; x \vdash\ x : \alpha \Rightarrow \lambda\gamma.t \quad \alpha <: nt}{\Gamma; x \vdash :: nt : \alpha \Rightarrow \lambda\gamma.t}$$

$$\frac{\Gamma; x \vdash\ x : \alpha \Rightarrow \lambda\gamma.t \quad \alpha \not<: nt}{\Gamma; x \vdash :: nt : () \Rightarrow \lambda\gamma.()} \qquad \frac{\Gamma; x \vdash\ x : \tau \Rightarrow \lambda\gamma.v \quad \Gamma; \tau \vdash\ p : \tau' \Rightarrow \lambda\gamma\ v.v'}{\Gamma; mx \vdash\ x\ /\ p : () \Rightarrow \lambda\gamma.v}$$

$$\frac{\begin{array}{c}\Gamma; mx \vdash\ p_1 : \tau_1 \Rightarrow \lambda\gamma.v_1 \quad x \notin dom(\Gamma) \\ \Gamma; mx \vdash_{\texttt{for}} x\ \texttt{in}\ \tau_1 \rightarrow x\ /\ p_2 : \tau_2 \Rightarrow \lambda\gamma\ v_1.v_2\end{array}}{\Gamma; mx \vdash\ p_1\ /\ p_2 : \tau_2 \Rightarrow \lambda\gamma.v_2}$$

Figure 14: Expression and path well-formedness and semantics.(I)

$$\boxed{\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \tau \rightarrow e : \tau' \Rightarrow \lambda\gamma\; v.v'}$$

$$\frac{}{\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; () \rightarrow e : () \Rightarrow \lambda\gamma\; ().()} \quad \frac{\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \tau \rightarrow e : \tau' \Rightarrow \lambda\gamma\; v_i.v_i'}{\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \tau^* \rightarrow e : \tau'^* \Rightarrow \lambda\gamma\; [v_1, \ldots, v_n].[v_1', \ldots, v_n']}$$

$$\frac{\begin{array}{c}\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \tau_1 \rightarrow e : \tau_1' \Rightarrow \lambda\gamma\; v_1.v_1' \\ \Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \tau_2 \rightarrow e : \tau_2' \Rightarrow \lambda\gamma\; v_2.v_2'\end{array}}{\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \tau_1, \tau_2 \rightarrow e : \tau_1', \tau_2' \Rightarrow \lambda\gamma\; (v_1, v_2).(v_1', v_2')} \quad \frac{\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; E(X) \rightarrow e : \tau' \Rightarrow \lambda\gamma\; v.v'}{\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; X \rightarrow e : \tau' \Rightarrow \lambda\gamma\; v.v'}$$

$$\frac{\begin{array}{c}\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \tau_1 \rightarrow e : \tau_1' \Rightarrow \lambda\gamma\; v_1.v_1' \\ \Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \tau_2 \rightarrow e : \tau_2' \Rightarrow \lambda\gamma\; v_2.v_2'\end{array}}{\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \tau_1 \mid \tau_2 \rightarrow e : \tau_1' \mid \tau_2' \Rightarrow \lambda\gamma\; v.\;\textbf{case}\; v \;\textbf{of}\; \{L\; v_1 \rightarrow L\; v_1'; R\; v_2 \rightarrow R\; v_2'\}}$$

$$\frac{\Gamma[x : \alpha]; mx \vdash\; e : \tau' \Rightarrow \lambda\gamma[x := t].v'}{\Gamma; mx \vdash_{\texttt{for}} x \;\texttt{in}\; \alpha \rightarrow e : \tau' \Rightarrow \lambda\gamma\; t.v'}$$

Figure 15: Expression and path well-formedness and semantics.(II)

$$\boxed{\vdash\; pat : \tau \Rightarrow \Pi_\tau}$$

$$\frac{}{\vdash\; \tau : \tau \Rightarrow \emptyset} \quad \frac{}{\vdash\; () : () \Rightarrow \emptyset}$$

$$\frac{}{\vdash\; x \;\textsf{as}\; \tau : \tau \Rightarrow \{x := (\mathsf{id}, \tau)\}} \quad \frac{\vdash\; pat : \tau \Rightarrow \Pi_\tau}{\vdash\; n[pat] : n[\tau] \Rightarrow \mathsf{in} \circ\!\!< \Pi_\tau}$$

$$\frac{\vdash\; pat_1 : \tau_1 \Rightarrow \Pi_{\tau_1} \quad \vdash\; pat_2 : \tau_2 \Rightarrow \Pi_{\tau_2}}{\vdash\; pat_1, pat_2 : \tau_1, \tau_2 \Rightarrow \mathsf{keepfst} \circ\!\!< \Pi_{\tau_1} \cup \mathsf{keepsnd} \circ\!\!< \Pi_{\tau_2}}$$

Figure 16: Pattern type inference.

$$\boxed{\Gamma \vdash \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}$$

$$\frac{\Gamma \vdash \{\tau_1\}\ u\ \{\tau_2'\} \Rightarrow \lambda\gamma\ v_1.v_2' \quad \vdash \tau_2' <: \tau_2 \Rightarrow c}{\Gamma \vdash \{\tau_1\}\ u\ \{\tau_2\} \Rightarrow \lambda\gamma\ v_1.ucast_c\ v_2'}$$

$$\frac{\Gamma \vdash \{\tau_1\}\ u\ \{\tau_2\} \Rightarrow \lambda\gamma\ v_1.v_2 \quad \Gamma \vdash \{\tau_2\}\ u'\ \{\tau_3\} \Rightarrow \lambda\gamma\ v_2.v_3}{\Gamma \vdash \{\tau_1\}\ u; u'\ \{\tau_3\} \Rightarrow \lambda\gamma\ v_1.v_3} \qquad \frac{}{\Gamma \vdash \{\tau_1\}\ \texttt{skip}\ \{\tau_1\} \Rightarrow \lambda\gamma\ v_1.v_1}$$

$$\frac{\Gamma \vdash \{()\}\ u\ \{\tau_2\} \Rightarrow \lambda\gamma\ ().v_2}{\Gamma \vdash \{\tau_1\}\ \texttt{left}[u]\ \{\tau_2, \tau_1\} \Rightarrow \lambda\gamma\ v_1.(v_2, v_1)} \qquad \frac{\Gamma \vdash \{()\}\ u\ \{\tau_2\} \Rightarrow \lambda\gamma\ ().v_2}{\Gamma \vdash \{\tau_1\}\ \texttt{right}[u]\ \{\tau_1, \tau_2\} \Rightarrow \lambda\gamma\ v_1.(v_1, v_2)}$$

$$\frac{\Gamma \vdash \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}{\Gamma \vdash \{n[\tau]\}\ \texttt{children}[u]\ \{n[\tau']\} \Rightarrow \lambda\gamma\ n[v].n[v']} \qquad \frac{\Gamma; \cdot \vdash e : \tau \Rightarrow \lambda\gamma.v}{\Gamma \vdash \{()\}\ \texttt{insert}\ e\ \{\tau\} \Rightarrow \lambda\gamma\ ().v}$$

$$\frac{}{\Gamma \vdash \{\tau\}\ \texttt{delete}\ \{()\} \Rightarrow \lambda\gamma\ v.()}$$

$$\frac{\Gamma; \tau \vdash e : \texttt{bool} \Rightarrow \lambda\gamma\ v.v' \quad \Gamma \vdash \{\tau\}\ u\ \{\tau_1\} \Rightarrow \lambda\gamma\ v.v_1 \quad \Gamma \vdash \{\tau\}\ u\ \{\tau_2\} \Rightarrow \lambda\gamma\ v.v_2}{\Gamma \vdash \{\tau\}\ \texttt{if}\ e\ \texttt{then}\ u\ \texttt{else}\ u'\ \{\tau_1 \mid \tau_2\} \Rightarrow \lambda\gamma\ v.\ \mathbf{if}\ v'\ \mathbf{then}\ L\ v_1\ \mathbf{else}\ R\ v_2}$$

$$\frac{\begin{array}{c}\Gamma; \tau \vdash e : \tau' \Rightarrow \lambda\gamma\ v.v' \quad \vdash pat_1 : \tau_1 \Rightarrow \Pi_{\tau_1} \quad \vdash pat_n : \tau_n \Rightarrow \Pi_{\tau_n} \quad \tau' <: \tau_1 \mid \ldots \mid \tau_n \Rightarrow c \\ \Gamma; \Pi_{\tau_1}, \ldots, \Pi_{\tau_n} \vdash_{\texttt{case}} \{\tau'\}\ \{\tau_1 \mid \ldots \mid \tau_n\}\ \vec{u}\ \{\tau''\} \Rightarrow \lambda\gamma\ v'\ (ucast_c\ v').v''\end{array}}{\Gamma \vdash \{\tau\}\ \texttt{case}\ e\ \texttt{of}\ \vec{pat} \to \vec{u}\ \{\,\} \Rightarrow \lambda\gamma\ v.v''}$$

$$\frac{\Gamma \vdash \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}{\Gamma \vdash \{\tau\}\ \texttt{self}[u]\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'} \qquad \frac{\Gamma \vdash_{\texttt{iter}} \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}{\Gamma \vdash \{n[\tau]\}\ \texttt{child}[u]\ \{\tau'\} \Rightarrow \lambda\gamma\ n[v].n[v']}$$

$$\frac{\Gamma \vdash \{\tau\}\ p[p'[u]]\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}{\Gamma \vdash \{\tau\}\ p\ /\ p'[u]\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'} \qquad \frac{\alpha <: nt \quad \Gamma \vdash \{\tau\}\ u\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}{\Gamma \vdash \{\alpha\}\ (::nt)[u]\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}$$

$$\frac{\alpha \not<: nt}{\Gamma \vdash \{\alpha\}\ (::nt)[u]\ \{\tau\} \Rightarrow \lambda\gamma\ v.v} \qquad \frac{\Gamma \vdash \{\tau\}\ \texttt{if}\ e\ \texttt{then}\ u\ \texttt{else skip}\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}{\Gamma \vdash \{\tau\}\ (\texttt{where}\ e)[u]\ \{\tau'\} \Rightarrow \lambda\gamma\ v.v'}$$

Figure 17: In-place update well-formedness and semantics.(I)

$$\boxed{\Gamma;\Pi_{\tau_1},\ldots,\Pi_{\tau_n}\vdash_{\mathtt{case}}\{\tau\}\ \{\tau_1\mid\ldots\mid\tau_n\}\ \vec{u}\ \{\tau'\}\Rightarrow\lambda\gamma\ v\ v_{1n}.v'}$$

$$\frac{\Gamma\cup\Pi_{\tau_1}\ \vdash\ \{\tau\}\ u_1\ \{\tau_1\}\Rightarrow\lambda\gamma_{v,\Pi_{\tau_1}}\ v.v_1'}{\Gamma;\Pi_{\tau_1};f\vdash_{\mathtt{case}}\{\tau\}\ \{\tau_1\}\ \vec{u}\ \{\tau'\}\Rightarrow\lambda\gamma\ v\ v_1.v_1'}$$

$$\frac{\begin{array}{c}\Gamma;\Pi_{\tau_1}\vdash_{\mathtt{case}}\{\tau\}\ \{\tau_1\}\ u_1\ \{\tau_1'\}\Rightarrow\lambda\gamma\ v\ v_1.v_1'\\ \Gamma;\Pi_{\tau_2},\ldots,\Pi_{\tau_n}\vdash_{\mathtt{case}}\{\tau\}\ \{\tau_2\mid\ldots\mid\tau_n\}\ u_2,\ldots,u_n\ \{\tau_2'\}\Rightarrow\lambda\gamma\ v\ v_2.v_2'\end{array}}{\begin{array}{c}\Gamma;\Pi_{\tau_1},\Pi_{\tau_2},\ldots,\Pi_{\tau_n};f\vdash_{\mathtt{case}}\{\tau\}\ \{\tau_1\mid\tau_2\mid\ldots\mid\tau_n\}\ \vec{u}\ \{\tau'\}\Rightarrow\\ \lambda\gamma\ v\ v_{1n}.\mathtt{case}\ v_{1n}\ \mathtt{of}\ \{L\ v_1\rightarrow v_1';R\ v_2\rightarrow v_2'\}\end{array}}$$

$$\boxed{\Gamma\vdash_{\mathtt{iter}}\{\tau\}\ u\ \{\tau'\}\Rightarrow\lambda\gamma\ v.v'}$$

$$\frac{}{\Gamma\vdash_{\mathtt{iter}}\{()\}\ u\ \{()\}\Rightarrow\lambda\gamma\ ().()}\qquad\frac{\Gamma\vdash_{\mathtt{iter}}\{\tau\}\ u\ \{\tau'\}\Rightarrow\lambda\gamma\ v_i.v_i'}{\Gamma\vdash_{\mathtt{iter}}\{\tau^*\}\ u\ \{\tau'^*\}\Rightarrow\lambda\gamma\ [v_1,\ldots,v_n].[v_1',\ldots,v_n']}$$

$$\frac{\Gamma\vdash_{\mathtt{iter}}\{\tau_1\}\ u\ \{\tau_1'\}\Rightarrow\lambda\gamma\ v_1.v_1'\quad\Gamma\vdash_{\mathtt{iter}}\{\tau_2\}\ u\ \{\tau_2'\}\Rightarrow\lambda\gamma\ v_2.v_2'}{\Gamma\vdash_{\mathtt{iter}}\{\tau_1,\tau_2\}\ u\ \{\tau_1',\tau_2'\}\Rightarrow\lambda\gamma\ (v_1,v_2).(v_1',v_2')}\qquad\frac{\Gamma\ \vdash\ \{\alpha\}\ u\ \{\tau\}\Rightarrow\lambda\gamma\ t.v}{\Gamma\vdash_{\mathtt{iter}}\{\alpha\}\ u\ \{\tau\}\Rightarrow\lambda\gamma\ t.v}$$

$$\frac{\Gamma\vdash_{\mathtt{iter}}\{\tau_1\}\ u\ \{\tau_1'\}\Rightarrow\lambda\gamma\ v_1.v_1'\quad\Gamma\vdash_{\mathtt{iter}}\{\tau_2\}\ u\ \{\tau_2'\}\Rightarrow\lambda\gamma\ v_2.v_2'}{\Gamma\vdash_{\mathtt{iter}}\{\tau_1\mid\tau_2\}\ u\ \{\tau_1'\mid\tau_2'\}\Rightarrow\lambda\gamma\ v.\ \mathbf{case}\ v\ \mathbf{of}\ \{L\ v_1\rightarrow L\ v_1';R\ v_2\rightarrow R\ v_2'\}}$$

$$\frac{\Gamma\vdash_{\mathtt{iter}}\{E(X)\}\ u\ \{\tau'\}\Rightarrow\lambda\gamma\ v.v'}{\Gamma\vdash_{\mathtt{iter}}\{X\}\ u\ \{\tau'\}\Rightarrow\lambda\gamma\ v.v'}$$

Figure 18: In-place update well-formedness and semantics.(II)

$$
\begin{array}{rl}
\mathsf{out} & : \tau \Leftrightarrow_\Gamma n[\tau] \\
\mathsf{remsndone} & : \tau \Leftrightarrow_\Gamma (\tau, ()) \\
\mathsf{param} & : (\tau_1 \to a) \to (a \to \tau_1 \Leftrightarrow_\Gamma \tau_2) \to (\tau_1 \Leftrightarrow_\Gamma \tau_2) \\
\mathsf{remfstone} & : \tau \Leftrightarrow_\Gamma ((), \tau) \\
\otimes & : (\tau_1 \Leftrightarrow_\Gamma \tau_3) \to (\tau_2 \Leftrightarrow_\Gamma \tau_4) \to ((\tau_1, \tau_2) \Leftrightarrow_\Gamma (\tau_3, \tau_4)) \\
\mathsf{assocl} & : ((\tau_1, \tau_2), \tau_3) \Leftrightarrow_\Gamma (\tau_1, (\tau_2, \tau_3)) \\
\oplus & : (\tau_1 \Leftrightarrow_\Gamma \tau_3) \to (\tau_2 \Leftrightarrow_\Gamma \tau_4) \to ((\tau_1 \mid \tau_2) \Leftrightarrow_\Gamma (\tau_3 \mid \tau_4)) \\
\mathsf{assocr} & : (\tau_1, (\tau_2, \tau_3)) \Leftrightarrow_\Gamma ((\tau_1, \tau_2), \tau_3) \\
\mathsf{eitherS} & : (\tau \to \mathtt{bool}) \to (\tau \Leftrightarrow_\Gamma (\tau \mid \tau)) \\
\mathsf{coswap} & : (\tau_1 \mid \tau_2) \Leftrightarrow_\Gamma (\tau_2 \mid \tau_1) \\
\uplus & : (\tau_1 \Leftrightarrow_\Gamma \tau_3) \to (\tau_2 \Leftrightarrow_\Gamma \tau_3) \to ((\tau_1 \mid \tau_2) \Leftrightarrow_\Gamma \tau_3) \\
\mathsf{distl} & : ((\tau_1, \tau_3) \mid (\tau_2, \tau_3)) \Leftrightarrow_\Gamma ((\tau_1 \mid \tau_2), \tau_3) \\
\mathsf{injl} & : (\tau_1 \mid \tau_2) \Leftrightarrow_\Gamma \tau_1 \\
\mathsf{undistl} & : ((\tau_1 \mid \tau_2), \tau_3) \Leftrightarrow_\Gamma ((\tau_1, \tau_3) \mid (\tau_2, \tau_3)) \\
\mathsf{uninjl} & : \tau_1 \Leftrightarrow_\Gamma (\tau_1 \mid \tau_2) \\
\mathsf{unnil} & : () \Leftrightarrow_\Gamma \tau^* \\
\mathsf{injr} & : (\tau_1 \mid \tau_2) \Leftrightarrow_\Gamma \tau_2 \\
\mathsf{uncons} & : (\tau, \tau^*) \Leftrightarrow_\Gamma \tau^* \\
\mathsf{uninjr} & : \tau_2 \Leftrightarrow_\Gamma (\tau_1 \mid \tau_2) \\
\mathsf{unwrap} & : \tau \Leftrightarrow_\Gamma \tau^* \\
\mathsf{ignore} & : \tau \to (() \Leftrightarrow_\Gamma \tau) \\
\mathsf{foldlist} & : ((() \mid (\tau_1, \tau_2)) \Leftrightarrow_\Gamma \tau_2) \to (\tau_1{}^* \Leftrightarrow_\Gamma \tau_2) \\
\mathsf{new} & : \tau_1 \to (\tau_1 \Leftrightarrow_\Gamma ()) \\
\mathsf{mergelist} & : (((\tau_1, \tau_3) \mid \tau_2) \Leftrightarrow_\Gamma \tau_3) \to ((\tau_1{}^*, \tau_2) \Leftrightarrow_\Gamma \tau_3) \\
\nabla & : (\tau \Leftrightarrow_\Gamma \tau_1) \to (\tau \Leftrightarrow_\Gamma \tau_2) \to (\tau \Leftrightarrow_\Gamma (\tau_1 \mid \tau_2))
\end{array}
$$

Figure 19: Language of put-based point-free lenses for translating core bidirectional updates.

$$
\begin{array}{ll}
C_{\mathsf{id}}(v) = () & C_{\mathsf{keep}}(v) = v \\
C_{\mathsf{in}}(v) = () & C_{\mathsf{keepfst}}(v_1, v_2) = v_1 \\
C_{\mathsf{out}}(v) = () & C_{\mathsf{keepsnd}}(v_1, v_2) = v_2 \\
C_{\mathsf{assocl}}(v) = () & C_{(l_1 \propto l_2)}(v) = (C_{l_1}(v), C_{l_2}(Q_{l_1}(v))) \\
C_{\mathsf{assocr}}(v) = () & C_{(l_1 \otimes l_2)}(v_1, v_2) = (C_{l_1}(v_1), C_{l_2}(v_2)) \\
C_{\mathsf{distl}}(v) = () & C_{(l_1 \oplus l_2)}(L\ v_1) = L(C_{l_1}(v_1)) \\
C_{\mathsf{undistl}}(v) = () & C_{(l_1 \oplus l_2)}(R\ v_2) = R(C_{l_2}(v_2)) \\
C_{\mathsf{injl}}(v) = () & C_{(\mathsf{eitherS}\ p\ l_1\ l_2)}(v) = \textbf{if } (p\ v) \textbf{ then } L\ C_{l_1}(v) \textbf{ else } R\ C_{l_2}(v) \\
C_{\mathsf{uninjl}}(v) = () & C_{(l_1 \bullet\!\nabla l_2)}(v) = \textbf{if } (v \in dom(Q_{l_1})) \textbf{ then } L\ C_{l_1}(v) \textbf{ else } R\ C_{l_2}(v) \\
C_{\mathsf{injr}}(v) = () & C_{(l_1 \uplus l_2)}(L\ v_1) = L\ C_{l_1}(v_1) \\
C_{\mathsf{uninjr}}(v) = () & C_{(l_1 \uplus l_2)}(R\ v12) = R\ C_{l_2}(v_2) \\
C_{\mathsf{unnil}}(v) = () & C_{(\mathsf{ignore}\ v')}(()) = () \\
C_{\mathsf{uncons}}(v) = () & C_{(\mathsf{ignore}\ v')}(v) = v \\
C_{\mathsf{unwrap}}(v) = () & C_{(\mathsf{foldlist}\ l)}(v) = C_{(\mathsf{in} \propto (id \oplus \mathsf{id} \otimes \mathsf{foldlist}\ l) \propto l)}(v) \\
C_{\mathsf{coswap}}(v) = () & \\
\multicolumn{2}{l}{C_{(\mathsf{mergelist}\ l)}(v) = C_{((\mathsf{in} \otimes \mathsf{id}) \propto \mathsf{undistl} \propto (\mathsf{keepfst} \oplus \mathsf{assocl}) \propto \mathsf{coswap} \propto (\mathsf{id} \otimes \mathsf{mergelist}\ l \oplus \mathsf{id}) \propto l)}(v)}
\end{array}
$$

Figure 20: Complement function.

$$\boxed{\Gamma;\Pi \vdash \{\tau\}\ s\ \{\nu\} \Rightarrow l}$$

$$\frac{\begin{array}{c} \mathit{listifyS}(\tau) = (\tau_1, l_1) \quad \mathit{listifyV}(\tau') = (\tau_2, l_2) \quad \Gamma[x_V : \tau_2]; \emptyset \vdash \{\tau_1\}\ b\ \{x_V : \tau_2\} \Rightarrow l \\ \Gamma; \tau_1 \vdash p_s : \tau_1' \Rightarrow f_1 \quad \Gamma; \tau_2 \vdash p_v : \tau_2' \Rightarrow f_2 \quad \tau_1 =: \tau_2 \quad \emptyset \vdash_S e : \mathtt{bool} \Rightarrow f_e \\ \Gamma \vdash_{\mathtt{create}} \{\tau_2\}\ c\ \{\tau_1\} \Rightarrow mf_c \quad \Gamma \vdash_{\mathtt{recover}} r\ \{\tau\} \Rightarrow f_r \quad f\ mv_s\ v_v\ \gamma = \gamma[x_V := v_v] \end{array}}{\begin{array}{c} \Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{alignkey}\ e\ b\ p_s\ p_v\ c\ r\ \{x_V : \tau'\} \Rightarrow \\ l_1 \circ\!< \mathtt{alignkey}\ f_1\ f_2\ (\lambda v_1\ v_2. v_1 \sim v_2)\ f_e\ mf_c\ f_r\ (\mathsf{withEnv}\ f\ l) \circ\!< l_2 \end{array}}$$

$$\frac{\begin{array}{c} \Pi_\tau \vdash_S \{\tau\}\ p\ \{\tau'\} \Rightarrow l_p \quad \vdash \mathit{pat}_1 : \tau_1 \Rightarrow \Pi_{\tau_1} \quad \ldots \quad \vdash \mathit{pat}_n : \tau_n \Rightarrow \Pi_{\tau_n} \\ \vdash \tau <: \tau_1 \mid \ldots \mid \tau_n \Rightarrow c \\ \Gamma; \Pi_\tau; \Pi_{\tau_1}, \ldots, \Pi_{\tau_n}; \mathit{ucast}_c \vdash_{\mathtt{caseS}} \{\tau\}\ \{\tau_1 \mid \ldots \mid \tau_n\}\ \vec{s}\ \{\nu\} \Rightarrow l \end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{caseS}\ p\ \mathtt{of}\ \vec{pat} \to \vec{s}\ \{\nu\} \Rightarrow l}$$

$$\frac{\begin{array}{c} \mathit{split}(\nu, \mathit{vars}(e), \mathit{vars}(\nu) \backslash \mathit{vars}(e)) = (\nu_1, \nu_2, l_{12}) \quad \Gamma \vdash_V \{\tau'\}\ e\ \{\nu_1\} \Rightarrow l_e \\ \vdash_V \mathit{pat}_1 : \tau_1 \Rightarrow \nu_1'; l_1' \quad \ldots \quad \vdash \mathit{pat}_n : \tau_n \Rightarrow \nu_n'; l_n' \quad \vdash \tau' \leqslant (\tau_1 \mid \ldots \mid \tau_n) \Rightarrow l' \\ \Gamma; \Pi_\tau \vdash_{\mathtt{caseV}} \{\tau\}\ \vec{s}\ \{\nu_2\}\ \{\nu_1' \mid \ldots \mid \nu_n'\} \Rightarrow l \end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{caseV}\ e\ \mathtt{of}\ \vec{pat} \to \vec{s}\ \{\nu\} \Rightarrow l \circ\!< ((l_1' \oplus \ldots \oplus l_n') \circ\!< l' \circ\!< l_e \otimes \mathsf{id})) \circ\!< l_{12}}$$

$$\frac{\begin{array}{c} \Gamma; \tau \vdash e : \tau' \Rightarrow \lambda\gamma\ v.v' \quad \vdash \mathit{pat}_1 : \tau_1 \Rightarrow \Pi_{\tau_1} \quad \ldots \quad \vdash \mathit{pat}_n : \tau_n \Rightarrow \Pi_{\tau_n} \\ \vdash \tau' <: \tau_1 \mid \ldots \mid \tau_n \Rightarrow c \quad \Gamma; \Pi_\tau; \Pi_{\tau_1}, \ldots, \Pi_{\tau_n}; \mathit{ucast}_c \vdash_{\mathtt{case}} \{\tau\}\ \{\tau_1 \mid \ldots \mid \tau_n\}\ \vec{s}\ \{\nu\} \Rightarrow l \end{array}}{\Gamma; \Pi_\tau \vdash \{\tau\}\ \mathtt{case}\ e\ \mathtt{of}\ \vec{pat} \to \vec{s}\ \{\nu\} \Rightarrow l}$$

$$\boxed{\Gamma; \Pi_\tau; \Pi_{\tau_1}, \ldots, \Pi_{\tau_n}; f \vdash_{\mathtt{caseS}} \{\tau\}\ \{\tau_1 \mid \ldots \mid \tau_n\}\ \vec{s}\ \{\nu\} \Rightarrow l}$$

$$\frac{\Gamma \cup \Pi_{\tau_1}; \Pi_\tau \cup \Pi_{\tau_1} \vdash \{\tau\}\ s_1\ \{\nu\} \Rightarrow l_1 \quad f_1\ v_s\ v_v\ \gamma = \gamma \cup \gamma_{v_s, \Pi_{\tau_1}}}{\Gamma; \Pi_\tau; \Pi_{\tau_1}; f \vdash_{\mathtt{caseS}} \{\tau\}\ \{\tau_1\}\ \vec{s}\ \{\nu\} \Rightarrow \mathsf{withEnv}\ f_1\ l_1}$$

$$\frac{\begin{array}{c} \Gamma; \Pi_\tau; \Pi_{\tau_1}; f_1 \vdash_{\mathtt{caseS}} \{\tau\}\ \{\tau_1\}\ s_1\ \{\nu\} \Rightarrow l_1 \quad f_1\ v_1 = \mathbf{case}\ f\ \gamma_1\ v_1\ \mathbf{of}\ L\ v_1' \to v_1' \\ \Gamma; \Pi_\tau; \Pi_{\tau_2}, \ldots, \Pi_{\tau_n}; f_2 \vdash_{\mathtt{caseS}} \{\tau\}\ \{\tau_2 \mid \ldots \mid \tau_n\}\ s_2, \ldots, s_n\ \{\nu\} \Rightarrow l_2 \quad f_2\ v_2 = \mathbf{case}\ f\ v_2\ \mathbf{of}\ L\ v_2' \to v_2' \\ f_{12}\ v_s = \mathbf{case}\ f\ v_s\ \mathbf{of}\ \{L\ v_1' \to \mathtt{true}, R\ v_2' \to \mathtt{false}\} \end{array}}{\Gamma; \Pi_\tau; \Pi_{\tau_1}, \Pi_{\tau_2}, \ldots, \Pi_{\tau_n}; f \vdash_{\mathtt{caseS}} \{\tau\}\ \{\tau_1 \mid \tau_2 \mid \ldots \mid \tau_n\}\ \vec{s}\ \{\nu\} \Rightarrow \mathsf{ifSthenelse}\ f_{12}\ l_1\ l_2}$$

$$\boxed{\Gamma; \Pi_\tau \vdash_{\mathtt{caseV}} \{\tau\}\ \vec{s}\ \{\nu\}\ \{\nu_1 \mid \ldots \mid \nu_n\} \Rightarrow l}$$

$$\frac{\Gamma \cup \Gamma_{\nu_1}; \Pi_\tau \vdash \{\tau\}\ s_1\ \{\nu_1, \nu\} \Rightarrow l_1 \quad f_1\ mv_s\ (v_1, v_v)\ \gamma = \gamma_{v_1 : \nu_1}}{\Gamma; \Pi_\tau \vdash_{\mathtt{caseV}} \{\tau\}\ \vec{s}\ \{\nu\}\ \{\nu_1\} \Rightarrow \mathsf{withEnv}\ f_1\ l_1 \circ\!< l}$$

$$\frac{\Gamma; \Pi_\tau \vdash_{\mathtt{caseV}} \{\tau\}\ s_1\ \{\nu\}\ \{\nu_1\} \Rightarrow l_1 \quad \Gamma; \Pi_\tau \vdash_{\mathtt{caseV}} \{\tau\}\ s_2, \ldots, s_n\ \{\nu\}\ \{\nu_2 \mid \ldots \mid \nu_n\} \Rightarrow l_2}{\Gamma; \Pi_\tau \vdash_{\mathtt{caseV}} \{\tau\}\ \vec{s}\ \{\nu\}\ \{\nu_1 \mid \nu_2 \mid \ldots \mid \nu_n\} \Rightarrow (l_1 \bullet\!\nabla\, l_2) \circ\!< \mathsf{distl}}$$

Figure 21: Statement well-formedness and lens semantics for key alignment, case expressions and procedure arguments.(I)

$$\boxed{\Gamma; \Pi_\tau; \Pi_{\tau_1}, \ldots, \Pi_{\tau_n}; f \vdash_{\mathsf{case}} \{\tau\}\ \{\tau_1 \mid \ldots \mid \tau_n\}\ \vec{s}\ \{\nu\} \Rightarrow l}$$

$$\frac{\Gamma \cup \Pi_{\tau_1}; \Pi_\tau \vdash \{\tau\}\ s_1\ \{\nu\} \Rightarrow l_1 \quad f_1\ v_s\ v_v\ \gamma = \gamma \cup \gamma_{f\ \gamma\ v_s, \Pi_{\tau_1}}}{\Gamma; \Pi_\tau; \Pi_{\tau_1}; f \vdash_{\mathsf{case}} \{\tau\}\ \{\tau_1\}\ \vec{s}\ \{\nu\} \Rightarrow \mathsf{withEnv}\ f_1\ l_1}$$

$$\frac{\begin{array}{c}\Gamma; \Pi_\tau; \Pi_{\tau_1}; f_1 \vdash_{\mathsf{case}} \{\tau\}\ \{\tau_1\}\ s_1\ \{\nu\} \Rightarrow l_1 \quad f_1\ v_1 = \mathbf{case}\ f\ v_1\ \mathbf{of}\ \{L\ v_1' \to v_1'\} \\ \Gamma; \Pi_\tau; \Pi_{\tau_2}, \ldots, \Pi_{\tau_n}; f_2 \vdash_{\mathsf{case}} \{\tau\}\ \{\tau_2 \mid \ldots \mid \tau_n\}\ s_2, \ldots, s_n\ \{\nu\} \Rightarrow l_2 \\ f_2\ v_2 = \mathbf{case}\ f\ v_2\ \mathbf{of}\ \{R\ v_2' \to v_2'\} \\ f_{12}\ v_s\ v_v\ \gamma = \mathbf{case}\ f\ v_s\ \mathbf{of}\ \{L\ v_1' \to \mathtt{true}, R\ v_2' \to \mathtt{false}\}\end{array}}{\Gamma; \Pi_\tau; \Pi_{\tau_1}, \Pi_{\tau_2}, \ldots, \Pi_{\tau_n}; f \vdash_{\mathsf{case}} \{\tau\}\ \{\tau_1 \mid \tau_2 \mid \ldots \mid \tau_n\}\ \vec{s}\ \{\nu\} \Rightarrow \mathsf{ifthenelse}\ f_{12}\ l_1\ l_2}$$

$$\boxed{\Pi_\tau \vdash_{\mathtt{procS}} \{\tau\}\ \vec{p}\ \{\nu_s\} \Rightarrow l}$$

$$\frac{\Pi_\tau \vdash_S \{\tau\}\ p_1\ \{\tau_1'\} \Rightarrow l_1 \quad \vdash \tau_1 \leqslant \tau_1' \Rightarrow l_1'}{\Pi_\tau \vdash_{\mathtt{procS}} \{\tau\}\ p_1\ \{x_1 : \tau_1\} \Rightarrow l_1 \circ\!\!< l_1'}$$

$$\frac{\Pi_\tau \vdash_{\mathtt{procS}} \{\tau\}\ p_1\ \{x_1 : \tau_1\} \Rightarrow l_1 \quad \Pi_\tau \vdash_{\mathtt{procS}} \{\tau\}\ p_2, \ldots, p_n\ \{\nu_2\} \Rightarrow l_2}{\Pi_\tau \vdash_{\mathtt{procS}} \{\tau\}\ p_1, p_2, \ldots, p_n\ \{x_1 : \tau_1, \nu_2\} \Rightarrow \mathsf{unfork}\ l_1\ l_2}$$

$$\boxed{\Gamma \vdash_{\mathtt{procV}} \{\nu_v\}\ \vec{e}\ \{\nu\} \Rightarrow l}$$

$$\frac{}{\Gamma \vdash_{\mathtt{procV}} \{()\}\ \cdot\ \{()\} \Rightarrow \mathsf{id}} \qquad \frac{\Gamma \vdash_V \{\tau_1'\}\ e_1\ \{\nu\} \Rightarrow l_1 \quad \vdash \tau_1' \leqslant \tau_1 \Rightarrow l_1'}{\Gamma \vdash_{\mathtt{procV}} \{x_1 : \tau_1\}\ e_1\ \{\nu\} \Rightarrow l_1' \circ\!\!< l_1}$$

$$\frac{\begin{array}{c} split(\nu, vars(e_1), vars(e_2, \ldots, e_n)) = (\nu_1', \nu_2', l_{12}) \quad \Gamma \vdash_{\mathtt{procV}} \{x_1 : \tau_1\}\ e_1\ \{\nu_1'\} \Rightarrow l_1 \\ \Gamma \vdash_{\mathtt{procV}} \{\nu_2\}\ e_2, \ldots, e_n\ \{\nu_2'\} \Rightarrow l_2 \end{array}}{\Gamma \vdash_{\mathtt{procV}} \{x_1 : \tau_1, \nu_2\}\ e_1, e_2, \ldots, e_n\ \{\nu\} \Rightarrow (l_1 \otimes l_2) \circ\!\!< l_{12}}$$

Figure 22: Statement well-formedness and lens semantics for key alignment, case expressions and procedure arguments.(II)

$$\boxed{\Gamma \vdash_{\texttt{recover}} r \; \{\tau\} \Rightarrow \lambda\gamma \; v.mv}$$

$$\frac{\begin{array}{c}\Gamma[x:\tau]; x \vdash e : \tau' \Rightarrow \lambda\gamma[x := v].v' \quad \vdash \\ pat_1 : \tau_1 \Rightarrow \Pi_{\tau_1} \quad \ldots \quad \vdash pat_n : \tau_n \Rightarrow \Pi_{\tau_n} \quad \vdash \tau' <: \tau_1 \mid \ldots \mid \tau_n \Rightarrow c \\ \Gamma; \Pi_{\tau_1}, \ldots, \Pi_{\tau_n} \vdash_{\texttt{case}} \{\tau_1 \mid \ldots \mid \tau_n\} \; \vec{r} \; \{\tau\} \Rightarrow \lambda\gamma \; v \; (ucast_c \; v').mv\end{array}}{\Gamma \vdash^{\texttt{case}}_{\texttt{recover}} \texttt{case} \; e \; \texttt{of} \; \vec{pat} \rightarrow \vec{r} \; \{\tau\} \Rightarrow \lambda\gamma \; v.mv}$$

$$\boxed{\Gamma; \Pi_{\tau_1}, \ldots, \Pi_{\tau_n} \vdash^{\texttt{case}}_{\texttt{recover}} \{\tau_1 \mid \ldots \mid \tau_n\} \; \vec{r} \; \{\tau\} \Rightarrow \lambda\gamma \; v \; v_{1n}.mv}$$

$$\frac{\Gamma \cup \Pi_{\tau_1} \vdash \{\tau\} \; r_1 \Rightarrow \lambda\gamma \cup \gamma_{v_1, \Pi_{\tau_1}} \; v.mv}{\Gamma; \Pi_{\tau_1} \vdash^{\texttt{case}}_{\texttt{recover}} \{\tau_1\} \; \vec{r} \; \{\tau\} \Rightarrow \lambda\gamma \; v_1 \; v.mv}$$

$$\frac{\begin{array}{c}\Gamma; \Pi_{\tau_1} \vdash^{\texttt{case}}_{\texttt{recover}} \{\tau_1\} \; r_1 \Rightarrow \lambda\gamma \; v \; v_1.mv' \\ \Gamma; \Pi_{\tau_2}, \ldots, \Pi_{\tau_n} \vdash^{\texttt{case}}_{\texttt{recover}} \{\tau_2 \mid \ldots \mid \tau_n\} \; r_2, \ldots, r_n \Rightarrow \lambda\gamma \; v_2 \; v.mv''\end{array}}{\begin{array}{c}\Gamma; \Pi_{\tau_1}, \Pi_{\tau_2}, \ldots, \Pi_{\tau_n} \vdash^{\texttt{case}}_{\texttt{recover}} \{\tau_1 \mid \tau_2 \mid \ldots \mid \tau_n\} \; \vec{r} \; \{\tau\} \Rightarrow \\ \lambda\gamma \; v \; v_{1n}.\; \textbf{case} \; v_{1n} \; \textbf{of} \; \{L \; v_1 \rightarrow mv'; R \; v_2 \rightarrow mv''\}\end{array}}$$

Figure 23: Recover statement well-formedness and semantics for case expressions.

$$\boxed{\Gamma \vdash_S \{\tau\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l}$$

$$\frac{}{\Gamma \vdash_S \{\tau\}\ \texttt{self}\ \{\tau\} \Rightarrow \lambda\gamma.\mathsf{id}} \quad \frac{}{\Gamma \vdash_S \{n[\tau]\}\ \texttt{child}\ \{\tau\} \Rightarrow \lambda\gamma.\mathsf{in}} \quad \frac{\alpha <: nt}{\Gamma \vdash_S \{\alpha\} :: nt\ \{\alpha\} \Rightarrow \lambda\gamma.\mathsf{id}}$$

$$\frac{\alpha \not<: nt}{\Gamma \vdash_S \{\alpha\} :: nt\ \{\ \boxed{\alpha}\ \} \Rightarrow \lambda\gamma.\mathsf{id}} \quad \frac{\Gamma; \tau \vdash e : () \Rightarrow \lambda\gamma\ v.()}{\Gamma \vdash_S \{\tau\}\ \texttt{where}\ e\ \{\ \boxed{\tau}\ \} \Rightarrow \lambda\gamma.\mathsf{id}}$$

$$\frac{\Gamma; \tau \vdash e : \mathsf{bool} \Rightarrow \lambda\gamma\ v.v_b}{\Gamma \vdash_S \{\tau\}\ \texttt{where}\ e\ \{\tau \mid \boxed{\tau}\ \} \Rightarrow \lambda\gamma.\mathsf{eitherS}\ (\lambda v.v_b)\ \mathsf{id}\ \mathsf{id}}$$

$$\frac{\Gamma \vdash_S \{\tau\}\ p_1\ \{\tau_1\} \Rightarrow \lambda\gamma.l_1 \quad \Gamma \vdash_S^{\texttt{iter}} \{\tau_1\}\ p_2\ \{\tau_2\} \Rightarrow \lambda\gamma.l_2}{\Gamma \vdash_S \{\tau\}\ p_1\ /\ p_2\ \{\tau'\} \Rightarrow \lambda\gamma.l_1 \circ\!\!< l_2} \quad \frac{x : \tau' \in \Gamma}{\Gamma \vdash_S \{\tau\}\ x\ \{\tau'\} \Rightarrow \lambda\gamma.\mathsf{keep} \circ\!\!< \mathsf{ignore}\ \gamma(x)}$$

$$\frac{}{\Gamma \vdash_S \{\tau\}\ w\ \{\texttt{string}\} \Rightarrow \lambda\gamma.\mathsf{keep} \circ\!\!< \mathsf{ignore}\ w} \quad \frac{}{\Gamma \vdash_S \{\tau\}\ b\ \{\texttt{bool}\} \Rightarrow \lambda\gamma.\mathsf{keep} \circ\!\!< \mathsf{ignore}\ b}$$

$$\boxed{\Gamma \vdash_S^{\texttt{iter}} \{\tau\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l}$$

$$\frac{}{\Gamma \vdash_S^{\texttt{iter}} \{()\}\ p\ \{()\} \Rightarrow \lambda\gamma.\mathsf{id}} \quad \frac{\Gamma \vdash_S \{\alpha\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l}{\Gamma \vdash_S^{\texttt{iter}} \{\alpha\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l} \quad \frac{\Gamma \vdash_S^{\texttt{iter}} \{\tau\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l}{\Gamma \vdash_S^{\texttt{iter}} \{\tau^*\}\ p\ \{\tau'^*\} \Rightarrow \lambda\gamma.\mathsf{map}\ l}$$

$$\frac{\begin{array}{c}\Gamma \vdash_S^{\texttt{iter}} \{\tau_1\}\ p\ \{\tau_1'\} \Rightarrow \lambda\gamma.l_1 \\ \Gamma \vdash_S^{\texttt{iter}} \{\tau_2\}\ p\ \{\tau_2'\} \Rightarrow \lambda\gamma.l_2\end{array}}{\Gamma \vdash_S^{\texttt{iter}} \{\tau_1, \tau_2\}\ p\ \{\tau_1', \tau_2'\} \Rightarrow \lambda\gamma.l_1 \otimes l_2} \quad \frac{\begin{array}{c}\Gamma \vdash_S^{\texttt{iter}} \{\tau_1\}\ p\ \{\tau_1'\} \Rightarrow \lambda\gamma.l_1 \\ \Gamma \vdash_S^{\texttt{iter}} \{\tau_2\}\ p\ \{\tau_2'\} \Rightarrow \lambda\gamma.l_2\end{array}}{\Gamma \vdash_S^{\texttt{iter}} \{\tau_1 \mid \tau_2\}\ p\ \{\tau_1' \mid \tau_2'\} \Rightarrow \lambda\gamma.l_1 \oplus l_2}$$

$$\frac{\Gamma \vdash_S^{\texttt{iter}} \{E(X)\}\ p\ \{\tau'\} \Rightarrow \lambda\gamma.l}{\Gamma \vdash_S^{\texttt{iter}} \{X\}\ p\ \{\tau'^*\} \Rightarrow \lambda\gamma.\mathsf{map}\ l}$$

Figure 24: Source path well-formedness and semantics as lenses.

$$\boxed{\Gamma \vdash_V \{\tau\}\ e\ \{\nu\} \Rightarrow l}$$

$$\frac{}{\Gamma \vdash_V \{()\}\ ()\ \{()\} \Rightarrow \mathsf{id}} \qquad \frac{\Gamma \vdash_V \{\tau\}\ e\ \{\nu\} \Rightarrow l}{\Gamma \vdash_V \{n[\tau]\}\ n[e]\ \{\nu\} \Rightarrow \mathsf{in} \circ\!< l} \qquad \frac{}{\Gamma \vdash_V \{\tau'\}\ x\ \{x : \tau'\} \Rightarrow \mathsf{id}}$$

$$\frac{\Gamma \vdash_V \{\tau\}\ p\ \{\tau'\} \Rightarrow l}{\Gamma \vdash_V \{\tau\}\ x\ /\ p\ \{x : \tau'\} \Rightarrow l} \qquad \frac{\begin{array}{c} \mathit{split}(\nu, \mathit{vars}(e_1), \mathit{vars}(e_2)) = (\nu_1, \nu_2, l_{12}) \\ \Gamma \vdash_V \{\tau_1\}\ e_1\ \{\nu_1\} \Rightarrow l_1 \quad \Gamma \vdash_V \{\tau_2\}\ e_2\ \{\nu_2\} \Rightarrow l_2 \end{array}}{\Gamma \vdash_V \{\tau_1, \tau_2\}\ e_1, e_2\ \{\nu\} \Rightarrow (l_1 \otimes l_2) \circ\!< l_{12}}$$

$$\frac{\begin{array}{c} \mathit{split}(\nu, \mathit{vars}(e), \mathit{vars}(\nu) \backslash \mathit{vars}(e)) = (\nu_1, \nu_2, l_{12}) \\ \Gamma \vdash_V \{\tau_1\}\ e\ \{\nu_1\} \Rightarrow l_1 \quad \vdash_V \mathit{pat} : \tau_1' \Rightarrow \nu_1'; l_1' \quad \vdash \tau_1 \leqslant \tau_1' \Rightarrow l' \\ \Gamma \vdash_V \{\tau_2\}\ e_2\ \{\nu_1', \nu_2\} \Rightarrow l_2 \end{array}}{\Gamma \vdash_V \{\tau_2\}\ \texttt{let}\ \mathit{pat} = e_1\ \texttt{in}\ e_2\ \{\nu\} \Rightarrow l_2 \circ\!< (l_1' \circ\!< l' \circ\!< l_1 \otimes \mathsf{id}) \circ\!< l_{12}}$$

$$\frac{\nu \vdash_V e : \texttt{bool} \Rightarrow f \quad \Gamma \vdash_V \{\tau_1\}\ e_1\ \{\nu\} \Rightarrow l_1 \quad \Gamma \vdash_V \{\tau_2\}\ e_2\ \{\nu\} \Rightarrow l_2 \quad \tau_1 \mid \tau_2 \text{ unambiguous}}{\Gamma \vdash_V \{\tau\}\ \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \{\nu\} \Rightarrow \mathsf{ifVthenelse}\, f\ (\mathsf{injl} \circ\!< l_1)\ (\mathsf{injr} \circ\!< l_2)}$$

$$\frac{\nu \vdash_V e : \texttt{bool} \Rightarrow f \quad \Gamma \vdash_V \{\tau_1\}\ e_1\ \{\nu\} \Rightarrow l_1 \quad \Gamma \vdash_V \{\tau_2\}\ e_2\ \{\nu\} \Rightarrow l_2 \quad \vdash (\tau_1 \leqslant \tau_2) \Rightarrow l}{\Gamma \vdash_V \{\tau\}\ \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \{\nu\} \Rightarrow \mathsf{ifVthenelse}\, f\ (l \circ\!< l_1)\ l_2}$$

$$\frac{\nu \vdash_V e : \texttt{bool} \Rightarrow f \quad \Gamma \vdash_V \{\tau_1\}\ e_1\ \{\nu\} \Rightarrow l_1 \quad \Gamma \vdash_V \{\tau_2\}\ e_2\ \{\nu\} \Rightarrow l_2 \quad \vdash (\tau_2 \leqslant \tau_1) \Rightarrow l \quad \tau_1 \not<: \tau_2}{\Gamma \vdash_V \{\tau\}\ \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \{\nu\} \Rightarrow \mathsf{ifVthenelse}\, f\ l_1\ (l \circ\!< l_2)}$$

$$\frac{\Gamma \vdash_V \{\tau\}\ e\ \{\nu\} \Rightarrow l_1 \quad \Gamma; x \vdash_V^{\mathsf{for}} \{\tau'\}\ e\ \{\tau\} \Rightarrow l_2}{\Gamma \vdash_V \{\tau'\}\ \texttt{for}\ x\ \texttt{in}\ e\ \texttt{return}\ e'\ \{\nu\} \Rightarrow l_2 \circ\!< l_1} \qquad \frac{}{\Gamma \vdash_V \{\texttt{string}\}\ w\ \{()\} \Rightarrow \mathsf{new}\ w}$$

$$\frac{}{\Gamma \vdash_V \{\texttt{bool}\}\ \texttt{true}\ \{()\} \Rightarrow \mathsf{new}\ \texttt{true}} \qquad \frac{}{\Gamma \vdash_V \{\texttt{bool}\}\ \texttt{false}\ \{()\} \Rightarrow \mathsf{new}\ \texttt{false}}$$

Figure 25: Non-source expression well-formedness and lens semantics.(I)

$$\boxed{\Gamma \vdash_V \{\tau'\}\ p\ \{\tau\} \Rightarrow l}$$

$$\frac{}{\Gamma \vdash_V \{\tau_2\}\ \texttt{self}\ \{\tau_2\} \Rightarrow \mathsf{id}} \quad \frac{}{\Gamma \vdash_V \{\tau\}\ \texttt{child}\ \{n[\tau]\} \Rightarrow \mathsf{out}}$$

$$\frac{\emptyset; \tau \vdash\ e : \texttt{bool} \Rightarrow \lambda\emptyset\ v.v_b}{\Gamma \vdash_V \{\tau\}\ \texttt{where}\ e\ \{\tau\} \Rightarrow \mathsf{ifVthenelse}\ (\lambda v.v_b)\ \mathsf{id}\ \mathsf{bot}} \quad \frac{\alpha <: nt}{\Gamma \vdash_V \{\alpha\} :: nt\ \{\alpha\} \Rightarrow \mathsf{id}}$$

$$\frac{\Gamma \vdash_V \{\tau_1\}\ p_1\ \{\tau\} \Rightarrow l_1 \quad x \notin dom(\Gamma) \quad \Gamma; x \vdash_V^{\texttt{for}} \{\tau_2\}\ x\ /\ p_2\ \{\tau_1\} \Rightarrow l_2}{\Gamma \vdash_V \{\tau_2\}\ p_1\ /\ p_2\ \{\tau\} \Rightarrow l_2 \circ\!\!< l_1}$$

$$\boxed{\Gamma; x \vdash_V^{\texttt{for}} \{\tau'\}\ e\ \{\tau\} \Rightarrow l}$$

$$\frac{}{\Gamma; x \vdash_V^{\texttt{for}} \{()\}\ e\ \{()\} \Rightarrow \mathsf{id}} \quad \frac{\Gamma[x : \alpha] \vdash_V \{\tau'\}\ e\ \{x : \alpha\} \Rightarrow l \quad f\ v_s\ v_v\ \gamma = \gamma[x := v_v]}{\Gamma; x \vdash_V^{\texttt{for}} \{\tau'\}\ e\ \{\alpha\} \Rightarrow \mathsf{withEnv}\ f\ l}$$

$$\frac{\Gamma; x \vdash_V^{\texttt{for}} \{\tau_1'\}\ e\ \{\tau_2\} \Rightarrow l_1 \quad \Gamma; x \vdash_V^{\texttt{for}} \{\tau_1'\}\ e\ \{\tau_2\} \Rightarrow l_2}{\Gamma; x \vdash_V^{\texttt{for}} \{\tau_1', \tau_2'\}\ e\ \{\tau_1, \tau_2\} \Rightarrow l_1 \otimes l_2} \quad \frac{\Gamma; x \vdash_V^{\texttt{for}} \{\tau'\}\ e\ \{\tau\} \Rightarrow l}{\Gamma; x \vdash_V^{\texttt{for}} \{\tau'^*\}\ e\ \{\tau^*\} \Rightarrow \mathsf{map}\ l}$$

$$\frac{\Gamma; x \vdash_V^{\texttt{for}} \{\tau_1'\}\ e\ \{\tau_2\} \Rightarrow l_1 \quad \Gamma; x \vdash_V^{\texttt{for}} \{\tau_1'\}\ e\ \{\tau_2\} \Rightarrow l_2}{\Gamma; x \vdash_V^{\texttt{for}} \{\tau_1' \mid \tau_2'\}\ e\ \{\tau_1 \mid \tau_2\} \Rightarrow l_1 \oplus l_2} \quad \frac{\Gamma; x \vdash_V^{\texttt{for}} \{\tau\}\ e\ \{E(X)\} \Rightarrow l}{\Gamma; x \vdash_V^{\texttt{for}} \{\tau\}\ e\ \{X\} \Rightarrow l}$$

Figure 26: Non-source expression well-formedness and lens semantics.(II)

$$\boxed{\vdash_V pat : \tau \Rightarrow \nu; l}$$

$$\frac{}{\vdash_V x\ \mathsf{as}\ \tau : \tau \Rightarrow x : \tau; \mathsf{id}}$$

$$\frac{}{\vdash_V () : () \Rightarrow (); \mathsf{id}}$$

$$\frac{\vdash_V pat : \tau \Rightarrow \nu; l}{\vdash_V n[pat] : n[\tau] \Rightarrow \nu; l \circ\!\!< \mathsf{out}}$$

$$\frac{\vdash_V pat_1 : \tau_1 \Rightarrow \nu_1; l_1 \quad \vdash_V pat_2 : \tau_2 \Rightarrow \nu_2; l_2 \quad \vdash \nu_1 \cdot \nu_2 \Rightarrow \nu; l}{\vdash_V pat_1, pat_2 : \tau_1, \tau_2 \Rightarrow \nu; l \circ\!\!< (l_1 \otimes l_2)}$$

Figure 27: View pattern type inference and lens semantics.

$$\boxed{\tau \leadsto \tau' = l}$$

$$\frac{}{\tau \leadsto \tau = \mathsf{id}} \qquad \frac{\tau \leadsto \tau' = l}{n[\tau] \leadsto n[\tau'] = \mathsf{in} \circ\!< l \circ\!< \mathsf{out}} \qquad \frac{\beta \leadsto \tau_1' = l_1}{\beta \leadsto \tau_1' \mid \tau_2' = l_1 \circ\!< \mathsf{uninjl}} \qquad \frac{\beta \leadsto \tau_2' = l_2}{\beta \leadsto \tau_1' \mid \tau_2' = l_2 \circ\!< \mathsf{uninjr}}$$

$$\frac{() \leadsto \tau_1' = l_1}{() \leadsto \tau_1' \mid \tau_2' = l_1 \circ\!< \mathsf{uninjl}} \qquad \frac{() \leadsto \tau_2' = l_2}{() \leadsto \tau_1' \mid \tau_2' = l_2 \circ\!< \mathsf{uninjr}} \qquad \frac{}{() \leadsto \tau'^* = \mathsf{unnil}} \qquad \frac{\tau_1 \leadsto \tau' = l_1 \quad \tau_2 \leadsto \tau' = l_2}{\tau_1 \mid \tau_2 \leadsto \tau' = l_1 \triangledown l_2}$$

$$\frac{\beta \leadsto \tau' = l}{\beta \leadsto \tau'^* = l \circ\!< \mathsf{unwrap}} \qquad \frac{\tau'/\beta = (l_1, \tau_R) \quad \tau_2 \leadsto \tau_R = l_2}{\beta, \tau_2 \leadsto \tau' = (id \otimes l_2) \circ\!< l_1} \qquad \frac{\tau_2 \leadsto \tau' = l}{(), \tau_2 \leadsto \tau' = \mathsf{keepfst} \circ\!< l} \qquad \frac{() \mid \tau, \tau' \leadsto \tau' = l}{\tau^* \leadsto \tau' = \mathsf{foldlist}\ l}$$

$$\frac{\tau_1, (\tau_2, \tau_3) \leadsto \tau' = l}{(\tau_1, \tau_2), \tau_3 \leadsto \tau' = \mathsf{assocl} \circ\!< l} \qquad \frac{\tau_1, \tau' \mid \tau_2 \leadsto \tau' = l}{\tau_1{}^*, \tau_2 \leadsto \tau' = \mathsf{mergelist}\ l}$$

$$\frac{\tau_{ne}, \tau_3 \leadsto \tau' = l_1 \quad \tau_e, \tau_3 \leadsto \tau' = l_2 \quad \mathit{splitEmpty}(\tau_1 \mid \tau_2) = (\tau_{ne} \mid \tau_e, l)}{(\tau_1 \mid \tau_2), \tau_3 \leadsto \tau' = (l \otimes \mathsf{id}) \circ\!< \mathsf{undistl} \circ\!< (l_1 \triangledown l_2)}$$

$$\frac{\tau_1, \tau_3 \leadsto \tau' = l_1 \quad \tau_2, \tau_3 \leadsto \tau' = l_2 \quad () \not<: \tau_1 \mid \tau_2}{(\tau_1 \mid \tau_2), \tau_3 \leadsto \tau' = \mathsf{undistl} \circ\!< (l_1 \triangledown l_2)}$$

$$\boxed{\tau/\beta = (l, \tau_R)}$$

$$\frac{}{\beta/\beta = (\mathsf{keepsnd}, ())} \qquad \frac{\tau_1/\beta = (l_1, \tau_{R_1})}{\tau_1 \mid \tau_2/\beta = (l_1 \circ\!< \mathsf{uninjl}, \tau_{R_1})} \qquad \frac{\tau_2/\beta = (l_2, \tau_{R_2})}{\tau_1 \mid \tau_2/\beta = (l_2 \circ\!< \mathsf{uninjr}, \tau_{R_2})} \qquad \frac{}{\beta, \tau_2/\beta = (\mathsf{id}, \tau_2)}$$

$$\frac{\tau \leadsto \tau_1 = l}{n[\tau_1], \tau_2/n[\tau] = (\mathsf{in} \circ\!< l \circ\!< \mathsf{out} \otimes \mathsf{id}, \tau_2)} \qquad \frac{\tau_1, \tau_3/\beta = (l_1, \tau_{R_1})}{(\tau_1 \mid \tau_2), \tau_3/\beta = (l_1 \circ\!< (\mathsf{uninjl} \otimes \mathsf{id}), \tau_{R_1})}$$

$$\frac{\tau_2, \tau_3/\beta = (l_2, \tau_{R_2})}{(\tau_1 \mid \tau_2), \tau_3/\beta = (l_2 \circ\!< (\mathsf{uninjr} \otimes \mathsf{id}), \tau_{R_2})} \qquad \frac{\tau_1, (\tau_2, \tau_3)/\beta = (l, \tau_R)}{(\tau_1, \tau_2), \tau_3/\beta = (l \circ\!< \mathsf{assocr}, \tau_R)}$$

$$\frac{\tau_1/\beta = (l_1, ())}{\tau_1{}^*, \tau_2/\beta = ((\mathsf{remsndone} \circ\!< l_1 \otimes \mathsf{id}) \circ\!< \mathsf{assocr} \circ\!< (\mathsf{uncons} \otimes \mathsf{id}), \tau_1{}^*, \tau_2)}$$

$$\frac{\tau_1/\beta = (l_1, \tau_{R_1}) \quad \tau_{R_1} \neq ()}{\tau_1{}^*, \tau_2/\beta = (\mathsf{assocr} \circ\!< (l_1 \otimes \mathsf{id}) \circ\!< \mathsf{assocr} \circ\!< (\mathsf{uncons} \otimes \mathsf{id}), \tau_{R_1}, \tau_1{}^*, \tau_2)}$$

$$\frac{\tau_2/\beta = (l_2, \tau_{R_2})}{\tau_1{}^*, \tau_2/\beta = (l_2 \circ\!< \mathsf{remfstone} \circ\!< (\mathsf{unnil} \otimes \mathsf{id}), \tau_{R_2})}$$

$$\frac{\tau/\beta = (l, ())}{\tau^*/\beta = ((\mathsf{remsndone} \circ\!< l \otimes \mathsf{id}) \circ\!< \mathsf{uncons}, \tau^*)} \qquad \frac{\tau/\beta = (l, \tau_R) \quad \tau_R \neq ()}{\tau^*/\beta = (\mathsf{assocr} \circ\!< (l \otimes \mathsf{id}) \circ\!< \mathsf{uncons}, (\tau_R, \tau^*))}$$

Figure 28: Source normalization procedure.

$$
\begin{array}{l}
norm_V(n[\tau]) = (n[\tau'], \mathsf{in} \circ\!< l \circ\!< \mathsf{out}) \quad \textbf{where } norm_V(\tau) = (\tau', l) \\
norm_V(\beta) = (\beta, \mathsf{id}) \quad \textbf{where } norm_V(\tau) = (\tau', l) \\
norm_V(()) = ((), \mathsf{id}) \\
norm_V(\tau_1 \mid \tau_2) = \{ \ (\tau_1' \mid \tau_2', l_1 \oplus l_2) \quad \text{if } \tau_1' \mid \tau_2' \text{ unambiguous} \\
\quad \textbf{where } norm_V(\tau_1) = (\tau_1', l_1) \ \ \textbf{and} \ \ norm_V(\tau_2) = (\tau_2', l_2) \\
norm_V(\tau_1, \tau_2) = \begin{cases} (\tau_2', \mathsf{remfstone} \circ\!< l_2) & \text{if } \tau_1 = () \\ (\tau_1', \mathsf{remsndone} \circ\!< l_1) & \text{if } \tau_2 = () \\ ((\tau_1', \tau_2'), l_1 \otimes l_2) & \text{if } \tau_1', \tau_2' \text{ unambiguous} \end{cases} \\
\quad \textbf{where } norm_V(\tau_1) = (\tau_1', l_1) \ \ \textbf{and} \ \ norm_V(\tau_2) = (\tau_2', l_2) \\
norm_V(\tau^*) = \{ \ (\tau'^*, \mathsf{map}\ l) \quad \text{if } \tau'^* \text{ unambiguous} \\
\quad \textbf{where } norm_V(\tau) = (\tau', l)
\end{array}
$$

Figure 29: View normalization procedure.

The function $elems : Type \rightarrow \{Atom\}$ that computes a set of all the top-level atomic types in a given sequence type is defined as follows:

$$
\begin{array}{l}
elems(\alpha) = \{\alpha\} \\
elems(()) = \emptyset \\
elems(\tau_1 \mid \tau_2) = elems(\tau_1) \cup elems(\tau_2) \\
elems(\tau_1, \tau_2) = elems(\tau_1) \cup elems(\tau_2) \\
elems(\tau^*) = elems(\tau) \\
elems(X) = elems(E(X))
\end{array}
$$

I Unidirectional update normalization

The syntax of ordinary unidirectional updates that we use in BiFluX differs slightly from that of Flux [8]. For example, there is no `iter` u update that iterates over a sequence by applying the same update u, and instead we support arbitrary paths as directions. In our design, iteration occurs automatically at the `child` axis (for updates of the form `child`$[u]$). The two different `let` $x = e$ `in` u and `snapshot` x `in` u updates in Flux, that respectively bind a variable to the result of evaluating an expression under the current environment and bind a variable to the current value of the focus, are both subsumed by our `case` e `of` $\vec{pat} \rightarrow \vec{u}$ update, that allows the expression e to depend on the current value of the focus.

The remaining rules for normalizing high-level bidirectional and unidirectional updates are shown in Figures 30, 31 and 32. We omit some cases for `UPDATE FOR VIEW` statements that can be easily inferred. The normalization for particular create and recover unidirectional updates are given in Figure 33.

$$
\begin{aligned}
[\![\, u \,]\!]^b_{Stmt} &= [\![\, u \,]\!]^b_{Upd}(\texttt{true}, \texttt{true}, \cdot) \\
[\![\, \{\} \,]\!]^b_{Stmt} &= \texttt{skip} \\
[\![\, \texttt{IF SOURCE } e \texttt{ THEN } s \texttt{ ELSE } s' \,]\!]^b_{Stmt} &= \texttt{ifS } e \texttt{ then } [\![\, s \,]\!]^b_{Stmt} \texttt{ else } [\![\, s' \,]\!]^b_{Stmt} \\
[\![\, \texttt{IF VIEW } e \texttt{ THEN } s \texttt{ ELSE } s' \,]\!]^b_{Stmt} &= \texttt{ifV } e \texttt{ then } [\![\, s \,]\!]^b_{Stmt} \texttt{ else } [\![\, s' \,]\!]^b_{Stmt} \\
[\![\, \texttt{IF } e \texttt{ THEN } s \texttt{ ELSE } s' \,]\!]^b_{Stmt} &= \texttt{if } e \texttt{ then } [\![\, s \,]\!]^b_{Stmt} \texttt{ else } [\![\, s' \,]\!]^b_{Stmt} \\
[\![\, \texttt{CASE SOURCE } p \texttt{ OF } \{pat_1 \to s_1 \mid \ldots \mid pat_n \to s_n\} \,]\!]^b_{Stmt} &= \texttt{caseS } p \texttt{ of } \vec{pat} \to [\![\, \vec{s} \,]\!]^b_{Stmt} \\
[\![\, \texttt{CASE VIEW } e \texttt{ OF } \{pat_1 \to s_1 \mid \ldots \mid pat_n \to s_n\} \,]\!]^b_{Stmt} &= \texttt{caseV } e \texttt{ of } \vec{pat} \to [\![\, \vec{s} \,]\!]^b_{Stmt} \\
[\![\, \texttt{CASE } e \texttt{ OF } \{pat_1 \to s_1 \mid \ldots \mid pat_n \to s_n\} \,]\!]^b_{Stmt} &= \texttt{case } e \texttt{ of } \vec{pat} \to [\![\, \vec{s} \,]\!]^b_{Stmt} \\
[\![\, \texttt{LET SOURCE } pat = p \texttt{ IN } s \,]\!]^b_{Stmt} &= \texttt{caseS } p \texttt{ of } p \to \ [\![\, s \,]\!]^b_{Stmt} \\
[\![\, \texttt{LET VIEW } pat = e \texttt{ IN } s \,]\!]^b_{Stmt} &= \texttt{caseV } p \texttt{ of } p \to \ [\![\, s \,]\!]^b_{Stmt} \\
[\![\, \texttt{LET } pat = e \texttt{ IN } s \,]\!]^b_{Stmt} &= \texttt{case } p \texttt{ of } p \to \ [\![\, s \,]\!]^b_{Stmt} \\
[\![\, \texttt{SOURCE } e_1; cs \,]\!]_{Conds} &= (e_1 \wedge e_S, e_V, \vec{x} = \vec{e}) \\
&\quad \textbf{where } [\![\, cs \,]\!]_{Conds} = (e_S, e_V, \vec{x} = \vec{e}) \\
[\![\, \texttt{VIEW } e_1; cs \,]\!]_{Conds} &= (e_S, e_1 \wedge e_V, \vec{x} = \vec{e}) \\
&\quad \textbf{where } \ [\![\, cs \,]\!]_{Conds} = (e_S, e_V, \vec{x} = \vec{e}) \\
[\![\, \texttt{VIEW } x_0 := e_0; cs \,]\!]_{Conds} &= (e_S, e_V, x_0, \vec{x} = e_0, \vec{e}) \\
&\quad \textbf{where } \ [\![\, cs \,]\!]_{Conds} = (e_S, e_V, \vec{x} = \vec{e}) \\
[\![\, \texttt{SOURCE } e \,]\!]_{Conds} &= (e, \texttt{true}, \cdot) \\
[\![\, \texttt{VIEW } e \,]\!]_{Conds} &= (\texttt{true}, e, \cdot) \\
[\![\, \texttt{VIEW } x := e \,]\!]_{Conds} &= (\texttt{true}, \texttt{true}, x = e)
\end{aligned}
$$

Figure 30: Bidirectional update normalization.(I)

$$\llbracket \texttt{UPDATE}\ pat\ \texttt{IN}\ p\ \texttt{BY}\ vs\ \texttt{FOR VIEW}\ pat'\ \texttt{IN}\ p' \rrbracket^{b}_{Upd}(e_S, e_V, \vec{x} = \vec{e}) =$$
$p[[b]p']$ **where**

$$(s_{SV}, ms_V, ms_S) = splitVStmt(vs)$$
$$b = \texttt{alignpos}\ (\texttt{case self of}\ pat \to e_S)\ \llbracket s_{SV} \rrbracket^{b}_{Stmt}\ \llbracket ms_V \rrbracket^{c}_{MStmt}(pat')\ \llbracket ms_S \rrbracket^{r}_{MStmt}(pat)$$

$$\llbracket \texttt{UPDATE}\ pat\ \texttt{IN}\ p\ \texttt{BY}\ vs\ \texttt{FOR VIEW}\ p'\ \texttt{MATCHING SOURCE BY}\ p_s\ \texttt{VIEW BY}\ p_v \rrbracket^{b}_{Upd}(e_S, e_V, \vec{x} = \vec{e}) =$$
$p[[b]p']$ **where**

$$(s_{SV}, ms_V, ms_S) = splitVStmt(vs)$$
$${p_s}' = \texttt{case self of}\ pat \to p_s$$
$$b = \texttt{alignkey}\ (\texttt{case self of}\ pat \to e_S)\ p'_s\ p_v\ \llbracket s_{SV} \rrbracket^{b}_{Stmt}\ \llbracket ms_V \rrbracket^{c}_{MStmt}(\cdot)\ \llbracket ms_S \rrbracket^{r}_{MStmt}(pat)$$

$$\llbracket \texttt{UPDATE}\ p\ \texttt{BY}\ vs\ \texttt{FOR VIEW}\ pat'\ \texttt{IN}\ p'\ \texttt{MATCHING SOURCE BY}\ p_s\ \texttt{VIEW BY}\ p_v \rrbracket^{b}_{Upd}(e_S, e_V, \vec{x} = \vec{e})$$
$= p[[b]p']$ **where**

$$(s_{SV}, ms_V, ms_S) = splitVStmt(vs)$$
$${p_v}' = \texttt{case self of}\ pat' \to p_v$$
$$b = \texttt{alignkey}\ e_S\ p_s\ p'_v\ \llbracket s_{SV} \rrbracket^{b}_{Stmt}\ \llbracket ms_V \rrbracket^{c}_{MStmt}(pat')\ \llbracket ms_S \rrbracket^{r}_{MStmt}(\cdot)$$

$$\llbracket \texttt{UPDATE}\ p\ \texttt{BY}\ vs\ \texttt{FOR VIEW}\ p'\ \texttt{MATCHING SOURCE BY}\ p_s\ \texttt{VIEW BY}\ p_v \rrbracket^{b}_{Upd}(e_S, e_V, \vec{x} = \vec{e}) =$$
$p[[b]p']$ **where**

$$(s_{SV}, ms_V, ms_S) = splitVStmt(vs)$$
$$b = \texttt{alignkey}\ e_S\ p_s\ p_v\ \llbracket s_{SV} \rrbracket^{b}_{Stmt}\ \llbracket ms_V \rrbracket^{c}_{MStmt}(\cdot)\ \llbracket ms_S \rrbracket^{r}_{MStmt}(\cdot)$$

$$\llbracket p \rrbracket^{\texttt{iter}}_{PatPath}(e_S, b) = (p\ /\ \texttt{where}\ e_S)[\texttt{iter}\ b]$$
$$\llbracket p \rrbracket^{S}_{PatPath}(e_S, b) = (p\ /\ \texttt{where}\ e_S)[b]$$

Figure 31: Bidirectional update normalization.(II)

$$
\begin{aligned}
\llbracket\, u \ \mathtt{WHERE}\ cs \,\rrbracket^u_{Stmt} &= \llbracket\, u \,\rrbracket^u_{Upd}(\llbracket\, cs \,\rrbracket^u_{Conds}) \\
\llbracket\, u \,\rrbracket^u_{Stmt} &= \llbracket\, u \,\rrbracket^u_{Upd}(\mathtt{true}) \\
\llbracket\, s; s' \,\rrbracket^u_{Stmt} &= \llbracket\, s \,\rrbracket^u_{Stmt}; \llbracket\, s' \,\rrbracket^u_{Stmt} \\
\llbracket\, \{\} \,\rrbracket^u_{Stmt} &= \mathtt{skip} \\
\llbracket\, \mathtt{IF}\ e\ \mathtt{THEN}\ s\ \mathtt{ELSE}\ s' \,\rrbracket^u_{Stmt} &= \mathtt{if}\ e\ \mathtt{then}\ \llbracket\, s \,\rrbracket^u_{Stmt}\ \mathtt{else}\ \llbracket\, s' \,\rrbracket^u_{Stmt} \\
\llbracket\, \mathtt{LET}\ pat = e\ \mathtt{IN}\ s \,\rrbracket^u_{Stmt} &= \mathtt{case}\ e\ \mathtt{of}\ pat \rightarrow \llbracket\, s \,\rrbracket^u_{Stmt} \\
\llbracket\, \mathtt{CASE}\ e\ \mathtt{OF}\ \{pat_1 \rightarrow s_1 \mid \ldots \mid pat_n \rightarrow s_n\} \,\rrbracket^u_{Stmt} &= \mathtt{case}\ e\ \mathtt{of}\ \vec{pat} \rightarrow \llbracket\, \vec{s} \,\rrbracket^u_{Stmt}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket\, pat\ \mathtt{IN}\ p \,\rrbracket^u_{PatPath}(u) &= p[\mathtt{case}\ \mathtt{self}\ \mathtt{of}\ pat \rightarrow u] \\
\llbracket\, p \,\rrbracket^u_{PatPath}(u) &= p[u] \\
\llbracket\, e; cs \,\rrbracket^u_{Conds} &= e \wedge \llbracket\, cs \,\rrbracket^u_{Conds} \\
\llbracket\, e \,\rrbracket^u_{Conds} &= e
\end{aligned}
$$

$$
\begin{aligned}
\llbracket\, \mathtt{UPDATE}\ patp\ \mathtt{BY}\ s \,\rrbracket^u_{Upd}(e) &= \llbracket\, patp \,\rrbracket^u_{PatPath}((\mathtt{where}\ e)[\llbracket\, s \,\rrbracket^u_{Stmt}]) \\
\llbracket\, \mathtt{DELETE}\ patp \,\rrbracket^u_{Upd}(e) &= \llbracket\, patp \,\rrbracket^u_{PatPath}((\mathtt{where}\ e)[\mathtt{delete}]) \\
\llbracket\, \mathtt{DELETE}\ patp \,\rrbracket^u_{Upd}(e) &= \llbracket\, patp \,\rrbracket^u_{PatPath}((\mathtt{where}\ e)[\mathtt{children}[\mathtt{delete}]]) \\
\llbracket\, \mathtt{REPLACE}\ patp\ \mathtt{WITH}\ e' \,\rrbracket^u_{Upd}(e) &= \llbracket\, patp \,\rrbracket^u_{PatPath}((\mathtt{where}\ e)[\mathtt{delete}; \mathtt{insert}\ e']) \\
\llbracket\, \mathtt{REPLACE\ IN}\ patp\ \mathtt{WITH}\ e' \,\rrbracket^u_{Upd}(e) &= \llbracket\, patp \,\rrbracket^u_{PatPath}((\mathtt{where}\ e)[\\
&\quad\ \mathtt{children}[\mathtt{delete}; \mathtt{insert}\ e']]) \\
\llbracket\, \mathtt{INSERT\ BEFORE}\ patp\ \mathtt{VALUE}\ e' \,\rrbracket^u_{Upd}(e) &= \llbracket\, patp \,\rrbracket^u_{PatPath}((\mathtt{where}\ e)[\mathtt{left}[\mathtt{insert}\ e']]) \\
\llbracket\, \mathtt{INSERT\ AFTER}\ patp\ \mathtt{VALUE}\ e' \,\rrbracket^u_{Upd}(e) &= \llbracket\, patp \,\rrbracket^u_{PatPath}((\mathtt{where}\ e)[\mathtt{right}[\mathtt{insert}\ e']]) \\
\llbracket\, \mathtt{INSERT\ AS\ FIRST\ INTO}\ patp\ \mathtt{VALUE}\ e' \,\rrbracket^u_{Upd}(e) &= \llbracket\, patp \,\rrbracket^u_{PatPath}((\mathtt{where}\ e)[\\
&\quad\ \mathtt{children}[\mathtt{left}[\mathtt{insert}\ e']]]) \\
\llbracket\, \mathtt{INSERT\ AS\ LAST\ INTO}\ patp\ \mathtt{VALUE}\ e' \,\rrbracket^u_{Upd}(e) &= \llbracket\, patp \,\rrbracket^u_{PatPath}((\mathtt{where}\ e)[\\
&\quad\ \mathtt{children}[\mathtt{right}[\mathtt{insert}\ e']]])
\end{aligned}
$$

Figure 32: Unidirectional update normalization.

$$
\begin{aligned}
[\![\{\}]\!]^r_{MStmt}(mpat) &= \texttt{delete}\\
[\![\cdot]\!]^r_{MStmt}(mpat) &= \texttt{delete}\\
[\![s]\!]^r_{MStmt}(\cdot) &= [\![s]\!]^r_{Stmt}\\
[\![s]\!]^r_{MStmt}(pat) &= \texttt{case self of } pat \rightarrow [\![s]\!]^r_{Stmt}\\
[\![\texttt{IF } e \texttt{ THEN } s \texttt{ ELSE } s']\!]^r_{Stmt} &= \texttt{if } e \texttt{ then } [\![s]\!]^r_{Stmt} \texttt{ else } [\![s']\!]^r_{Stmt}\\
[\![\texttt{LET } pat = e \texttt{ IN } s]\!]^r_{Stmt} &= \texttt{case } e \texttt{ of } pat \rightarrow \ [\![s]\!]^r_{Stmt}\\
[\![\texttt{CASE } e \texttt{ OF } \{pat_1 \rightarrow r_1 \mid \ldots \mid pat_n \rightarrow r_n\}]\!]^r_{Stmt} &= \texttt{case } e \texttt{ of } \vec{pat} \rightarrow \ [\![\vec{r}]\!]^r_{Cases}\\
[\![\texttt{KEEP self}; s]\!]^r_{Stmt} &= \texttt{keep } [\![s]\!]^u_{Stmt}\\
[\![\texttt{DELETE self}]\!]^r_{Stmt} &= \texttt{delete}
\end{aligned}
$$

$$
\begin{aligned}
[\![\{\}]\!]^c_{MStmt}(mpat) &= \cdot\\
[\![\cdot]\!]^c_{MStmt}(mpat) &= \cdot\\
[\![s]\!]^c_{MStmt}(\cdot) &= \texttt{delete}; [\![s]\!]^c_{Stmt}\\
[\![s]\!]^c_{MStmt}(pat) &= \texttt{case } pat \texttt{ of self} \rightarrow \texttt{delete}; [\![s]\!]^c_{Stmt}\\
[\![\texttt{IF } e \texttt{ THEN } s \texttt{ ELSE } s']\!]^c_{Stmt} &= \texttt{if } e \texttt{ then } [\![s]\!]^c_{Stmt} \texttt{ else } [\![s']\!]^c_{Stmt}\\
[\![\texttt{LET } pat = e \texttt{ IN } s]\!]^c_{Stmt} &= \texttt{case } e \texttt{ of } pat \rightarrow \ [\![s]\!]^c_{Stmt}\\
[\![\texttt{CASE } e \texttt{ OF } \{pat_1 \rightarrow r_1 \mid \ldots \mid pat_n \rightarrow r_n\}]\!]^c_{Stmt} &= \texttt{case } e \texttt{ of } \vec{pat} \rightarrow \ [\![\vec{r}]\!]^c_{Cases}\\
[\![\texttt{CREATE } e; s]\!]^c_{Stmt} &= \texttt{insert } e; [\![s]\!]^u_{Stmt}
\end{aligned}
$$

Figure 33: Create and recover unidirectional update normalization.