Advances in Programming Languages
APL15: Bidirectional Programming

David Aspinall

School of Informatics
The University of Edinburgh

Friday 19 November 2010
Semester 1 Week 9
This block of lectures covers some language techniques and tools for manipulating structured data and text.

- **Motivations, simple bidirectional transformations**
- **Boomerang and complex transformations**
- **XML processing with CDuce**

This lecture introduces some of the motivations and basic concepts behind bidirectional programming.
Outline

1. Motivations
2. Language design
3. Semantics
4. Boomerang example
5. Summary
1 Motivations
2 Language design
3 Semantics
4 Boomerang example
5 Summary
A classic problem in databases: how can we propagate changes in a `view` on the data back into the database itself?

### University Staff Database (Confidential)

<table>
<thead>
<tr>
<th><strong>Name:</strong></th>
<th>David Aspinall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Email:</strong></td>
<td><a href="mailto:da@inf.ed.ac.uk">da@inf.ed.ac.uk</a></td>
</tr>
<tr>
<td><strong>Staff Number:</strong></td>
<td>1230935</td>
</tr>
<tr>
<td><strong>Pay grade:</strong></td>
<td>pt 6.II</td>
</tr>
<tr>
<td><strong>Home Address:</strong></td>
<td>10 London Road, E7 5QA</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
View Update Problem

A classic problem in databases: how can we propagate changes in a view on the data back into the database itself?

University Cycle to Work Scheme

<table>
<thead>
<tr>
<th>Name:</th>
<th>David Aspinall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home Address:</td>
<td>10 London Road, E7 5QA</td>
</tr>
<tr>
<td>Distance to work:</td>
<td>418 miles</td>
</tr>
</tbody>
</table>
View Update Problem

A classic problem in databases: how can we propagate changes in a view on the data back into the database itself?

University Cycle to Work Scheme

**Name:** David Aspinall  
**Home Address:** 10 London Road, E7 5QA  
**Distance to work:** 418 miles

A bit odd!
View Update Problem

A classic problem in databases: how can we propagate changes in a view on the data back into the database itself?

University Cycle to Work Scheme

<table>
<thead>
<tr>
<th>Name:</th>
<th>David Aspinall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home Address:</td>
<td>10 London Road, EH7 5QA</td>
</tr>
<tr>
<td>Distance to work:</td>
<td>2.6 miles</td>
</tr>
</tbody>
</table>

Corrected. A more feasible candidate for cycling to work.
A classic problem in databases: how can we propagate changes in a *view* on the data back into the database itself?

University Staff Database (Confidential)

- **Name:** David Aspinall
- **Email:** da@inf.ed.ac.uk
- **Staff Number:** 1230935
- **Pay grade:** pt 6.II
- **Home Address:** 10 London Road, **EH7** 5QA

This fix should be updated in the staff database.
A view \( v \) is generated by an arbitrary query \( q \) on the source database;
A view $\nu$ is generated by an arbitrary query $q$ on the source database;
The view is updated by an update function $u$ to $\nu'$;
A view $v$ is generated by an arbitrary query $q$ on the source database; The view is updated by an update function $u$ to $v'$; The source must be updated *correspondingly* to $s'$ by a translation function $t$, so that the same query $q$ yields $v'$ again.
The view update problem has been a research challenge for a long time. Since query q is arbitrary, it may be

non-injective: a view update has many possible source updates e.g., imagine updating “distance to work” instead of postcode

non-surjective: an update may have no possible source update e.g., suppose the view included “nearest quiet road”

In database world, present state-of-the-art is to use triggers which are custom programmed for particular views. Drawbacks:

must be re-programmed for each query/allowed update
duplicates information from the query
error prone: must check consistency with query, maintain in tandem.
The view update problem has been a research challenge for a long time. Since query \( q \) is arbitrary, it may be

- *non-injective*: a view update has many possible source updates e.g., imagine updating “distance to work” instead of postcode
The view update problem has been a research challenge for a long time. Since query $q$ is arbitrary, it may be

- **non-injective**: a view update has many possible source updates
  e.g., imagine updating “distance to work” instead of postcode

- **non-surjective**: an update may have no possible source update
  e.g., suppose the view included “nearest quiet road”
The view update problem has been a research challenge for a long time. Since query $q$ is arbitrary, it may be

- **non-injective**: a view update has many possible source updates
e.g., imagine updating “distance to work” instead of postcode
- **non-surjective**: an update may have no possible source update
e.g., suppose the view included “nearest quiet road”

In database world, present state-of-the-art is to use *triggers* which are custom programmed for particular views. Drawbacks:

- must be re-programmed for each query/allowed update
- duplicates information from the query
- error prone: must check consistency with query, maintain in tandem.
Solution: Bidirectional programming

Idea: write one program \textit{get} for the query \( q \), and automatically derive another one \textit{put} which propagates view changes back to the source data, whenever it is possible.

Advantages:

- no need to maintain separate programs
- ideally, consistency is ensured automatically too.

The \textit{put} function goes in the opposite direction to \textit{get}. So when both exist, we have a \textit{bidirectional transformation}.

Hence \textit{bidirectional} programming, where we write bidirectional transformations. Ordinary programs, of course, run only in one direction.
Bidirectional transformations have a myriad of applications.

Some examples:
Bidirectional transformations have a myriad of applications.

Some examples:

- **software engineering**: solving the “round-trip problem” of model-driven development.
Other applications

Bidirectional transformations have a myriad of applications.

Some examples:

- **software engineering**: solving the “round-trip problem” of model-driven development.
- **user interfaces**: helping to implement the model-view-controller paradigm, by ensuring that view updates consistently change the model and vice-versa.
Bidirectional transformations have a myriad of applications.

Some examples:

- **software engineering**: solving the “round-trip problem” of model-driven development.

- **user interfaces**: helping to implement the model-view-controller paradigm, by ensuring that view updates consistently change the model and vice-versa.

- **data synchronization**: unifying and mediating between data held in different formats, such as address book data.
Bidirectional transformations have a myriad of applications.

Some examples:

- **software engineering**: solving the “round-trip problem” of model-driven development.
- **user interfaces**: helping to implement the model-view-controller paradigm, by ensuring that view updates consistently change the model and vice-versa.
- **data synchronization**: unifying and mediating between data held in different formats, such as address book data.
- **marshalling**: transferring data across networks, or mediating between different applications, allowing changes in a safe way.
Outline

1 Motivations

2 Language design

3 Semantics

4 Boomerang example

5 Summary
Designing a bidirectional language

We could solve the bidirectional problem by:

- **meta-programming**: trying to generate *put* from *get*, case-by-case.

- designing a **new special purpose language** or DSL abstraction, for writing *put* and *get* at once.
Designing a bidirectional language

We could solve the bidirectional problem by:

- **meta-programming**: trying to generate `put` from `get`, case-by-case.
  - use an existing language and meta-mechanism
    - difficult; impossible to solve for all updates
    - must explain failures to programmer

- designing a **new special purpose language** or DSL abstraction, for writing `put` and `get` at once.
We could solve the bidirectional problem by:

- **meta-programming**: trying to generate `put` from `get`, case-by-case.
  - use an existing language and meta-mechanism
  - difficult; impossible to solve for all updates
  - must explain failures to programmer

- designing a **new special purpose language** or DSL abstraction, for writing `put` and `get` at once.
  - can easily restrict syntactically what is expressed
  - programmer must learn new syntax/abstraction
Designing a bidirectional language

We could solve the bidirectional problem by:

- **meta-programming**: trying to generate `put` from `get`, case-by-case.
  - use an existing language and meta-mechanism
  - difficult; impossible to solve for all updates
  - must explain failures to programmer

- designing a **new special purpose language** or DSL abstraction, for writing `put` and `get` at once.
  - can easily restrict syntactically what is expressed
  - programmer must learn new syntax/abstraction
Boomerang: A Programming Language Approach

Ideas behind **Boomerang**: 
- design a special purpose bidirectional programming language 
- every expressible program denotes a bidirectional transformation 
- error messages are specific to domain 
- can ensure all programs have correct bidirectional behaviour 
- take a *functional* approach (ex: why?)

History at University of Pennsylvania, Benjamin Pierce: 
- late 1990s, early 2000s: popular *Unison* file synchronization tool built on carefully designed semantic foundations. 
- mid 2000s: *Harmony* project, investigating view updates for XML and then bidirectional programming.

See J. Nathan Foster’s, PhD thesis *Bidirectional Programming Languages*, University of Pennsylvania, 2009. The diagram on p.35 and some of the following content is adapted from this PhD thesis and earlier papers co-authored with Benjamin Pierce and other collaborators.
Putting and Getting

Suppose we have a set of source values $S$ and view values $V$.

The basic bidirectional property we want is that given some get function (database query),

$$ get : S \rightarrow V $$

we should have a way to compute updates on $S$ from altered views, i.e., find a corresponding put function with type:

$$ put : V, S \rightarrow S $$

which transforms a changed view into an update on $S$, i.e., a function from $S$ to $S$. 
Putting and Getting

Suppose we have a set of source values $S$ and view values $V$.

The basic bidirectional property we want is that given some get function (database query),

$$\text{get} : S \rightarrow V$$

we should have a way to compute updates on $S$ from altered views, i.e., find a corresponding put function with type:

$$\text{put} : V, S \rightarrow S$$

which transforms a changed view into an update on $S$, i.e., a function from $S$ to $S$.

An alternative type for put is possible: we might instead try to record and characterise the update operations and make put take as its argument a delta. This might allow more accurate source changes, can you think of an example?
Put and Get laws

To make this commute we want this equation to be satisfied: for all view elements $v'$ and source elements $s$,

$$get(put(v', s)) = v'$$

A put followed by a get must give us back the thing we put in: the **PutGet** law.
Put and Get laws

To make this commute we want this equation to be satisfied: for all view elements \( v' \) and source elements \( s \),

\[
get(put(v', s)) = v'
\]

A put followed by a get must give us back the thing we put in: the \textbf{PutGet} law.

On the other hand, if we put back the same thing that we got out, we don’t expect any change to the source:

\[
put(get(s), s) = s
\]

This is the \textbf{GetPut} law.

PutGet and GetPut together are a rather loose specification...
Creating from a view

It’s useful to also be able to synthesise a source element from a view element, perhaps giving default values to parts of the source that are not manifest in the view.

This motivates a third type of function:

\[ create : \mathcal{V} \rightarrow \mathcal{S} \]

Create must satisfy the obvious CreateGet law:

\[ \text{get}(create(v)) = v \]
A *lens* is an abstraction which captures all these pieces.

A lens $l$ is written $l \in S \Leftrightarrow V$ to show its set of source values $S$ and set of view values $V$. 
Boomerang is a programming language for constructing lenses.

- simple lenses are easy to express
- lenses can be combined using *combinators*
- larger lenses can be expressed more easily using *grammars*
- a library of useful pre-defined lenses is supplied

A fundamental design decision is to make the functions that comprise lenses always *total*. If a program compiles, then *put* can never go wrong at run time due to a forbidden update.

The abstraction is always maintained: combinations of lenses construct new lenses which again satisfy the required laws.

The language has a strong type system which helps ensure these things statically. In particular, every lens has a fixed source domain $S$ and view domain $V$, described by types. These are often built from *regular expressions* denoting sets of strings.
Let $\Sigma$ be an alphabet of characters $c \in \Sigma$. Strings over the alphabet $\Sigma$ are ranged over by $s \in \Sigma^*$. The empty string is denoted $\epsilon$. Given two strings $s_1$ and $s_2$, their concatenation is $s_1 \cdot s_2$.

Recall the language of regular expressions $R$ used to describe sets of strings:

$$ R ::= s \mid R \cdot R \mid R|R \mid R^* $$

with familiar meanings.

($R_1|R_2$ stands for the union of the sets denoted by $R_1$ and $R_2$).
Simple Lenses: Copy

Given a regular expression $R$, then

$$\text{copy } R \in R \Leftrightarrow R$$

defines a lens with source domain $R$ and target (view) domain $R$, such that for $s, v \in R$

$$\begin{align*}
\text{get}(s) &= s \\
\text{put}(v, s) &= v \\
\text{create}(v) &= v
\end{align*}$$

This lens is an identity, it simply copies from source to the view. Since the source and view domains are the same, no information is hidden.
Simple Lenses: Constant

Given a regular expression $R$, and any string $k$, then the constant lens

$$\text{const } R \ k \ \in \ R \ \Leftrightarrow \ \{k\}$$

such that for $s, v \in R$

$$get(s) = k$$
$$put(v, s) = s$$
$$create(v) = \text{default}(R)$$

Going forwards, this lens ignores its source and always produces the view $k$. Going backwards, it ignores any (necessarily vacuous) updates and leaves the source unchanged.

The create an element in the source, we have to pick one. The function $\text{default}(R)$ stands for the choice of an arbitrary value from the set $R$ (in practice this may be defined by the programmer).
Deletion and Insertion

Lenses to insert and delete are defined using the constant lens.

\[
\begin{align*}
\text{del } R & \quad \in \quad R \leftrightarrow \{\epsilon\} \\
\text{del } R & \quad = \quad \text{const } R \epsilon \\
\text{ins } v & \quad \in \quad \{\epsilon\} \leftrightarrow \{v\} \\
\text{ins } v & \quad = \quad \text{const } \{\epsilon\} v
\end{align*}
\]
Given two lenses \( l_1 \in S_1 \Leftrightarrow V_1 \) and \( l_2 \in S_2 \Leftrightarrow V_2 \), their concatenation

\[
l_1 . l_2 \quad \in \quad S_1 \cdot S_2 \Leftrightarrow V_1 \cdot V_2
\]
is defined, provided both \( S_1 \cdot S_2 \) and \( V_1 \cdot V_2 \) are \emph{splittable}.

i.e., given \( s \in S_1 \cdot S_2 \) we can find unique \( s_1 \in S_1, s_2 \in S_2 \) such that \( s_1 \cdot s_2 = s \).

The underlying functions of \( l_1 . l_2 \) each split their inputs and pass to the underlying functions from \( l_1 \) and \( l_2 \) respectively, and then concatenate the results.

For example:

\[
\text{get}(s_1 \cdot s_2) = (\text{get}(s_1)) \cdot (\text{get}(s_2))
\]
Outline

1 Motivations
2 Language design
3 Semantics
4 Boomerang example
5 Summary
module Staffdb =
    (copy NAME) . "", "
    . del EMAIL . del "", "
    . del STAFFNUM . del ", " . del SALARY . del ", " 
    . del ADDRESS . del ", " 
    . (ins "Map—distance—from: ") 
    . (copy POSTCODE) 
let cycleinfos : lens = "" | cycleinfo . (newline . cycleinfo)*
Testing get

```plaintext
let staffdb : string =
<<
David Aspinall, da@inf.ed.ac.uk, 1230935, 6.II, 10 London Road, E7 5QA
Ian Stark, stark@inf.ed.ac.uk, 0579035, 7.II, 14A Queen Anne Street, EH1 FZM
>>

test cycleinfos.get staffdb = ?

Produces:

Test result:
"David Aspinall, Map-distance-from: E7 5QA
Ian Stark, Map-distance-from: EH1 FZM"
```
Testing put

test cycleinfos.put
<<
David Aspinall, Map—distance—from: EH7 5QA
Ian Stark, Map—distance—from: EH1 FZM
>>
into staffdb = ?

Produce:

Test result:
"David Aspinall, da@inf.ed.ac.uk, 1230935, 6.II,
  10 London Road, EH7 5QA
Ian Stark, stark@inf.ed.ac.uk, 0579035, 7.II,
  14A Queen Anne Street, EH1 FZM"

(newlines added to fit on slide)
Summary

Bidirectional Programming

- Bidirectional transformations map view updates back to source
- Applications: database views, MDD, UIs, sync, ...
- Foundations: get, put, create, and laws.

Next Lecture

- Boomerang: positions and normalization
- A magic way to get bidirectional transformations

Homework

- Check that the simple lenses shown define functions satisfying the GetPut, PutGet, and CreateGet laws.
- Download Boomerang and try it out.