Advances in Programming Languages

APL16: Bidirectional Programming II

David Aspinall

School of Informatics
The University of Edinburgh

Tuesday 23rd November 2010
Semester 1 Week 10

Topic: Bidirectional Programming and Text Processing

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 more advanced aspects of Boomerang.

Outline

Outline

Boomerang again

```
David Aspinall, da@inf.ed.ac.uk, IF 4.04A
Ian Stark, stark@inf.ed.ac.uk, IF 5.04
Philip Wadler, wadler@inf.ed.ac.uk, IF 5.31
```

A lens with a view of only name and email

```
module Nameemail =
  let NAME = [a−zA−Z ]+ let EMAIL = [a−zA−Z@.]+
  let OFFICE = [AZ0−9. ]+

  let nameemail : lens = NAME . ", " . EMAIL . del ", " . del OFFICE

  let nameemails : lens =
        "" | nameemail . (newline . nameemail)∗
```

Recall: a regexp R is coerced to the lens copy R.

Boomerang failure modes

A Boomerang program can go wrong in these ways:

- **static type-checking error**: the program is malformed.

  For example, the languages defined by regular expressions do not meet semantic requirements (e.g., splittable concatenations).

- **dynamic runtime error**: the program hits an erroneous state during execution.

  For example, a `get` or `put` operation is applied to a string which is not in its domain, because it doesn't belong to the source or view value sets.

  But: strong typing for the lens combinators means that the only place that this may happen will be on inputs directly supplied by the user. In particular, the usual benefit of static typing applies: there should be no obscure failures deep in the program.

Boomerang: testing get

You can write unit tests in Boomerang which specify the expected results:

```
test nameemails.get staffdb =
<<
David Aspinall, da@inf.ed.ac.uk
Ian Stark, stark@inf.ed.ac.uk
Philip Wadler, wadler@inf.ed.ac.uk
>>
```

Putting this together with previous definitions into a file `nameemails.boom` and then executing:

```
boomerang nameemails.boom
```

produces no output, indicating that the test succeeded.

Boomerang: testing put

This test also succeeds:

```
test nameemails.put
<<
David Aspinall, da@inf.ed.ac.uk
Ian Stark, Ian.Stark@ed.ac.uk
Philip Wadler, wadler@inf.ed.ac.uk
>>
into staffdb =
<<
David Aspinall, da@inf.ed.ac.uk, IF 4.04A
Ian Stark, Ian.Stark@ed.ac.uk, IF 5.04
Philip Wadler, wadler@inf.ed.ac.uk, IF 5.31
>>
```

Boomerang: testing put

This test also succeeds:

```
test nameemails.put
<<
David Aspinall, da@inf.ed.ac.uk
Ian Stark, Ian.Stark@ed.ac.uk
Philip Wadler, wadler@inf.ed.ac.uk
>>
into staffdb =
<<
David Aspinall, da@inf.ed.ac.uk, IF 4.04A
Ian Stark, Ian.Stark@ed.ac.uk, IF 5.04
Philip Wadler, wadler@inf.ed.ac.uk, IF 5.31
>>
```

Boomerang: testing put

This test also succeeds:

```
test nameemails.put
<<
David Aspinall, da@inf.ed.ac.uk
Ian Stark, Ian.Stark@ed.ac.uk
Philip Wadler, wadler@inf.ed.ac.uk
>>
into staffdb =
<<
David Aspinall, da@inf.ed.ac.uk, IF 4.04A
Ian Stark, Ian.Stark@ed.ac.uk, IF 5.04
Philip Wadler, wadler@inf.ed.ac.uk, IF 5.31
>>
```

Outline

A failed test

```
test nameemails.put
<<
Ian Stark, Ian.Stark@ed.ac.uk
>>
into staffdb =
<<
David Aspinall, da@inf.ed.ac.uk, IF 4.04A
Ian Stark, Ian.Stark@ed.ac.uk, IF 5.04
Philip Wadler, wadler@inf.ed.ac.uk, IF 5.31
>>
```

The put operation *is* defined, but the output is not what is claimed.

Why not and what is the value of put?

Rearranging the input

Here is another test, where we do not delete records but rearrange them.

```
test nameemails.put
<<
Ian Stark, Ian.Stark@ed.ac.uk
David Aspinall, da@inf.ed.ac.uk
Philip Wadler, wadler@inf.ed.ac.uk
>>
into staffdb = ?
```

Rearranging the input

Here is another test, where we do not delete records but rearrange them.

```
test nameemails.put
<<
Ian Stark, Ian.Stark@ed.ac.uk
David Aspinall, da@inf.ed.ac.uk
Philip Wadler, wadler@inf.ed.ac.uk
>>
into staffdb =
<<
Ian Stark, Ian.Stark@ed.ac.uk, IF 4.04A
David Aspinall, da@inf.ed.ac.uk, IF 5.04
Philip Wadler, wadler@inf.ed.ac.uk, IF 5.31
>>
```

This is the answer we get: but Ian and I have swapped rooms!

Rearranging data

This fails because the interpretation of the lens for the Kleene $*$ (sequence) operator maps updates by position.

So, the first position in the view update is mapped to the first position in the source, and so on.

But in practice, the order of some parts of the data may not be important.

Rearranging data

This fails because the interpretation of the lens for the Kleene $*$ (sequence) operator maps updates by position.

So, the first position in the view update is mapped to the first position in the source, and so on.

But in practice, the order of some parts of the data may not be important.

Can you think of a way to solve this?

Rearranging data

This fails because the interpretation of the lens for the Kleene $*$ (sequence) operator maps updates by position.

So, the first position in the view update is mapped to the first position in the source, and so on.

But in practice, the order of some parts of the data may not be important.

Can you think of a way to solve this?

Note: with a semantics based on *update operations* rather than a single view state, this would be handled automatically, because data insertions, deletions and movement could be recorded. This option is not (automatically) available to Boomerang, so rearrangements must be deduced.

Resourceful Lenses in Boomerang

To solve this, Boomerang introduces "chunks" $\langle s \rangle$, which are pieces of data inside imagined markers $\langle$ and $\rangle$. Data can be reordered between chunks.

The source and view data sets are split up into:

- the *rigid* parts — a "skeleton" describing the shape of the data, where chunks are placeholders for data;
- the *resource* — the data carried by the skeleton, which fill in the chunks.

The programmer specifies where chunks may appear and how they are aligned. The underlying functions for the lens compute transformations to reorder the data as needed.

Resourceful Lenses in Boomerang

To solve this, Boomerang introduces “chunks” $\langle s \rangle$, which are pieces of data inside imagined markers $\langle$ and $\rangle$. Data can be reordered between chunks.

The source and view data sets are split up into:

- the *rigid* parts — a “skeleton” describing the shape of the data, where chunks are placeholders for data;
- the *resource* — the data carried by the skeleton, which fill in the chunks.

The programmer specifies where chunks may appear and how they are aligned. The underlying functions for the lens compute transformations to reorder the data as needed.

The precise semantics is involved: the data language is extended with chunk markers and erasure and skeleton functions; each lens requires a new function and is constrained by 8 new laws.

A Reorderable Address List

The programmer gains a few new language features.

```
module Nameemail2 =

  let NAME = [a-zA-Z ]+
  let EMAIL = [a-zA-Z@.]+
  let OFFICE = [A-Z0-9. ]+

  let nameemail : lens =
         key NAME . ", " . EMAIL . del ", " . del OFFICE

  let nameemails : lens =
      "" | <nameemail> . (newline . <nameemail>)*
```

A Reorderable Address List: testing

With this alteration, we get a correct behaviour from the previous update example.

```
test nameemails.put
<<
Ian Stark, Ian.Stark@ed.ac.uk
David Aspinall, da@inf.ed.ac.uk
Philip Wadler, wadler@inf.ed.ac.uk
>>
into staffdb =
<<
Ian Stark, Ian.Stark@ed.ac.uk, IF 5.04
David Aspinall, da@inf.ed.ac.uk, IF 4.04A
Philip Wadler, wadler@inf.ed.ac.uk, IF 5.31
>>
```

Heuristics for alignment

Boomerang extends this basic idea with additional practical features.

- The data sets may contain multiple skeletons, with different kinds of data chunks aligned independently. These are specified with **tags**:

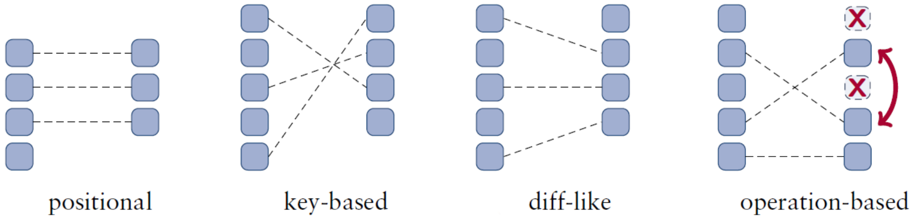
  ```
  <tag : lens>
  ```

  A tag contains a name, but can also specify alignment strategy.

- Alignment strategies available include:

positional	the default, aligning sequentially
key-based	using keys where unique, then a cost heuristic
diff-like	minimising edit distances
operation-based	if the operations are recorded somehow, the reordering can be supplied to the update.

Varieties of alignment



See the Boomerang user manual for more on how to program with these different mechanisms.

Credit: picture above taken from J. N. Foster's PhD thesis *Bidirectional Programming Languages*, University of Pennsylvania, 2009.

Outline

The need for normalisation

```
test nameemails.put
<<
Ian Stark,  Ian.Stark@ed.ac.uk
David Aspinall, da@inf.ed.ac.uk
Philip Wadler, wadler@inf.ed.ac.uk
>>
into staffdb = ?
```

Produces a long error message:

```
File "nameemail2.boom", ... Test result: error
File "nameemail2.boom", ... run−time checking error
v="Ian Stark, Ian.Stark@ed.ac.uk...
did not satisfy  ...  string does not match ...
 ...  [Ian Stark, ] AROUND HERE [ Ian.Stark ...
```

The need for normalisation

```
test␣nameemails.put
<<
Ian␣Stark,␣␣Ian.Stark@ed.ac.uk
David␣Aspinall,␣da@inf.ed.ac.uk
Philip␣Wadler,␣wadler@inf.ed.ac.uk
>>
into␣staffdb␣=␣?
```

Produces a long error message:

```
File "nameemail2.boom", ... Test result: error
File "nameemail2.boom", ... run−time checking error
v="Ian Stark, Ian.Stark@ed.ac.uk...
did not satisfy  ...  string does not match ...
 ... [Ian Stark, ] AROUND HERE [ Ian.Stark ...
```

— the view update had one too many spaces!

Approaches to Normalisation

In practice we want to allow for extra white space, noisy input, etc, where it doesn't matter: inputs should be normalised or made *canonical* before being converted.

However, this may break the PutGet law. (Ex: why?)

Solutions that researchers have considered include:

- give up or alter the laws;
- add functions to lenses which convert to/from canonical form;
- use *equivalence relations* in the laws.

Quotient Lenses: lenses with equivalence relations

Equivalence relations can be added easily to the semantic framework, by supposing that each lens has a pair of equivalence relations built in:

- $\sim_{\mathrm{S}}$ on its source S;
- $\sim_{\mathrm{V}}$ on its view V.

Recall that an equivalence relation $\sim$ is a relation that behaves like equality: it is reflexive, symmetric and transitive.

A subscript on $\sim$ like $\sim_{\mathrm{T}}$ is denotes the set the relation belongs to.

The lens functions must *respect equivalence*, meaning that equivalent inputs are mapped to equivalent outputs.

Equality (identity) itself is an equivalence relation; it is the default choice for lenses.

Laws for Quotient Lenses

The laws are modified to hold up to equivalence:

PutGet	$get(put(v', s))$	$\sim_V$	v'
GetPut	$put(get(s), s)$	$\sim_S$	s
CreateGet	$get(create(v))$	$\sim_V$	v

The programming language is then extended with constructs to allow coarser equivalences to be defined on lenses.

For example, we may want a particular lens to consider all sequences of whitespace characters as equivalent to a single whitespace character.

Building quotient lenses: canonizers

Suppose S and U are sets with equivalences, where the equivalence on S is coarser (e.g., equating sequences of whitespace) than on U (which only has canonical amounts of whitespace).

Quotient lenses can be introduced with *canonizers* for S and U, a pair of functions:

- $canonize : S \to U$
- $choose : U \to S$

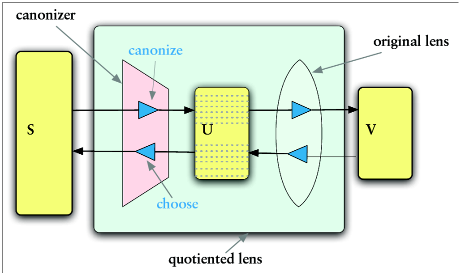
such that

$$canonize(choose(u)) \sim_U u$$

for all $u \in U$. In other words, choosing a representative element for u and then canonizing again must give us back the same element.

A canonizer is a bit like a lens without put. Indeed, lenses can be used to define canonizers using their get and $create$ functions.

Quotient on the left



The *left quotient* operation makes a new lens from an old one by joining a canonizer on the left (it composes the get function from S with the $canonize$ function).

Intuitively, this collapses some of the source values in S.

Picture credit: Foster's PhD thesis.

Quotient delete

Recall del, which deletes a string from the source with get, and restores it with put.

Since deletion removes an ignored part of the input, it's a natural place to collapse some input values.

The (quotient) lens $qdel\ \mathrm{R}\ s$ deletes *any* string from R in the forwards direction, treating strings from R as equivalent. In the reverse direction, it restores s as the chosen string.

(For well-formedness, it must be the case that $s \in \mathrm{R}$).

Quotient insert

Symmetrically, a *right quotient* collapses some of the view values. It composes a lens with a canonizer on the right. The canonization happens in the put function.

Recall $ins\, v$, which inserts a fixed string v into the view and has the empty string as its source set.

The quotient lens version $qins\, R\, v$ is like $ins\, v$ in the get direction. In the put direction, it accepts any string from R, not just v.

(For well-formedness, it must be the case that $v \in R$).

Addresses with quotients: flexible, normalised whitespace

```
let NAME = [a-zA-Z ]+
let EMAIL = [a-zA-Z@.]+
let OFFICE = [A-Z0-9.] . [A-Z0-9. ]+
let SEP = "," . [ \t]*

let nameemail : lens =
        key NAME .
        (qdel SEP "," . qins SEP ",\t\t") . EMAIL .
        qdel SEP "," . del OFFICE
```

```
let staffdb : string =
<<
David Aspinall, da@inf.ed.ac.uk,     IF 4.04A
Ian Stark,      stark@inf.ed.ac.uk,  IF 5.04
Philip Wadler,  wadler@inf.ed.ac.uk, IF 5.31
>>
```

Addresses with quotients: flexible, normalised whitespace

```
let NAME = [a−zA−Z ]+
let EMAIL = [a−zA−Z@.]+
let OFFICE = [A−Z0−9.] . [A−Z0−9. ]+
let SEP = "," . [ \t]*

let nameemail : lens =
        key NAME .
        (qdel SEP "," . qins SEP ",\t\t") . EMAIL .
        qdel SEP "," . del OFFICE
```

```
nameemails.get staffdb =
```

```
David Aspinall,         da@inf.ed.ac.uk
Ian Stark,              stark@inf.ed.ac.uk
Philip Wadler,          wadler@inf.ed.ac.uk"
```

Addresses with quotients: flexible, normalised whitespace

```
let NAME = [a−zA−Z ]+
let EMAIL = [a−zA−Z@.]+
let OFFICE = [A−Z0−9.] . [A−Z0−9. ]+
let SEP = "," . [ \t]∗

let nameemail : lens =
        key NAME .
        (qdel SEP "," . qins SEP ",\t\t") . EMAIL .
        qdel SEP "," . del OFFICE
```

```
nameemails.put
<<
Ian Stark,Ian.Stark@ed.ac.uk
David Aspinall,       da@inf.ed.ac.uk
Philip Wadler,    wadler@inf.ed.ac.uk
>> into staffdb = ?
```

Addresses with quotients: flexible, normalised whitespace

```
let NAME = [a−zA−Z ]+
let EMAIL = [a−zA−Z@.]+
let OFFICE = [A−Z0−9.] . [A−Z0−9. ]+
let SEP = "," . [ \t]*

let nameemail : lens =
        key NAME .
        (qdel SEP "," . qins SEP ",\t\t") . EMAIL .
        qdel SEP "," . del OFFICE
```

```
Ian Stark,Ian.Stark@ed.ac.uk,IF 5.04
David Aspinall,da@inf.ed.ac.uk,IF 4.04A
Philip Wadler,wadler@inf.ed.ac.uk,IF 5.31
```

Outline

Another route to bidirectional programming

We suggested solving bidirectional programming by these ways:

- **meta-programming** with a general purpose language
- using a **DSL** with a custom abstraction

Boomerang follows the second approach.

Is there a third approach?

- **fully automatically**: derive bidirectional programs from ordinary one-way programs, without *meta*-programming

Sounds tricky...

Bidirectionalization for Free!

Janis Voigtländer.
Bidirectionalization for free! (Pearl).
In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA.*, pages 165–176, ACM, 2009.

Voigtländer's paper shows how to write automatic bidirectionalizers in Haskell. Given a `get` function with a polymorphic type, his higher order function will return a corresponding `put` function as if by magic, *without examining the syntax* of `get`.

Bidirectionalization for Free!

Janis Voigtländer.
Bidirectionalization for free! (Pearl).
In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA.*, pages 165–176, ACM, 2009.

Voigtländer's paper shows how to write automatic bidirectionalizers in Haskell. Given a `get` function with a polymorphic type, his higher order function will return a corresponding `put` function as if by magic, *without examining the syntax* of `get`.

How it works. Behind the scenes, Voigtländer's function exploits the polymorphic type of `get` to execute it on test input data. This allows tracking rearrangements of the input into the output, automatically finding mappings like those in Boomerang's resourceful lenses. The mappings are used to define a corresponding `put` function. The solution is ingenious, especially in that the PutGet and GetPut laws are proved. However, efficiency reasons mean that it is best used for prototyping rather than for final versions of bidirectional transformations.

Outline

Summary

Bidirectional programming

- Bidirectional transformations map view updates back to source.
- Applications: database views, MDD, UIs, sync, ...
- Foundations: get, put, create, and their laws.
- Boomerang: resourceful lenses and quotient lenses.
- Bidirectionalization for free.

Next lecture

- XML processing with CDuce

Homework

- Try out more advanced examples in Boomerang, including the supplied examples and devising your own.