Fun With String Lenses

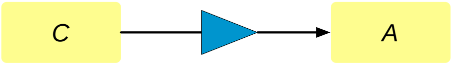
Benjamin C. Pierce
University of Pennsylvania

WG 2.8, July 2007

My usual obsession...

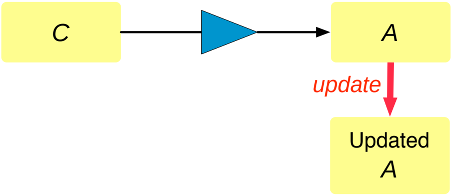
The View Update Problem

- We transform source structure C to target structure A



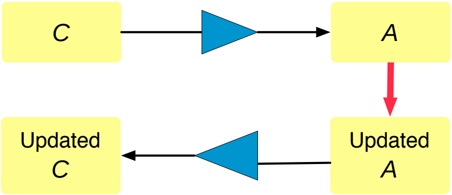
The View Update Problem

- We transform source structure C to target structure A
- Someone updates A



The View Update Problem

- We transform source structure C to target structure A
- Someone updates A
- We must now translate this update to obtain an appropriately updated C



A Bad “Solution”

We *could* just write such pairs of functions in any old programming language.

- But this would be ugly and unmaintainable!

A Good Solution

Better: take a linguistic approach.

- Design a bi-directional programming language, in which every expression can be read...
  - from left to right as a *get* function
  - from right to left as the corresponding *put* function

Pieces of the puzzle:

- A semantic space of pairs of functions that "behave well together" (dubbed lenses)
- Natural, convenient syntax with a compositional semantics
- Static type system guaranteeing well-behavedness and totality

Lenses For Trees [POPL 2005, PLANX 2007]

Data model: Trees (XML, etc.)

Computation model: Local tree manipulation combinators, plus mapping, conditionals, recursion.

Type system: Based on regular tree automata

- with some interesting side-conditions

Lenses For Relations [PODS 2006]

Data model: Relational databases (named collections of tables)

Computation model: Operators from relational algebra, each augmented with enough parameters to determine *put* behavior.

Type system: Built using standard tools from databases

- predicates on rows of tables
- functional dependencies between columns

Lenses for Strings [in progress]

Data model: Strings over a finite alphabet

Computation model: Finite-state string transducers, described using regular-expression-like operators

Type system: Regular expressions

- with some interesting side conditions

What We're Up To

1. *string lens combinators*
   - intuitive semantics and typing rules
   - based on familiar regular operators (union, concatenation, Kleene-star).
2. *dictionary lenses* (and two more combinators) for dealing with ordered data
3. *Boomerang*: a full-blown *bidirectional programming language*
4. Pretty big examples
   - e.g., SwissProt ascii $\longleftrightarrow$ XML (2Kloc)

Bottom line: Finally, a bi-directional language that is (pretty) easy to learn and (a lot of) fun to use.

String Lenses

Semantics of Basic Lenses

A basic lens l from C to A is a triple of functions

$$\begin{array}{lcl} l.get & \in & C \longrightarrow A \\ l.put & \in & A \longrightarrow C \longrightarrow C \\ l.create & \in & A \longrightarrow C \end{array}$$

obeying three "round-tripping" laws:

$$l.put\ (l.get\ c)\ c = c \qquad (\textsc{GetPut})$$

$$l.get\ (l.put\ a\ c) = a \qquad (\textsc{PutGet})$$

$$l.get\ (l.create\ a) = a \qquad (\textsc{CreateGet})$$

[Switch to demo]

String Lens Primitives

Copy

$$\frac{E \in \mathcal{R}}{cp\ E \in [\![E]\!] \Longleftrightarrow [\![E]\!]}$$

$$\begin{array}{lcl} get\ c & = & c \\ put\ a\ c & = & a \\ create\ a & = & a \end{array}$$

Const

$$\frac{E \in \mathcal{R} \qquad u \in \Sigma^* \qquad v \in [\![E]\!]}{const\ E\ u\ v \in [\![E]\!] \Longleftrightarrow \{u\}}$$

$$\begin{aligned} get\ c &= u \\ put\ a\ c &= c \\ create\ a &= v \end{aligned}$$

Derived Forms

$$
\begin{array}{rcl}
E \leftrightarrow u & \in & [\![E]\!] \Longleftrightarrow \{u\} \\
E \leftrightarrow u & = & const\ E\ u\ (choose(E)) \\
del\ E & \in & [\![E]\!] \Longleftrightarrow \{\epsilon\} \\
del\ E & = & E\ \leftrightarrow \epsilon \\
ins\ u & \in & \{\epsilon\} \Longleftrightarrow \{u\} \\
ins\ u & = & \epsilon \leftrightarrow u
\end{array}
$$

Concatenation

$$\frac{C_1 \cdot^! C_2 \qquad A_1 \cdot^! A_2 \\ l_1 \in C_1 \Longleftrightarrow A_1 \qquad l_2 \in C_2 \Longleftrightarrow A_2}{l_1 \cdot l_2 \in C_1 \cdot C_2 \Longleftrightarrow A_1 \cdot A_2}$$

$$\begin{array}{lcl}
get\ (c_1 \cdot c_2) & = & (l_1.get\ c_1) \cdot (l_2.get\ c_2) \\
put\ (a_1 \cdot a_2)\ (c_1 \cdot c_2) & = & (l_1.put\ a_1\ c_1) \cdot (l_2.put\ a_2\ c_2) \\
create\ (a_1 \cdot a_2) & = & (l_1.create\ a_1) \cdot (l_2.create\ a_2)
\end{array}$$

Iteration

$$\frac{l \in C \Longleftrightarrow A \qquad C^{!*} \qquad A^{!*}}{l^* \in C^* \Longleftrightarrow A^*}$$

$$\begin{array}{lcl}
\mathit{get}\ (c_1 \cdots c_n) & = & (l.\mathit{get}\ c_1) \cdots (l.\mathit{get}\ c_n) \\
\mathit{put}\ (a_1 \cdots a_n)\ (c_1 \cdots c_m) & = & c'_1 \cdots c'_n \\
\quad \text{where } c'_i = \begin{cases} l.\mathit{put}\ a_i\ c_i & i \in \{1, \ldots, \min(m, n)\} \\ l.\mathit{create}\ a_i & i \in \{m+1, \ldots, n\} \end{cases} & & \\
\mathit{create}\ (a_1 \cdots a_n) & = & (l.\mathit{create}\ a_1) \cdots (l.\mathit{create}\ a_n)
\end{array}$$

Union

$$\frac{C_1 \cap C_2 = \emptyset \qquad l_1 \;\in\; C_1 \Longleftrightarrow A_1 \qquad l_2 \;\in\; C_2 \Longleftrightarrow A_2}{l_1 \mid l_2 \;\in\; C_1 \cup C_2 \Longleftrightarrow A_1 \cup A_2}$$

$$\begin{aligned}
get\ c &= \begin{cases} l_1.get\ c & \text{if } c \in C_1 \\ l_2.get\ c & \text{if } c \in C_2 \end{cases} \\
put\ a\ c &= \begin{cases} l_1.put\ a\ c & \text{if } c \in C_1 \wedge a \in A_1 \\ l_2.put\ a\ c & \text{if } c \in C_2 \wedge a \in A_2 \\ l_1.create\ a & \text{if } c \in C_2 \wedge a \in A_1 \setminus A_2 \\ l_2.create\ a & \text{if } c \in C_1 \wedge a \in A_2 \setminus A_1 \end{cases} \\
create\ a &= \begin{cases} l_1.create\ a & \text{if } a \in A_1 \\ l_2.create\ a & \text{if } a \in A_2 \setminus A_1 \end{cases}
\end{aligned}$$

[back to demo]

Dictionary Lenses

Semantics of Dictionary Lenses

$l \in C \overset{S,D}{\Longleftrightarrow} A$ if...

$$\begin{aligned} l.get &\in C \longrightarrow A \\ l.parse &\in C \longrightarrow S \times D \\ l.key &\in A \longrightarrow K \\ l.create &\in A \longrightarrow D \longrightarrow C \times D \\ l.put &\in A \longrightarrow S \times D \longrightarrow C \times D \end{aligned}$$

...obeying...

$$\frac{s, d' = l.parse\ c \qquad d \in D}{l.put\ (l.get\ c)\ (s, (d' +\!\!+\ d)) = c, d} \quad (\textsc{GetPut})$$

$$\frac{c, d' = l.put\ a\ (s, d)}{l.get\ c = a} \quad (\textsc{PutGet})$$

$$\frac{c, d' = l.create\ a\ d}{l.get\ c = a} \quad (\textsc{CreateGet})$$

Boomerang

A full-blown language based on dictionary lenses

- A simply typed functional language with base types:
  - `string`
  - `regexp`
  - `dlens`
- ... and primitives:

```
get : dlens -> string -> string
put : dlens -> string -> string -> string
create : dlens -> string -> string

union : dlens -> dlens -> dlens
concat : dlens -> dlens -> dlens
...
```

Two-stage typechecking

Problem:

- Our lens combinators have types involving regular expressions
- The functional component of Boomerang involves arrow types
- Not clear how to mix them!

Two-stage typechecking

A pretty reasonable solution:

- Typecheck functional program (using simple types)
- Executing it involves applying operators like `concat` to `dlens` values
- `dlens` values include (functional) components `get`, `put`, etc., and (regular expression) components `domain`, `codomain`, etc.
- evaluating `concat` *dynamically* applies the *static* typing rule for lens concatenation (using
- if this succeeds, then the resulting `dlens` can be further composed, or applied to a string using `get`, etc.

Thank You!

Collaborators on this work: Aaron Bohannon, Nate Foster, Alexandre Pilkiewicz, Alan Schmitt

Other Harmony contributors: Ravi Chugh, Malo Denielou, Michael Greenwald, Owen Gunden, Martin Hofmann, Sanjeev Khanna, Keshav Kunal, Stéphane Lescuyer, Jon Moore, Jeff Vaughan, Zhe Yang

Resources: Papers, slides, (open) source code, and online demos:

http://www.seas.upenn.edu/~harmony/

The Real Semantics of Dictionary Lenses

A dictionary lens from C to A with skeleton type S and dictionary type D has components...

$$
\begin{array}{rcl}
l.get & \in & C \longrightarrow A \\
l.parse & \in & C \longrightarrow S \times D(L) \\
l.key & \in & A \longrightarrow K \\
l.create & \in & A \longrightarrow D(L) \longrightarrow C \times D(L) \\
l.put & \in & A \longrightarrow S \times D(L) \longrightarrow C \times D(L)
\end{array}
$$

... where...

$$
\frac{s, d' = l.parse\ c \qquad d \in D(L)}{l.put\ (l.get\ c)\ (s, (d' \mathbin{++} d)) = c, d} \qquad \text{(GetPut)}
$$

$$
\frac{c, d' = l.put\ a\ (s, d)}{l.get\ c = a} \qquad \text{(PutGet)}
$$

$$
\frac{c, d' = l.create\ a\ d}{l.get\ c = a} \qquad \text{(CreateGet)}
$$