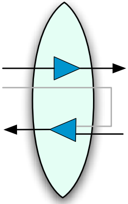
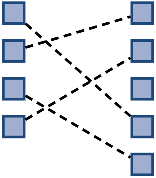
Matching Lenses



Davi M. J. Barbosa (Polytechnique)
Julien Cretin (Polytechnique/INRIA)
Nate Foster (Cornell)
Michael Greenberg (Penn)
Benjamin C. Pierce (Penn)



ICFP '10

Example

```
=History (5 pts)=
List the inventors of the
following programming languages.
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58    (* McCarthy *)
* ML 73      (* Gordon,Milner *)
=Scoping (2 pts)=
Which of these terms are closed?
* λx.λy.x           (* Yes *)
* (λx.λz.x) λx.λy.z (* No *)
=Lambda Calculus (3 pts)=
Give a weakly normalizing term
which is not strongly normalizing.
(* (λx.λy.y) ((λx.x x) λx.x x) *)
```

Example

```
=History (5 pts)=
List the inventors of the
following programming languages.
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58    (* McCarthy *)
* ML 73      (* Gordon,Milner *)
=Scoping (2 pts)=
Which of these terms are closed?
* λx.λy.x           (* Yes *)
* (λx.λz.x) λx.λy.z (* No *)
=Lambda Calculus (3 pts)=
Give a weakly normalizing term
which is not strongly normalizing.
(* (λx.λy.y) ((λx.x x) λx.x x) *)
```

```
=History=
List the inventors of the
following programming languages.
* Haskell 98
* LISP 58
* ML 73
=Scoping=
Which of these terms are closed?
* λx.λy.x
* (λx.λz.x) λx.λy.z
=Lambda Calculus=
Give a weakly normalizing term
which is not strongly normalizing.
```

Example

```
=History (5 pts)=
List the inventors of the
following programming languages.
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58    (* McCarthy *)
* ML 73      (* Gordon,Milner *)
=Scoping (2 pts)=
Which of these terms are closed?
* λx.λy.x           (* Yes *)
* (λx.λz.x) λx.λy.z (* No *)
=Lambda Calculus (3 pts)=
Give a weakly normalizing term
which is not strongly normalizing.
(* (λx.λy.y) ((λx.x x) λx.x x) *)
```

=History=
List the inventors of the

```
=History=
List the inventors of the
following programming languages.
* LISP 58
* ML 73
* OCaml 87
* Haskell 90
=Combinators=
Give the equations for S and K in
a combinatory algebra.
=Scoping=
Which of these terms are closed?
* λx.λy.x
* λx.(λy.y) y
* (λx.λz.x) λx.λy.z
=Lambda Calculus=
Give a weakly normalizing term
which is not strongly normalizing.
```

Example

```
=History (5 pts)=
List the inventors of the
following programming languages.
* LISP 58     (* McCarthy *)
* ML 73       (* Gordon,Milner *)
* OCaml 87 (* TODO: answer *)
* Haskell 90 (* Hudak,PJ,Wadler *)
=Combinators (? pts)=
Give the equations for S and K in
a combinatory algebra.
(* TODO: write the answer *)
=Scoping (2 pts)=
Which of these terms are closed?
* λx.λy.x              (* Yes *)
* λx.(λy.y) y (* TODO: answer *)
* (λx.λz.x) λx.λy.z (* No *)
=Lambda Calculus (3 pts)=
Give a weakly normalizing term
which is not strongly normalizing.
(* (λx.λy.y) ((λx.x x) λx.x x) *)
```

```
=History=
List the inventors of the
following programming languages.
* LISP 58
* ML 73
* OCaml 87
* Haskell 90
=Combinators=
Give the equations for S and K in
a combinatory algebra.
=Scoping=
Which of these terms are closed?
* λx.λy.x
* λx.(λy.y) y
* (λx.λz.x) λx.λy.z
=Lambda Calculus=
Give a weakly normalizing term
which is not strongly normalizing.
```

Example

```
=History (5 pts)=
List the inventors of the
following programming languages.
* LISP 58    (* McCarthy *)
* ML 73      (* Gordon,Milner *)
* OCaml 87 (* TODO: answer *)
* Haskell 90 (* Hudak,PJ,Wadler *)
=Combinators (? pts)
Give the equations f
a combinatory algebr
(* TODO: write the a
=Scoping (2 pts)=
Which of these terms are closed?
* λx.λy.x           (* Yes *)
* λx.(λy.y) y (* TODO: answer *)
* (λx.λz.x) λx.λy.z (* No *)
=Lambda Calculus (3 pts)=
Give a weakly normalizing term
which is not strongly normalizing.
(* (λx.λy.y) ((λx.x x) λx.x x) *)
```

```
=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
```

```
=History=
List the inventors of the
following programming languages.
* LISP 58
* ML 73
* OCaml 87
* Haskell 90
s for S and K in
ebra.
Which of these terms are closed?
* λx.λy.x
* λx.(λy.y) y
* (λx.λz.x) λx.λy.z
=Lambda Calculus=
Give a weakly normalizing term
which is not strongly normalizing.
```

Basic lens with complement

A lens l is between a source set S and a view set V, and over a complement set C.

Notation: $l \in S \overset{C}{\Longleftrightarrow} V$

Basic lens with complement

A lens l is between a source set S and a view set V, and over a complement set C.

Notation: $l \in S \stackrel{C}{\Longleftrightarrow} V$

The source S contains all the information (the full exam).

```
=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
```

Basic lens with complement

A lens l is between a source set S and a view set V, and over a complement set C.

Notation: $l \in S \overset{C}{\Longleftrightarrow} V$

The view V has less information than the source (we don't show the answers and number of points).

```
=History=
* Haskell 98
* LISP 58
* ML 73
```

Basic lens with complement

A lens l is between a source set S and a view set V, and over a complement set C.

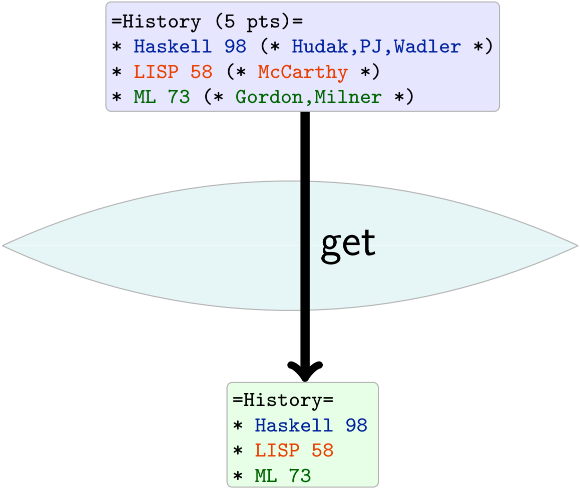
Notation: $l \in S \overset{C}{\Longleftrightarrow} V$

The complement C represents the missing information (the answers and number of points).

5 pts	Hudak,PJ,Wadler McCarthy Gordon,Milner

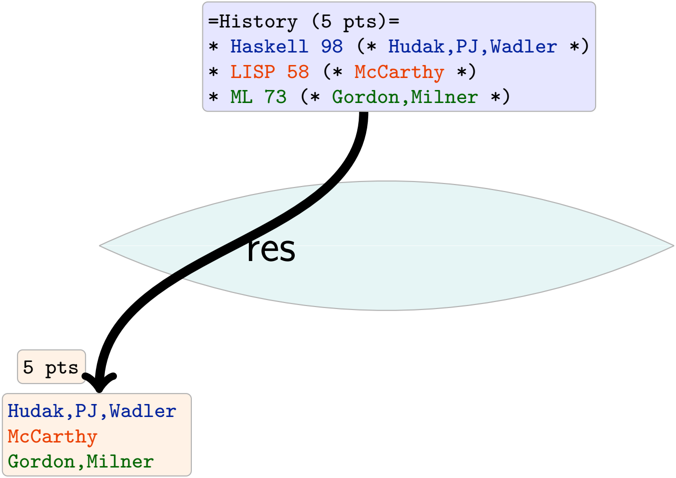
Basic lens with complement

A lens comes with three functions: get,



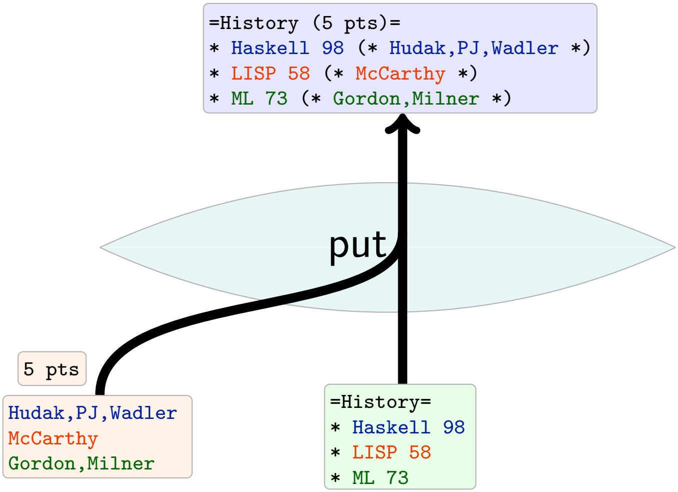
Basic lens with complement

A lens comes with three functions: get, res



Basic lens with complement

A lens comes with three functions: get, res and put.



Basic lens with complement

$$
\begin{array}{lcl}
l.get & \in & S \to V \\
l.res & \in & S \to C \\
l.put & \in & V \to C \to S
\end{array}
$$

These functions obey two round-tripping laws, explaining the interoperation between get, res and put.

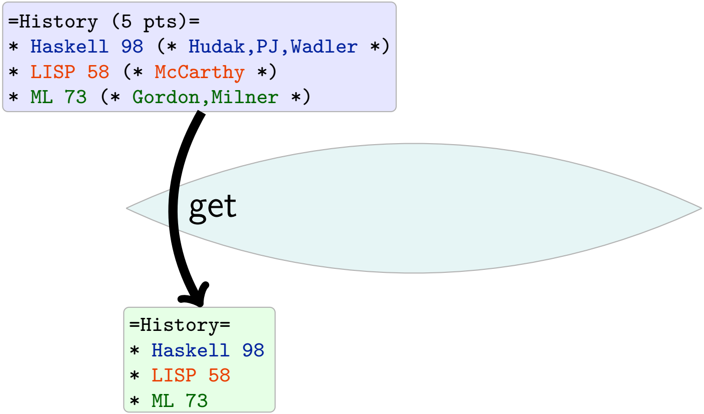
$$l.get\ (l.put\ v\ c) = v \qquad (\textsc{PutGet})$$

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

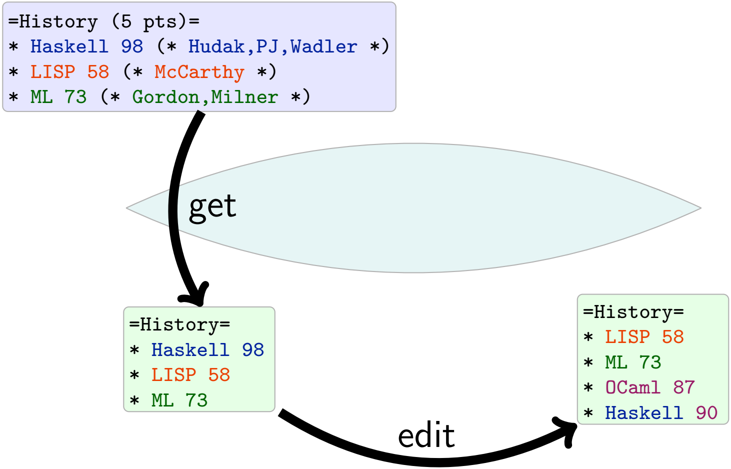
The alignment problem

```
=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
```

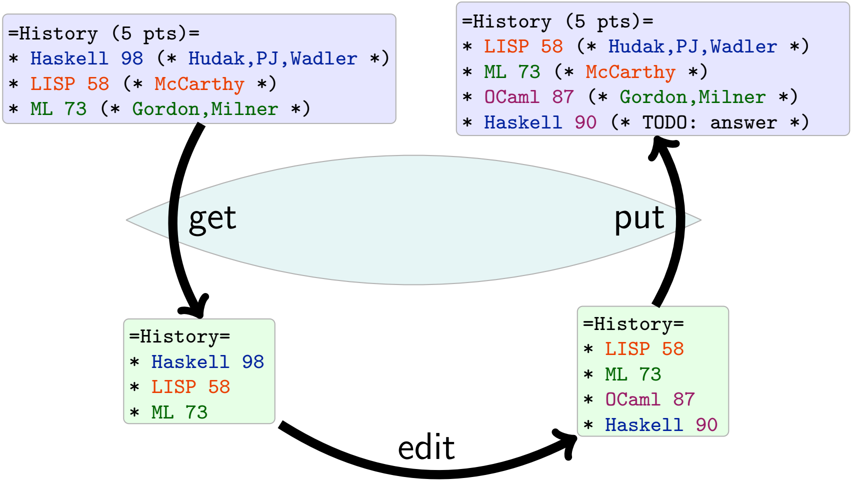
The alignment problem



The alignment problem



The alignment problem



Challenges

- This problem is fundamentally heuristic
  - "state-based" lens only sees the result of edit
  - user intent must be inferred
- Appropriate heuristic depends on the application
- How to fit these heuristic behaviors into our principled lens framework?
  - how to formulate clean semantic laws involving "user intent"?

[No previous work fully solves this problem.]

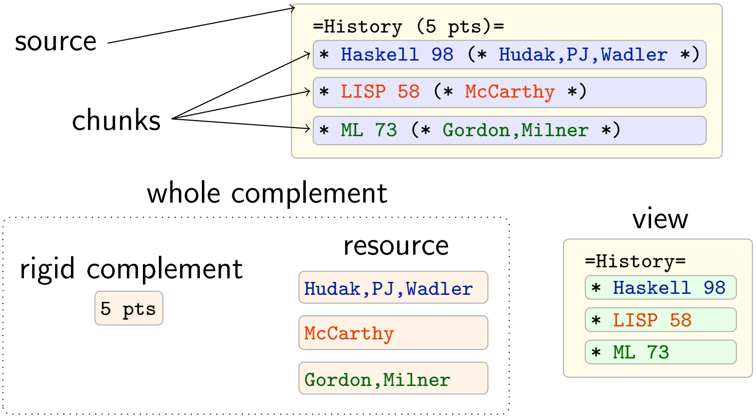
Matching Lenses

Goals:

- General solution (applicable to many heuristics)
- Clean theory (core laws parametrized on heuristics)

Structures with chunks

In order to express the behavior of the put function in presence of view edits, we need to add structure to the source, view and complement types.



Plan

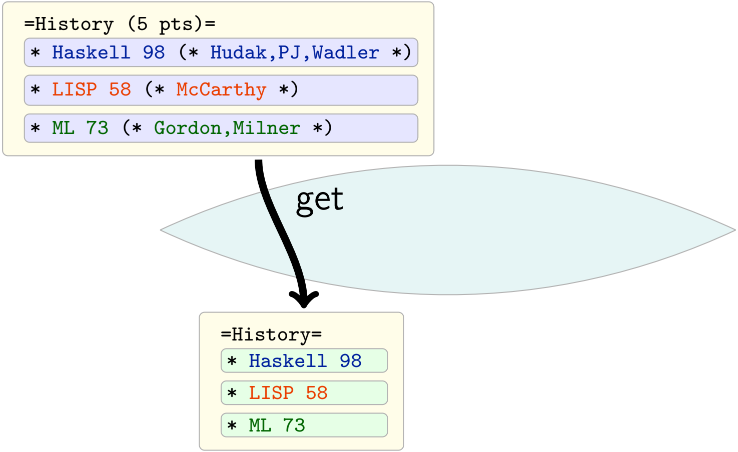
- Start with something simple
  - *get* does not permute the items
  - items are not nested
  - only one sublens is used for all items
- Understand it fully
- Relax these simplifications

Simple matching lenses

Mechanism

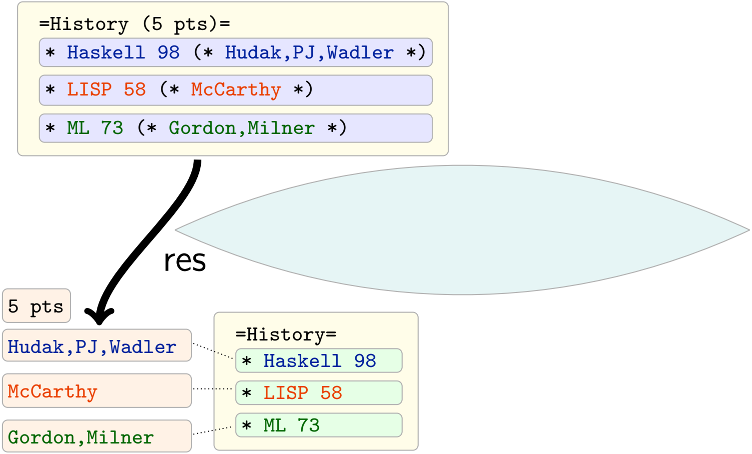
```
=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
```

Mechanism

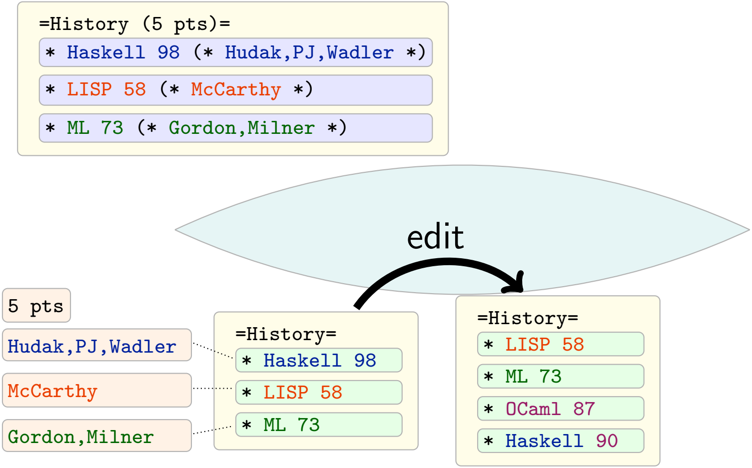


=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
get
=History=
* Haskell 98
* LISP 58
* ML 73

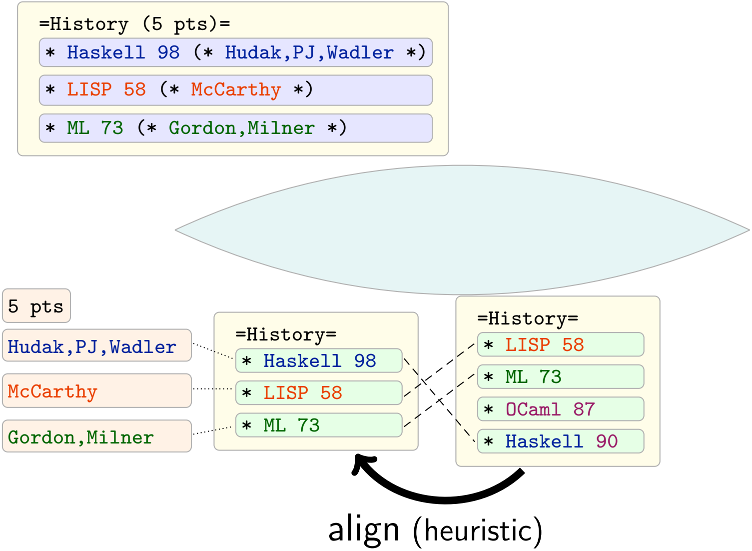
Mechanism


=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
res
5 pts
Hudak,PJ,Wadler
McCarthy
Gordon,Milner
=History=
* Haskell 98
* LISP 58
* ML 73

Mechanism

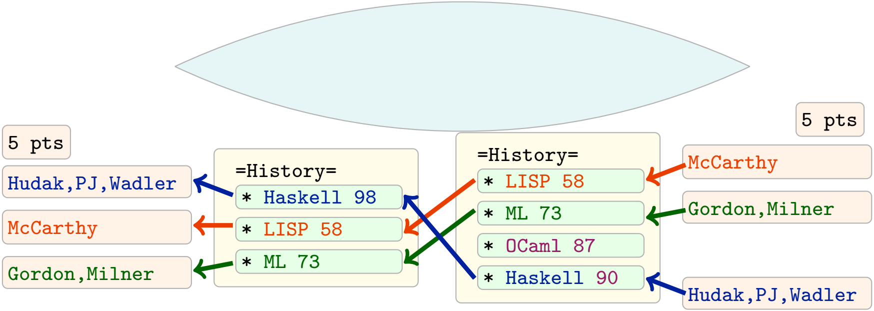

=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
edit
5 pts
Hudak,PJ,Wadler
McCarthy
Gordon,Milner
=History=
* Haskell 98
* LISP 58
* ML 73
=History=
* LISP 58
* ML 73
* OCaml 87
* Haskell 90

Mechanism

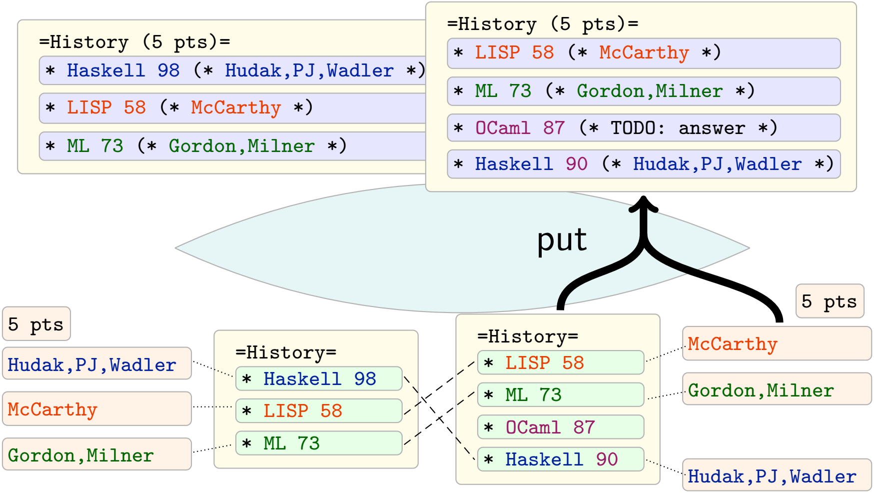

=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
5 pts
Hudak,PJ,Wadler
McCarthy
Gordon,Milner
=History=
* Haskell 98
* LISP 58
* ML 73
=History=
* LISP 58
* ML 73
* OCaml 87
* Haskell 90
align (heuristic)

Mechanism

```
=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
```



Mechanism



=History (5 pts)=
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
=History (5 pts)=
* LISP 58 (* McCarthy *)
* ML 73 (* Gordon,Milner *)
* OCaml 87 (* TODO: answer *)
* Haskell 90 (* Hudak,PJ,Wadler *)
put
5 pts
Hudak,PJ,Wadler
McCarthy
Gordon,Milner
=History=
* Haskell 98
* LISP 58
* ML 73
=History=
* LISP 58
* ML 73
* OCaml 87
* Haskell 90
5 pts
McCarthy
Gordon,Milner
Hudak,PJ,Wadler

Matching lenses

A matching lens l is between S and V, and over a *rigid* complement C and a basic lens k.

We split the complement in two parts: a rigid complement C, and a resource (reorderable part) $\{|\mathbb{N} \mapsto C_k|\}$.

$$
\begin{aligned}
l.get &\in S \to V \\
l.res &\in S \to \boxed{\boxed{C} \times \boxed{\{|\mathbb{N} \mapsto C_k|\}}} \\
l.put &\in V \to \boxed{\boxed{C} \times \boxed{\{|\mathbb{N} \mapsto C_k|\}}} \to S
\end{aligned}
$$

whole complement

Matching lenses

A matching lens l is between S and V, and over a *rigid* complement C and a basic lens k.

We split the complement in two parts: a rigid complement C, and a resource (reorderable part) $\{|\mathbb{N} \mapsto C_k|\}$.

$$
\begin{array}{lcl}
l.get & \in & S \to V \\
l.res & \in & S \to C \times \{|\mathbb{N} \mapsto C_k|\} \\
l.put & \in & V \to C \times \{|\mathbb{N} \mapsto C_k|\} \to S
\end{array}
$$

whole complement

rigid complement

Matching lenses

A matching lens l is between S and V, and over a *rigid* complement C and a basic lens k.

We split the complement in two parts: a rigid complement C, and a resource (reorderable part) $\{|\mathbb{N} \mapsto C_k|\}$.

$$
\begin{array}{lcl}
l.get & \in & S \to V \\
l.res & \in & S \to C \times \{|\mathbb{N} \mapsto C_k|\} \\
l.put & \in & V \to C \times \{|\mathbb{N} \mapsto C_k|\} \to S
\end{array}
$$

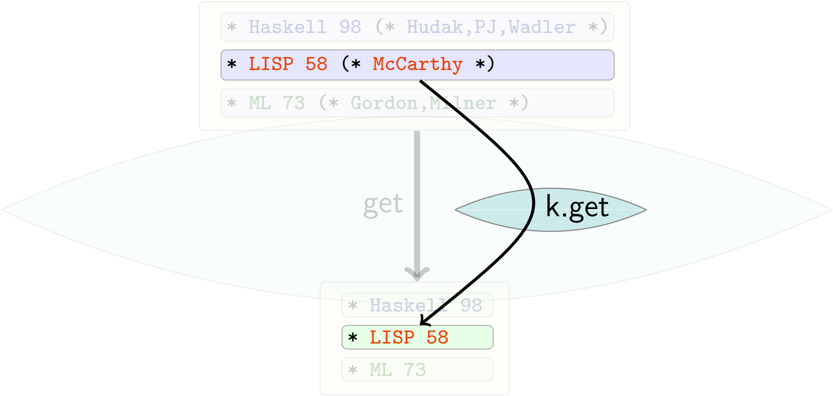
whole complement

rigid complement

resource (reorderable part)

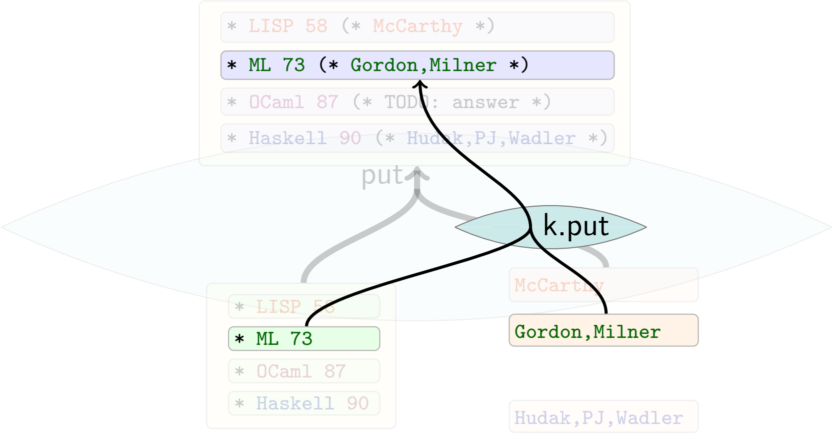
ChunkGet

We add new laws guiding how the lens operate.



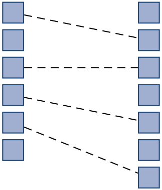
ChunkPut

We add new laws guiding how the lens operate in presence of view edits.

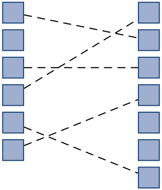


Heuristics

We can now get benefit of our framework, by considering several heuristics.
We have implemented heuristics that minimize a cost function on an alignment search space.



diffy



setlike

Syntax of the example

```
module Example =

let field (b:bool) =
  let c = [A-Za-z0-9():,.?] | "" in
  let w =
    match b with
    | true -> [ \n]
    | false -> ' '
    :regexp
  in ( c | c . (c | w)* . c )

let topic =
    copy ("=" . field false)
  . default
      (del (" (" . [?1-9] . " pts)"))
      " (? pts)"
  . copy "=\n"

let question =
    copy ("* " . field false)
  . default
      (del (" (* " . field false . " *)"))
      " (* TODO: answer *)"
  . copy "\n"
```

```
let subject = field true . "\n"

let exercise1 =
  let q = setlike 0 "question" in
  topic . subject . <q:key question > +

let exercise2 =
    topic . subject
  . default
      (del ("(* " . field true . " *)\n"))
      "(* TODO: write the answer *)\n"

let main_lens =
  let e1 = setlike 0 "exercise1" in
  let e2 = setlike 0 "exercise2" in
  ( <e1:key ( align exercise1) >
  | <e2:key ( align exercise2) > )*
```

Extensions

Nested chunks

We can handle several levels of chunks.

```
=History (5 pts)=
List the inventors of the
following programming languages.
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58    (* McCarthy *)
* ML 73      (* Gordon,Milner *)

=Scoping (2 pts)=
Which of these terms are closed?
* λx.λy.x             (* Yes *)
* (λx.λz.x) λx.λy.z (* No *)

=Lambda Calculus (3 pts)=
Give a weakly normalizing term
which is not strongly normalizing.
(* (λx.λy.y) ((λx.x x) λx.x x) *)
```

Tags

We can also have several kinds of chunks which are processed in different ways.

```
=History (5 pts)=
List the inventors of the
following programming languages.
* Haskell 98 (* Hudak,PJ,Wadler *)
* LISP 58    (* McCarthy *)
* ML 73      (* Gordon,Milner *)
```

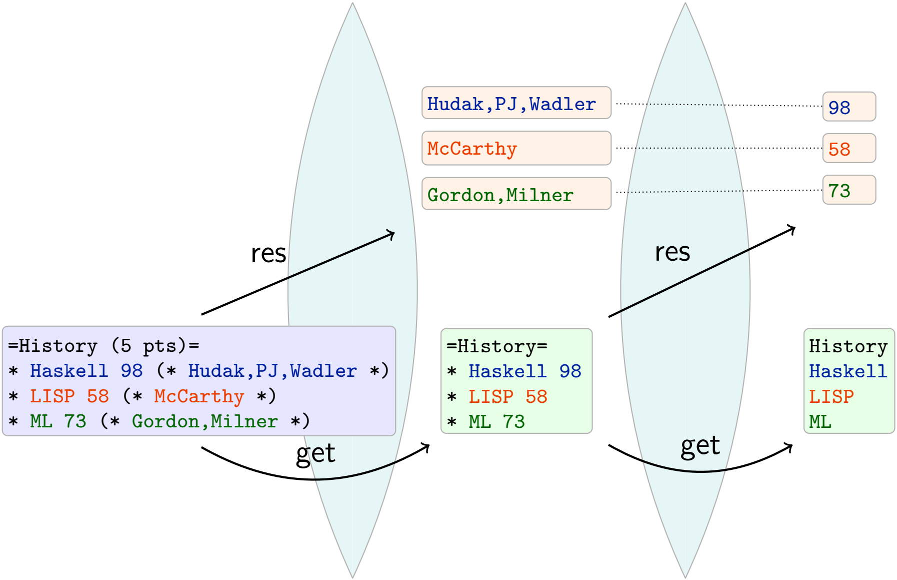
```
=Scoping (2 pts)=
Which of these terms are closed?
* λx.λy.x              (* Yes *)
* (λx.λz.x) λx.λy.z (* No *)
```

```
=Lambda Calculus (3 pts)=
Give a weakly normalizing term
which is not strongly normalizing.
(* (λx.λy.y) ((λx.x x) λx.x x) *)
```

k1

k2

Composition



Related work

Positional

- Focal — [TOPLAS '07]
- semantic bidirectionalization — [Vogtlaender '09]
- syntactic bidirectionalization — [Matsuda '07]
- point free lenses — [Pacheco and Cunha '10]

Update-based

- most databases
- X and Inv — [Hu, Mu and Takeichi '04]
- constraint maintainers — [Merteens '98]
- u-lenses — [Diskin, Xiong and Czarnecki '10]

Dictionary lenses

Idea: use keys for alignment [POPL '08]

Mechanism: build a dictionary, thread it through put

Limitations:

- we don't necessarily have keys,
- the update can change keys, and
- weird composition

Benefits of matching lenses:

- modularity
- enable use of global heuristics
- stronger semantic laws

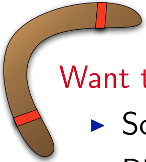
Conclusion

- The alignment problem was an often eluded and not well understood issue arising whenever we handle a list of items in a lossy way, which is the case in many applications.
- The notion of chunks allows to precisely tell which parts of the source are linked.
- Abstracting the alignment from the lens's work makes the distinction between them clear.
- The behavior of put with edits on the view is now specified in the semantic using new laws
- The lens theory still remains quite simple

Thank You!

Collaborators: Davi Barbosa, Nate Foster, Michael Greenberg, Benjamin Pierce

Boomerang contributors: Aaron Bohannon, Martin Hofmann, Alexandre Pilkiewicz, Alan Schmitt, and Daniel Wagner.



Want to play? Boomerang is available for download:

- Source code (LGPL)
- Binaries for OS X, Linux
- Research papers
- Tutorial, manual and demos

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

Extra slides

Matching lens laws (1/2)

$$locations(s) = locations(l.get\ s) \quad (\textsc{GetChunks})$$

$$\frac{c, r = l.res\ s}{locations(s) = \mathrm{dom}(r)} \quad (\textsc{ResChunks})$$

$$\frac{n \in (locations(v) \cap \mathrm{dom}(r))}{(l.put\ v\ (c, r))[n] = k.put\ v[n]\ (r(n))} \quad (\textsc{ChunkPut})$$

$$\frac{n \in (locations(v) \setminus \mathrm{dom}(r))}{(l.put\ v\ (c, r))[n] = k.create\ v[n]} \quad (\textsc{NoChunkPut})$$

$$\frac{skel(v) = skel(v')}{skel(l.put\ v\ (c, r)) = skel(l.put\ v'\ (c, r'))} \quad (\textsc{SkelPut})$$

Matching lens laws (2/2)

$$l.get\ (l.create\ v\ \ r) = v \qquad \text{(CreateGet)}$$

$$\frac{n \in (locations(v) \cap \mathrm{dom}(r))}{(l.create\ v\ \ r)[n] = k.put\ v[n]\ (r(n))} \ \text{(ChunkCreate)}$$

$$\frac{n \in (locations(v) \setminus \mathrm{dom}(r))}{(l.create\ v\ \ r)[n] = k.create\ v[n]} \ \text{(NoChunkCreate)}$$

$$\frac{skel(v) = skel(v')}{skel(l.create\ v\ \ r) = skel(l.create\ v'\ \ r')} \ \text{(SkelCreate)}$$

$$l.get\ (l.put\ v\ (c, r)) = v \qquad \text{(PutGet)}$$

$$l.put\ (l.get\ s)\ (l.res\ s) = s \qquad \text{(GetPut)}$$