Transformation of Structure-Shy Programs

Applied to XPath Queries and Strategic Functions

Alcino Cunha and Joost Visser

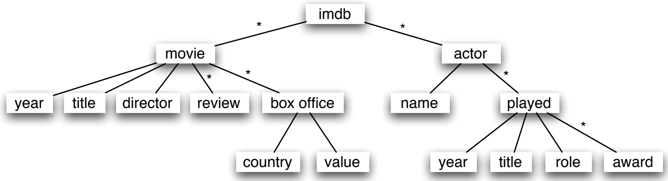
Universidade do Minho, Portugal

PEPM'07, January 15th

Structure-Shy Programming

- A structure-shy program specifies type-specific behavior for a selected set of data constructors only. For the remaining structure, generic behavior is provided.
- It comes in many flavors: adaptive programming, strategic programming (Stratego, Strafunsky, SYB), polytypic programming (PolyP, GH), XML processing (XSLT, XPath).
- Advantages: programs are more concise, easy to understand, reusable, no boilerplate.
- Disadvantages: structure-shy programs have potentially worse space and time behavior then equivalent structure-sensitive programs (dynamic checks, unnecessary traversals).

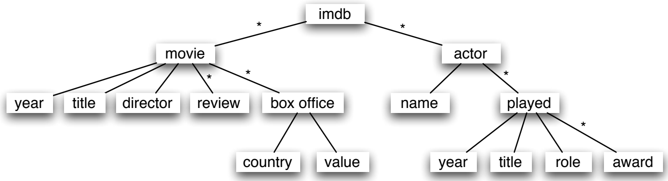
Examples



- XPath

```
//movie/director    //movie[//actor]
```

Examples



- XPath

```
//movie/director    //movie[//actor]
```

- Scrap Your Boilerplate

$$
\begin{aligned}
&\mathit{trunc} = \mathit{everywhere}\ (\mathit{mkT}_{\mathit{Review}}\ \mathit{take100}) \\
&\mathit{count} = \mathit{everything}\ (\mathit{mkQ}_{\mathit{Review}}\ \mathit{size}) \\
&\mathit{take100}\ (\mathit{Review}\ r) = \mathit{Review}\ (\mathit{take}\ 100\ r) \\
&\mathit{size}\ (\mathit{Review}\ r) = \mathit{length}\ r
\end{aligned}
$$

Motivation

- Using knowledge of the schema we would like to transform

  `//movie/director`

  into the less structure-shy but more efficient

  `imdb/movie/director`

Motivation

- Using knowledge of the schema we would like to transform

  `//movie/director`

  into the less structure-shy but more efficient

  `imdb/movie/director`

  or into the more structure-shy

  `//director`

Motivation

- Using knowledge of the schema we would like to transform

  `//movie/director`

  into the less structure-shy but more efficient

  `imdb/movie/director`

  or into the more structure-shy

  `//director`

- Concerning SYB we would like to eliminate strategic combinators and produce the type-specific functions

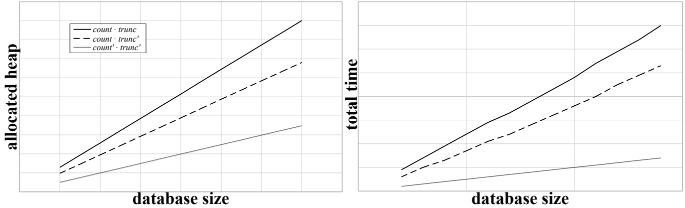
$$
\begin{aligned}
trunc' &= imdb\ (map\ (movie\ id\ id\ id\ (map\ take100)\ id))\ id \\
count' &= sum \circ map\ (sum \circ map\ size \circ reviews) \circ movies
\end{aligned}
$$

  using congruence and selector functions such as:

$$
\begin{aligned}
&imdb\ f\ \ g\ (Imdb\ m\ a) = Imdb\ (f\ \ m)\ (g\ a) \\
&movies\ (Imdb\ m\ a) = m
\end{aligned}
$$

Motivation

- In this SYB example, type-specialization implies an improvement in space and time by factors of 2.6 and 4.8.



- For a fair comparison, we have not used a type-class based implementation of strategic combinators, but our own, GADT-based implementation (14 times faster).

Methodology

- Type-specialization will be achieved by transforming structure-shy programs into structure-sensitive ones, defined using a fixed set of point-free combinators.
- Point-free programming is particular suited to algebraic manipulation, and will potentiate further simplification after specialization.

Methodology

- Type-specialization will be achieved by transforming structure-shy programs into structure-sensitive ones, defined using a fixed set of point-free combinators.
- Point-free programming is particular suited to algebraic manipulation, and will potentiate further simplification after specialization.
- Program transformation laws for structure-shy programs and for the generalization of structure-sensitive ones will also be defined.

Point-free Combinators

$$
\begin{array}{ll}
id & :: A \to A \\
(\circ) & :: (B \to C) \to (A \to B) \to (A \to C) \\
fst & :: A \times B \to A \\
snd & :: A \times B \to B \\
(\triangle) & :: (A \to B) \to (A \to C) \to (A \to B \times C) \\
(\times) & :: (A \to B) \to (C \to D) \to (A \times C \to B \times D) \\
map & :: (A \to B) \to ([A] \to [B]) \\
wrap & :: A \to [A] \\
filter & :: (A \to Bool) \to ([A] \to [A]) \\
zero & :: B \to A \\
plus & :: A \times A \to A \\
fold & :: [A] \to A \\
cond & :: (A \to Bool) \to (A \to B) \to (A \to B) \to (A \to B) \\
true & :: A \to Bool
\end{array}
$$

Point-free Calculation Laws

$f \circ id = f$	$\circ$-IdR
$id \circ f = f$	$\circ$-IdL
$f \circ (g \circ h) = (f \circ g) \circ h$	$\circ$-Assoc
$f \times g = (f \circ fst) \triangle (g \circ snd)$	$\times$-Def
$fst \circ (f \triangle g) = f$	$\times$-CancelL
$snd \circ (f \triangle g) = g$	$\times$-CancelR
$(f \triangle g) \circ h = (f \circ h) \triangle (g \circ h)$	$\times$-Fusion
$fst \triangle snd = id$	$\times$-Reflex
$map\ id = id$	*map*-Id
$map\ f \circ zero = zero$	*map*-Zero
$map\ f \circ map\ g = map\ (f \circ g)$	*map*-Fusion
$map\ f \circ wrap = wrap \circ f$	*map*-Wrap

Point-free Calculation Laws

$true \circ f = true$	*true*-Fusion
$cond\ true\ f\ g = f$	*cond*-True
$cond\ zero\ f\ g = g$	*cond*-False
$(cond\ f\ l\ r) \circ g = cond\ (f \circ g)\ (l \circ g)\ (r \circ g)$	*cond*-Fusion
$filter\ true = id$	*filter*-True
$filter\ zero = zero$	*filter*-False
$filter\ f \circ zero = zero$	*filter*-Zero
$filter\ f \circ plus = plus \circ (filter\ f \times filter\ f)$	*filter*-Plus
$filter\ f \circ map\ g = map\ g \circ filter\ (f \circ g)$	*filter*-Map
$filter\ f \circ fold = fold \circ map\ (filter\ f)$	*filter*-Fold
$filter\ f \circ wrap = cond\ f\ wrap\ zero$	*filter*-Wrap

Point-free Calculation Laws

$plus \circ (zero \triangle f) = f$	$plus$-ZeroL
$plus \circ (f \triangle zero) = f$	$plus$-ZeroR
$zero \circ f = zero$	$zero$-Fusion
$fold \circ (map\ zero) = zero$	$fold$-MapZero
$fold \circ wrap = id$	$fold$-Wrap
$fold \circ map\ wrap = id$	$fold$-MapWrap
$fold \circ plus = plus \circ (fold \times fold)$	$fold$-Plus
$map\ f \circ plus = plus \circ (map\ f \times map\ f)$	map-Plus
$map\ f \circ zero = zero$	map-Zero
$fold \circ zero = zero$	$fold$-Zero
$map\ f \circ fold = fold \circ map\ (map\ f)$	map-Fold
$fold \circ fold \circ map\ f = fold \circ map\ (fold \circ f)$	$fold$-FoldMap

Strategic Programming

- Combinators for type-preserving generic functions:

nop	$:: \mathrm{T}$	-- identity
$(\triangleright)$	$:: \mathrm{T} \to \mathrm{T} \to \mathrm{T}$	-- sequence
$mapT$	$:: \mathrm{T} \to \mathrm{T}$	-- map over children
mkT_A	$:: (A \to A) \to \mathrm{T}$	-- creation
apT_A	$:: \mathrm{T} \to (A \to A)$	-- application
$everywhere$	$:: \mathrm{T} \to \mathrm{T}$	-- top-down traversal

- Combinators for type-unifying generic functions:

$\emptyset$	$:: \mathrm{Q}\ R$	-- empty result
$(\cup)$	$:: \mathrm{Q}\ R \to \mathrm{Q}\ R \to \mathrm{Q}\ R$	-- union of results
$mapQ$	$:: \mathrm{Q}\ R \to \mathrm{Q}\ R$	-- fold over children
mkQ_A	$:: (A \to R) \to \mathrm{Q}\ R$	-- creation
apQ_A	$:: \mathrm{Q}\ R \to (A \to R)$	-- application
$everything$	$:: \mathrm{Q}\ R \to \mathrm{Q}\ R$	-- top-down crush

Strategic Programming Laws

$f \triangleright nop = f$	$\triangleright$-IDR
$nop \triangleright f = f$	$\triangleright$-IDL
$f \triangleright (g \triangleright h) = (f \triangleright g) \triangleright h$	$\triangleright$-ASSOC
$mapT\ nop = nop$	$mapT$-NOP
$mapT\ f \triangleright mapT\ g = mapT\ (f \triangleright g)$	$mapT$-FUSION
$f \cup \emptyset = f$	$\cup$-EMPTYR
$\emptyset \cup f = f$	$\cup$-EMPTYL
$f \cup (g \cup h) = (f \cup g) \cup h$	$\cup$-ASSOC
$mapQ\ \emptyset = \emptyset$	$mapQ$-EMPTY
$mapQ\ f \cup mapQ\ g = mapQ\ (f \cup g)$	$mapQ$-FUSION
$everywhere\ f = f \triangleright mapT\ (everywhere\ f)$	$everyw$-DEF
$everything\ f = f \cup mapQ\ (everything\ f)$	$everyt$-DEF

From Strategic to Point-free Programs

$$
\begin{array}{lr}
apT_A \; nop = id & nop\text{-}\textsc{Apply} \\
apT_A \; (f \triangleright g) = apT_A \; f \circ apT_A \; g & \triangleright\text{-}\textsc{Apply} \\
\left.\begin{array}{l} apT_A \; (mkT_A \; f) = f \\ apT_A \; (mkT_B \; f) = id, \textbf{ if } A \not\equiv B \end{array}\right\} & mkT\text{-}\textsc{Apply} \\
\left.\begin{array}{l} apT_{(A \times B)} \; (mapT \; f) = apT_A \; f \times apT_B \; f \\ apT_{[A]} \; (mapT \; f) = map \; (apT_A \; f) \\ apT_A \; (mapT \; f) = id, \textbf{ if } A \text{ simple type} \end{array}\right\} & mapT\text{-}\textsc{Apply} \\
\hline
apQ_A \; \emptyset = zero & \emptyset\text{-}\textsc{Apply} \\
apQ_A \; (f \cup g) = plus \circ (apQ_A \; f \triangle apQ_A \; g) & \cup\text{-}\textsc{Apply} \\
\left.\begin{array}{l} apQ_A \; (mkQ_A \; f) = f \\ apQ_A \; (mkQ_B \; f) = zero, \textbf{ if } A \not\equiv B \end{array}\right\} & mkQ\text{-}\textsc{Apply} \\
\left.\begin{array}{l} apQ_{(A \times B)} \; (mapQ \; f) \\ \quad = plus \circ ((apQ_A \; f) \times (apQ_B \; f)) \\ apQ_{[A]} \; (mapQ \; f) = fold \circ map \; (apQ_A \; f) \\ apQ_A \; (mapQ \; f) = zero, \textbf{ if } A \text{ simple type} \end{array}\right\} & mapQ\text{-}\textsc{Apply}
\end{array}
$$

From Point-free to Strategic Programs

$$
\begin{array}{|rr|}
\hline
mkT_A\ id = nop & id\text{-}\textsc{PullT} \\
mkT_A\ (f \circ g) = mkT_A\ g \rhd mkT_A\ f & \circ\text{-}\textsc{PullT} \\
mkT_{[A]}\ (map\ g) = mapT\ (mkT_A\ f) & map\text{-}\textsc{PullT} \\
\left.\begin{array}{l} mkT_{(A\times A)}\ (f \times f) = mapT\ (mkT_A\ f) \\ mkT_{(A\times B)}\ (f \times g) \\ \quad = mapT\ (mkT_A\ f \rhd mkT_B\ g),\ \textbf{if}\ A \not\equiv B \end{array}\right\} & \times\text{-}\textsc{PullT} \\
\hline
mkQ_A\ zero = \emptyset & \emptyset\text{-}\textsc{PullQ} \\
mkQ_A\ (plus \circ (f \triangle g)) = mkQ_A\ f \cup mkQ_A\ g & plus\text{-}\textsc{PullQ} \\
\left.\begin{array}{l} mkQ_{[A]}\ (fold \circ map\ f) = mapQ\ (mkQ_A\ f) \\ mkQ_{[A]}\ (map\ f) = mapQ\ (mkQ_A\ (wrap \circ f)) \end{array}\right\} & map\text{-}\textsc{PullQ} \\
\left.\begin{array}{l} mkQ_{(A\times B)}\ (f \circ fst) \\ \quad = mapQ\ (mkQ_A\ f),\ \textbf{if}\ A \not\equiv B \\ mkQ_{(A\times B)}\ (f \circ snd) \\ \quad = mapQ\ (mkQ_B\ f),\ \textbf{if}\ A \not\equiv B \end{array}\right\} & \times\text{-}\textsc{PullQ} \\
\hline
\end{array}
$$

XPath Queries

- Many XPath constructs can be expressed directly as strategic combinators of type $\mathrm{Q}\ [\star]$, where $\star$ represents a universal node type.
- Function $mkAny_A :: A \rightarrow \star$ is used to inject any type A into the universal type.

$self$	$:: \mathrm{Q}\ [\star]$	`-- self::node()`
$child$	$:: \mathrm{Q}\ [\star]$	`-- child::node()`
$desc$	$:: \mathrm{Q}\ [\star]$	`-- descendant::node()`
$descself$	$:: \mathrm{Q}\ [\star]$	`-- descendant-or-self::node()`
$name$	$:: String \rightarrow \mathrm{Q}\ [\star]$	`-- self::name`
$(/)$	$:: \mathrm{Q}\ [\star] \rightarrow \mathrm{Q}\ R \rightarrow \mathrm{Q}\ R$	-- q/r
$(?)$	$:: \mathrm{Q}\ [\star] \rightarrow \mathrm{Q}\ Bool \rightarrow \mathrm{Q}\ [\star]$	-- $q[p]$
$nonempty$	$:: \mathrm{Q}\ Bool$	

Some XPath Laws

$(f \cup g) \mathbin{/} h = (f \mathbin{/} h) \cup (g \mathbin{/} h)$	$\cup$-DIST
$\emptyset \mathbin{/} f = \emptyset$	/-EMPTYL
$f \mathbin{/} \emptyset = \emptyset$	/-EMPTYR
$\mathit{self} \mathbin{/} f = f$	/-SELFL
$f \mathbin{/} \mathit{self} = f$	/-SELFR
$\mathit{name}\ n \mathbin{/} \mathit{name}\ n = \mathit{name}\ n$	/-NAME
$(f \mathbin{/} g) \mathbin{/} h = f \mathbin{/} (g \mathbin{/} h)$	/-ASSOC
$\emptyset \mathbin{?} p = \emptyset$	?-EMPTY
$f \mathbin{?} \mathit{nonempty} = f$	?-NONEMPTY
$(f \mathbin{?} p) \mathbin{?} q = (f \mathbin{?} q) \mathbin{?} p$	?-COMUT
$f \mathbin{?} (\mathit{name}\ n \mathbin{/} \mathit{nonempty}) = f \mathbin{/} \mathit{name}\ n$	?-NAME

From XPath to Point-free Programs and Back

$$\begin{array}{|c|r|}
\hline
\begin{array}{rcl} \mathit{child} &=& \mathit{mapQ}\ \mathit{self} \\ \mathit{desc} &=& \mathit{everything}\ \mathit{child} \\ \mathit{descself} &=& \mathit{self} \cup \mathit{desc} \\ \mathit{mapQ}\ f &=& \mathit{child}\ /\ f \end{array} & \begin{array}{r} \mathit{child}\text{-}\mathrm{DEF} \\ \mathit{desc}\text{-}\mathrm{DEF} \\ \mathit{descself}\text{-}\mathrm{DEF} \\ \mathit{mapQ}\text{-}\mathrm{DEF} \end{array} \\
\hline
\begin{array}{c} \mathit{apQ}_A\ (f\ /\ g) = \mathit{fold} \circ \mathit{map}\ (\mathit{apQ}_\star\ g) \circ \mathit{apQ}_A\ f \\ \mathit{apQ}_A\ (f\ ?\ p) = \mathit{filter}\ (\mathit{apQ}_\star\ p) \circ \mathit{apQ}_A\ f \\ \mathit{apQ}_A\ \mathit{nonempty} = \mathit{true} \\ \mathit{apQ}_A\ \mathit{self} = \mathit{wrap} \circ \mathit{mkAny}_A \\ \left.\begin{array}{l} \mathit{apQ}_A\ (\mathit{name}\ n) = \mathit{apQ}_A\ \mathit{self},\ \textbf{if}\ A\ \text{has name}\ n \\ \mathit{apQ}_A\ (\mathit{name}\ n) = \mathit{zero},\ \textbf{otherwise} \end{array}\right\} \\ \mathit{apQ}_\star\ f \circ \mathit{mkAny}_A = \mathit{apQ}_A\ f \end{array} & \begin{array}{r} /\text{-}\mathrm{APPLY} \\ ?\text{-}\mathrm{APPLY} \\ \mathit{nempt}\text{-}\mathrm{APPLY} \\ \mathit{self}\text{-}\mathrm{APPLY} \\ \mathit{name}\text{-}\mathrm{APPLY} \\ \star\text{-}\mathrm{APPLY} \end{array} \\
\hline
\left.\begin{array}{l} \mathit{mkQ}_A\ (\mathit{wrap} \circ \mathit{mkAny}_A) = \mathit{name}\ n, \\ \qquad\qquad \textbf{if}\ A\ \text{has name}\ n \\ \mathit{mkQ}_A\ (\mathit{wrap} \circ \mathit{mkAny}_A) = \mathit{self},\ \textbf{otherwise} \end{array}\right\} & \star\text{-}\mathrm{PULLQ} \\
\hline
\end{array}$$

Type-safe Representation of Types

$$
\begin{array}{l}
\textbf{data}\ \mathit{Type}\ a\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{Int} & :: & \mathit{Type}\ \mathit{Int} \\
\mathit{Bool} & :: & \mathit{Type}\ \mathit{Bool} \\
\mathit{String} & :: & \mathit{Type}\ \mathit{String} \\
\mathit{Any} & :: & \mathit{Type}\ \star \\
\mathit{List} & :: & \mathit{Type}\ a \rightarrow \mathit{Type}\ [a] \\
\mathit{Prod} & :: & \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (a, b) \\
\mathit{Either} & :: & \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (\mathit{Either}\ a\ b) \\
\mathit{Func} & :: & \mathit{Type}\ a \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ (a \rightarrow b) \\
\mathit{Data} & :: & \mathit{String} \rightarrow \mathit{EP}\ a\ b \rightarrow \mathit{Type}\ b \rightarrow \mathit{Type}\ a
\end{array} \\
\textbf{data}\ \mathit{EP}\ a\ b = \mathit{EP}\{\mathit{to} :: a \rightarrow b, \mathit{from} :: b \rightarrow a\}
\end{array}
$$

Type-safe Representation of Types

$$
\begin{array}{l}
\textbf{class } \mathit{Typeable}\ a\ \textbf{where } \mathit{typeof} :: \mathit{Type}\ a \\
\textbf{instance } \mathit{Typeable}\ \mathit{Int}\ \textbf{where } \mathit{typeof} = \mathit{Int} \\
\textbf{instance } (\mathit{Typeable}\ a, \mathit{Typeable}\ b) \Rightarrow \mathit{Typeable}\ (a \to b) \\
\quad \textbf{where } \mathit{typeof} = \mathit{Func}\ \mathit{typeof}\ \mathit{typeof} \\
\textbf{data } \mathit{Imdb} = \mathit{Imdb}\ [\mathit{Movie}]\ [\mathit{Actor}] \\
\textbf{instance } \mathit{Typeable}\ \mathit{Imdb}\ \textbf{where} \\
\quad \mathit{typeof} = \mathit{Data}\ \texttt{"Imdb"}\ (\mathit{EP}\ \mathit{to}\ \mathit{from})\ \mathit{rep} \\
\qquad \textbf{where } \mathit{rep} = \mathit{Prod}\ (\mathit{List}\ \mathit{typeof})\ (\mathit{List}\ \mathit{typeof}) \\
\qquad\qquad\quad \mathit{to}\ (\mathit{Imdb}\ \mathit{ms}\ \mathit{as}) = (\mathit{ms}, \mathit{as}) \\
\qquad\qquad\quad \mathit{from}\ (\mathit{ms}, \mathit{as}) = \mathit{Imdb}\ \mathit{ms}\ \mathit{as}
\end{array}
$$

Type-safe Representation of Types

$$\textbf{class}\ \mathit{Typeable}\ a\ \textbf{where}\ \mathit{typeof} :: \mathit{Type}\ a$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Typeable}\ \mathit{Int}\ \textbf{where}\ \mathit{typeof} = \mathit{Int} \\
\textbf{instance}\ (\mathit{Typeable}\ a, \mathit{Typeable}\ b) \Rightarrow \mathit{Typeable}\ (a \to b) \\
\quad \textbf{where}\ \mathit{typeof} = \mathit{Func}\ \mathit{typeof}\ \mathit{typeof} \\
\textbf{data}\ \mathit{Imdb} = \mathit{Imdb}\ [\mathit{Movie}]\ [\mathit{Actor}]
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Typeable}\ \mathit{Imdb}\ \textbf{where} \\
\quad \mathit{typeof} = \mathit{Data}\ \texttt{"Imdb"}\ (\mathit{EP}\ \mathit{to}\ \mathit{from})\ \mathit{rep} \\
\qquad \textbf{where}\ \mathit{rep} = \mathit{Prod}\ (\mathit{List}\ \mathit{typeof})\ (\mathit{List}\ \mathit{typeof}) \\
\qquad\qquad\quad \mathit{to}\ (\mathit{Imdb}\ \mathit{ms}\ \mathit{as}) = (\mathit{ms}, \mathit{as}) \\
\qquad\qquad\quad \mathit{from}\ (\mathit{ms}, \mathit{as}) = \mathit{Imdb}\ \mathit{ms}\ \mathit{as}
\end{array}$$

$$\textbf{data}\ \mathit{Equal}\ a\ b\ \textbf{where}\ \mathit{Eq} :: \mathit{Equal}\ a\ a$$

$$\mathit{teq} :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Maybe}\ (\mathit{Equal}\ a\ b)$$

Type-safe Representation of Functions

$$
\begin{array}{l}
\textbf{data}\ \mathsf{F}\ f\ \textbf{where} \\
\quad \begin{array}{ll}
Id & :: \mathsf{F}\ (a {\rightarrow} a) \\
Comp & :: Type\ b \rightarrow \mathsf{F}\ (b {\rightarrow} c) \rightarrow \mathsf{F}\ (a {\rightarrow} b) \rightarrow \mathsf{F}\ (a {\rightarrow} c) \\
Fst & :: \mathsf{F}\ ((a, b) {\rightarrow} a) \\
Snd & :: \mathsf{F}\ ((a, b) {\rightarrow} b) \\
(\triangle) & :: \mathsf{F}\ (a {\rightarrow} b) \rightarrow \mathsf{F}\ (a {\rightarrow} c) \rightarrow \mathsf{F}\ (a {\rightarrow} (b, c)) \\
Plus & :: Monoid\ a \rightarrow \mathsf{F}\ ((a, a) {\rightarrow} a) \\
Datamap & :: Type\ b \rightarrow \mathsf{F}\ (b {\rightarrow} b) \rightarrow \mathsf{F}\ (a {\rightarrow} a) \\
unData & :: \mathsf{F}\ (a {\rightarrow} b) \\
MkAny & :: \mathsf{F}\ (a {\rightarrow} \star) \\
Fun & :: String \rightarrow (a {\rightarrow} b) \rightarrow \mathsf{F}\ (a {\rightarrow} b)
\end{array} \\
\quad \ldots \\
\textbf{data}\ \star\ \textbf{where}\ Any :: Type\ a \rightarrow a \rightarrow \star \\
\textbf{data}\ Monoid\ r = Monoid \{ zero :: r, plus :: r \rightarrow r \rightarrow r \}
\end{array}
$$

Type-safe Representation of Functions

$$
\begin{array}{l}
\textbf{data}\ \mathsf{F}\ f\ \textbf{where} \\
\quad \ldots \\
\quad \begin{array}{ll}
\mathit{Nop} & :: \mathsf{F}\ \mathsf{T} \\
\mathit{Seq} & :: \mathsf{F}\ \mathsf{T} \to \mathsf{F}\ \mathsf{T} \to \mathsf{F}\ \mathsf{T} \\
\mathit{ApT} & :: \mathit{Type}\ a \to \mathsf{F}\ \mathsf{T} \to \mathsf{F}\ (a \to a) \\
\mathit{MkT} & :: \mathit{Type}\ a \to \mathsf{F}\ (a \to a) \to \mathsf{F}\ \mathsf{T} \\
\mathit{MkQ} & :: \mathit{Monoid}\ r \to \mathit{Type}\ a \to \mathsf{F}\ (a \to r) \to \mathsf{F}\ (\mathsf{Q}\ r) \\
\mathit{Empty} & :: \mathit{Monoid}\ r \to \mathsf{F}\ (\mathsf{Q}\ r) \\
\ldots & \\
\mathit{Self} & :: \mathsf{F}\ (\mathsf{Q}\ [\star]) \\
\mathit{Name} & :: \mathit{String} \to \mathsf{F}\ (\mathsf{Q}\ [\star]) \\
(:/:) & :: \mathsf{F}\ (\mathsf{Q}\ [\star]) \to \mathsf{F}\ (\mathsf{Q}\ r) \to \mathsf{F}\ (\mathsf{Q}\ r) \\
(:?:) & :: \mathsf{F}\ (\mathsf{Q}\ [\star]) \to \mathsf{F}\ (\mathsf{Q}\ \mathit{Bool}) \to \mathsf{F}\ (\mathsf{Q}\ [\star]) \\
\mathit{Nonempty} & :: \mathsf{F}\ (\mathsf{Q}\ \mathit{Bool})
\end{array}
\end{array}
$$

Rewrite Rules

$$\textbf{type}\ Rule = \forall f\ .\ Type\ f \longrightarrow \mathsf{F}\ f \longrightarrow RewriteM\ (\mathsf{F}\ f)$$

Rewrite Rules

$$\textbf{type}\ Rule = \forall f\ .\ Type\ f \rightarrow \mathsf{F}\ f \rightarrow RewriteM\ (\mathsf{F}\ f)$$

$$
\begin{aligned}
&nat\_id :: Rule\\
&nat\_id\ \_\ (Comp\ \_\ Id\ f) = return\ f\\
&nat\_id\ \_\ (Comp\ \_\ f\ Id) = return\ f\\
&nat\_id\ \_\ \_ \qquad\qquad\qquad\quad = mzero\\
&prod\_def :: Rule\\
&prod\_def\ (Func\ (Prod\ a\ b)\ \_)\ (f \times g)\\
&\quad = return\ ((Comp\ a\ f\ Fst) \triangle (Comp\ b\ g\ Snd))\\
&prod\_def\ \_\ \_\ mzero
\end{aligned}
$$

Rewrite Rules

$$
\begin{aligned}
&\mathit{mkT\_apply} :: \mathit{Rule} \\
&\mathit{mkT\_apply}\ \_\ (\mathit{ApT}\ a\ (\mathit{MkT}\ b\ f)) \\
&\quad = \mathbf{case}\ \mathit{teq}\ a\ b\ \mathbf{of}\ \mathit{Just\ Eq} \rightarrow \mathit{return}\ f \\
&\qquad\qquad\qquad\qquad\qquad\ \ \mathit{Nothing} \rightarrow \mathit{return}\ \mathit{Id} \\
&\mathit{mkT\_apply}\ \_\ \_ = \mathit{mzero} \\
&\mathit{mapT\_apply}\ \_\ (\mathit{ApT}\ t\ (\mathit{MapT}\ f)) = \mathit{return}\ (\mathit{aux}\ t) \\
&\quad \mathbf{where}\ \mathit{aux} :: \mathit{Type}\ a \rightarrow \mathrm{F}\ (a \rightarrow a) \\
&\qquad\qquad \mathit{aux}\ (\mathit{Prod}\ a\ b) = (\mathit{ApT}\ a\ f) \times (\mathit{ApT}\ b\ f) \\
&\qquad\qquad \mathit{aux}\ (\mathit{List}\ a) \quad\ = \mathit{Listmap}\ (\mathit{ApT}\ a\ f) \\
&\qquad\qquad \mathit{aux}\ \mathit{Int} \qquad\quad\ \ = \mathit{Id} \\
&\qquad\qquad \ldots \\
&\mathit{mapT\_apply}\ \_\ \_ = \mathit{mzero}
\end{aligned}
$$

Strategic Rule Combinators

$$
\begin{array}{ll}
nop :: Rule & \text{-- identity rule} \\
(\circledgtr) :: Rule \rightarrow Rule \rightarrow Rule & \text{-- sequential composition} \\
(\oslash) :: Rule \rightarrow Rule \rightarrow Rule & \text{-- choice} \\
all :: Rule \rightarrow Rule & \text{-- map on all children} \\
one :: Rule \rightarrow Rule & \text{-- map on one child} \\
rewrite :: Rule \rightarrow F\ f \rightarrow F\ f & \text{-- top-level application}
\end{array}
$$

$$
\begin{array}{l}
many\ r = (r \circledgtr (many\ r)) \oslash nop \\
once\ r = r \oslash one\ (once\ r) \\
innermost\ r = all\ (innermost\ r) \circledgtr ((r \circledgtr innermost\ r) \oslash nop)
\end{array}
$$

Some Transformation Strategies

- Optimization of point-free programs

$$
\begin{aligned}
&\mathit{optimize\_pf} = \mathit{innermost\ opt} \varogreaterthan \mathit{innermost\ inv}\\
&\quad \textbf{where}\ \mathit{opt} = \mathit{nat\_id} \oslash \mathit{prod\_def} \oslash \mathit{prod\_cancel} \oslash\\
&\qquad\qquad\qquad \mathit{map\_zero} \oslash \mathit{map\_fusion} \oslash \ldots\\
&\qquad\qquad\ \ \mathit{inv} = \mathit{prod\_dev\_inv} \oslash \mathit{prod\_fusion\_inv} \oslash \ldots
\end{aligned}
$$

Some Transformation Strategies

- Optimization of point-free programs

$$\begin{aligned} &\mathit{optimize\_pf} = \mathit{innermost\ opt} \odot \mathit{innermost\ inv} \\ &\quad \textbf{where}\ \mathit{opt} = \mathit{nat\_id} \oslash \mathit{prod\_def} \oslash \mathit{prod\_cancel} \oslash \\ &\qquad\qquad\quad \mathit{map\_zero} \oslash \mathit{map\_fusion} \oslash \ldots \\ &\qquad\qquad \mathit{inv} = \mathit{prod\_dev\_inv} \oslash \mathit{prod\_fusion\_inv} \oslash \ldots \end{aligned}$$

- Specialization of structure-shy programs

$$\begin{aligned} &\mathit{optimize\_t} = \mathit{t2pf} \odot \mathit{optimize\_pf} \\ &\mathit{t2pf} = \mathit{innermost}\ (\mathit{mapT\_apply} \oslash \mathit{mkT\_apply} \oslash \ldots) \end{aligned}$$

Increasing Structure-shyness

$mapT\ (everywhere\ f) \stackrel{?}{=} everywhere\ f$	$mapT$-ELIM
$mkT_A\ f \stackrel{?}{=} everywhere\ (mkT_A\ f)$	$everyw$-INTRO
$mapQ\ (everything\ f) \stackrel{?}{=} everything\ f$	$mapQ$-ELIM
$mkQ_A\ f \stackrel{?}{=} everything\ (mkQ_A\ f)$	$everyw$-INTRO
$self \stackrel{?}{=} descself$	$self$-ELIM
$child\ /\ descself \stackrel{?}{=} descself$	$child$-ELIM

Increasing Structure-shyness

$$
\begin{array}{l}
guardT :: Rule \rightarrow Rule \\
guardT\ r\ t\ f = \mathbf{do} \\
\quad g \leftarrow r\ t\ f \\
\quad f' \leftarrow optimize\_t\ t\ f; g' \leftarrow optimize\_t\ t\ g \\
\quad \mathbf{if}\ (f' \equiv g')\ \mathbf{then}\ return\ g\ \mathbf{else}\ mzero
\end{array}
$$

Increasing Structure-shyness

$$
\begin{array}{l}
guardT :: Rule \rightarrow Rule \\
guardT\ r\ t\ f = \mathbf{do} \\
\quad g \leftarrow r\ t\ f \\
\quad f' \leftarrow optimize\_t\ t\ f; g' \leftarrow optimize\_t\ t\ g \\
\quad \mathbf{if}\ (f' \equiv g')\ \mathbf{then}\ return\ g\ \mathbf{else}\ mzero
\end{array}
$$

$$
\begin{array}{l}
generalize\_t = \\
\quad optimize\_t \oslash\!\!> mkT\_apply\_inv \\
\quad innermost\ (id\_pullT \oslash comp\_pullT \oslash \ldots) \oslash\!\!> \\
\quad many\ (once\ (seq\_id \oslash mapT\_fusion \oslash \ldots) \\
\qquad\qquad \oslash\ guardT\ (once\ (mapT\_elim \oslash everyw\_intro)))
\end{array}
$$

Strategic Transformations

$> \textbf{let } trunc = everywhere\ (mkT_{Review}\ take100)$

$> rewrite\ optimize\_t\ (apT_{Imdb}\ trunc)$

$imdb\ (map\ (movie\ (id \times id \times id \times map\ take100 \times id)) \times id)$

$> rewrite\ optimize\_t\ (apT_{Actor}\ trunc)$

id

$> rewrite\ optimize\_t\ (apT_{Review}\ trunc)$

$take100$

Strategic Transformations

$> \textbf{let}\ trunc = everywhere\ (mkT_{Review}\ take100)$
$> rewrite\ optimize\_t\ (apT_{Imdb}\ trunc)$
$imdb\ (map\ (movie\ (id \times id \times id \times map\ take100 \times id)) \times id)$
$> rewrite\ optimize\_t\ (apT_{Actor}\ trunc)$
id
$> rewrite\ optimize\_t\ (apT_{Review}\ trunc)$
$take100$

$> \textbf{let}\ up = apT_{Actor}\ (everywhere\ (mkT_{Award}\ upper))$
$\quad \textbf{where}\ upper\ (Award\ t) = Award\ (map\ toUpper\ t)$
$> \textbf{let}\ bigawards = everywhere\ (mkT_{Actor}\ up)$
$> rewrite\ generalize\_t\ (apT_{Imdb}\ bigawards)$
$apT_{Imdb}\ (everywhere\ (mkT_{Award}\ upper))$

Strategic Queries

$$
\begin{array}{l}
> \textbf{let } \mathit{count} = \mathit{everything}\ (\mathit{mkQ}_{\mathit{Review}}\ \mathit{size}) \\
> \mathit{rewrite}\ \mathit{optimize\_q}\ (\mathit{apQ}_{\mathit{Imdb}}\ \mathit{count}) \\
\mathit{sum} \circ \mathit{map}\ (\mathit{sum} \circ \mathit{map}\ \mathit{size} \circ \mathit{reviews}) \circ \mathit{movies} \\
\quad \textbf{where } \mathit{movies} = \mathit{fst} \circ \mathit{unImdb} \\
\qquad\qquad \mathit{reviews} = \mathit{fst} \circ \mathit{snd} \circ \mathit{snd} \circ \mathit{snd} \circ \mathit{unMovie} \\
> \mathit{rewrite}\ \mathit{optimize\_q}\ (\mathit{apQ}_{\mathit{Actor}}\ \mathit{count}) \\
\mathit{zero}
\end{array}
$$

XPath Queries

$$> \textbf{let}\ directors = descself\ /\ child\ /\ \langle director \rangle$$

$$> rewrite\ optimize\_xp\ (apQ_{[Imdb]}\ directors)$$

$$\begin{aligned}
&concat \circ map\ (map\ (mkAny_{.} \circ director) \circ movies)\\
&\quad \textbf{where}\ movies = fst \circ unImdb\\
&\qquad\qquad director = fst \circ snd \circ snd \circ unMovie
\end{aligned}$$

$$\begin{aligned}
> \textbf{let}\ movactors = {} & descself\ /\ \langle movie \rangle\ ?\\
& descself\ /\ \langle actor \rangle\ /\ nonempty
\end{aligned}$$

$$> rewrite\ optimize\_xp\ (apQ_{[Imdb]}\ movactors)$$

$$nil$$

$$> \textbf{let}\ dirparents = descself\ ?\ child\ /\ \langle director \rangle\ /\ nonempty$$

$$> rewrite\ optimize\_xp\ (apQ_{[Imdb]}\ dirparents)$$

$$concat \circ map\ (map\ mkDyn \circ fst \circ unImdb)$$

Demo

Do you want a demo?

Contributions

- Laws for strategic programs and for converting between these and point-free programs.
- GADT encoding of the XPath language in terms of strategic program combinators, augmented with a universal node type.
- Laws for XPath queries and for converting between these and point-free programs.
- Implementation of the algebraic laws in a type-safe rewriting system, encoded in Haskell, that can be used for specialization, generalization, and optimization.
- A unified framework for point-free, strategic, and XPath transformations, where structure-sensitive point-free programs are used as the solution space for transformation of structure-shy programs.

Future Work

- Prove all the laws used. Characterize formally the normal forms and termination behavior of the rewrite strategies.
- Handle (mutually) recursive data types.
- Expand XPath coverage.
- Tackle other languages such as XQuery or SQL.
- Front-end for parsing and pretty-printing of XPath.