A Model of Functional Traversal-Based Generic Programming

Bryan Chadwick and Karl Lieberherr

Presenter: Jesse Tov

Mitchfest, 23 August 2009

A Model of Functional Traversal-Based Generic Programming

Bryan Chadwick and Karl Lieberherr

Presenter: Jesse Tov

Mitchfest, 23 August 2009

A Model of Functional Traversal-Based Generic Programming

Bryan Chadwick and Karl Lieberherr

Presenter: Jesse Tov

Mitchfest, 23 August 2009

Motivation

We like folds and maps:

```
(foldr + 0 ’(1 2 3 4))
(map add1 ’(1 2 3 4))
```

But!

- fixed recursion scheme
- accumulators awkward
- requires too many parameters

Motivation

We like folds and maps:

```
(foldr + 0 '(1 2 3 4))
(map add1 '(1 2 3 4))
```

But!

- fixed recursion scheme
  $\longrightarrow$ go everwhere + declarative control
- accumulators awkward
- requires too many parameters

Motivation

We like folds and maps:

```
(foldr + 0 '(1 2 3 4))
(map add1 '(1 2 3 4))
```

But!

- fixed recursion scheme
  $\longrightarrow$ go everwhere + declarative control
- accumulators awkward
  $\longrightarrow$ orthogonal accumulators
- requires too many parameters

Motivation

We like folds and maps:

```
(foldr + 0 '(1 2 3 4))
(map add1 '(1 2 3 4))
```

But!

- fixed recursion scheme
  $\longrightarrow$ go everwhere + declarative control
- accumulators awkward
  $\longrightarrow$ orthogonal accumulators
- requires too many parameters
  $\longrightarrow$ say only what you need

Related Work

- Scrap Your Boilerplate (Lämmel and Peyton Jones 2003, 2004)
- Polytypic Programming (Jansson and Jeuring 1997)
- Generic Programming (Hinze 1999; Loeh et al. 2005)
- Generalized Folds (Sheard and Fegaras 1993; Meijer et al. 1991)
- Strategic Programming (Lämmel et al. 2004)

Beyond the Fold: Adaptive Traversal

```
(traverse val funcs)
```

- Function set with multiple dispatch
- Adapts to the data
- Control and accumulators

TBGP: Examples

A flat list:

```
(foldr + 0 '(1 2 3 4))
```

TBGP: Examples

A flat list:

```
(foldr + 0 '(1 2 3 4))
```

A nested list:

```
(let loop ([lst '(1 (2 3) ((4)))])
  (foldr (lambda (a b)
           (cond [(number? a) (+ a b)]
                 [else        (+ (loop a) b)]))
         0 lst))
```

TBGP: Examples

A flat list:

```
(foldr + 0 '(1 2 3 4))
```

A nested list:

```
(traverse '(1 (2 3) ((4)))
    (funcset [(cons number number) (c n m) (+ n m)]
             [(empty)              (e)     0]))
```

TBGP: Examples

A flat list:

```
(foldr + 0 '(1 2 3 4))
```

A nested list:

```
(traverse '(1 (2 3) ((4)))
     (funcset [(cons number number) (c n m) (+ n m)]
              [(empty)              (e)     0]))
```

Fully generic:

```
(traverse '(1 (2 3) ((4)))
     (union-func (build-TU number + 0)
                 (funcset [(number) (n) n])))
```

A Whirlwind Tour

- Data definitions
- Function sets
- Generic traversal & dispatch

Data Definitions

```
(concrete Unit [])
(concrete Arr  [(dom type) (cod type)])
(concrete TV   [(id symbol)])
(concrete All  [(id TV) (body type)])
```

concrete

- Defines new value constructors
- (Like PLT Scheme's define-struct)

Data Definitions

```
(concrete Unit []) 
(concrete Arr  [(dom type) (cod type)])
(concrete TV   [(id symbol)])
(concrete All  [(id TV) (body type)])

(abstract type [Unit Arr TV All])
```

concrete

- Defines new value constructors
- (Like PLT Scheme's define-struct)

abstract

- Introduces named unions
- Used in patterns for function dispatch

Function Sets

```
(funcset
  [(TV)            (tv)        ...]
  [(Arr set set)   (a s2 s2)   ...])
```

- Each function case:

  [($ctor_1 \ldots ctor_n$) ($arg_1 \ldots arg_n$) *exp*]
- Each $ctor_i$ may be concrete or abstract
- Chooses the “best” function

Function Sets

```
(funcset
  [(TV)            (tv)        ...]
  [(Arr set set)   (a s2 s2)   ...]
  [(any set set)   (a s2 s2)   ...])
```

- Each function case:

  [($ctor_1 \ldots ctor_n$) ($arg_1 \ldots arg_n$) *exp*]
- Each $ctor_i$ may be concrete or abstract
- Chooses the “best” function

Traversals

- Depth-first, bottom-up folds
- At each data constructor:
  - Recur on all the fields
  - Dispatch on original value and field results

TBGP: Example: Free Type Variables

```
;; type → set
;; What are the free type variables?
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set-∪ set-∅)
      (funcset
        [(TV)            (tv)       (set-sing tv)]
        [(All set set)   (a t bdy)  (set-\ bdy t)]))))
```

TBGP: Example: Free Type Variables

```
;; type → set
;; What are the free type variables?
(define (ftv t)
  (traverse t
    (union-func
⇒     (build-TU set set-∪ set-∅)
      (funcset
        [(TV)              (tv)       (set-sing tv)]
        [(All set set)     (a t bdy) (set-\ bdy t)]))))
```

TBGP: Example: Free Type Variables

```
;; type → set
;; What are the free type variables?
(define (ftv t)
   (traverse t
      (union-func
         (build-TU set set-∪ set-∅)
         (funcset
⇒           [(TV)               (tv)        (set-sing tv)]
            [(All set set)      (a t bdy) (set-\ bdy t)])))))
```

TBGP: Example: Free Type Variables

```
;; type → set
;; What are the free type variables?
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set-∪ set-∅)
      (funcset
        [(TV)            (tv)        (set-sing tv)]
⇒       [(All set set)   (a t bdy) (set-\ bdy t)]))))
```

TBGP: Example: Free Type Variables

```
;; type → set
;; What are the free type variables?
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set-∪ set-∅)
      (funcset
        [(TV)               (tv)         (set-sing tv)]
        [(All set set)      (a t bdy) (set-\ bdy t)])))))


(abstract type [Unit Arr TV All])
```

TBGP: Example: Free Type Variables

```
;; type → set
;; What are the free type variables?
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set-∪ set-∅)
      (funcset
        [(TV)            (tv)       (set-sing tv)]
        [(All set set)   (a t bdy) (set-\ bdy t)])))))


(abstract type [Unit Arr TV All Int])
```

TBGP: Example: Free Type Variables

```
;; type → set
;; What are the free type variables?
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set-∪ set-∅)
      (funcset
        [(TV)             (tv)       (set-sing tv)]
        [(All set set)    (a t bdy) (set-\ bdy t)])))))


(abstract type [Unit Arr TV All Int])
(abstract expr [Var Lam App TLam TApp])
```

TBGP: Example: Free Type Variables

```
;; any → set
;; What are the free type variables?
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set-∪ set-∅)
      (funcset
        [(TV)             (tv)       (set-sing tv)]
        [(All set set)    (a t bdy) (set-\ bdy t)]
        [(TLam set set)   (a t bdy) (set-\ bdy t)]))))

(abstract type [Unit Arr TV All Int])
(abstract expr [Var Lam App TLam TApp])
```

TBGP: Example: Free Type Variables

```
;; any → set
;; What are the free type variables?
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set-∪ set-∅)
      (funcset
        [(TV)            (tv)       (set-sing tv)]
        [(All set set)   (a t bdy)  (set-\ bdy t)]
        [(TLam set set)  (a t bdy)  (set-\ bdy t)]))))

(abstract type [Unit Arr TV All Int Refn])
(abstract expr [Var Lam App TLam TApp])
(concrete Refn [(ty type) (pred expr)])
```

Going Wrong

```
(traverse (Unit)
  (funcset [(Unit) (u) #t]))
```

Going Wrong

```
(traverse (Unit)
  (funcset [(Unit) (u) #t]))

(traverse (Unit)
  (funcset [(TV string) (tv id) id]))
```

Going Wrong

```
(traverse (Unit)
  (funcset [(Unit) (u) #t]))
```

```
(traverse (Unit)
  (funcset [(TV string) (tv id) id]))
```

```
(traverse (TV 'a)
  (funcset [(TV string) (tv id) id]))
```

Going Wrong

```
(traverse (Unit)
  (funcset [(Unit) (u) #t]))
```

```
(traverse (Unit)
  (funcset [(TV string) (tv id) id]))
error: No applicable function found
```

```
(traverse (TV 'a)
  (funcset [(TV string) (tv id) id]))
error: No applicable function found
```

A Typed Model: AP-F

- Data definitions (concrete and abstract)
- Function sets with multiple dispatch
- Structurally recursive traversal

Syntax

x	$\in$	Var	variable names
C	$\in$	$CVar$	concrete type names
A	$\in$	$AVar$	abstract type names

Syntax

$$
\begin{array}{lcll}
x & \in & \mathit{Var} & \text{variable names} \\
C & \in & \mathit{CVar} & \text{concrete type names} \\
A & \in & \mathit{AVar} & \text{abstract type names} \\
T & ::= & C \mid A & \text{types} \\
D & ::= & \texttt{(concrete } C \texttt{ [} T_1 \ldots T_n \texttt{])} & \text{structure definitions} \\
  & \mid & \texttt{(abstract } A \texttt{ [} T_1 \ldots T_n \texttt{])} & \text{union definitions}
\end{array}
$$

Syntax

x	$\in$	Var	variable names
C	$\in$	$CVar$	concrete type names
A	$\in$	$AVar$	abstract type names
T	$::=$	$C \mid A$	types
D	$::=$	`(concrete` C `[` $T_1 \ldots T_n$ `])`	structure definitions
	$\mid$	`(abstract` A `[` $T_1 \ldots T_n$ `])`	union definitions
e	$::=$	x	variable expressions
	$\mid$	`(`$C\ e_1 \ldots e_n$`)`	structure expressions
	$\mid$	`(traverse` $e\ F$ `)`	traversal expressions
F	$::=$	`(funcset` $f_1 \ldots f_n$`)`	function sets
f	$::=$	`[(`$T_0 \ldots T_n$`) (`$x_0 \ldots x_n$`)` e `]`	single functions

Syntax

x	$\in$	Var	variable names
C	$\in$	$CVar$	concrete type names
A	$\in$	$AVar$	abstract type names
T	$::=$	$C \mid A$	types
D	$::=$	(concrete C [$T_1 \ldots T_n$])	structure definitions
	$\mid$	(abstract A [$T_1 \ldots T_n$])	union definitions
e	$::=$	x	variable expressions
	$\mid$	($C\ e_1 \ldots e_n$)	structure expressions
	$\mid$	(traverse e F)	traversal expressions
F	$::=$	(funcset $f_1 \ldots f_n$)	function sets
f	$::=$	[($T_0 \ldots T_n$) ($x_0 \ldots x_n$) e]	single functions
P	$::=$	$D_1 \ldots D_n\ e$	programs

Dynamics: Additional Syntax

Values

$$v ::= (C\ v_1\ \ldots\ v_n)$$

Run-time expressions

$$\begin{array}{rcl} e & ::= & \cdots \\ & | & (\texttt{dispatch}\ F\ v_0\ e_1\ \ldots\ e_n) \\ & | & (\texttt{apply}\ f\ v_0\ v_1\ \ldots\ v_n) \end{array}$$

Evaluation contexts

$$\begin{array}{rcl} E & ::= & [\,] \\ & | & (C\ v\ \ldots\ E\ e\ \ldots) \\ & | & (\texttt{traverse}\ E\ F) \\ & | & (\texttt{dispatch}\ F\ v_0\ v\ \ldots\ E\ e\ \ldots) \end{array}$$

The Subtype Relation

$$\frac{(\texttt{abstract}\ A\ T_1\ \ldots\ T_i\ \ldots\ T_n) \in P}{T_i \le A}$$

$$\frac{}{T \le T} \qquad\qquad \frac{T \le T'' \qquad T'' \le T'}{T \le T'}$$

Dynamics: Reduction Relation

Structural Recursion

$$E[\,\texttt{(traverse}\ \texttt{(}C\ v_1\ \ldots\ v_n\texttt{)}\ \ F\ \texttt{)}\,]$$
$$\to\ E[\,\texttt{(dispatch}\ F\ \texttt{(}C\ v_1\ \ldots\ v_n\texttt{)}$$
$$\texttt{(traverse}\ v_1\ F\ \texttt{)}\ \ldots\ \texttt{(traverse}\ v_n\ F\ \texttt{))}\,]$$

Dispatch/Apply

$$E[\,\texttt{(dispatch}\ F\ v_0\ v_1\ \ldots\ v_n\texttt{)}\,]$$
$$\to\ E[\,\texttt{(apply}\ \ \mathit{choose}(F,\ \mathit{types}(\,v_0\ v_1\ \ldots\ v_n))\ \ v_0\ v_1\ \ldots\ v_n\ \texttt{)}\,]$$

$$E[\,\texttt{(apply}\ \texttt{[(}T_0\ \ldots\ T_n\texttt{)}\ \texttt{(}x_0\ \ldots\ x_n\texttt{)}\ e\ \texttt{]}\ \ v_0\ v_1\ \ldots\ v_n\ \texttt{)}\,]$$

$$\to\ E[\ e[\overline{v_i/x_i}]\ ]$$

Dynamics: Metafunctions

Are the signatures compatible?

$$possible((T_1 \; \ldots \; T_n),\, (T'_1 \; \ldots \; T'_n)) = \bigwedge T_i \leq T'_i \vee T'_i \leq T_i$$

Filter for the functions compatible with a signature:

$$possibleFs(F,\, (T_1 \; \ldots \; T_n)) = \{f \in F : possible(sig(f), (T_1 \; \ldots \; T_n)\}$$

Choose the "best" possible function:

$$choose(F, (C_1 \; \ldots \; C_n)) = \bigsqcap possibleFs(F, (C_1 \; \ldots \; C_n))$$

Type System

- No dispatch errors
- (No bogus values)

Type Judgments

$$\Gamma ::= \emptyset \mid \Gamma, x{:}T \qquad \text{value contexts}$$

$$\mathcal{X} ::= \emptyset \mid \mathcal{X}, T{:}T' \qquad \text{recursion contexts}$$

$\Gamma \vdash_e e : T$ — e has type T

$\Gamma \vdash_f f : T$ — f returns type T

$\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle T_0, F \rangle : T$ — traversing a T_0 using F evaluates to a T

Type Rules

[T-VAR]

$$\frac{}{\Gamma, x{:}T, \Gamma' \vdash_e x : T}$$

[T-NEW]

$$\frac{(\texttt{concrete}\ C\ \texttt{[}\ T_1\ \ldots\ T_n\,\texttt{]})\in P \qquad \Gamma \vdash_e e_i : T_i' \qquad T_i' \leq T_i \quad {}^{i\in\{1,\ldots,n\}}}{\Gamma \vdash_e \texttt{(}C\ e_1\ \ldots\ e_n\texttt{)} : C}$$

[T-FUNC]

$$\frac{\Gamma, x_0{:}T_0, \ldots, x_n{:}T_n \vdash_e e_0 : T}{\Gamma \vdash_f \texttt{[}\,\texttt{(}T_0\ \ldots\ T_n\texttt{)}\ \texttt{(}x_0\ \ldots\ x_n\texttt{)}\ e_0\,\texttt{]} : T}$$

Type Rules

[T-Var]

$$\frac{}{\Gamma, x{:}T, \Gamma' \vdash_e x : T}$$

[T-New]

$$\frac{\texttt{(concrete } C \texttt{ [} T_1 \ldots T_n \texttt{])} \in P \qquad \Gamma \vdash_e e_i : T_i' \quad T_i' \leq T_i \quad ^{i \in \{1,\ldots,n\}}}{\Gamma \vdash_e \texttt{(} C\ e_1 \ldots e_n \texttt{)} : C}$$

[T-Func]

$$\frac{\Gamma, x_0{:}T_0, \ldots, x_n{:}T_n \vdash_e e_0 : T}{\Gamma \vdash_f \texttt{[(} T_0 \ldots T_n \texttt{) (} x_0 \ldots x_n \texttt{)}\ e_0 \texttt{]} : T}$$

Type Rules

[T-VAR]

$$\frac{}{\Gamma, x{:}T, \Gamma' \vdash_e x : T}$$

[T-NEW]

$$\frac{\texttt{(concrete}\ C\ \texttt{[}\ T_1\ \ldots\ T_n\ \texttt{])} \in P \qquad \Gamma \vdash_e e_i : T_i' \quad T_i' \leq T_i \quad {}^{i \in \{1,\ldots,n\}}}{\Gamma \vdash_e \texttt{(}C\ e_1\ \ldots\ e_n\texttt{)} : C}$$

[T-FUNC]

$$\frac{\Gamma, x_0{:}T_0, \ldots, x_n{:}T_n \vdash_e e_0 : T}{\Gamma \vdash_f \texttt{[(}T_0\ \ldots\ T_n\texttt{)}\ \texttt{(}x_0\ \ldots\ x_n\texttt{)}\ e_0\ \texttt{]} : T}$$

Type Rules

[T-VAR]

$$\frac{}{\Gamma, x{:}T, \Gamma' \vdash_e x : T}$$

[T-NEW]

$$\frac{(\texttt{concrete}\ C\ [\ T_1\ \ldots\ T_n\ ])\in P \qquad \Gamma \vdash_e e_i : T'_i \qquad T'_i \leq T_i \quad {}^{i\in\{1,\ldots,n\}}}{\Gamma \vdash_e (C\ e_1\ \ldots\ e_n) : C}$$

[T-FUNC]

$$\frac{\Gamma, x_0{:}T_0, \ldots, x_n{:}T_n \vdash_e e_0 : T}{\Gamma \vdash_f [\,(T_0\ \ldots\ T_n)\ (x_0\ \ldots\ x_n)\ e_0\,] : T}$$

Type Rules

[T-Var]

$$\frac{}{\Gamma, x{:}T, \Gamma' \vdash_e x : T}$$

[T-New]

$$\frac{(\texttt{concrete}\ C\ [\ T_1\ \ldots\ T_n\ ])\in P \qquad \Gamma \vdash_e e_i : T_i' \qquad T_i' \leq T_i \qquad {}^{i\in\{1,\ldots,n\}}}{\Gamma \vdash_e (C\ e_1\ \ldots\ e_n) : C}$$

[T-Func]

$$\frac{\Gamma, x_0{:}T_0, \ldots, x_n{:}T_n \vdash_e e_0 : T}{\Gamma \vdash_f [\,(T_0\ \ldots\ T_n)\ (x_0\ \ldots\ x_n)\ e_0\,] : T}$$

Type Rules: Runtime Syntax

[T-Apply]

$$\frac{\Gamma \vdash_f f : T \qquad f = \texttt{[(}T_0 \ \ldots \ T_n\texttt{)}\ \texttt{(}x_0 \ \ldots \ x_n\texttt{)}\ e\ \texttt{]} \qquad \emptyset \vdash_e v_i : T_i' \qquad T_i' \leq T_i \quad {}^{i \in \{0,\ldots,n\}}}{\Gamma \vdash_e \texttt{(apply}\ f\ v_0\ v_1\ \ldots\ v_n\texttt{)} : T}$$

[T-Dispatch]

$$\frac{\emptyset \vdash_e v_0 : C \qquad \Gamma \vdash_e e_i : T_i' \quad {}^{i \in \{1,\ldots,n\}} \qquad \Gamma \vdash_f f : T_f \qquad T_f \leq T \quad {}^{f \in possibleFs(F, (C\ T_1' \ldots T_n'))}}{\Gamma \vdash_e \texttt{(dispatch}\ F\ v_0\ e_1\ \ldots\ e_n\ \texttt{)} : T}$$

Type Rules: Runtime Syntax

[T-Apply]

$$\frac{\Gamma \vdash_f f : T \qquad f = \texttt{[(}T_0 \; \ldots \; T_n\texttt{)}\;\texttt{(}x_0 \; \ldots \; x_n\texttt{)}\; e\;\texttt{]} \qquad \emptyset \vdash_e v_i : T'_i \qquad T'_i \leq T_i \quad {}^{i \in \{0,\ldots,n\}}}{\Gamma \vdash_e \texttt{(apply}\; f\; v_0\; v_1\; \ldots\; v_n\texttt{)} : T}$$

[T-Dispatch]

$$\frac{\emptyset \vdash_e v_0 : C \qquad \Gamma \vdash_e e_i : T'_i \quad {}^{i \in \{1,\ldots,n\}} \qquad \Gamma \vdash_f f : T_f \qquad T_f \leq T \quad {}^{f \in possibleFs(F, (C\; T'_1 \ldots T'_n))}}{\Gamma \vdash_e \texttt{(dispatch}\; F\; v_0\; e_1\; \ldots\; e_n\;\texttt{)} : T}$$

Type Rules: Runtime Syntax

[T-Apply]

$$\frac{\Gamma \vdash_f f : T \qquad f = \texttt{[(}T_0 \ \ldots \ T_n\texttt{)}\ \texttt{(}x_0 \ \ldots \ x_n\texttt{)}\ e\ \texttt{]} \qquad \emptyset \vdash_e v_i : T'_i \qquad T'_i \leq T_i \quad {}^{i \in \{0,\ldots,n\}}}{\Gamma \vdash_e \texttt{(apply}\ f\ v_0\ v_1\ \ldots\ v_n\texttt{)} : T}$$

[T-Dispatch]

$$\frac{\emptyset \vdash_e v_0 : C \qquad \Gamma \vdash_e e_i : T'_i \quad {}^{i \in \{1,\ldots,n\}} \qquad \Gamma \vdash_f f : T_f \qquad T_f \leq T \quad {}^{f \in possibleFs(F, (C\ T'_1 \ldots T'_n))}}{\Gamma \vdash_e \texttt{(dispatch}\ F\ v_0\ e_1\ \ldots\ e_n\ \texttt{)} : T}$$

Type Rules: Traversal Expression

$$\textsf{[T-Trav]}\quad \frac{\Gamma \vdash_e e_0 : T_0 \quad \Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0, F \rangle : T}{\Gamma \vdash_e (\texttt{traverse}\; e_0\; F) : T}$$

Starting with $\mathcal{X} = \emptyset$,
traversing T_0 with F returns T

Type Rules: Abstract Traversal

[T-ATRAV]

$$\frac{\begin{array}{c}(\texttt{abstract}\ A\ \texttt{[}\ T_1\ \ldots\ T_n\ \texttt{]})\in P \\ T_i \notin \mathrm{dom}\,\mathcal{X} \Rightarrow \Gamma;\mathcal{X}, A{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \quad {}^{i\in\{1,\ldots,n\}} \\ T_i \in \mathrm{dom}\,\mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad {}^{i\in\{1,\ldots,n\}} \\ T_i' \leq T \quad {}^{i\in\{1,\ldots,n\}}\end{array}}{\Gamma;\mathcal{X} \vdash_{\mathcal{T}} \langle A, F\rangle : T}$$

- Each T_i' is the traversal result for T_i
- Each T_i' is subsumed by T

Type Rules: Abstract Traversal

[T-ATRAV]

$$\frac{\begin{array}{c}(\texttt{abstract}\ A\ \texttt{[}\ T_1\ \ldots\ T_n\ \texttt{]})\in P\\ T_i\notin \mathrm{dom}\,\mathcal{X}\Rightarrow \Gamma;\mathcal{X},A{:}T\vdash_{\mathcal{T}}\langle T_i,F\rangle : T_i' \quad {}^{i\in\{1,\ldots,n\}}\\ T_i\in \mathrm{dom}\,\mathcal{X}\Rightarrow T_i'=\mathcal{X}(T_i) \quad {}^{i\in\{1,\ldots,n\}}\\ T_i'\leq T \quad {}^{i\in\{1,\ldots,n\}}\end{array}}{\Gamma;\mathcal{X}\vdash_{\mathcal{T}}\langle A,F\rangle : T}$$

- Each T_i' is the traversal result for T_i
- Each T_i' is subsumed by T

Type Rules: Abstract Traversal

$$\begin{array}{l}[\text{T-AT}\textsc{rav}]\\ \dfrac{\begin{array}{c}(\texttt{abstract}\ A\ \texttt{[}\ T_1\ \ldots\ T_n\ \texttt{]})\in P\\ T_i\notin\operatorname{dom}\mathcal{X}\Rightarrow\Gamma;\mathcal{X},A{:}T\vdash_{\mathcal{T}}\langle T_i,F\rangle : T'_i\quad{}^{i\in\{1,\ldots,n\}}\\ T_i\in\operatorname{dom}\mathcal{X}\Rightarrow T'_i=\mathcal{X}(T_i)\quad{}^{i\in\{1,\ldots,n\}}\\ T'_i\leq T\quad{}^{i\in\{1,\ldots,n\}}\end{array}}{\Gamma;\mathcal{X}\vdash_{\mathcal{T}}\langle A,F\rangle : T}\end{array}$$

- Each T'_i is the traversal result for T_i
- Each T'_i is subsumed by T

Type Rules: Abstract Traversal

[T-ATRAV]

$$\frac{\begin{array}{c}(\texttt{abstract}\ A\ \texttt{[}\ T_1 \ldots T_n\ \texttt{]}) \in P \\ T_i \notin \mathrm{dom}\,\mathcal{X} \Rightarrow \Gamma; \mathcal{X}, A{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \quad {}^{i\in\{1,\ldots,n\}} \\ T_i \in \mathrm{dom}\,\mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad {}^{i\in\{1,\ldots,n\}} \\ T_i' \leq T \quad {}^{i\in\{1,\ldots,n\}}\end{array}}{\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle A, F\rangle : T}$$

- Each T_i' is the traversal result for T_i
- Each T_i' is subsumed by T

Type Rules: Abstract Traversal

[T-ATRAV]

$$\frac{\begin{array}{c}(\texttt{abstract}\ A\ \texttt{[}\ T_1\ \ldots\ T_n\ \texttt{]}) \in P \\ T_i \notin \operatorname{dom} \mathcal{X} \Rightarrow \Gamma; \mathcal{X}, A{:}T \vdash_{\mathcal{T}} \langle T_i, F \rangle : T_i' \quad {}^{i \in \{1,\ldots,n\}} \\ T_i \in \operatorname{dom} \mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad {}^{i \in \{1,\ldots,n\}} \\ T_i' \leq T \quad {}^{i \in \{1,\ldots,n\}}\end{array}}{\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle A, F \rangle : T}$$

- Each T_i' is the traversal result for T_i
- Each T_i' is subsumed by T

Type Rules: Abstract Traversal

[T-ATRAV]

$$\frac{\begin{array}{c}\texttt{(abstract}\ A\ \texttt{[}\ T_1\ \ldots\ T_n\ \texttt{])} \in P \\ T_i \notin \mathrm{dom}\,\mathcal{X} \Rightarrow \Gamma; \mathcal{X}, A{:}T \vdash_{\mathcal{T}} \langle T_i, F \rangle : T_i' \quad {}^{i \in \{1,\ldots,n\}} \\ T_i \in \mathrm{dom}\,\mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad {}^{i \in \{1,\ldots,n\}} \\ T_i' \leq T \quad {}^{i \in \{1,\ldots,n\}}\end{array}}{\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle A, F \rangle : T}$$

- Each T_i' is the traversal result for T_i
- Each T_i' is subsumed by T

Type Rules: Abstract Traversal

$$
\textsc{[T-ATrav]}\quad
\frac{
\begin{array}{c}
(\texttt{abstract}\ A\ \texttt{[}\ T_1 \ldots T_n\ \texttt{]}) \in P \\
T_i \notin \mathrm{dom}\,\mathcal{X} \Rightarrow \Gamma; \mathcal{X}, A{:}T \vdash_{\mathcal{T}} \langle T_i, F \rangle : T'_i \quad {}^{i \in \{1,\ldots,n\}} \\
T_i \in \mathrm{dom}\,\mathcal{X} \Rightarrow T'_i = \mathcal{X}(T_i) \quad {}^{i \in \{1,\ldots,n\}} \\
T'_i \leq T \quad {}^{i \in \{1,\ldots,n\}}
\end{array}
}{
\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle A, F \rangle : T
}
$$

- Each T'_i is the traversal result for T_i
- Each T'_i is subsumed by T

Type Rules: Concrete Traversal

[T-CTRAV]

$$\frac{\begin{array}{c}\texttt{(concrete}\ C\ \texttt{[}\ T_1\ \ldots\ T_n\texttt{])} \in P \\ T_i \notin \operatorname{dom}\mathcal{X} \Rightarrow \Gamma; \mathcal{X}, C{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \quad {}^{i\in\{1,\ldots,n\}} \\ T_i \in \operatorname{dom}\mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad {}^{i\in\{1,\ldots,n\}} \\ \Gamma \vdash_f f : T_f \qquad T_f \leq T \quad {}^{f\in possibleFs(F,(C\ T_1' \ldots T_n'))} \\ covers(F, (C\ T_1'\ \ldots\ T_n'))\end{array}}{\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T}$$

- Each T_i' is the traversal result for T_i
- Each function T_j is subsumed by T
- The function set *covers* its possible invocation

Type Rules: Concrete Traversal

[T-CTRAV]

$$\frac{\begin{array}{c}\texttt{(concrete}\ C\ \texttt{[}\ T_1\ \ldots\ T_n\ \texttt{])} \in P \\ T_i \notin \operatorname{dom} \mathcal{X} \Rightarrow \Gamma; \mathcal{X}, C{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \quad ^{i\in\{1,\ldots,n\}} \\ T_i \in \operatorname{dom} \mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad ^{i\in\{1,\ldots,n\}} \\ \Gamma \vdash_f f : T_f \qquad T_f \leq T \quad ^{f\in possibleFs(F,(C\ T_1' \ldots T_n'))} \\ covers(F, (C\ T_1'\ \ldots\ T_n'))\end{array}}{\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T}$$

- Each T_i' is the traversal result for T_i
- Each function T_j is subsumed by T
- The function set *covers* its possible invocation

Type Rules: Concrete Traversal

$$
\textsc{[T-CTrav]}\quad
\dfrac{
\begin{array}{c}
(\texttt{concrete}\ C\ \texttt{[}\ T_1\ \ldots\ T_n\texttt{]})\in P \\
T_i \notin \mathrm{dom}\,\mathcal{X} \Rightarrow \Gamma;\mathcal{X}, C{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \quad {}^{i\in\{1,\ldots,n\}} \\
T_i \in \mathrm{dom}\,\mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad {}^{i\in\{1,\ldots,n\}} \\
\Gamma \vdash_f f : T_f \qquad T_f \le T \quad {}^{f\in possibleFs(F,(C\ T_1' \ldots T_n'))} \\
covers(F, (C\ T_1'\ \ldots\ T_n'))
\end{array}
}{
\Gamma;\mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T
}
$$

- Each T_i' is the traversal result for T_i
- Each function T_j is subsumed by T
- The function set *covers* its possible invocation

Type Rules: Concrete Traversal

[T-CTRAV]

$$\frac{\begin{array}{c}(\texttt{concrete}\ C\ \texttt{[}\ T_1\ \ldots\ T_n\texttt{]})\in P \\ T_i \notin \operatorname{dom}\mathcal{X} \Rightarrow \Gamma;\mathcal{X},C{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T_i' \quad {}^{i\in\{1,\ldots,n\}} \\ T_i \in \operatorname{dom}\mathcal{X} \Rightarrow T_i' = \mathcal{X}(T_i) \quad {}^{i\in\{1,\ldots,n\}} \\ \Gamma \vdash_f f : T_f \qquad T_f \leq T \quad {}^{f\in possibleFs(F,(C\ T_1' \ldots T_n'))} \\ covers(F, (C\ T_1'\ \ldots\ T_n'))\end{array}}{\Gamma;\mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T}$$

- Each T_i' is the traversal result for T_i
- Each function T_j is subsumed by T
- The function set *covers* its possible invocation

Type Rules: Concrete Traversal

[T-CTrav]

$$\frac{\begin{array}{c}(\texttt{concrete}\ C\ \texttt{[}\ T_1\ \ldots\ T_n\texttt{]})\in P \\ T_i \notin \mathrm{dom}\,\mathcal{X} \Rightarrow \Gamma;\mathcal{X},C{:}T \vdash_{\mathcal{T}} \langle T_i, F\rangle : T'_i \quad {}^{i\in\{1,\ldots,n\}} \\ T_i \in \mathrm{dom}\,\mathcal{X} \Rightarrow T'_i = \mathcal{X}(T_i) \quad {}^{i\in\{1,\ldots,n\}} \\ \Gamma \vdash_f f : T_f \qquad T_f \leq T \quad {}^{f\in possibleFs(F,(C\ T'_1\ \ldots\ T'_n))} \\ covers(F,(C\ T'_1\ \ldots\ T'_n))\end{array}}{\Gamma;\mathcal{X} \vdash_{\mathcal{T}} \langle C, F\rangle : T}$$

- Each T'_i is the traversal result for T_i
- Each function T_j is subsumed by T
- The function set *covers* its possible invocation

Leaf Covering

$$covers(F, (C\ T'_1\ \ldots\ T'_n))$$

Does F contain an applicable function for each *concrete instance* of $(C\ T'_1\ \ldots\ T'_n)$?

Leaf Covering: A Graph Problem

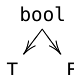
```
(concrete T [])
(concrete F [])
(abstract bool [T F])
(concrete Pair [bool bool])
```

$$covers(F, \texttt{(Pair bool bool)})$$

Leaf Covering: A Graph Problem

```
(concrete T [])
(concrete F [])
(abstract bool [T F])
(concrete Pair [bool bool])
```

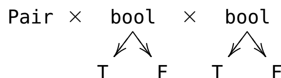
$$covers(F, \texttt{(Pair bool bool)})$$



Leaf Covering: A Graph Problem

```
(concrete T [])
(concrete F [])
(abstract bool [T F])
(concrete Pair [bool bool])
```

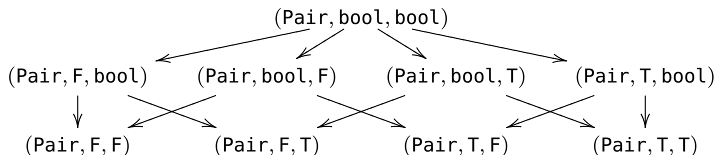
$$covers(F, \texttt{(Pair bool bool)})$$



Leaf Covering: A Graph Problem

```
(concrete T [])
(concrete F [])
(abstract bool [T F])
(concrete Pair [bool bool])
```

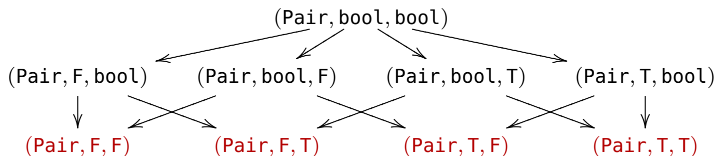
$$covers(F, \texttt{(Pair bool bool)})$$



Leaf Covering: A Graph Problem

$F =$ `(funcset)`

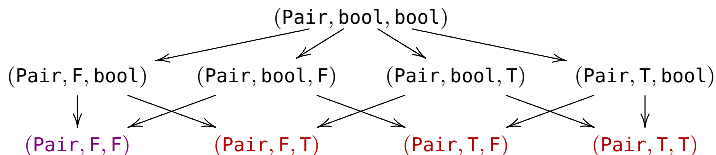
$covers(F,$ `(Pair bool bool)`$)$



Leaf Covering: A Graph Problem

$F =$ `(funcset`
`[(Pair F    F)` $\ldots$ `])`

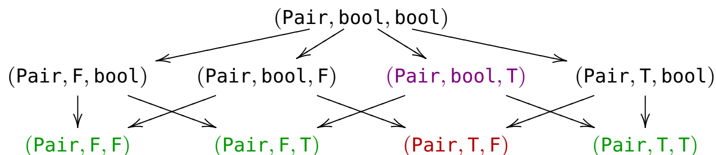
$$covers(F, \texttt{(Pair bool bool)})$$



Leaf Covering: A Graph Problem

```
F = (funcset
      [(Pair F    F)    ...]
      [(Pair bool T)    ...])
```

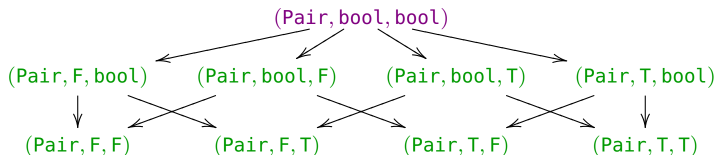
$$covers(F, \texttt{(Pair bool bool)})$$



Leaf Covering: A Graph Problem

```
F = (funcset
      [(Pair F    F)    ...]
      [(Pair bool T)    ...]
      [(Pair bool bool) ...])
```

$$covers(F, \texttt{(Pair bool bool)})$$



Soundness Sketch

Lemma (Function specialization).
If $\forall i \in \{1, \ldots, n\}, T'_i \leq T_i$ *then*
$$possibleFs(F, (T'_1 \ \ldots \ T'_n)) \subseteq possibleFs(F, (T_1 \ \ldots \ T_n))$$

Lemma (Traversals of subtypes).
For any well-typed traversal of a type T_0 *with* $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T_0, F \rangle : T$*, the traversal of any subtype* $T'_0 \leq T_0$ *gives us* $\Gamma; \emptyset \vdash_{\mathcal{T}} \langle T'_0, F \rangle : T'$ *for some* $T' \leq T$*.*

Theorem (Type Soundness).
Progress & Preservation

Thanks

Bryan Chadwick

chadwick@ccs.neu.edu

Karl Lieberherr

lieber@ccs.neu.edu

PLT Scheme implementation

http://www.ccs.neu.edu/home/chadwick/demeterf/apf-lib/

References

Ralf Hinze. A new approach to generic functional programming. In *In The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 1999.

P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *ACM Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

R. Lämmel, E. Visser, and J. Visser. The essence of strategic programming, 2004. URL `citeseer.ist.psu.edu/lammel02essence.html`.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the ACM SIGPLAN (TLDI 2003)*, volume 38, pages 26–37. ACM Press, March 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN (ICFP 2004)*, pages 244–255. ACM Press, 2004.

Andres Loeh, Johan Jeuring (editors); Dave Clarke, Ralf Hinze, Alexey Rodriguez, and Jan de Wit. Generic haskell user's guide – version 1.42 (coral). Technical Report UU-CS-2005-004, Department of Information and Computing Sciences, Utrecht University, 2005.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523, pages 124–144. Springer-Verlag, Berlin, 1991.

Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM Press, New York, 1993.

TBGP: Examples (2)

⇒ A flat list:

```
(map add1 '(1 2 3 4))
```

Nested lists:

```
(traverse '(1 (2 3) ((4)))
     (funcset [(cons any list) (c f r) (cons f r)]
              [(number)        (n)     (+ n 1)]))
```

Fully generic:

```
(traverse '(1 (2 3) ((4)))
     (union-func TP
                  (funcset [(number) (n) (+ n 1)])))
```

TBGP: Examples (2)

A flat list:

```
(map add1 '(1 2 3 4))
```

⇒ Nested lists:

```
(traverse '(1 (2 3) ((4)))
     (funcset [(cons any list) (c f r) (cons f r)]
              [(number)        (n)     (+ n 1)]))
```

Fully generic:

```
(traverse '(1 (2 3) ((4)))
     (union-func TP
                  (funcset [(number) (n) (+ n 1)])))
```

TBGP: Examples (2)

A flat list:

```
(map add1 '(1 2 3 4))
```

Nested lists:

```
(traverse '(1 (2 3) ((4)))
     (funcset [(cons any list) (c f r) (cons f r)]
              [(number)        (n)     (+ n 1)]))
```

⇒ Fully generic:

```
(traverse '(1 (2 3) ((4)))
     (union-func TP
                  (funcset [(number) (n) (+ n 1)])))
```

TBGP: Example: Pretty-printing

```
;; type → string
(define (type->string t)
  (traverse t
    (funcset
      [(Unit)                (u)        "unit"]
      [(TV symbol)           (tv id)    (symbol->string id)]
      [(All string string)  (a v bdy)
          (string-append "(∀" v ". " bdy ")")]
      [(Arr string string) (a d c)
          (string-append "(" d " → " c ")")])))
```

TBGP: Example: Pretty-printing

```
;; type → string
(define (type->string t)
  (traverse t
    (funcset
⇒     [(Unit)                (u)        "unit"]
      [(TV symbol)           (tv id)    (symbol->string id)]
      [(All string string)  (a v bdy)
          (string-append "(∀" v ". " bdy ")")]
      [(Arr string string) (a d c)
          (string-append "(" d " → " c ")")])))
```

TBGP: Example: Pretty-printing

```
;; type → string
(define (type->string t)
  (traverse t
    (funcset
      [(Unit)               (u)       "unit"]
⇒     [(TV symbol)          (tv id)   (symbol->string id)]
      [(All string string)  (a v bdy)
          (string-append "(∀" v ". " bdy ")")]
      [(Arr string string) (a d c)
          (string-append "(" d " → " c ")")])))
```

TBGP: Example: Pretty-printing

```
;; type → string
(define (type->string t)
  (traverse t
    (funcset
      [(Unit)                (u)        "unit"]
      [(TV symbol)           (tv id)    (symbol->string id)]
⇒     [(All string string)  (a v bdy)
         (string-append "(∀" v ". " bdy ")")]
      [(Arr string string) (a d c)
         (string-append "(" d " → " c ")")])))
```

TBGP: Example: Pretty-printing

```
;; type → string
(define (type->string t)
  (traverse t
    (funcset
      [(Unit)               (u)        "unit"]
      [(TV symbol)          (tv id)    (symbol->string id)]
      [(All string string)  (a v bdy)
          (string-append "(∀" v ". " bdy ")")]
⇒     [(Arr string string) (a d c)
          (string-append "(" d " → " c ")")])))
```