

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
→ go everywhere + 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
 - go everywhere + declarative control
- accumulators awkward
 - 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
 - go everywhere + declarative control
- accumulators awkward
 - orthogonal accumulators
- requires too many parameters
 - 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 ... *ctor*_{*n*}) (*arg*₁ ... *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:

```
[(ctor1 ... ctorn) (arg1 ... argn) 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  $\rightarrow$  set
```

```
;; What are the free type variables?
```

```
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set- $\cup$  set- $\emptyset$ )
      (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- $\cup$  set- $\emptyset$ )
```

```
      (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- $\cup$  set- $\emptyset$ )
```

```
      (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 \rightarrow set

;; What are the free type variables?

```
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set- $\cup$  set- $\emptyset$ )
      (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 \rightarrow set

;; What are the free type variables?

```
(define (ftv t)
  (traverse t
    (union-func
      (build-TU set set- $\cup$  set- $\emptyset$ )
      (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

$x \in \text{Var}$
 $C \in \text{CVar}$
 $A \in \text{AVar}$

variable names
concrete type names
abstract type names

Syntax

x	\in	Var	variable names
C	\in	$CVar$	concrete type names
A	\in	$AVar$	abstract type names
T	$::=$	$C \mid A$	types
D	$::=$	$(\text{concrete } C [T_1 \dots T_n])$ \mid $(\text{abstract } A [T_1 \dots T_n])$	structure definitions union definitions

Syntax

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

Dynamics: Additional Syntax

Values

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

Run-time expressions

$$e ::= \dots$$

- | (dispatch $F v_0 e_1 \dots e_n$)
- | (apply $f v_0 v_1 \dots v_n$)

Evaluation contexts

$$E ::= []$$

- | (C $v \dots E e \dots$)
- | (traverse $E F$)
- | (dispatch $F v_0 v \dots E e \dots$)

The Subtype Relation

$$\frac{(\text{abstract } A \ T_1 \ \dots \ T_i \ \dots \ T_n) \in P}{T_i \leq A}$$

$$\frac{}{T \leq T} \qquad \frac{T \leq T'' \quad T'' \leq T'}{T \leq T'}$$

Dynamics: Reduction Relation

Structural Recursion

$$\begin{aligned} E[(\text{traverse } (C \ v_1 \ \dots \ v_n) \ F)] \\ \rightarrow E[(\text{dispatch } F \ (C \ v_1 \ \dots \ v_n) \\ (\text{traverse } v_1 \ F) \ \dots \ (\text{traverse } v_n \ F))] \end{aligned}$$

Dispatch/Apply

$$\begin{aligned} E[(\text{dispatch } F \ v_0 \ v_1 \ \dots \ v_n)] \\ \rightarrow E[(\text{apply } \text{choose}(F, \text{types}(v_0 \ v_1 \ \dots \ v_n)) \ v_0 \ v_1 \ \dots \ v_n)] \end{aligned}$$

$$\begin{aligned} E[(\text{apply } [(T_0 \ \dots \ T_n) \ (x_0 \ \dots \ x_n) \ e] \ v_0 \ v_1 \ \dots \ v_n)] \\ \rightarrow E[e[\overline{v_i/x_i}]] \end{aligned}$$

Dynamics: Metafunctions

Are the signatures compatible?

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

Filter for the functions compatible with a signature:

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

Choose the “best” possible function:

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

Type System

- No dispatch errors
- (No bogus values)

Type Judgments

$\Gamma ::= \emptyset \mid \Gamma, x:T$

value contexts

$\mathcal{X} ::= \emptyset \mid \mathcal{X}, T:T'$

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]

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

[T-NEW]

$$\frac{\begin{array}{l} \text{(concrete } C [T_1 \dots T_n]) \in P \\ \Gamma \vdash_e e_i : T'_i \quad T'_i \leq T_i \quad i \in \{1, \dots, n\} \end{array}}{\Gamma \vdash_e (C e_1 \dots e_n) : C}$$

[T-FUNC]

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

Type Rules

[T-VAR]

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

[T-NEW]

$$\frac{\Gamma \vdash_e e_i : T'_i \quad T'_i \leq T_i \quad i \in \{1, \dots, n\}}{\Gamma \vdash_e (C \ e_1 \ \dots \ e_n) : C}$$

(concrete $C [T_1 \ \dots \ T_n] \in P$)

[T-FUNC]

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

Type Rules

[T-VAR]

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

[T-NEW]

$$\frac{\begin{array}{l} \text{(concrete } C [T_1 \dots T_n]) \in P \\ \Gamma \vdash_e e_i : T'_i \quad T'_i \leq T_i \quad i \in \{1, \dots, n\} \end{array}}{\Gamma \vdash_e (C e_1 \dots e_n) : C}$$

[T-FUNC]

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

Type Rules

[T-VAR]

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

[T-NEW]

$$\frac{\begin{array}{l} \text{(concrete } C [T_1 \dots T_n]) \in P \\ \Gamma \vdash_e e_i : T'_i \quad T'_i \leq T_i \quad i \in \{1, \dots, n\} \end{array}}{\Gamma \vdash_e (C e_1 \dots e_n) : C}$$

[T-FUNC]

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

Type Rules

[T-VAR]

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

[T-NEW]

$$\frac{\begin{array}{l} \text{(concrete } C [T_1 \dots T_n]) \in P \\ \Gamma \vdash_e e_i : T'_i \quad T'_i \leq T_i \quad i \in \{1, \dots, n\} \end{array}}{\Gamma \vdash_e (C e_1 \dots e_n) : C}$$

[T-FUNC]

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

Type Rules: Runtime Syntax

[T-APPLY]

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

[T-DISPATCH]

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

Type Rules: Runtime Syntax

[T-APPLY]

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

[T-DISPATCH]

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

Type Rules: Runtime Syntax

[T-APPLY]

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

[T-DISPATCH]

$$\frac{\begin{array}{l} \emptyset \vdash_e v_0 : C \\ \Gamma \vdash_e e_i : T'_i \quad i \in \{1, \dots, n\} \\ \Gamma \vdash_f f : T_f \quad T_f \leq T \quad f \in \text{possibleFs}(F, (C \ T'_1 \ \dots \ T'_n)) \end{array}}{\Gamma \vdash_e (\text{dispatch } F \ v_0 \ e_1 \ \dots \ e_n) : T}$$

Type Rules: Traversal Expression

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

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

Type Rules: Abstract Traversal

[T-ATRAV]

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

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

Type Rules: Abstract Traversal

[T-ATRAV]

(abstract $A [T_1 \dots T_n]$) $\in P$

$T_i \notin \text{dom } \mathcal{X} \Rightarrow \Gamma; \mathcal{X}, A:T \vdash_{\mathcal{T}} \langle T_i, F \rangle : T'_i \quad i \in \{1, \dots, n\}$

$T_i \in \text{dom } \mathcal{X} \Rightarrow T'_i = \mathcal{X}(T_i) \quad i \in \{1, \dots, n\}$

$T'_i \leq T \quad i \in \{1, \dots, n\}$

$\Gamma; \mathcal{X} \vdash_{\mathcal{T}} \langle A, F \rangle : T$

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

Type Rules: Abstract Traversal

[T-ATRAV]

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

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

Type Rules: Abstract Traversal

[T-ATRAV]

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

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

Type Rules: Abstract Traversal

[T-ATRAV]

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

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

Type Rules: Abstract Traversal

[T-ATRAV]

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

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

Type Rules: Abstract Traversal

[T-ATRAV]

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

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

Type Rules: Concrete Traversal

[T-CTRAV]

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

- 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]

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

- 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]

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

- 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]

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

- 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]

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

- 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

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

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

Leaf Covering: A Graph Problem

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

covers(F , (Pair bool bool))

Leaf Covering: A Graph Problem

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

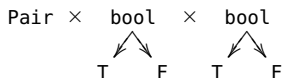
$\text{covers}(F, (\text{Pair bool bool}))$



Leaf Covering: A Graph Problem

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

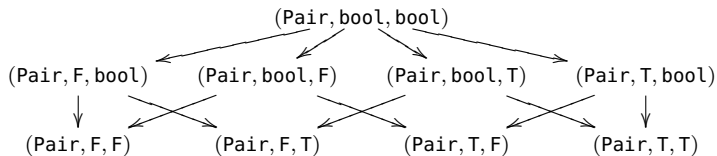
$\text{covers}(F, (\text{Pair bool bool}))$



Leaf Covering: A Graph Problem

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

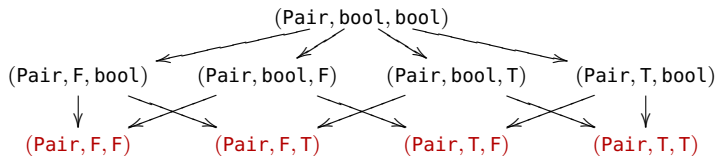
covers(*F*, (Pair bool bool))



Leaf Covering: A Graph Problem

$F = (\text{funcset})$

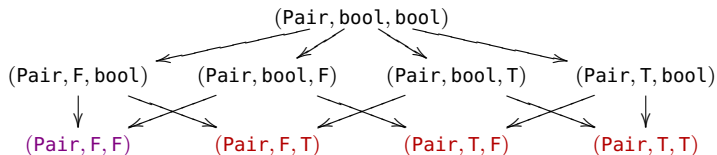
$\text{covers}(F, (\text{Pair bool bool}))$



Leaf Covering: A Graph Problem

$F = (\text{funcset}$
 $[(\text{Pair } F \quad F) \quad \dots])$

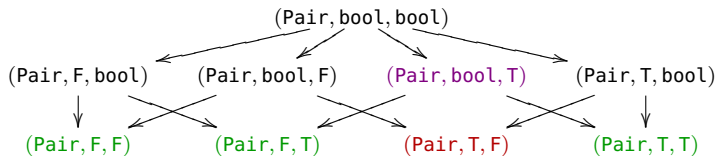
$\text{covers}(F, (\text{Pair } \text{bool } \text{bool}))$



Leaf Covering: A Graph Problem

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

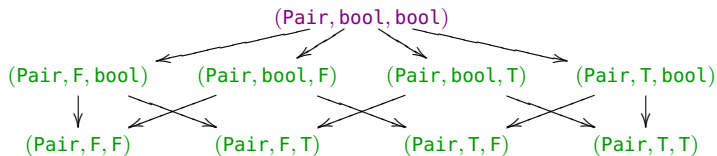
$\text{covers}(F, (\text{Pair bool bool}))$



Leaf Covering: A Graph Problem

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

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



Soundness Sketch

Lemma (Function specialization).

If $\forall i \in \{1, \dots, n\}, T'_i \leq T_i$ then

$possibleFs(F, (T'_1 \dots T'_n)) \subseteq possibleFs(F, (T_1 \dots T_n))$

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 ")")]))]
```