Practical Variable-Arity Polymorphism

Another Tread on the Stairway to Heaven

T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen

PLT @ Northeastern University

ESOP 2009

Adding Types Module by Module

lexer

;; *lex* **:** *Input-Port* → **Listof** *token*
(**define** (*lex port*) . . .)

parser

(**require** *lexer*)
;; *parse* **:** *Input-Port* → *ast*
(**define** (*parse port*) (**let** ([*tokens* (*lex port*)]) . . .))

compiler

(**require** *parser*)
;; *compiler* **:** *Input-Port* → *Target*
(**define** (*compiler port*) (**let**∗ ([*ast* (*parse tokens*)]) . . .))

lexer

;; *lex* **:** *Input-Port* → **Listof** *token*
(**define** (*lex port*) . . .)

parser

(**require/typed** *lexer* [*lex* (*Input-Port* → (**Listof** *token*))])
(**:** *parse* (*Input-Port* → *ast*))
(**define** (*parse port*) (**let** ([*tokens* (*lex port*)]) . . .))

compiler

(**require** *parser*)
;; *compiler* **:** *Input-Port* → *Target*
(**define** (*compiler port*) (**let**∗ ([*ast* (*parse tokens*)]) . . .))

lexer

(**:** *lex* (*Input-Port* → (**Listof** *token*)))
(**define** (*lex port*) . . .)

parser

(**require** *lexer*)
(**:** *parse* (*Input-Port* → *ast*))
(**define** (*parse port*) (**let** ([*tokens* (*lex port*)]) . . .))

compiler

(**require** *parser*)
;; *compiler* **:** *Input-Port* → *Target*
(**define** (*compiler port*) (**let**∗ ([*ast* (*parse tokens*)]) . . .))

lexer

```
(: lex (Input-Port → (Listof token)))
(define (lex port) . . . )
```

parser

```
(require lexer)
(: parse (Input-Port → ast))
(define (parse port) (let ([tokens (lex port)]) . . . ))
```

compiler

```
(require parser)
(: compiler (Input-Port → Target))
(define (compiler port) (let∗ ([ast (parse tokens)]) . . . ))
```

Typed Scheme

A module-based extension of PLT Scheme that can be used to gradually move Scheme code from untyped modules to typed modules.

- True union types
- Occurrence typing [POPL '08]
- Subtyping
- Polymorphism (and local inference)

Typed Scheme

A module-based extension of PLT Scheme that can be used to gradually move Scheme code from untyped modules to typed modules.

- True union types
- Occurrence typing [POPL '08]
- Subtyping
- Polymorphism (and local inference)
- Variadic Functions...?

Homogenous Variadic Functions

Already solved!

In Typed Scheme:

```
(: max (Num Num* → Num))
(define (max n0 . ns)
  (foldl (λ: ([n : Num] [i : Num]) (if (> n i) n i)) n0 ns))
```

In Java:

```
public static int max(int arg, int... args) {
  int res = arg;
  for (int n : args) { if (n > res) res = n; }
  return res;
}
```

Homogenous Variadic Functions

Already solved!

In Typed Scheme:

```
(: max (Num Num* → Num))
(define (max n0 . ns)
  (foldl (λ: ([n : Num] [i : Num]) (if (> n i) n i)) n0 ns))
```

In Java:

```
public static int max(int arg, int... args) {
  int res = arg;
  for (int n : args) { if (n > res) res = n; }
  return res;
}
```

Heterogenous Variadic Functions

Scheme's *map*:

$(\mathit{map}\ (\lambda\ (x)\ (+\ x\ 1))$
$\qquad (\mathit{list}\ 1\ 2\ 3))$

Heterogenous Variadic Functions

Scheme's *map*:

```
(map (λ (x) (+ x 1))
     (list 1 2 3))

(map (λ (s n) (string-append s " " (number->string n)))
     (list "France" "Germany" "UK" "US")
     (list 33 49 44 1))
```

Heterogenous Variadic Functions

Scheme's *map*:

```
(map (λ (x) (+ x 1))
     (list 1 2 3))

(map (λ (s n) (string-append s " " (number->string n)))
     (list "France" "Germany" "UK" "US")
     (list 33 49 44 1))
```

How to type *map*?

Typed Scheme's Previous Type for *map*

$$
\begin{array}{l}
(\forall\ (R\ A\ B\ C) \\
\quad (\textbf{case-lambda} \\
\quad\quad ((A \quad\quad \to\ R)\ (\textbf{Listof}\ A) \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \to\ (\textbf{Listof}\ R)) \\
\quad\quad ((A\ B \quad \to\ R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B) \quad\quad\quad\quad\quad\quad \to\ (\textbf{Listof}\ R)) \\
\quad\quad ((A\ B\ C \to\ R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B)\ (\textbf{Listof}\ C) \to\ (\textbf{Listof}\ R))))
\end{array}
$$

Haskell's *map*

In Haskell:

$$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

Haskell's *map*

In Haskell:

$$map :: (a \to b) \to [a] \to [b]$$
$$zipWith :: (a \to b \to c) \to [a] \to [b] \to [c]$$

Haskell's *map*

In Haskell:

$$map :: (a \to b) \to [a] \to [b]$$
$$zipWith :: (a \to b \to c) \to [a] \to [b] \to [c]$$
$$zipWith3 :: (a \to b \to c \to d) \to [a] \to [b] \to [c] \to [d]$$

Haskell's *map*

In Haskell:

$$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$
$$zipWith :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$$
$$zipWith3 :: (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow [a] \rightarrow [b] \rightarrow [c] \rightarrow [d]$$
$$\ldots$$
$$zipWith7 :: (a \rightarrow b \rightarrow \ldots \rightarrow h) \rightarrow [a] \rightarrow [b] \rightarrow \ldots \rightarrow [h]$$

Not Just Functions

In Scala:

Function0$<$*R*$>$

Function1$<$*R*, *A*$>$

Function2$<$*R*, *A*, *B*$>$

...

Function9$<$*R*, *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*$>$

Not Just Uses

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))
```

Not Just Uses

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))

(: fold-left
  (∀ (α β)
    ((α β* → α) α (Listof β)* → α)))
```

Heterogenous Variadic Types

A Type for *map*

$(\forall\ (R\ A)$
$\quad ((A \rightarrow R)\ (\textbf{Listof}\ A) \rightarrow (\textbf{Listof}\ R)))$
$(\forall\ (R\ A\ B)$
$\quad ((A\ B \rightarrow R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B) \rightarrow (\textbf{Listof}\ R)))$
$(\forall\ (R\ A\ B\ C)$
$\quad ((A\ B\ C \rightarrow R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B)\ (\textbf{Listof}\ C) \rightarrow (\textbf{Listof}\ R)))$
...

A Type for *map*

$(\forall\ (R\ A)$
$\quad ((A \rightarrow R)\ (\textbf{Listof}\ A) \rightarrow (\textbf{Listof}\ R)))$
$(\forall\ (R\ A\ B)$
$\quad ((A\ B \rightarrow R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B) \rightarrow (\textbf{Listof}\ R)))$
$(\forall\ (R\ A\ B\ C)$
$\quad ((A\ B\ C \rightarrow R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B)\ (\textbf{Listof}\ C) \rightarrow (\textbf{Listof}\ R)))$
$\ldots$
$(\forall\ (R\ A \qquad\qquad)$
$\quad ((A \qquad\qquad \rightarrow R)\ (\textbf{Listof}\ A) \qquad\qquad\qquad\qquad\qquad \rightarrow (\textbf{Listof}\ R)))$

A Type for *map*

$(\forall\ (R\ A)$
$\quad((A \to R)\ (\textbf{Listof}\ A) \to (\textbf{Listof}\ R)))$
$(\forall\ (R\ A\ B)$
$\quad((A\ B \to R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B) \to (\textbf{Listof}\ R)))$
$(\forall\ (R\ A\ B\ C)$
$\quad((A\ B\ C \to R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B)\ (\textbf{Listof}\ C) \to (\textbf{Listof}\ R)))$
$\ldots$
$(\forall\ (R\ A\ B \ldots)$
$\quad((A \qquad \to R)\ (\textbf{Listof}\ A) \qquad\qquad \to (\textbf{Listof}\ R)))$

A Type for *map*

$(\forall\ (R\ A)$
$\quad ((A \rightarrow R)\ (\textbf{Listof}\ A) \rightarrow (\textbf{Listof}\ R)))$
$(\forall\ (R\ A\ B)$
$\quad ((A\ B \rightarrow R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B) \rightarrow (\textbf{Listof}\ R)))$
$(\forall\ (R\ A\ B\ C)$
$\quad ((A\ B\ C \rightarrow R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B)\ (\textbf{Listof}\ C) \rightarrow (\textbf{Listof}\ R)))$
$\ldots$
$(\forall\ (R\ A\ B \ldots)$
$\quad ((A\ B \ldots \rightarrow R)\ (\textbf{Listof}\ A) \qquad\qquad\qquad \rightarrow (\textbf{Listof}\ R)))$

A Type for *map*

(∀ (*R* *A*)
  ((*A* → *R*) (**Listof** *A*) → (**Listof** *R*)))
(∀ (*R* *A* *B*)
  ((*A* *B* → *R*) (**Listof** *A*) (**Listof** *B*) → (**Listof** *R*)))
(∀ (*R* *A* *B* *C*)
  ((*A* *B* *C* → *R*) (**Listof** *A*) (**Listof** *B*) (**Listof** *C*) → (**Listof** *R*)))
. . .
(∀ (*R* *A* *B* . . .)
  ((*A* *B* . . . → *R*) (**Listof** *A*) (**Listof** *B*) . . . → (**Listof** *R*)))

Instantiating *map*

$(\forall\ (R\ A\ B \ldots)$
$((A\ B \ldots \to R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B) \ldots \to (\textbf{Listof}\ R)))$
@ [Int Bool Char Str]
$\Rightarrow$
$((A\ B \ldots \to R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B) \ldots$
$\to (\textbf{Listof}\ R))$

Instantiating *map*

$(\forall\ (R\ A\ B \ldots)$
$((A\ B \ldots \rightarrow R)$ **(Listof** A**)** **(Listof** B**)** $\ldots \rightarrow$ **(Listof** R**)))**

@ [Int Bool Char Str]

$\Rightarrow$

$((A\ B \ldots \rightarrow$ **Int**$)$ **(Listof** A**)** **(Listof** B**)** $\ldots$
$\rightarrow$ **(Listof Int))**

Instantiating *map*

$(\forall\ (R\ A\ B \ldots)$
$\quad((A\ B \ldots \to R)$ **(Listof** A**)** **(Listof** B**)** $\ldots \to$ **(Listof** R**)))**
@ [Int Bool Char Str]
$\Rightarrow$
((**Bool** $B \ldots \to$ **Int**) **(Listof Bool)** (**Listof** B) $\ldots$
$\to$ (**Listof Int**))

Instantiating *map*

(∀ (*R* *A* *B* . . .)
  ((*A* *B* . . . → *R*) (**Listof** *A*) (**Listof** *B*) . . . → (**Listof** *R*)))
@ [**Int** **Bool** **Char** **Str**]
⇒
((**Bool** *B* . . . → **Int**) (**Listof Bool**) (**Listof** *B*) . . .
 → (**Listof Int**))

Instantiating *map*

$(\forall\ (R\ A\ B\ \ldots)$
$((A\ B\ \ldots\ \to R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B)\ \ldots\ \to (\textbf{Listof}\ R)))$
@ [**Int Bool Char Str**]
$\Rightarrow$
$((\textbf{Bool}\ B_1\ B_2 \to \textbf{Int})\ (\textbf{Listof Bool})\ (\textbf{Listof}\ B_1)\ (\textbf{Listof}\ B_2)$
$\to (\textbf{Listof Int}))$

Instantiating *map*

$(\forall\ (R\ A\ B \ldots)$
$\quad((A\ B \ldots \rightarrow R)\ (\mathbf{Listof}\ A)\ (\mathbf{Listof}\ B) \ldots \rightarrow (\mathbf{Listof}\ R)))$
$\mathbf{@}\ [\mathbf{Int\ Bool\ Char\ Str}]$
$\Rightarrow$
$((\mathbf{Bool\ Char}\ B_2 \rightarrow \mathbf{Int})\ (\mathbf{Listof\ Bool})\ (\mathbf{Listof\ Char})\ (\mathbf{Listof}\ B_2)$
$\ \rightarrow (\mathbf{Listof\ Int}))$

Instantiating *map*

$(\forall\ (R\ A\ B \ldots)$
$\quad ((A\ B \ldots \rightarrow R)\ (\textbf{Listof}\ A)\ (\textbf{Listof}\ B) \ldots \rightarrow (\textbf{Listof}\ R)))$
$\textbf{@ [Int Bool Char Str]}$
$\Rightarrow$
$((\textbf{Bool Char Str} \rightarrow \textbf{Int})\ (\textbf{Listof Bool})\ (\textbf{Listof Char})\ (\textbf{Listof Str})$
$\ \rightarrow (\textbf{Listof Int}))$

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))
```

```
(: fold-left
  (∀ (α β)
    ((α β* → α) α (Listof β)* → α)))
```

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))
```

```
(: fold-left
  (∀ (α β ...)
    ((α β ... → α) α (Listof β) ... → α)))
```

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))

(: fold-left
  (∀ (α β ...)
    ((α β ... → α) α (Listof β) ... → α)))
```

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss) ;; (Listof β) ... ⇒ (Listof (Listof Any))
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))

(: fold-left
  (∀ (α β ...)
    ((α β ... → α) α (Listof β) ... → α)))
```

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))
```

```
(: fold-left
  (∀ (α β ...)
    ((α β ... → α) α (Listof β) ... → α)))
```

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))
```

```
(: fold-left
  (∀ (α β ...)
    ((α β ... → α) α (Listof β) ... → α)))
```

TD-MAP

$$\frac{\Gamma, \Delta, \Sigma \vdash e_r \triangleright \tau_r \ \ldots\ \alpha \qquad \Gamma, \Delta \cup \{\alpha\}, \Sigma \vdash e_f : (\tau_r \rightarrow \tau)}{\Gamma, \Delta, \Sigma \vdash (map\ \ e_f\ e_r) \triangleright \tau \ \ldots\ \alpha}$$

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))
```

```
(: fold-left
  (∀ (α β ...)
    ((α β ... → α) α (Listof β) ... → α)))
```

TD-MAP

$$\frac{\Gamma, \Delta, \Sigma \vdash \mathit{bss} \triangleright \tau_r \ldots \alpha \qquad \Gamma, \Delta \cup \{\alpha\}, \Sigma \vdash \mathit{car} : (\tau_r \rightarrow \tau)}{\Gamma, \Delta, \Sigma \vdash (\mathit{map}\ \mathit{car}\ \mathit{bss}) \triangleright \tau \ldots \alpha}$$

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))
```

```
(: fold-left
  (∀ (α β ...)
    ((α β ... → α) α (Listof β) ... → α)))
```

$$\textsc{TD-Map}\quad \frac{\Gamma, \Delta, \Sigma \vdash \mathit{bss} \triangleright (\textbf{Listof}\ \beta) \ldots \qquad \Gamma, \Delta \cup \{\alpha\}, \Sigma \vdash \mathit{car} : (\tau_r \to \tau)}{\Gamma, \Delta, \Sigma \vdash (\mathit{map}\ \mathit{car}\ \mathit{bss}) \triangleright \tau \ldots \alpha}$$

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))
```

```
(: fold-left
  (∀ (α β ...)
    ((α β ... → α) α (Listof β) ... → α)))
```

TD-MAP

$$\frac{\Gamma, \Delta, \Sigma \vdash \textit{bss} \triangleright (\textbf{Listof}\ \beta) \ldots \qquad \Gamma, \Delta \cup \{\beta\}, \Sigma \vdash \textit{car} : ((\textbf{Listof}\ \beta) \rightarrow \beta)}{\Gamma, \Delta, \Sigma \vdash (\textit{map}\ \textit{car}\ \textit{bss}) \triangleright \tau \ldots \alpha}$$

Defining Variadic Functions

```
(define (fold-left f a . bss)
  (if (ormap null? bss)
      a
      (apply fold-left
             f
             (apply f a (map car bss))
             (map cdr bss))))
```

```
(: fold-left
  (∀ (α β ...)
    ((α β ... → α) α (Listof β) ... → α)))
```

TD-Map

$$\frac{\Gamma, \Delta, \Sigma \vdash \mathit{bss} \triangleright (\textbf{Listof}\ \beta) \ldots \qquad \Gamma, \Delta \cup \{\beta\}, \Sigma \vdash \mathit{car} : ((\textbf{Listof}\ \beta) \rightarrow \beta)}{\Gamma, \Delta, \Sigma \vdash (\mathit{map}\ \mathit{car}\ \mathit{bss}) \triangleright \beta \ldots}$$

Multiple Dotted Type Variables

```
(define (length-results . fs)
  (λ args
    (length (apply append (map (λ (f) (f args)) fs)))))
```

Multiple Dotted Type Variables

```
(define (length-results . fs)
  (λ args
    (length (apply append (map (λ (f) (f args)) fs)))))
```

Multiple Dotted Type Variables

(**define** (*length-results* . *fs*)
  (λ *args*
    (*length* (*apply* *append* (*map* (λ (*f*) (*f* *args*)) *fs*)))))

($\forall$ (α ...)
  ((α ... $\rightarrow$ (**Listof Any**)) ... $\rightarrow$ (α ... $\rightarrow$ **Num**)))

Multiple Dotted Type Variables

```
(define (length-results . fs)
  (λ args
    (length (apply append (map (λ (f) (f args)) fs)))))
```

```
(∀ (α ...)
  (∀ (β ...)
    ((α ... → (Listof β)) ... → (α ... → Num))))
```

Multiple Dotted Type Variables

```
(define (length-results . fs)
  (λ args
    (length (apply append (map (λ (f) (f args)) fs)))))
```

$(\forall\ (\alpha\ \ldots)$
$\quad(\forall\ (\beta\ \ldots)$
$\qquad((\alpha\ \ldots\ \alpha \rightarrow (\textbf{Listof}\ \beta))\ \ldots\ \rightarrow (\alpha\ \ldots\ \alpha \rightarrow \textbf{Num}))))$

Multiple Dotted Type Variables

```
(define (length-results . fs)
  (λ args
    (length (apply append (map (λ (f) (f args)) fs)))))
```

$(\forall\ (\alpha\ \ldots)$
$\quad(\forall\ (\beta\ \ldots)$
$\quad\quad((\alpha\ \ldots\ \alpha \rightarrow (\textbf{Listof}\ \beta))\ \ldots\ \beta \rightarrow (\alpha\ \ldots\ \alpha \rightarrow \textbf{Num}))))$

Evaluation

The PLT Scheme Code Base

- $>$3000 Scheme source files
- $>$600,000 lines of code
- 5–10% of all functions are variadic

Uses of Built-In Heterogenous Variadic Functions

∼500 non-trivial uses

	30	Chosen at random
for-each	10	Precisely Typeable
andmap	10	Precisely Typeable
map	9	Precisely Typeable
	1	Uses a list as a 4-tuple

Uses of Built-In Heterogenous Variadic Functions

∼500 non-trivial uses

	30	Chosen at random
for-each	10	Precisely Typeable
andmap	10	Precisely Typeable
map	9	Precisely Typeable
	1	Uses a list as a 4-tuple

Variadic Function Definitions

∼1700 variadic definitions

	120	Chosen at random
Already Precise	68	do not use the rest argument
	26	have homogenous variadic types
	10	simulate optional arguments
Now Precise	12	require heterogenous variadic types
Imprecise	4	require further extensions

Variadic Function Definitions

∼1700 variadic definitions

	120	Chosen at random
Already Precise	68	do not use the rest argument
	26	have homogenous variadic types
	10	simulate optional arguments
Now Precise	12	require heterogenous variadic types
Imprecise	4	require further extensions

Variadic Function Definitions

$\sim$1700 variadic definitions

	120	Chosen at random
Already Precise	68	do not use the rest argument
	26	have homogenous variadic types
	10	simulate optional arguments
Now Precise	12	require heterogenous variadic types
Imprecise	4	require further extensions

Future Work

What Do We Need?

```
(define (zip . lsts)
  (apply map list lsts))
```

What Do We Need?

(**define** (*zip* . *lsts*)
  (*apply map list lsts*))

((**Listof Any**)* $\rightarrow$ (**Listof** (**Listof Any**))))

($\forall$ (α) ((**Listof** α)* $\rightarrow$ (**Listof** (**Listof** α))))

What Do We Need?

(**define** (*zip* . *lsts*)
  (*apply map list lsts*))

((**Listof Any**)* $\rightarrow$ (**Listof** (**Listof Any**))))

($\forall$ (α) ((**Listof** α)* $\rightarrow$ (**Listof** (**Listof** α))))

($\forall$ (α . . .)
  ($\forall$ (β . . .)
    ((**List** β . . . α) . . . β $\rightarrow$ (**List** (**List** β . . . β) . . . α))))

What Do You Need?

For Haskell:

> We've seen a solution for *map* using type classes.
>
> How would you write our *fold-left* in your language?

For Scala:

> We already have solutions for your language. See our technical report.

Conclusion

Conclusion

We have:

- designed a type system for heterogenous variadic functions and proven it sound;
- added hetereogenous variadic types to Typed Scheme; and
- shown the usefulness of the type system.

Conclusion

We have:

- designed a type system for heterogenous variadic functions and proven it sound;
- added hetereogenous variadic types to Typed Scheme; and
- shown the usefulness of the type system.

Conclusion

We have:

- designed a type system for heterogenous variadic functions and proven it sound;
- added hetereogenous variadic types to Typed Scheme; and
- shown the usefulness of the type system.

Thank you.

`http://www.plt-scheme.org/`