# PType System : A Featherweight Parallelizability Detector

**Dana N. Xu**[*]

**National University of Singapore**

joint work with

**Siau-Cheng Khoo**

**National University of Singapore**

**Zhenjiang Hu**

**University of Tokyo**

[*] *Now working at Computer Laboratory, University of Cambridge*

# Motivation

- Sequential programming is hard.
- Parallel programming is much, much harder.
- Multiprocessor systems have become increasingly available.
- Our approach (a good compromise)

  - Infer parallelizability of sequential functions via type system
- Parallelizability : if $F_s$ is parallelizable, then

  $\exists F_p \;.\; runtime(F_p)/runtime(F_s) \;=\; O(log\; m \;/\; m)$

  *where*

  $F_s$ - a sequential function.

  $F_p$ - parallel counterpart of $F_s$.

  $m$ - size of the input data.

# Existing Parallelization Approach - Using Skeletons

Skeleton functions: `map`, `reduce`, `scan`, etc

Code of the form

```
f xs = map g xs
```

can be parallelized as

```
f [] = []
f [a] = g a
f (x ++ y) = f x ++ f y
```

Code of the form

```
f xs = reduce op e xs
```

can be parallelized as

```
f [] = e
f [a] = a
f (x ++ y) = f x ‘op‘ f y
```

Note: `op` must be associative!

## Skeletons - Example

User's Sequential Definition:

```
f1 [] = 0
f1 (a:x) = (g a) + f1 x
```

Rewrite it with skeleton:

```
f1 xs = reduce (+) 0 (map g xs)
```

Parallel code generated:

```
f1 [] = 0
f1 [a] = g a
f1 (x ++ y) = f1 x + f1 y
```

## Life is not always that simple

User's Sequential Definition:

```
poly [a] c = a
poly (a:x) c = a + c * (poly x c)
```

Example: $p(x) = 2x^2 + 3x + 1$

```
p(5) = 2(25)+3(5)+1=66
poly [1,3,2] 5 = 1+5*(3+5*(2)) = 66
```

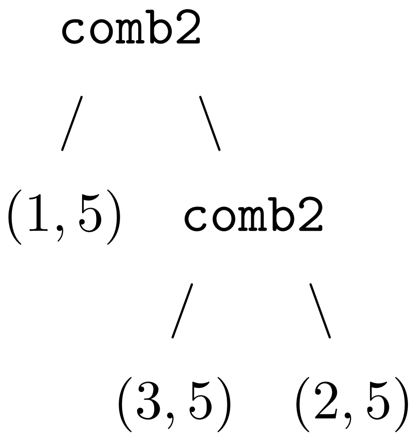
Not obvious how to use skeletons.

Thinking hard in bathtub .................

## Eureka Step! - invent an associative operator `comb2`

comb2 (p1,u1) (p2,u2) = (p1+p2*u1, u1*u2)

$p(x) = 2x^2 + 3x + 1$



`comb2` $(1, 5)$ (`comb2` $(3, 5)$ $(2, 5)$)

$= $ `comb2` $(1, 5)\ (3 + 2 * 5, 5 * 5)$

$= (1 + (3 + 2 * 5) * 5, 5 * 5 * 5)$

`comb2` (`comb2` $(1, 5)$ $(3, 5)$) $(2, 5)$

$= $ `comb2` $(1 + 3 * 5, 5 * 5)\ (2, 5)$

$= (1 + 3 * 5 + 2 * 5 * 5, 5 * 5 * 5)$

Rewrite to:

```
poly xs c = fst (polytup xs c)
polytup [a] c = (a,c)
polytup (a:x) c = (a,c) `comb2` (polytup x c)
```

## Eureka Step - Cont.

Rewrite it with skeleton:

```
poly xs c = fst (reduce comb2 (map (\x -> (x,c)) xs))
```

Parallel code generated:

```
poly [a] c = a
poly (xl ++ xr) c = poly xl c + (prod xl c)*(poly xr c)
prod [a] c = c
prod (xl ++ xr) c = (prod xl c)*(prod xr c)
```

This talk: Let's use type inference to replace the eureka step.

# Our Approach

- Given `f`, a function at-a-time
- type check `f` to derive a parallelizable type

  e.g. $R_{[+,*]}$ ("Recursion of `f` within $+$ and $*$") for `f`
- if this fails, do not parallelize `f`
- if OK, automatically transform `f` to a skeleton form and hence to parallel code.

# Extended-Ring Property

Let S $= [\oplus_1, \ldots, \oplus_n]$ be a sequence of $n$ binary operators. We say that S possesses the extended-ring property iff

**1.** all operators are associative;

**2.** each operator $\oplus$ has an identity, $\iota_\oplus$ such that

$$\forall v : \iota_\oplus \oplus v = v \oplus \iota_\oplus = v;$$

**3.** $\oplus_j$ is distributive over $\oplus_i$ $\forall i, j : 1 \leq i < j \leq n$

| | | |
|---|---|---|
| Example: | (Nat, [max, +, *], [0, 0, 1]) | Yes |
| | (Int, [+, *, ^ ], [0, 1, 1]) | No |

## Language Syntax

First Order, Strict Functional Language.

$$\begin{array}{rcl} e,t & \in & \textbf{Expressions} \\ e,t & ::= & n \mid v \mid c\ e_1\ \ldots\ e_n \mid e_1 \oplus e_2 \mid \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\ & & \mid f\ e_1\ \ldots\ e_n \mid \textbf{let}\ v = e_1\ \textbf{in}\ e_2 \\ p & \in & \textbf{Patterns} \\ p & ::= & v \mid c\ v_1\ \ldots\ v_n \\ \sigma & \in & \textbf{Programs} \\ \sigma & ::= & \gamma_i^*, (f_i\ p_1\ \ldots\ p_n = e)^*\ \forall\, i.\, i \geq 1 \end{array}$$

where $f_1$ is the main function.

$\gamma \in$ **Annotations** ( Declarations for Library Operators)

$\gamma ::= \#(\tau, [\oplus_1, \ldots, \oplus_n], [\iota_{\oplus_1}, \ldots, \iota_{\oplus_n}])$

# Skeleton Expressions Syntax

- $\underline{\bullet}$ denotes a recursive call.
- $\hat{e}$ is an expression $e$ which does not contain $\underline{\bullet}$.

$$
\begin{array}{lll}
sv & \in & \textbf{S–Values} \subseteq \textbf{Expressions} \\
sv & ::= & bv \mid \textbf{if}\ \hat{e_a}\ \textbf{then}\ \hat{e_b}\ \textbf{else}\ bv \\
bv & ::= & \underline{\bullet} \mid (\hat{e_1}\ \oplus_1\ \ldots\ \oplus_{n-1}\ \hat{e_n} \oplus_n\ \underline{\bullet}) \\
 & & \text{where } [\oplus_1, \ldots, \oplus_n] \text{ possesses the extended-ring property}
\end{array}
$$

# Examples of S-Value

```
f1 [a]   = a
f1 (a:x) = a + f1 x                                        Yes

f2 [a]   = a
f2 (a:x) = 2 * (a + f2 x)                                  No

f3 [a]   = a
f3 (a:x) = (2*a) + (2 * f3 x)                              Yes

f4 [a]   = a
f4 (a:x) = (double a + f2 x) + (sumlist x) * f4 x          Yes
```

# Type Expression

$$\begin{array}{lllllllll} \rho & \in & \texttt{PType} & \psi & \in & \texttt{NType} & \phi & \in & \texttt{RType} \\ \rho & ::= & \psi \mid \phi & \psi & ::= & N & \phi & ::= & R_S \end{array}$$

where $S$ is a sequence of operators

Example:

```
poly [a] c = a
poly (a:x) c = a + c * (poly x c)
```

Both `a` and `c` have PType $N$.

Expression `(a + c * (poly x c))` has PType $R_{[+,*]}$.

# Type Judgement

$$\Gamma \vdash_{\kappa} e :: \rho$$

$\Gamma$ - binds program variables to their `PTypes`.

$\kappa$ - is either a self-recursive call or a reference to such a call.

Example:

```
   :
f [a] = a
f (a:x) = e
where e = let v = a + f x
          in if (a>0) then v
                else 2 * (f x)
```

$$\Gamma \cup \{\mathtt{a} :: N, \mathtt{x} :: N\}$$
$$\vdash_{\{(\mathtt{f\ x}),\mathtt{v}\}} (\mathtt{if}\ (\mathtt{a} > 0)\ \mathtt{then}\ \mathtt{v}\ \mathtt{else}\ 2 * (\mathtt{f\ x})) :: R_{[+,*]}$$

# Type Checking Rules - I

$$\frac{v \neq \kappa}{\Gamma \cup \{v :: N\} \vdash_\kappa v :: N} \quad (\texttt{var} - \texttt{N}) \qquad \frac{v = \kappa}{\Gamma \cup \{v :: R_S\} \vdash_\kappa v :: R_S} \quad (\texttt{var}-\texttt{R})$$

$$\frac{}{\Gamma \vdash_\kappa n :: N} \quad (\texttt{con}) \qquad \frac{}{\Gamma \vdash_{(f\ x)} (f\ x) :: R_S} \quad (\texttt{rec})$$

$$\frac{\Gamma \vdash_\kappa e_1 :: N \quad \Gamma \vdash_\kappa e_2 :: \rho \quad (\rho = N) \vee (\rho = R_S \wedge \oplus \in S)}{\Gamma \vdash_\kappa (e_1 \oplus e_2) :: \rho} \quad (\texttt{op})$$

$$\frac{\Gamma \vdash_\kappa e :: N \quad g \notin FV(\kappa)}{\Gamma \vdash_\kappa (g\ e) :: N} \quad (\texttt{app}) \qquad \frac{\Gamma \vdash_\kappa e : \rho \quad \rho <: \rho'}{\Gamma \vdash_\kappa e :: \rho'} \quad (\texttt{sub})$$

# Type Checking Rules - II

$$\frac{\Gamma \vdash_{\kappa} e_1 :: N \quad \Gamma \cup \{v :: N\} \vdash_{\kappa} e_2 :: \rho}{\Gamma \vdash_{\kappa} (\textbf{let}\ v = e_1\ \textbf{in}\ e_2) :: \rho} \qquad (\texttt{let} - \texttt{N})$$

$$\frac{\Gamma \vdash_{\kappa} e_1 :: R_S \quad \Gamma \cup \{v :: R_S\} \vdash_{v} e_2 :: R_S}{\Gamma \vdash_{\kappa} (\textbf{let}\ v = e_1\ \textbf{in}\ e_2) :: R_S} \qquad (\texttt{let} - \texttt{R})$$

$$\frac{\Gamma \vdash_{\kappa} e_0 :: N \quad \Gamma \vdash_{\kappa} e_1 :: \rho_1 \quad \Gamma \vdash_{\kappa} e_2 :: \rho_2 \quad \bigtriangledown_{\texttt{if}}(\rho, \rho_1, \rho_2)}{\Gamma \vdash_{\kappa} (\textbf{if}\ e_0\ \textbf{then}\ e_1 \textbf{else}\ e_2) :: \rho} \qquad (\texttt{if})$$

$$\frac{}{\bigtriangledown_{\texttt{if}}(\rho, \rho, \rho)} \qquad \frac{}{\bigtriangledown_{\texttt{if}}(R_S, N, R_S)} \qquad \frac{}{\bigtriangledown_{\texttt{if}}(R_S, R_S, N)} \qquad (\texttt{if} - \texttt{merge})$$

# Soundness of PType System

$\leadsto$ : one step transformation of an expression.
$s{-}value$ : skeleton form which can be mapped directly to parallel code.

**Theorem 1 (Progress)** *If* $\Gamma \vdash_\kappa e :: R_S$, *then either* $e$ *is an s-value or* $e \leadsto \ldots \leadsto e'$ *where* $e'$ *is an s-value.*

**Theorem 2 (Preservation)** *If* $e :: R_S$ *and* $e \leadsto e'$, *then* $e' :: R_S$.

# Example 1 - The `mss` Problem

`mis` - maximum initial sum

`mss` - maximum segment sum

```
#(Int,[max,+],[0,0])
mis [a] = a
mis (a:x) = a ‘max‘ (a + mis x)
mss [a] = a
mss (a:x) = (a ‘max‘ (a + mis x)) ‘max‘ mss x
```

`mis` :: $R_{[\mathtt{max},+]}$

`mss` :: $R_{[\mathtt{max}]}$

# Example 2 - Fractal Image Decompression

`tr` - applies a list of transformations to a pixel.

`k` - applies these transformations to a set of pixels.

```
#(List,[++],[Nil])
#(Set,[union],[Nil])
tr :: [a -> a] -> a -> [a]
tr [f] p = [f p]
tr (f:fs) p = [f p] ++ tr fs p
k :: [[a]] -> [a]
k [a] fs = nodup (tr fs a)
k (a:x) fs = nodup (tr fs a) ‘union‘ (k x)
```

`tr` :: $R_{[\texttt{++}]}$

`k` :: $R_{[\texttt{union}]}$

## Relationship with Skeletons

```
map f [a] = [f a]
map f (a:x) = [f a] ++ map f x
reduce op e [a] = e ‘op‘ a
reduce op e (a:x) = a ‘op‘ reduce op e x
```

`map` :: $R_{[++]}$

`reduce` :: $R_{[op]}$

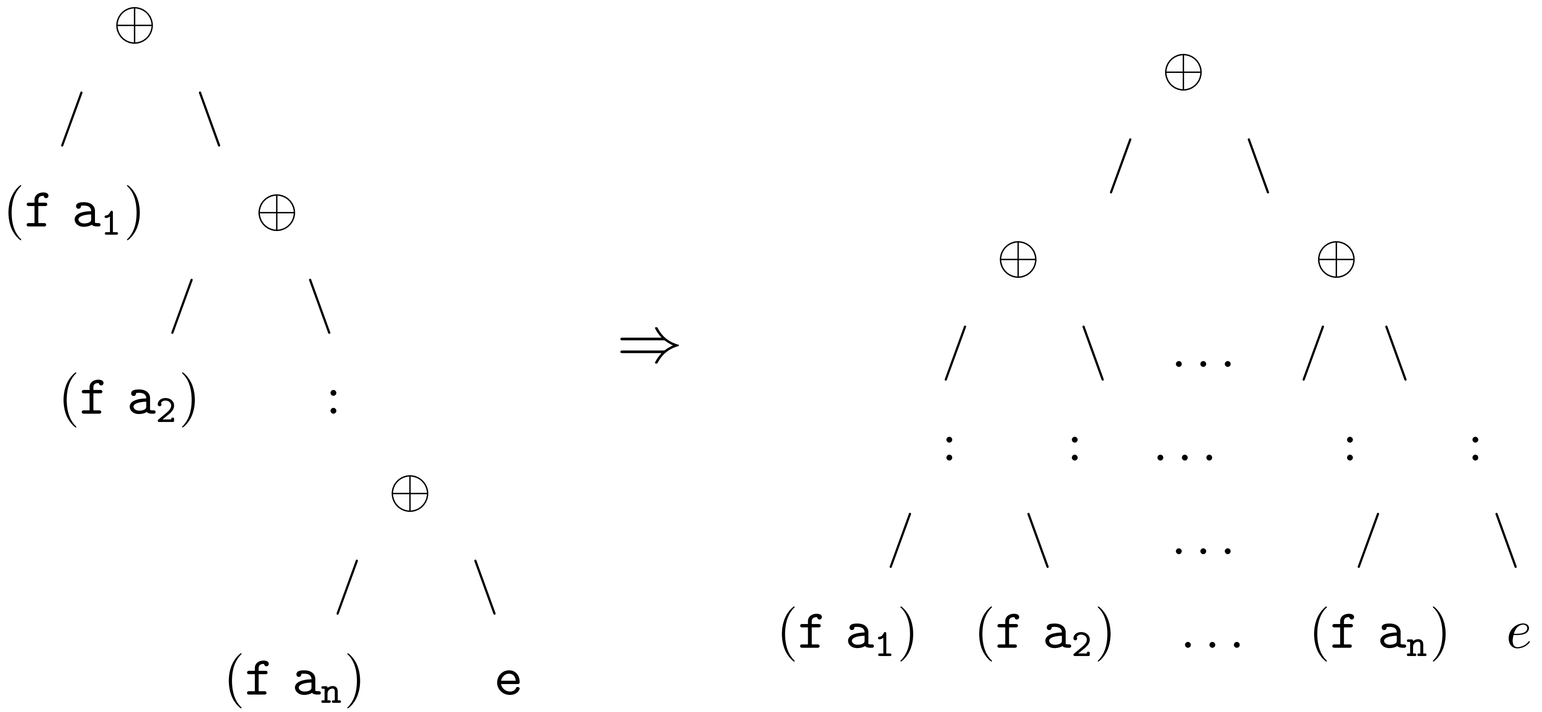
# Enhancements (done in the paper)

- Multiple Recursion Parameters
  - can handle zip-like functions.
- Accumulating Parameters
  - type-check parameters before type-check function body.
- Non-linear Mutual Recursion
  - commutativity is required.

# Conclusions

- Type system giving a novel insight into parallelizability.
- Modular : typecheck functions independently of callers.
- High level interface for programmers.
  - Do not need to explicitly write parallel program
  - Do not need to understand non-trivial concept (eg. skeletons and type system).
  - Only need to focus on extended ring property.
- A prototype system can be found at http://loris-4.ddns.comp.nus.edu.sg/~xun

# Parallelization



# Multiple Recursion Parameters

```
#(List Float, [++],[Nil])
polyadd [] ys = ys
polyadd xs [] = xs
polyadd (a:x) (b:y) = [(a + b)] ++ polyadd x y
```

`polyadd` $:: R_{[++]}$

# Accumulating Parameters

```
#(Bool,[&&],[True])               #(Int,[+,*],[0,1])
sbp x = sbp’ x 0
sbp’ [] c = c==0
sbp’ (a:x) c = if (a == ‘(‘)
                then sbp’ x (1 + c)
                else if (a == ‘)‘)
                         then (c>0) && (sbp’ x ((-1) + c)
                         else sbp’ x c
```

$\mathcal{C}[\![$ RHS of `sbp’` $]\!]_c =$

```
if (a == ’(’)  then 1+c
               else if (a == ’)’) then (-1) + c
                             else c
```

`c` :: $R_{[+]}$ `sbp` :: $N$ `sbp’` :: $R_{[\&\&]}$

# Example - Technical Indicators in Financial Analysis

```
#(Indicator Price, [+,*],[0,1])
ema (a:x)=(close a):ema' (a:x) (close a)
ema' [] p = []
ema' (a:x) p  = let r = (0.2 * (close a) + 0.8 * p)
                in [r] ++ ema' x r
```

`p` :: $R_{[+,*]}$
`ema'` :: $R_{[++]}$

# Non-linear Mutual Recursion

```
lfib [] = 1
lfib (a:x) = lfib x + lfib’ x
lfib’ [] = 0
lfib’ (a:x) = lfib x
```

Sketch of the type-checking process:

$$\Gamma \cup \{\mathtt{a} :: N, \mathtt{x} :: N\} \vdash_{\{(\mathtt{lfib\ x}),(\mathtt{lfib'\ x})\}} (\mathtt{lfib\ x} + \mathtt{lfib'\ x}) :: R_{[+]}$$

$$\Gamma \cup \{\mathtt{a} :: N, \mathtt{x} :: N\} \vdash_{\{(\mathtt{lfib\ x}),(\mathtt{lfib'\ x})\}} (\mathtt{lfib\ x}) :: R_{[\,]}$$

$$\vdash_{\{(\mathtt{lfib\ x}),(\mathtt{lfib'\ x})\}} (\mathtt{lfib\ x}) :: R_{[+]} \ \textit{since}\ R_{[\,]} <: R_{[+]}$$

$$\Gamma \cup \{\mathtt{a} :: N, \mathtt{x} :: N\} \vdash_{\{(\mathtt{lfib\ x}),(\mathtt{lfib'\ x})\}} ((\mathtt{lfib\ x} + \mathtt{lfib'\ x}), (\mathtt{lfib\ x})) :: R_{[+]}$$

# More on `List`

```
#(List,[++,map2],[Nil,Nil])
y `map2` z = map (\x -> y ++ x) z
```

**1.** map2 distributive over `++`

```
x `map2` (y ++ z)= x `map2` y ++ x `map2` z
```

**2.** map2 is semi-associative (i.e. `x op (y op z) = (x op' y) op z`)

```
x `map2` (y `map2` z) = (x ++ y) `map2` z
```

```
scan [a] = [[a]]
scan (a:x) = [[a] ++ ([a] `map2` (scan x))
```

`scan` $::R_{[++,map2]}$