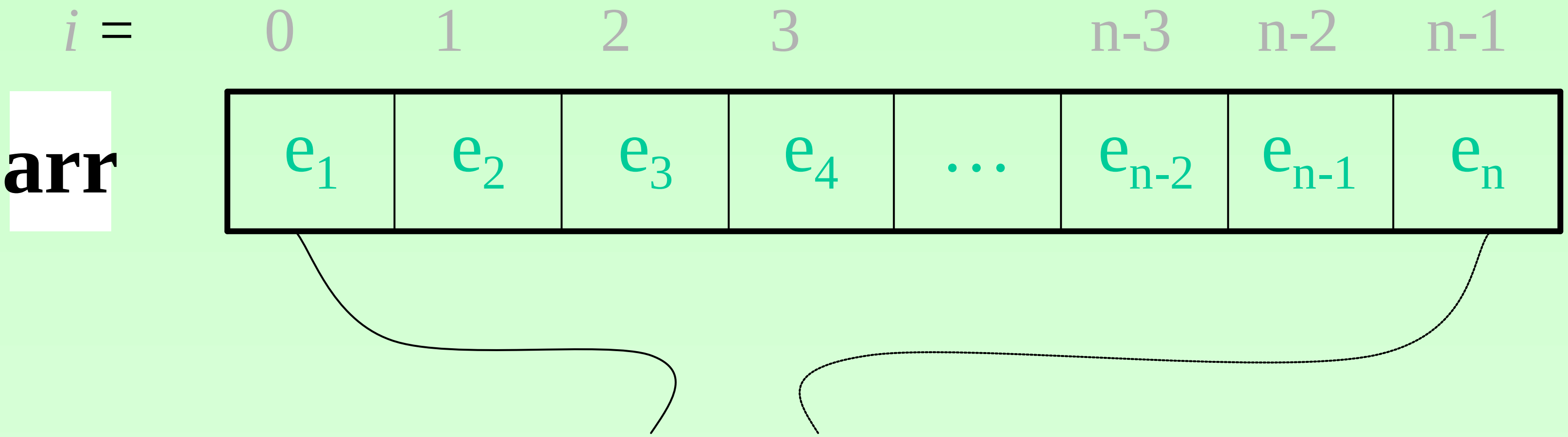
Deriving Preconditions for Array Bound Check Elimination

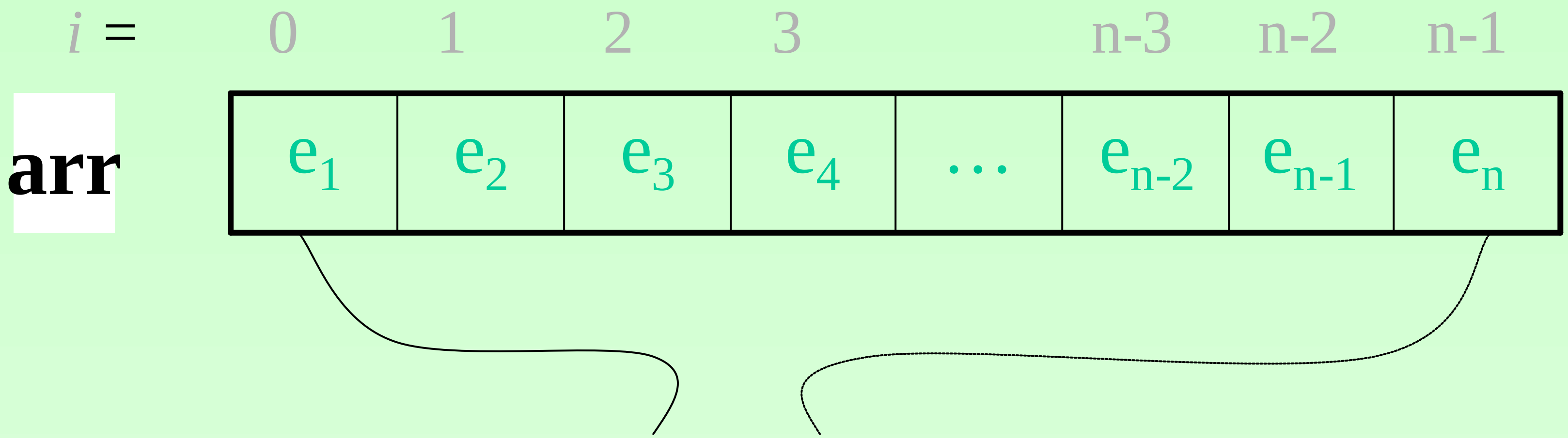
Dana N. XU
Joint work with W.N. CHIN and S.C. KHOO
Dept of Computer Science
School of Computing
National University of Singapore

Array Bound Checks



sub arr i = *if* (0 ≤ *i* < n)
then primeSub (arr, i);
else error "out of bound"

Array Bound Checks Elimination



sub arr i = ~~*if* (0 ≤ *i* < n)~~
~~*then*~~ primeSub(**A,** i);
~~*else* error “out of bound”~~

Motivation

- Checks are expensive.
- Precise Exception + Unsafe Checks => Less Optimisation
- Main difficulties
  - recursive procedures
  - partial redundancy

Our Solutions

- Base on Sized Typing
- Presburger Constraint Solving
- Partial Redundancy via Pre-conditions Derivation
- Utilize Recursive Invariants

Outline of Talk

- Motivation
- Language, Sized Types & Presburger Solver
- Key Idea
- Bound Checks Elimination Procedure
  - Context Synthesis
  - Deriving Weakest Pre-Condition
  - Converting Preconditions to Checks
  - Bound Check Specialisation

Language

$x \in \textbf{Var}$ ⟨Variables⟩
$a \in \textbf{Arr}$ ⟨Array Names⟩
$f \in \textbf{Var}$ ⟨Function Names⟩
$n \in \textbf{Int}$ ⟨Integer Constants⟩
$L \in \textbf{Label}$ ⟨Labels for checks⟩
$p \in \textbf{Prim}$ ⟨Primitives⟩

$$p ::= + \mid - \mid * \mid / \mid > \mid = \mid \; != \mid < \mid >= \mid <= \mid \mathit{not} \mid \mathit{or} \mid \mathit{and} \mid \mathit{length} \mid \mathit{newArr}$$

$\kappa \in \textbf{Call}$ ⟨Calls⟩

$$\kappa ::= L@\kappa \mid f\,(x_1,\ldots,x_n) \mid \mathit{sub}(a,x) \mid \mathit{update}(a,x_1,x_2)$$

$e \in \textbf{Exp}$ ⟨Expressions⟩

$$e ::= x \mid n \mid p\,(x_1,\ldots,x_n) \mid \kappa \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \mid \textbf{let } x = e_1 \textbf{ in } e_2$$

$d \in \textbf{Def}$ ⟨Function Definition⟩

$$d ::= f\;(x_1,\ldots,x_n) \;=\; e$$

Sized Type and Presburger Arithmetic

Sized Type = (AnnType, F)

Annotated Type Expressions:

$v \in \mathbf{V}$ ⟨Size Variables⟩

$t \in \mathbf{TVar}$ ⟨Type Variables⟩

$\sigma \in \mathbf{AnnType}$ ⟨Annotated Types⟩

$\sigma ::= \forall\, t\,.\,\sigma \mid \tau \mid \tau \rightarrow \tau$

$\tau \in \mathbf{Basic}$ ⟨Basic Type⟩

$\tau ::= t \mid (\tau_1, .., \tau_n) \mid \mathtt{Arr}^v\, \tau \mid \mathtt{Int}^v \mid \mathtt{Bool}^v$

Presburger Formulae:

$n \in \mathcal{Z}$ ⟨Integer constants⟩

$v \in \mathbf{V}$ ⟨Variable⟩

$\phi \in \mathbf{F}$ ⟨Presburger Formulae⟩

$\phi ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\, \phi$
$\quad \mid \exists\, v\,.\,\phi \mid \forall\, v.\,\phi$

$b \in \mathbf{BExp}$ ⟨Boolean Expression⟩

$b ::= \mathit{True} \mid \mathit{False} \mid a_1 = a_2 \mid$
$\quad a_1 \neq a_2 \mid a_1 < a_2 \mid a_1 > a_2 \mid$
$\quad a_1 \leq a_2 \mid a_1 \geq a_2$

$a \in \mathbf{AExp}$ ⟨Arithmetic Expression⟩

$a ::= n \mid v \mid n * a \mid a_1 + a_2 \mid -a$

Binary Search Example

```
getmid :: (Arr^a Int, Int^l, Int^h) → (Int^m, Int)
        Size  a ≥ 0 ∧ 2m ≤ l + h ∧ l + h ≤ 1 + 2m
getmid(arr, lo, hi) = let m = (lo + hi)/2
                      in   let x = L3@H3@sub arr m
                           in   (m, x)
cmp :: (Int^i, Int^j) → Int^r
        Size  (i < j ∧ r = −1) ∨ (i = j ∧ r = 0)
              ∨ (i > j ∧ r = 1)
cmp(k, x)  =  if k < x then −1
              else if k = x then 0 else 1
look :: (Arr^a Int, Int^l, Int^h, Int) → Int^r
        Size (a ≥ 0) ∧ ((l ≤ h) ∨ (l > h ∧ r = −1))
        Inv  a* = a ∧ l ≤ h, l* ∧ h* ≤ h ∧
             2 + 2l + 2h* ≤ h + 3l* ∧ l + 2h* < h + 2l*
look(arr, lo, hi, key)  =
        if (lo <= hi) then
           let (m, x) = L4@H4@getmid(arr, lo, hi)
           in   let t = cmp(key, x)
                in   if t < 0 then look(arr, lo, m − 1, key)
                     else if (t == 0) then m
                          else look(arr, m + 1, hi, key)
        else    −1
bsearch :: (Arr^a Int, Int) → Int
        Size (a ≥ 0)
bsearch(arr, key)  =  let v = length(arr)
                      in   L5@H5@look(arr, 0, v − 1, key)
```

Example:

```
getmid(arr,lo,hi)
            = let m=(lo+hi)/2 in
              let x=L@H@sub(arr,m) in (m,x)
```

Polymorphic type:

```
getmid :: (Arr α,Int,Int) → (Int,α)
sub :: (Arr α,Int) → α
```

Example:

```
getmid(arr,lo,hi)
            = let m=(lo+hi)/2 in
              let x=L@H@sub(arr,m) in (m,x)
```

Sized type

sub :: (Arr^{a} α,Int^{i}) $\rightarrow$ (α)

Size $(a>=0)$

getmid :: (Arr^{a} α,Int^{l},Int^{h}) $\rightarrow$ (Int^{m},α)

Size $(a>=0) \wedge (2m<=l+h) \wedge (1+2m>=l+h)$

Key Idea

`f(v`$_1$`,..,v`$_n$`) = … L@e …`

`ctx(L)`   `chk(L)`

Weakest pre-condition that can ensure that **`chk`** is *safe* under given context **`ctx`** is:

`pre = ¬ ctx ∨ chk`

pre ≡ ¬ ctx ∨ chk

f a x = if (x>=5) then a!x
else 0

It is safe to remove lower bound check under the condition

: (x>=5) Ç x ¸ 0

It is safe to remove higher bound check under the condition

: (x>=5) Ç x<(length a -1)

Example:

```
newsub :: (Arr^a α,Int^i,Int^j) → Int^r
newsub(arr,i,j) = if (0<=i<=j) then L1@H1@sub(arr,i)
                          else -1
```

We have:

```
ctx(L1) = (a>=0) ∧ (0<=i<=j)
chk(L1) = (i>=0)
```

```
pre(L1) = ¬ ctx(L1) ∨ chk(L1)
        = ¬(a>=0 ∧ 0<=i<=j) ∨ (i>=0)
        = True
```

Example:

```
newsub :: (Arr^a α,Int^i,Int^j) → Int^r
newsub(arr,i,j) = if (0<=i<=j) then L1@H1@sub(arr,i)
                           else -1
```

We have:

```
ctx(H1) = (a>=0 ∧ 0<=i<=j)
chk(H1)  = (i<a)
```

```
pre(H1) = ¬ ctx(H1) ∨ chk(H1)
        = ¬(a>=0 ∧ 0<=i<=j) ∨ (i<a)
        = (i<=-1) ∨ (j<i ∧ 0<=i) ∨ (i<a)
```

avoidance (under (i<=-1) ∨ (j<i ∧ 0<=i)) — *check* (under (i<a))

Check Classification

- Totally redundant `pre(L) = True`
- Unsafe/unknown `pre(L) = False`
- Partially redundant $pre(L) \wedge ctx(L) \Rightarrow chk(L)$

Example:

```
getmid(arr,lo,hi)   = let m=(lo+hi)/2 in
                      let x=L@H@sub(arr,m) in (m,x)
```

Sized type

sub :: (Arr^{a} a **,Int**^{i}**)** ® a

Size (a>=0)

(1) Required Checks

Req L : (i>=0), **H** : (i < a)

(2) Context Synthesis

ctx(L) = ctx(H)

= (2m<=l+h) ∧ (1+2m>=l+h)

getmid :: (Arr^{a} a **,Int**^{l}**,Int**^{h}**)** ® **(Int**^{m}**,** a **)**

Size (a>=0) ∧ (2m<=l+h) ∧ (1+2m>=l+h)

(3) Derived Precondition

Req **pre(L)** : (0<= l+h), **pre(H)** : (l+h < 2a)

Check Elimination : Steps

1. Context Synthesis
2. Pre-condition Derivation
3. From Pre-condition to Check
4. Bound Check Specialisation

Context Synthesis Algorithm

$$
\begin{array}{l}
\mathcal{C} \;::\; \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{F} \rightarrow (\mathbf{AnnType} \times \mathcal{P}((\mathbf{Label}, \mathbf{F})) \times \mathbf{F}) \\
\qquad\quad \textit{where}\ \mathbf{Env} = \mathbf{Var} \rightarrow \mathbf{AnnType} \times \mathbf{F} \\
\mathcal{C}[\![x]\!]\ \Gamma\ \psi = \textit{let}\ (\tau, \phi) = \Gamma[\![x]\!]\ \textit{in}\ (\tau, \emptyset, \phi) \\
\mathcal{C}[\![n]\!]\ \Gamma\ \psi = \textit{let}\ v = \textit{newVar}\ \textit{in}\ (\mathtt{Int}^v, \emptyset, (v = n)) \\
\mathcal{C}[\![f\ (x_1, \ldots, x_n)]\!]\ \Gamma\ \psi = \textit{let}\ \begin{array}[t]{lcl}
((\tau_1, \ldots, \tau_n) \rightarrow \tau, \phi_f) & = & \alpha(\Gamma[\![f]\!]) \\
X & = & \cup_{i=1}^{n} \{fv(\tau_i)\} \\
(\tau_i', \phi_i) & = & \Gamma[\![x_i]\!]\ \forall i \in \{1, \ldots, n\} \\
\phi & = & \exists X\ .\ \phi_f \wedge (\wedge_{i=1}^{n} (\phi_i \wedge (eq\ \tau_i'\ \tau_i)))
\end{array} \\
\qquad\qquad\qquad\qquad\qquad\quad \textit{in}\ (\tau, \emptyset, \phi)
\end{array}
$$

(Treatment of primitive operations is the same as that of function application.)

$$
\begin{array}{l}
\mathcal{C}[\![L@e]\!]\ \Gamma\ \psi = \textit{let}\ (\tau, \beta, \phi) = \mathcal{C}[\![e]\!]\ \Gamma\ \psi \\
\qquad\qquad\qquad\quad \textit{in}\ (\tau, \{(L, \mathcal{F}_{\Gamma,\psi})\} \cup \beta, \phi) \\
\mathcal{C}[\![\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]\ \Gamma\ \psi = \\
\qquad \textit{let}\ \begin{array}[t]{lcl}
(\mathtt{Bool}^v, \beta_0, \phi) & = & \mathcal{C}[\![e_0]\!]\ \Gamma\ \psi \\
(\tau_1, \beta_1, \phi_1) & = & \mathcal{C}[\![e_1]\!]\ \Gamma\ (\psi \wedge \phi \wedge (v = 1)) \\
(\tau_2, \beta_2, \phi_2) & = & \mathcal{C}[\![e_2]\!]\ \Gamma\ (\psi \wedge \phi \wedge (v = 0)) \\
\tau_3 & = & \alpha(\tau_1) \\
Y & = & \{v\} \cup fv(\tau_1) \cup fv(\tau_2) \\
\phi_3 & = & \exists Y\ .\ \phi \wedge (((eq\ \tau_1\ \tau_3) \wedge (v = 1) \wedge \phi_1) \\
 & & \qquad\qquad \vee ((eq\ \tau_2\ \tau_3) \wedge (v = 0) \wedge \phi_2))
\end{array} \\
\qquad \textit{in}\ (\tau_3, \beta_0 \cup \beta_1 \cup \beta_2, \phi_3) \\
\mathcal{C}[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]\ \Gamma\ \psi = \\
\qquad \textit{let}\ \begin{array}[t]{lcl}
(\tau_1, \beta_1, \phi_1) & = & \mathcal{C}[\![e_1]\!]\ \Gamma\ \psi \\
(\tau, \beta, \phi) & = & \mathcal{C}[\![e_2]\!]\ \Gamma[x :: (\tau_1, \phi_1)]\ \psi \\
Y & = & fv(\tau_1) \\
\phi_2 & = & \exists Y\ .\ (\phi_1 \wedge \phi)
\end{array} \\
\qquad \textit{in}\ (\tau, \beta_1 \cup \beta, \phi_2)
\end{array}
$$

Check Elimination : Steps

1. Context Synthesis
2. Pre-condition Derivation
3. From Pre-condition to Check
4. Bound Check Specialisation

Precondition of Recursion

- Make use of size invariant
- Separate analyses for
    - first recursive call
    - other recursive calls

Sized Invariant

```
look(arr,lo,hi,key) = if (lo<=hi) then
                        let (m,x)=L4@H4@getmid(arr,lo,hi) in
                        let t=cmp(key,x0) in
                          if (t<0) then look(arr,lo,m-1,key)
                          else if (t==0) then m
                          else look(arr,m+1,hi,key)
                      else -1
```

sized type:

$\texttt{look :: (Arr}^{a}\ \texttt{Int,Int}^{l}\texttt{,Int}^{h}\texttt{,Int)} \rightarrow \texttt{Int}^{r}$

size $(a>=0) \wedge ((l<=h) \vee ((l>h) \wedge (r = -1)))$

inv $(a^* = a) \wedge (l <= h,l^*) \wedge (h^* <=h)$

$\wedge\ (2+2h+2h^* <= l+3l^*) \wedge (l+2h^* <= h+2l^*)$

Recursive Procedure

Two Checks

```
chkFst(L4)  = 0<=l+h
chkRec(L4)  = 0<=l*+h*
```

Two Contexts

```
ctxFst(L4)  =  (l<=h)
ctxRec(L4)  =  (l*<=h*) ∧ (a*=a) ∧ (l<=h,l*)
               ∧ (h*<=h) ∧ (2+2h+2h*<=l+3l*) ∧ (l+2h*<=h+2l*)
```

Two Preconditions

```
preFst(L4)   = ¬ctxFst(L4) ∨ chkFst(L4)
             = (h<l) ∨ (0<=l+h)
preRec(L4)   = ¬ctxRec(L4) ∨ chkRec(L4)
             = (h<=l) ∨ (0<=l<h) ∨ (l=-1 ∧ h=0)
```

Combined Precondition

```
pre(L4)      = preFst(L4) ∧ preRec(L4)
             = (h<l) ∨ (0<=l+h ∧ 0<=l)
```

Check Elimination : Steps

1. Context Synthesis
2. Pre-condition Derivation
3. From Pre-condition to Check
4. Bound Check Specialisation

Converting Preconditions to Checks

- Interprocedural propagation of *safety pre-condition* to become *check.*
- Conversion Formulae used:

  `chk(C)= ∃ X. pre(L) ∧ subs(C)`

Converting Preconditions to Checks

```
look(arr,lo,hi,k) = … L4@H4@getmid(arr,lo,hi)…
```

```
pre(L4) = (h<l) ∨ (0<=l+h ∧ 0<=l)
pre(H4) = (h<=l) ∨ (h<a ∧ l+h<2a)
```

```
subs(L5) = subs(H5)
         = (l=0) ∧ (h=v-1)
```

```
bsearch(arr,key)    = let v=length(arr) in
                       L5@H5@look(arr,0,v-1,key)
```

```
chk(L5)  = ∃ l,h. pre(L4) ∧ subs(L5)
         = (v<=0) ∨ (1<=v)
chk(H5)  = ∃ l,h. pre(H4) ∧ subs(H5)
         = (v<=0) ∨ (v<=a,2a)
```

Interprocedural Propagation

```
ctx(L5) = ctx(H5)
        = (a>=0 ∧ v=a)
```

```
bsearch(arr,key)    = let v=length(arr) in
                       L5@H5@look(arr,0,v-1,key)
```

```
chk(L5) = (v<=0 ∨ 1<=v)
chk(H5) = (v<=0 ∨ v<=a,2a)
```

```
pre(L5) = ¬ctx(L5) ∨ chk(L5)
        = ∀v ¬(a>=0 ∧ v=a) ∨ (v<=0 ∨ 1<=v)
        = True
pre(H5) = ¬ctx(H5) ∨ chk(H5)
        = ∀v ¬(a>=0 ∧ v=a) ∨ (v<=0 ∨ v<=a,2a)
        = True
```

Check Elimination : Steps

1. Context Synthesis
2. Pre-condition Derivation
3. From Pre-condition to Check
4. Bound Check Specialisation

Bound Check Specialisation

Guided by each set of *Satisfiable Pre-Conditions*

```
getmid(arr,l,h)      =   … L@H@sub(arr,m)…
look(arr,lo,hi,k)    =      … L4@H4@getmid(arr,lo,hi)…
bsearch(arr,key)     = let v=length(arr) in
                        L5@H5@look(arr,0,v-1,key)
```

```
pre(L5) = True
pre(H5) = True
```

```
lookL4H4(arr,l,h,k) = look(arr,l,h,k) st pre(L4) ∧ pre(H4)
                    = … getmidLH(arr,lo,hi)…
getmidLH(arr,l,h)   = getmid(arr,l,h) st pre(L) ∧ pre(H)
                    = … subLH(arr,m)…
```

Bound Check Specialization

- Space-Time Trade-Off
  - Polyvariant (a version for each context of use)
  - Monovariant (a common minimal version)
  - Duovariant (a minimal and a maximal version)

Cost of Analysis (Constraint Solving)

	Forward	Backward
bcopy	0.03	0.21
binary search	0.54	0.07
bubble sort	0.05	0.31
dot product	0.03	0.21
hanoi	1.59	2.74
matrix mult	0.12	0.98
queens	0.19	0.53
sumarray	0.03	0.42

Contributions

- Combined Analysis
  - Forward Analysis for Context
  - Backward Analysis for Pre-condition

- Recursive Procedures

- Partial Redundancy without Code Motion.

- Guided Bound Check Specialisation.

Future Work

- Higher-order and polymorphic extension
- Other Safety Checks.
- Component Analysis.
- Imperative Languages