Modular Inference for Array Checks Optimization

Dana N. Xu
Computer Laboratory, University of Cambridge

Joint work with

Corneliu Popeea, Siau-Cheng Khoo, Wei-Ngan Chin
School of Computing
National University of Singapore

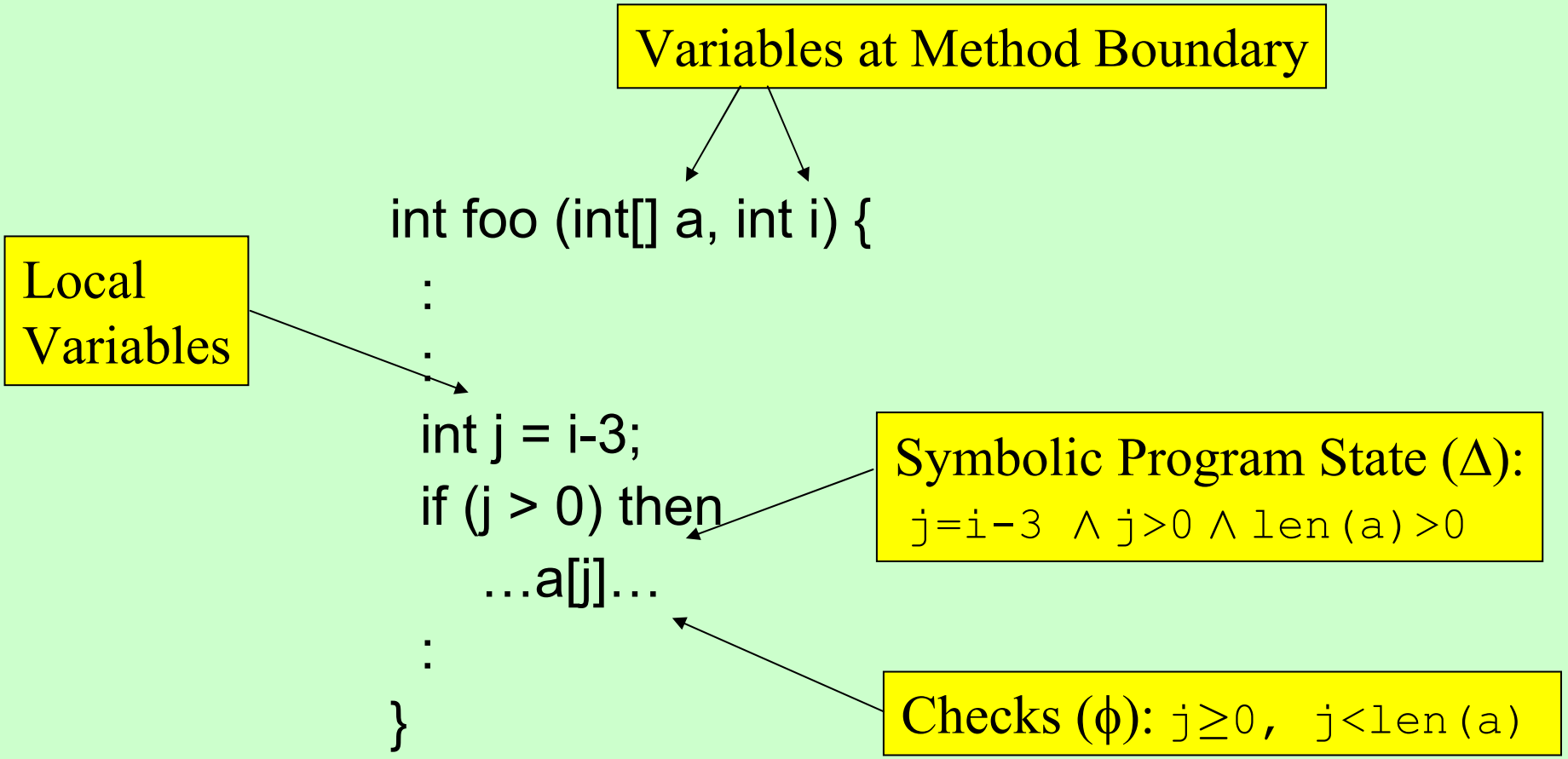
Array Bound Check Elimination

- Problem:
  - without array bound checks (e.g. C), programs may be unsafe.
  - with array bound checks (e.g. Java), programs are slowed down.
- Solution: remove redundant checks.

Current status

- Data flow analysis
  - Intra-procedural analysis [Kolte,Wolfe:*PLDI'95*; Bodik,Gupta,Sarkar:*PLDI'00*]
- Verification-Based Methods
  - Type Checking [Xi,Pfenning:*PLDI'98*]

Main Ideas


Variables at Method Boundary
Local Variables
int foo (int[] a, int i) {
:
.
int j = i-3;
if (j > 0) then
…a[j]…
:
}
Symbolic Program State (Δ):
j=i-3 ∧ j>0 ∧ len(a)>0
Checks (ϕ): j≥0, j<len(a)

Totally-safe Check

ϕ is *totally-safe* if $\Delta \Rightarrow \phi$

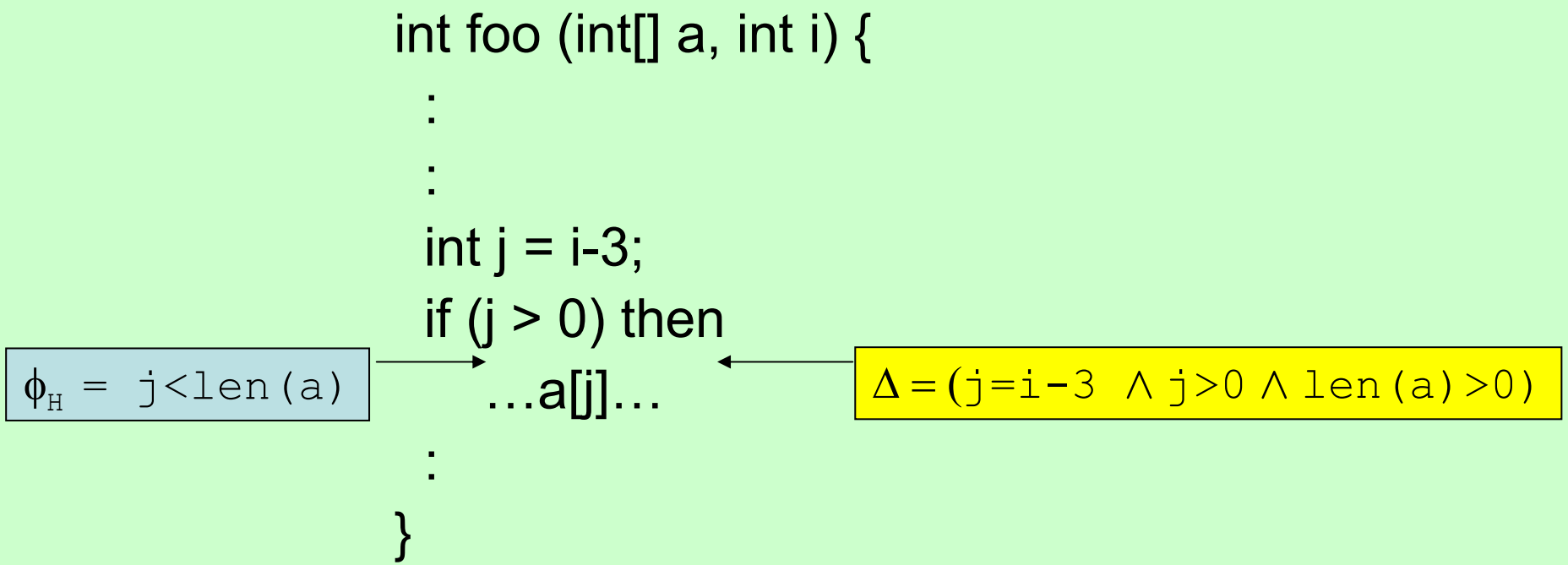
```
int foo (int[] a, int i) {
  :
  :
  int j = i-3;
  if (j > 0) then
     …a[j]…
  :
}
```

$\phi_L = j \geq 0$

$\Delta = (j=i-3 \wedge j>0 \wedge len(a)>0)$

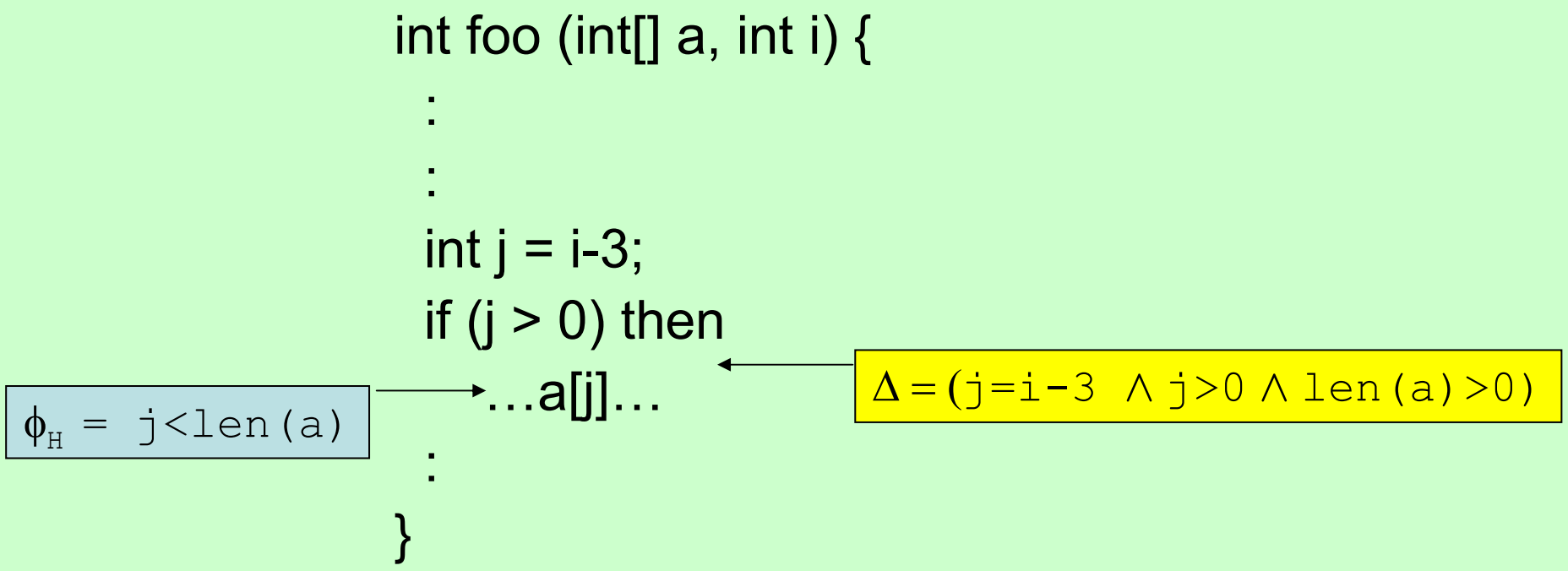
ϕ_L is **totally-safe**

Partially-safe Check



ϕ is not totally-safe because $\Delta \Rightarrow \phi$ is not a tautology

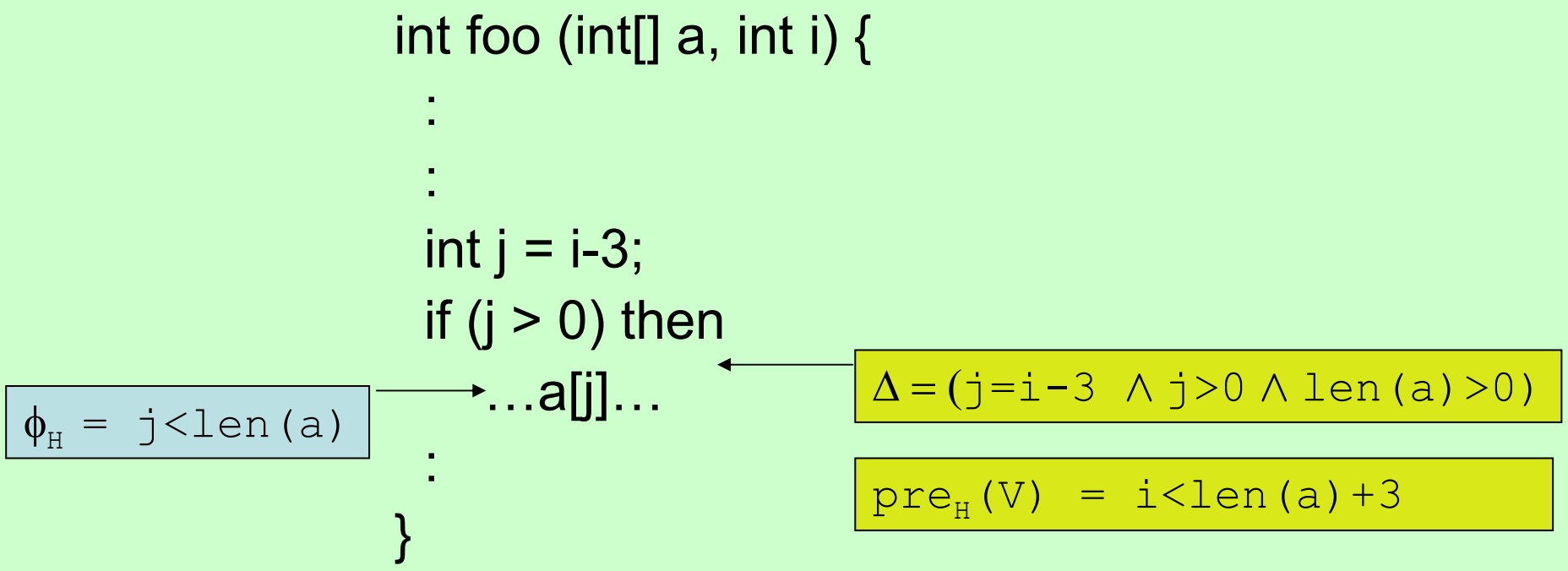
Partially-safe Check



But ϕ may be made safe when foo is called with a suitable state ψ: $\exists\psi.\ \psi \wedge \Delta \Rightarrow \phi$

We call ϕ a **partially-safe check.**

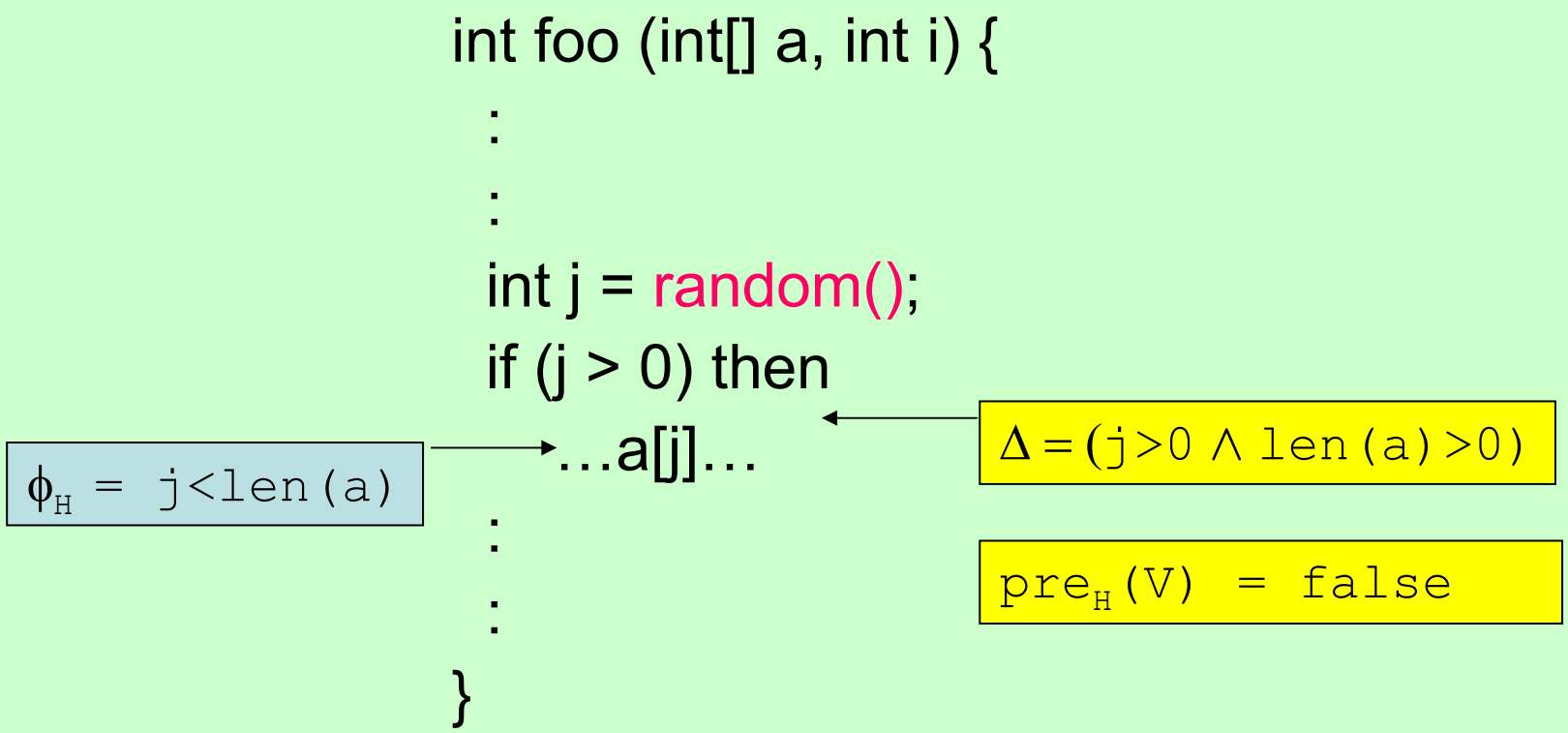
Partially-safe Check



Goal: Find the weakest **pre** such that: **pre** $\wedge \Delta \Rightarrow \phi$

$$pre(V) = \forall(L).\ \neg\ \Delta\ \vee\ \phi$$

Unsafe Check



There is no call context that can make ϕ totally-safe.
We call ϕ an **unsafe check**

3-way Check Classification

- Totally-safe
  - Run-time check is redundant, and can be eliminated.
- Partially-safe
  - Weakest precondition derivation is required.
- Unsafe
  - Run-time check is required.

Talk overview

- IMP – core language.
- Modular inference.
- Specialization.
- Experimental results.

IMP source language

- Input program:

  P ::= meth* prim*

  meth ::= t m (t_1 v_1, … t_n v_n) { e }

  e ::= c | v | v = e | t v = e_1; e_2

  | if v then e_1 else e_2 | m(v_1,…v_n)

- Base and array types:

  τ ::= Void | Int | Bool | Float

  t ::= τ | τ[Int,…,Int]

Sized Type

- Sized type associates size information to values in a type
- Types carry *size* annotations:
  - Int^{n}
    - Size of integer value k : $n = k$
  - Bool^{b}
    - Size of value True : $b = 1$; False : $b = 0$
  - Void, Float
  - $\text{Int}[\text{Int}^{a}]$
    - Size of array [1,1,1,1] : $a = 4$

Size Constraints

Δ, ϕ – Presburger constraints

$\Delta, \phi ::= \beta \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \exists n \cdot \phi \mid \forall n \cdot \phi$

$\beta ::= \text{True} \mid \text{False} \mid \alpha = \alpha \mid \alpha \leq \alpha$

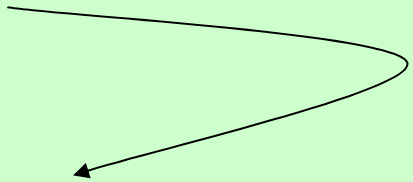
$\alpha ::= c \mid n \mid c * \alpha \mid \alpha + \alpha \mid - \alpha$

c – integer constant

n – size variable

Array Primitive Operations

prim ::= t m (t_1 v_1, … t_n v_n) where Δ,Φ



Postcondition, Preconditions

float[n] a = v

a[i] = v

a[i]

len(a)

Float[Int^a] **newArray** (Int^n n, Float v)
where (a=n),{S:n>0}

Void **assign** (Float[Int^a] a, Int^i i, Float v)
where ($0 \leq i < a$), {L: $0 \leq i$, H: i<a}

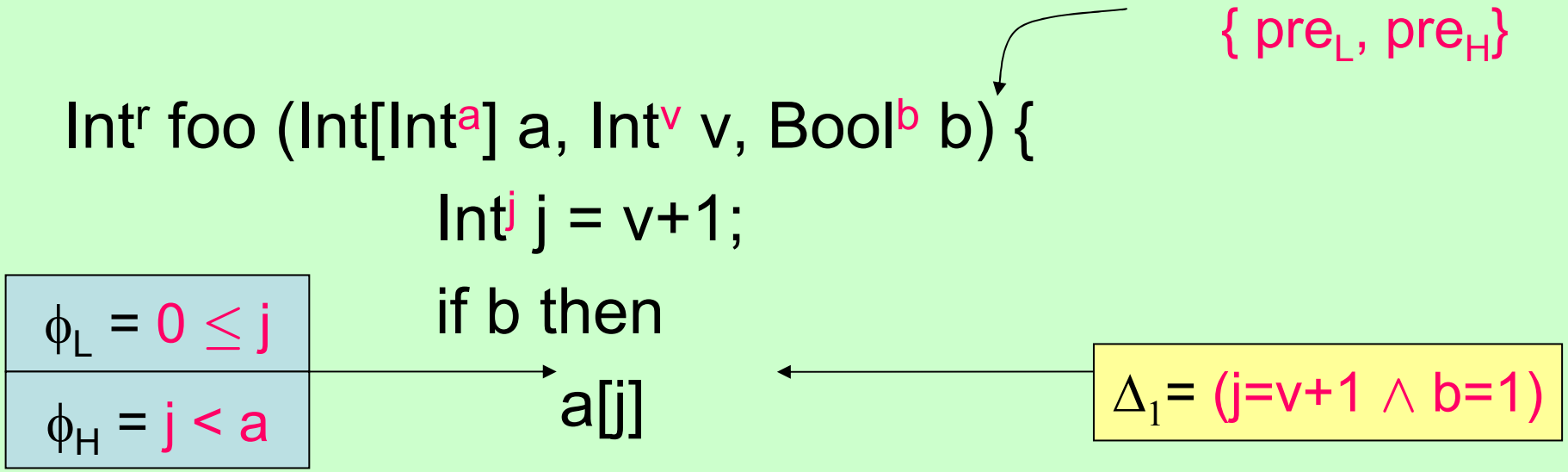
Float **sub** (Float[Int^a] a, Int^i i)
where ($0 \leq i < a$), {L: $0 \leq i$, H: i<a}

Int^r **len** (Float[Int^a] a) where (a=r),{ }

Modular Inference

- Compute for each method:
  - Preconditions
    - Derive minimal call context
  - Postcondition
    - Derive program states

Precondition Derivation

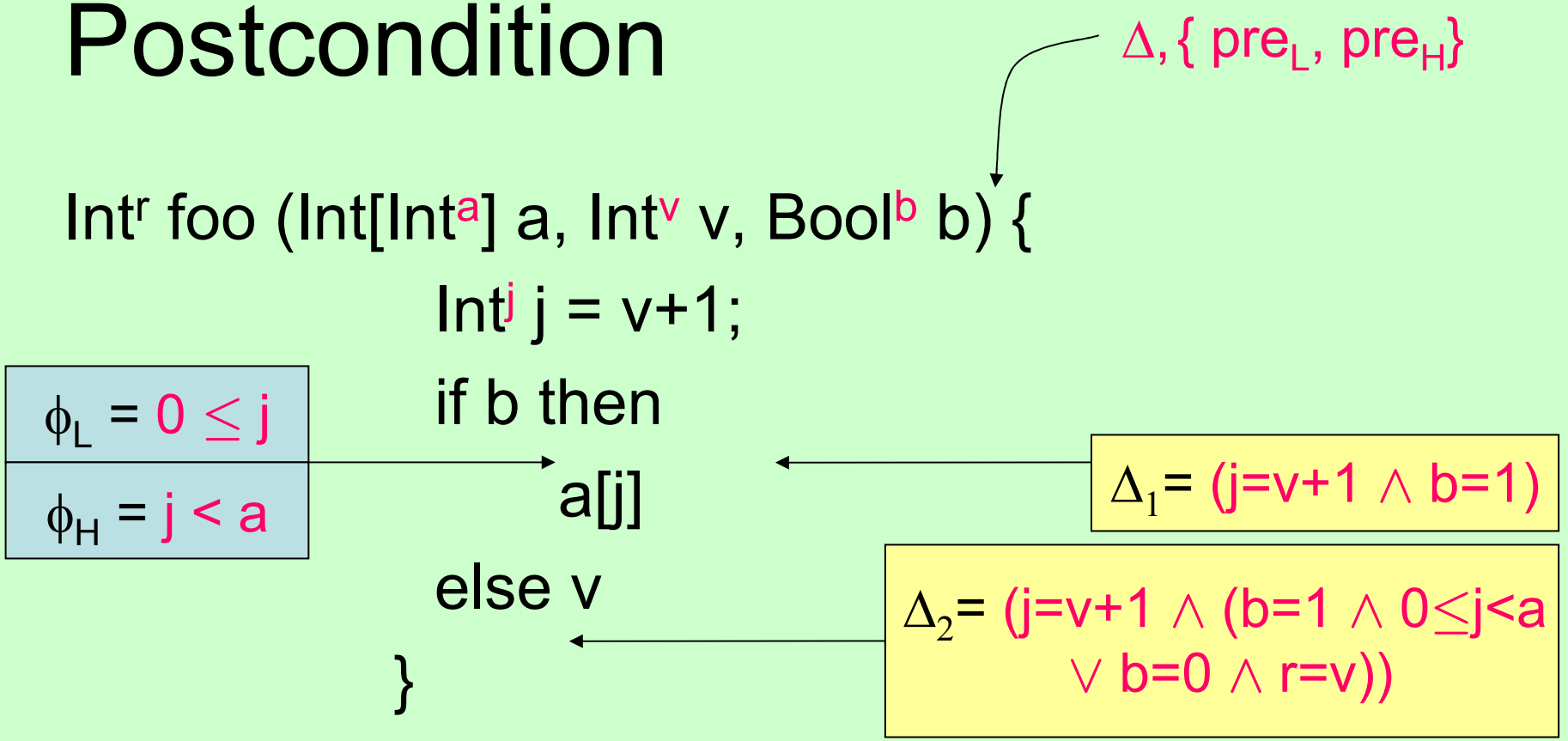


$\Delta_1 \Rightarrow \phi_L$? $\quad pre_L \wedge \Delta_1 \Rightarrow \phi_L$

$pre_L = \forall Local.(\neg\Delta_1 \vee \phi_L)$

$pre_L = (b=0 \vee 0 \leq v+1)$

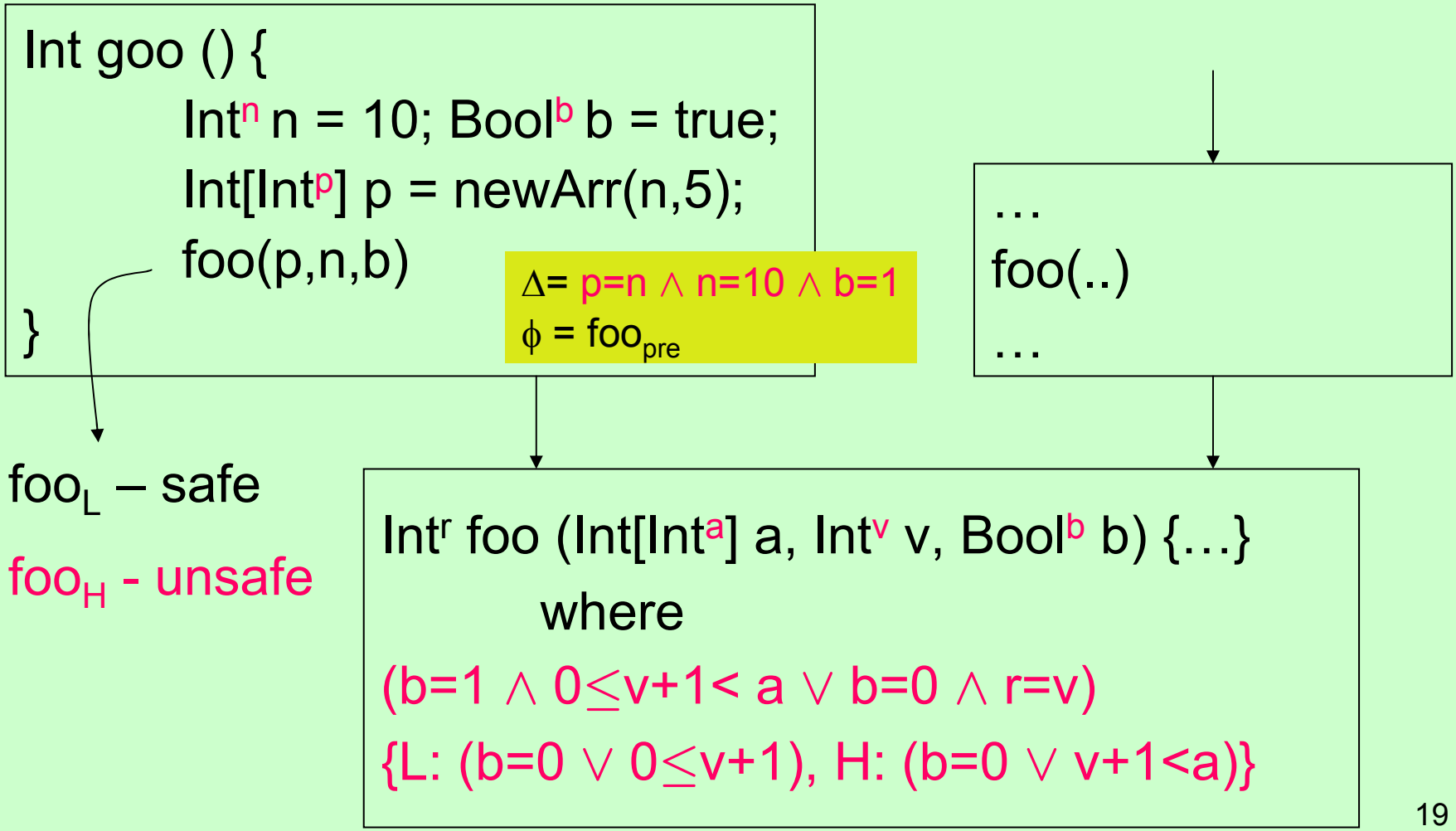
Postcondition



$\Delta = \exists \text{Local}.\ \Delta_2$

$\Delta = (b=1 \wedge 0 \leq v+1 < a \vee b=0 \wedge r=v)$

Precondition Propagation

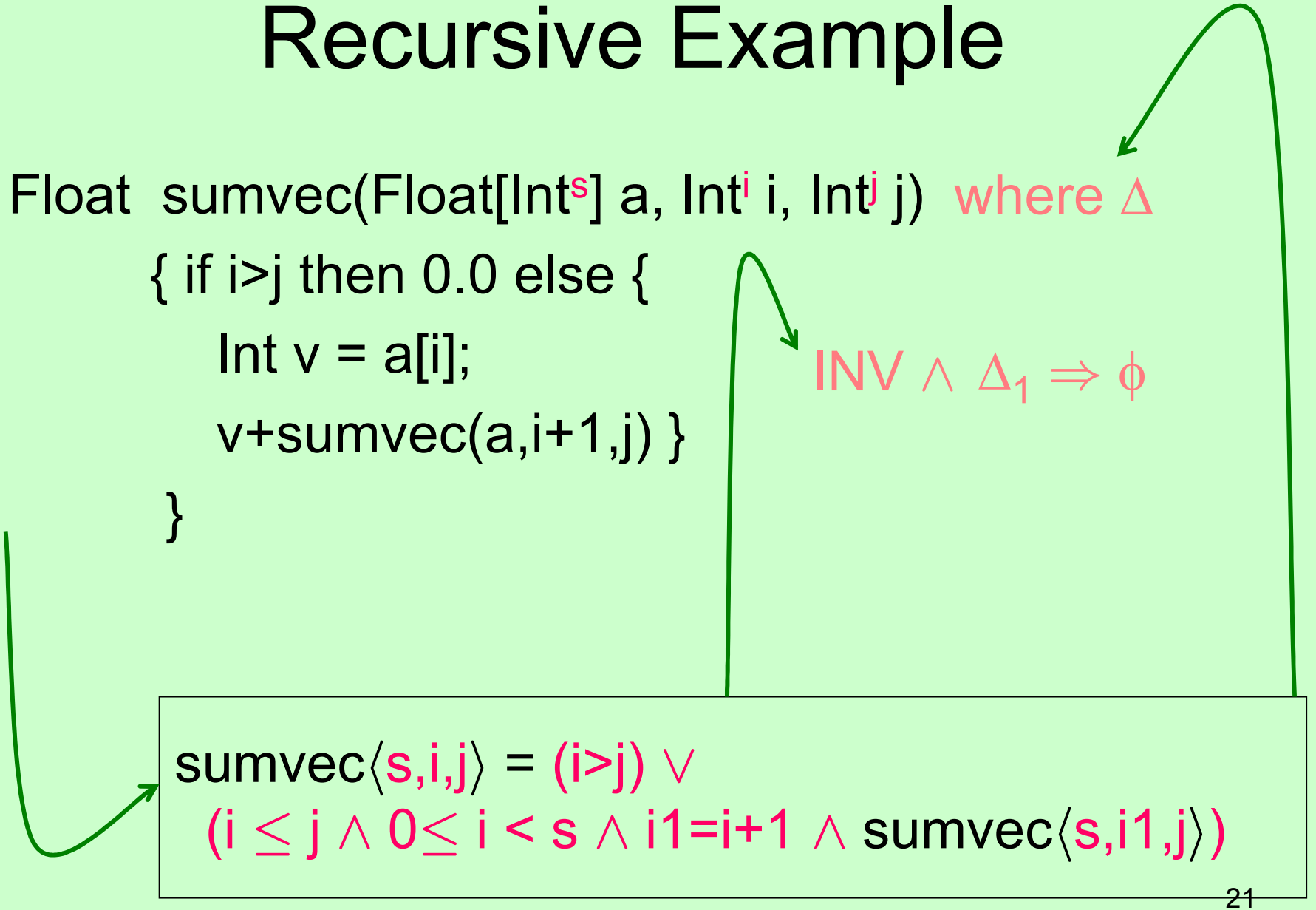


Recursive Methods

- Method body is abstracted to a recursive constraint.
- Two kinds of fixpoint computations:
  - postcondition.
  - recursive invariant.
    - represents the change of input arguments at nested recursive calls.
- The recursive invariant is used to determine safety of checks inside recursion.

Skip

Recursive Example



Recursive Example

```
Float  sumvec(Float[Int^s] a, Int^i i, Int^j j)
        { if i>j then 0.0 else {
            Int v = a[i];
            v+sumvec(a,i+1,j) }
         }
```

$\Delta = (i>j \vee 0 \leq i \leq j \wedge i<s)$

$\{\phi_L: (j<i \vee 0 \leq i),\ \phi_H: (j<s \vee \ldots)\}$

$\text{sumvec}\langle s,i,j\rangle = (i>j) \vee$
$(i \leq j \wedge 0 \leq i < s \wedge i1=i+1 \wedge \text{sumvec}\langle s,i1,j\rangle)$

Recursive Example (ii)

- Bottom-up fixpoint
  - start with base case:

  $sumvec_0\langle s,i,j\rangle = (i>j) \vee \mathit{false}$

  - iterate over:

  $sumvec_{i+1}\langle s,i,j\rangle = (i>j) \vee$
  $(i \leq j \wedge 0 \leq i < s \wedge i1=i+1 \wedge sumvec_i\langle s,i1,j\rangle)$

  - reach fixpoint:

  $sumvec_4 = (i>j) \vee (0 \leq i \leq j,s-1)$

Recursive Example (iii)

$\text{sumvec}\langle s,i,j\rangle = (i>j) \vee (i \leq j \wedge 0 \leq i < s \wedge i1=i+1 \wedge \text{sumvec}\langle s,i1,j\rangle)$

- Top-down fixpoint
  - start with one-step relation:

$I_1\langle s,i,j,s',i',j'\rangle = (i \leq j \wedge 0 \leq i < s \wedge s'=s \wedge i'=i+1 \wedge j'=j \wedge \mathit{true})$

  - iterate over:

$I_{i+1}\langle s,i,j,s',i',j'\rangle = I_i\langle s,i,j,s',i',j'\rangle \vee (\exists s1,i1,j1 \cdot I_i\langle s,i,j,s1,i1,j1\rangle \wedge I_1\langle s1,i1,j1,s',i',j'\rangle$

  - reach fixpoint:

$\text{sumvecl}_4\langle s,i,j,s',i',j'\rangle = (s'=s \wedge j'=j \wedge 0 \leq i < i' \leq s,j+1)$

Type Judgment

$$\Gamma; \Delta \vdash e :: t, \Delta_1, \Phi, \Upsilon$$

Γ – type-mapping between program variables (e) and size variables (Δ)

Δ (Δ_1) – pre (post-) state for e.

Φ – partially-safe preconditions to be classified at the caller site.

Υ – unsatisfiable checks.

Summary – Type System

- Able to capture imperative updates
- Flow, path, context sensitive $\rightarrow$ *precision*
- Allow partially-safe checks and compute corresponding preconditions

  pre = $\forall L.\ (\neg\Delta \vee \phi)$

How to Handle Unsafe Checks

Unsafe Checks

```
Int goo () {
  Int^n n = 10; Bool^b b = true;
  Int[Int^p] p = newArr(n,5);
  foo(p,n,b)
}
```

Possible reason?

- imprecise analysis
- real bug

foo_H – unsafe

Int^r foo ($Int[Int^a]$ a, Int^v v, $Bool^b$ b) {…}

where

$(b=1 \wedge 0 \leq v+1 < a \vee b=0 \wedge r=v)$

$\{L: (b=0 \vee 0 \leq v+1), H: (b=0 \vee v+1<a)\}$

Unsafe Checks

```
Int goo () {
  Int^n n = 10; Bool^b b = true;
  Int[Int^p] p = newArr(n,5);
  foo(p,n,b)
}
```

Solution 1: give type-error

Solution 2: insert run-time check

foo_H – unsafe

Int^r foo ($Int[Int^a]$ a, Int^v v, $Bool^b$ b) {…}

where

$(b=1 \wedge 0 \leq v+1 < a \vee b=0 \wedge r=v)$

$\{L: (b=0 \vee 0 \leq v+1), H: (b=0 \vee v+1<a)\}$

Unsafe Checks

```
Int goo () {
  Int^n n = 10; Bool^b b = true;
  Int[Int^p] p = newArr(n,5);
  if (..) foo(p,n,b) else error
}
```

foo_H – unsafe

Int^r foo (Int[Int^a] a, Int^v v, $Bool^b$ b) {…}

where

$(b=1 \wedge 0 \leq v+1 < a \vee b=0 \wedge r=v)$

$\{L: (b=0 \vee 0 \leq v+1), H: (b=0 \vee v+1<a)\}$

Solution 2a: specialize caller

- check motion
- what guard test?
- see: Output Constraint Specialization (*Khoo, Shi* – AsiaPEPM'02)

Unsafe Checks

```
Int goo () {
  Int n = 10; Bool b = true;
  Int[Int] A = newArr(n,5);
  foo(A,n,b)
}
```

Solution 2b: specialize callee

- guard the primitive operation.

foo_H – unsafe

```
Int^r foo(Int[Int^a] a, Int^v v, Bool^b b)
 {..
    if (j<len(a)) then sub(a,j) else error
 .. }
```

Specialization

- Monovariant specializer
  - Single optimized code for each method.
  - Uses the lower bound of all optimization.
  - Trades optimization for code space.
- Polyvariant specializer
  - Multiple optimized code for each method.
  - Each call site is replaced by its suitably optimized code.

Example

```
Int goo () {
  Int n = 10; Bool b = true;
  Int[Int] p = newArr(n,5);
  Int i = l2: foo(p,n,b);
  l3: foo(p,i,b)
}
```

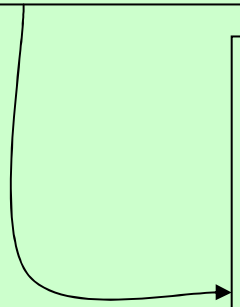
```
Int foo(Int[Int] a, Int v, Bool b) {
  Int j = v+1;
  if b then
    l1: a[j]
  else v
}
```

Pre-Conditions of `foo` computed			
ℓ_1.L	ℓ_1.H		
partially safe	partially safe		
Pre-Conditions of `goo` computed			
$\ell_2.\ell_1$.L	$\ell_2.\ell_1$.H	$\ell_3.\ell_1$.L	$\ell_3.\ell_1$.H
safe	unsafe	unsafe	unsafe

Monovariant Specialization

```
Int goo () {
  Int n = 10; Bool b = true;
  Int[Int] p = newArr(n,5);
  Int i = foo1(p,n,b);
  foo1(p,i,b)
}
```

```
Int foo(Int[Int] a, Int v, Bool b) {
  Int j = v+1;
  if b then
        a[j]
  else v
}
```



```
Int foo1(Int[Int] a, Int v, Bool b)
{.. if b then
    if (0 ≤ j) then
      if (j < len(a)) then sub(a,j) else error
    else error
    else v }
```

Polyvariant Specialization

```
Int goo () {
  Int n = 10; Bool b = true;
  Int[Int] p = newArr(n,5);
  Int i = foo1(p,n,b);
  foo2(p,i,b)
}
```

```
Int foo1(Int[Int] a, Int v, Bool b) {
{.. if b then
      if (j < len(a)) then a[j]
      else error
    else v }
}
```

```
Int foo2(Int[Int] a, Int v, Bool b)
{.. if b then
    if (0 ≤ j) then
      if (j < len(a)) then a[j] else error
    else error
    else v }
```

Soundness

- Inference + specialization =
  Well-typed program

If a method is *well-typed*, the execution of its body never incurs *invalid array-accesses*.

Experiments

- Prototype built using:
  - Haskell (GHC)
  - Omega constraint solving library (*Pugh et al*)
  - interface C-Haskell
- Test programs:
  - Complex recursion
  - Scientific benchmarks
- The method is viable
  - Sufficiently precise to eliminate checks.
  - Sufficiently fast to be usable.

Experimental Results

Benchmark Programs	Source (lines)	Static Checks	Checking (secs)	Inference (secs)	Dynamic Checks Eliminated
binary search	50	2	0.17	1.81	100%
bubble sort	59	12	0.43	1.51	100%
hanoi tower	38	16	3.73	11.47	100%
merge sort	80	16	7.70	13.07	100%
queen	45	8	0.52	2.10	100%
quick sort	67	12	0.38	1.76	100%
sentinel	26	4	0.05	0.16	50%
sparse multiply	46	12	3.27	7.09	100%
sumvec	33	2	0.11	0.47	100%
FFT	336	62	9.58	28.74	100%
LU Decomp.	191	80	13.10	72.91	100%
SOR	84	24	1.15	3.8	100%
Linpack	903	166	42.26	162.2	100%

Conclusion

- Imperative sized type inference
- Modular analysis:
  - 3-way check classification
  - precondition propagation
  - specialization to insert runtime tests
- Implementation
  - Web interface

    http://loris-4.ddns.comp.nus.edu.sg/~popeeaco