Automatic Parallelization Techniques and the Cetus Source-to-Source Compiler Infrastructure

Cetus version 1.2

Rudi Eigenmann, Sam Midkiff

1:30 – 3:00 Introduction to parallelization: Analyses and transformations, their use in Cetus, IR traversal, symbol table interface

3:30 – 5:30 Working on IR (expressions, statements, annotations), data dependence interface

What is Cetus?

- Source-to-source C compiler written in Java and maintained at Purdue University
- Provides an internal C parser (Antlr, 12K + lines)
- Intermediate representations (22K+ lines)
- Compiler passes (32K+ lines and growing)
- Supported by the National Science Foundation
- Drivers of the Cetus effort:
  - automatic parallelization
  - many other applications have emerged

Why Cetus?

- Wanted source-to-source C translator, because
  - Parallelization techniques tend to be source transformations
  - Allows comparison of original and transformed code
  - Cetus use as an instrumentation tool
- Alternatives did not fit
  - Predecessor (Polaris) only works on Fortran77
  - GCC has no source-to-source capability
  - SUIF is for C; last major update in 2001
  - Open64's 5 IRs are more complex than we needed
- Wanted a compiler written in modern language
  - Polaris and SUIF use old dialects of C++
  - Portability is important for user community
- Best alternative was to write our own

Download and License

- Cetus.ecn.purdue.edu
- Modified Artistic License

  Key points:

  – Open source with most of the common rules

  – OK to distribute modifications, if you make them public and let us know

  – Redistribution of Cetus must be free

What must a parallelizing compiler do?

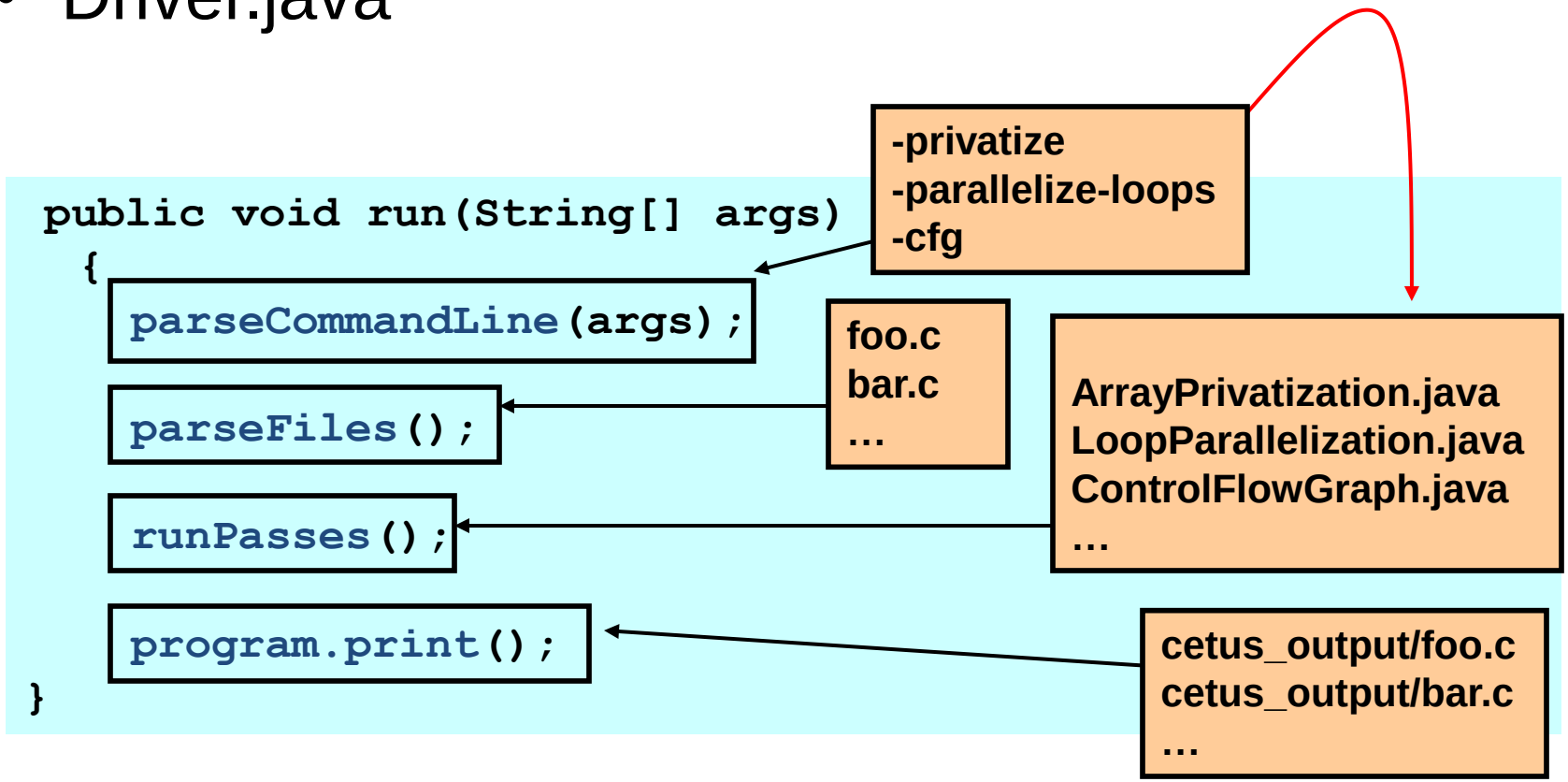
1. Determine parts of the program that can execute independently of one another
2. Transform dependent parts of a program so that they are independent
3. Perform new optimizations to efficient parallel execution
4. Perform traditional optimizations to enable good performance on each node
5. Generate parallel code

Roadmap for the rest of the tutorial

- High level control of compilation and using standard functionality in Cetus
- Existing analyses and annotations in Cetus – what they do and how to invoke them
  - Largely example-driven
  - *Will give an overview of many important compiler transformations*
- Cetus internals
  - Intermediate representation of programs (IR) within Cetus
  - How to traverse and manipulate the IR
  - Invoking analyses within Cetus
- Using the IR to write new transformations

Cetus Driver exercises high-level control

- Driver.java



Example of "built-in" functionality

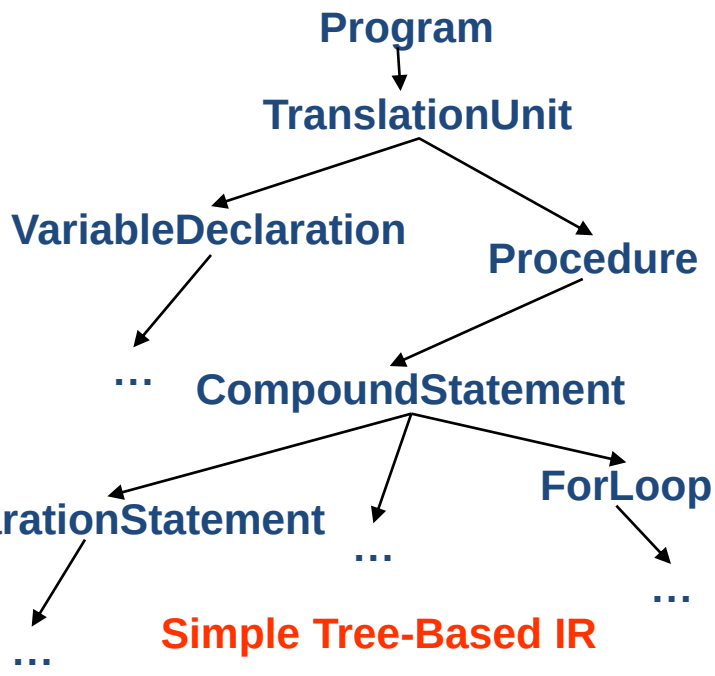
1. Cetus source-to-source translation – "parse and print"

Input

```
int temp;
int main(void) {
        int i,j,c,a[100];
        c = 2;
        for (i=0; i<100; i++) {
                a[i] = c*a[i]+(a[i]-1);
        }
}
```

Output

```
int temp;
int
main (void)
{
  int i, j, c, a[100];
  c = 2;
  for (i = 0; i < 100; i++)
    {
      a[i] = ((c * a[i]) + (a[i] - 1));
    }
}
```



Simple Tree-Based IR

As closely associated with original program structure as possible for regeneration of source code

Automatic parallelization – another "built-in" capability

Input

```
int foo(void)
{
  int i;
  double t, s, a[100];
  for ( i=0; i<50; ++i )
  {
    t = a[i];
    a[i+50] = t +
         (a[i]+a[i+50])/2.0;
    s = s + 2*a[i];
  }
  return 0;
}
```

```
$ cetus -parallelize-loops foo.c
```

Output

```
int foo(void )
{
  int i;
  double t, s, a[100];
  #pragma cetus private(i, t)
  #pragma cetus reduction(+: s)
  #pragma cetus parallel
  #pragma omp parallel for reduction(+: s)
                           private(i, t)
  for (i=0; i<50;  ++ i)
  {
    t=a[i];
    a[(i+50)]=(t+((a[i]+a[(i+50)])/2.0));
    s=(s+(2*a[i]));
  }
  return 0;
}
```

Cetus Passes for Program Analysis and Transformation

Analysis Passes

- Data dependence analysis
  - Banerjee-Wolfe Test
  - Range Test
- Pointer alias analysis
  - Currently: very simple flow/context-insensitive alias map
- Symbolic range analysis
  - Generates a map from variables to their valid symbolic bounds at each program point
- Array privatization
  - Computes privatizable scalars and arrays

Analysis Passes (cont.)

- Reduction recognition
  - Detects reduction operations
- Symbolic expression manipulators
  - Normalizes and simplifies expressions
  - Examples
    - `1+2*a+4-a` → `5+a` (folding)
    - `a*(b+c)` → `a*b+a*c` (distribution)
    - `(a*2)/(8*c)` → `a/(4*c)` (division)
- Call Graph and CFG generators
  - CFG provided either at basic-block level or at statement level
- Basic use/def set computation for both scalars and array sections

Transformation Passes

- Induction variable substitution pass
  - Identifies and substitutes induction variables
- Loop parallelizer
  - Depends on induction variable substitution, reduction recognition, and array privatization
  - Performs loop dependence analysis
  - Generates “parallel loop” annotations
- Program normalizers
  - Single call, single declarator, singe return normalizers
- Loop outliner
  - Extracts loops out into separate subroutines

Data Dependence Analysis

- Loops are the primary source of parallelism in scientific and engineering applications.
- Compilers detect loops that have independent iterations, i.e. iterations access disjoint data

```
for (i=1; i<n; i++) {
    A[expression1] = …
      … = A[expression2]
}
```

Necessary condition for this loop to be parallel:

expression1 in any iteration i' is different from *expression2* in any other iteration i''

Cetus includes the Banerjee-Wolfe DD test (v1.0) and the Range test (v1.1)

Motivating examples

Is it legal to run the loop in parallel?

```
Do i = 1, n
   a(i) = b(i)      S1
   c(i) = a(i-1)    S2
End do
```

To answer these questions it is necessary to determine

1. *If two references access the same memory location*
2. *The order of the accesses at runtime*

This allows us to determine if there are dependences and, if so, if a transformation violates those dependences

An example of dependences

Do i = 1, n
  a(i) = b(i) S_1
  c(i) = a(i-1) S_2
End do

→ Indicates dependences, i.e. the statement at the head of the arc is somehow dependent on the statement at the tail

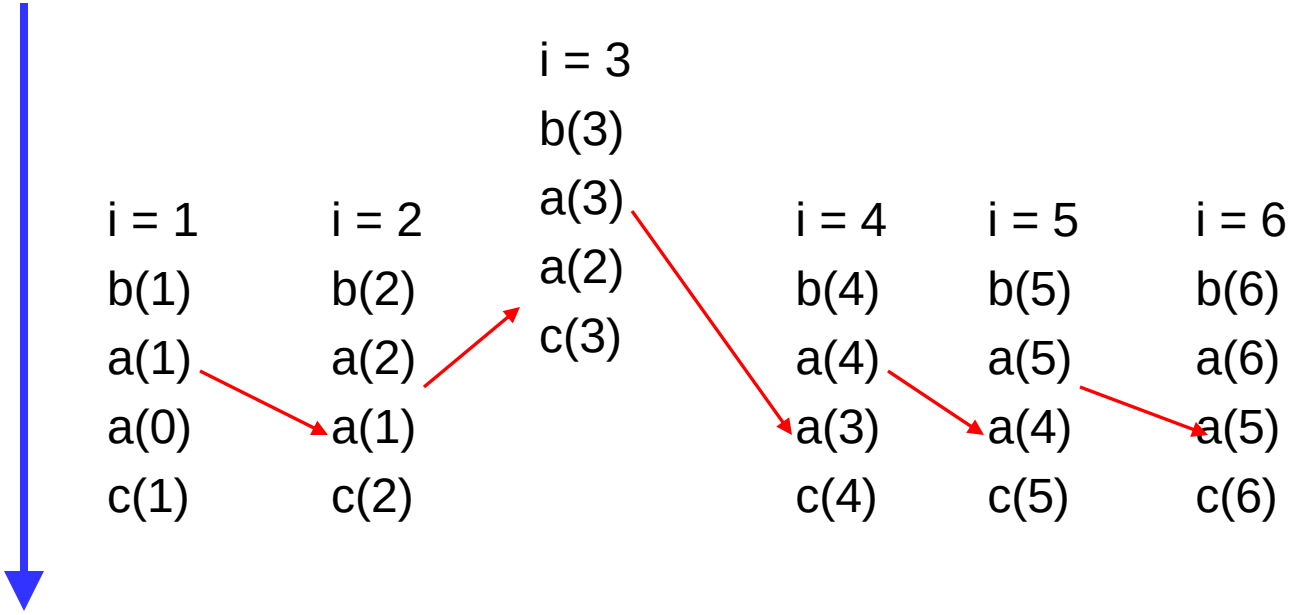
i = 1	i = 2	i = 3	i = 4	i = 5	i = 6
b(1)	b(2)	b(3)	b(4)	b(5)	b(6)
a(1)	a(2)	a(3)	a(4)	a(5)	a(6)
a(0)	a(1)	a(2)	a(3)	a(4)	a(5)
c(1)	c(2)	c(3)	c(4)	c(5)	c(6)

Can we run the loop in parallel?

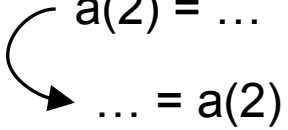
```
Do i = 1, n
  a(i) = b(i)      S1
  c(i) = a(i-1)    S2
End do
```

Assume 1 iteration per processor, then if for some reason some iterations execute out of lock-step, bad things can happen

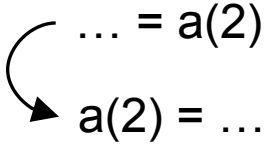
In this case, read of a(2) in i=3 will get an invalid value!



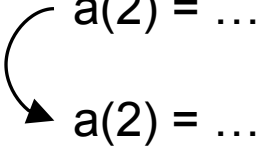
Types of dependence



Flow or true dependence – data for a read comes from a previous write (write/read hazard in hardware terms_



Anti-dependence – write to a location cannot occur before a previous read is finished



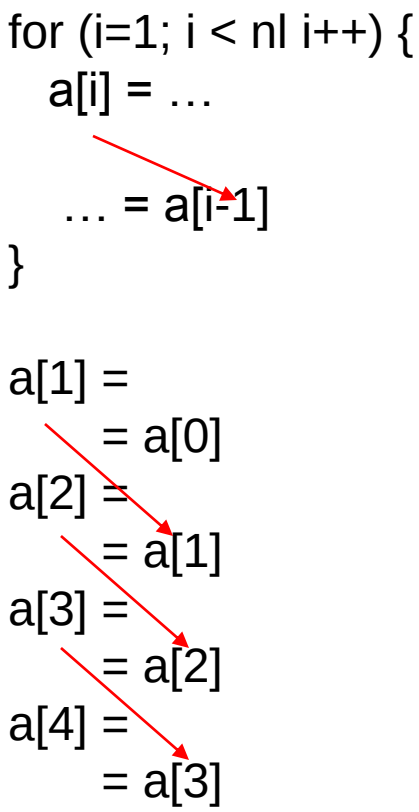
Output dependence – write a location must wait for a previous write to finish

Dependences always go from earlier in a program execution to later in the execution

Anti and output dependences can be eliminated by using more storage.

Dependence sources and sinks

- The *sink* of a dependence is the statement at the head of the dependence arrow
- The *source* is the statement at the tail of the dependence arrow



Data Dependence Tests: Concepts

Terms for data dependences between statements of loop iterations.

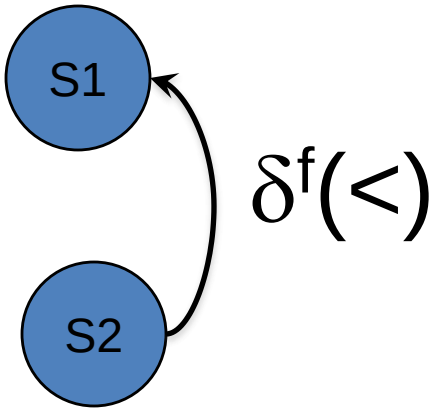
- Distance (vector): indicates how many iterations apart are source and sink of dependence.
- Direction (vector): is basically the sign of the distance. There are different notations: (<,=,>) or (+1,0,-1) meaning dependence (from earlier to later, within the same, from later to earlier) iteration.
- Loop-carried (or cross-iteration) dependence and non-loop-carried (or loop-independent) dependence: indicates whether or not a dependence exists within one iteration or across iterations.
  - For detecting parallel loops, only cross-iteration dependences matter.
  - *equal* dependences are relevant for optimizations such as statement reordering and loop distribution.
- Iteration space graphs: the un-abstracted form of a dependence graph with one node per statement instance.

Loop parallelization

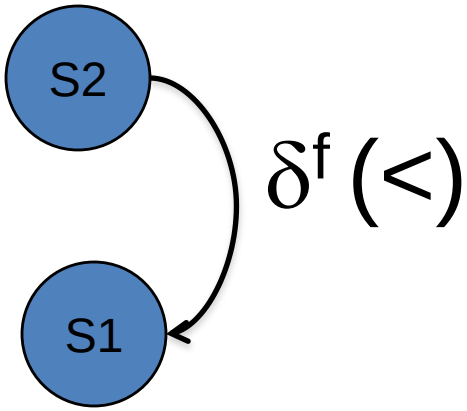
- If there are no *cross-iteration* dependences, can parallelize the loop
  - No data flow from iteration to iteration, so iterations can execute independently, and in parallel
- Most loops cannot be parallelized, need to eliminate dependences where possible.

Dependence graphs & vectorization

```
DO j = 1,n
   S1: c(j) = a(j)
   S2: a(j-1) = b(j)
ENDDO
```




```
DO j = 1,n
   S2: a(j) = b(j)
ENDDO

DO k=1,n
  S1: c(k) = a(k-1)
ENDDO
```



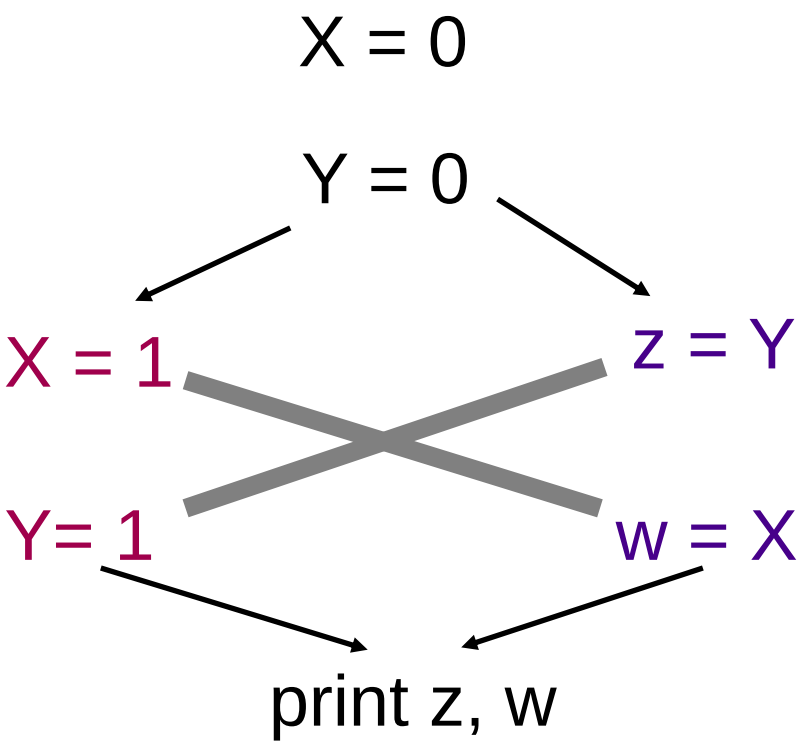
Problems with explicit parallelism

- Normal dependence analysis techniques may need to be modified when analyzing explicitly parallel programs
- Depends on the consistency model
  - Relaxed models are easier to compile
  - Sequential consistency is harder

Consider the Java Memory Model

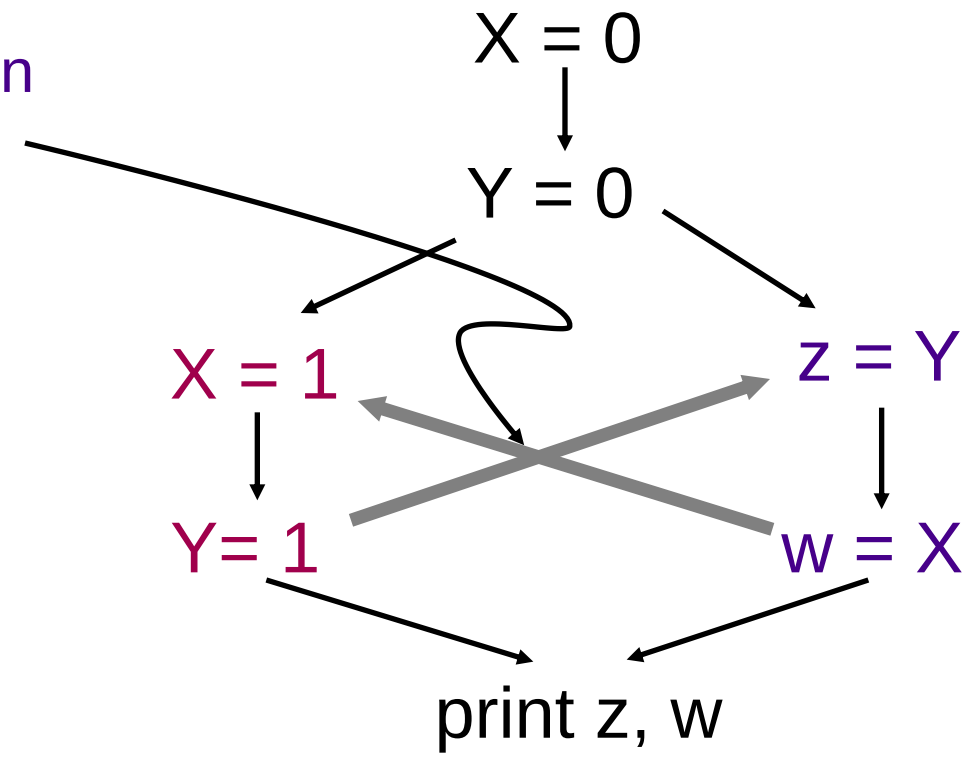
Question: Is it legal for ***z == 1*** and ***w == 0***?

Answer: YES



Sequential Consistency

Relative execution order implied by assigned value



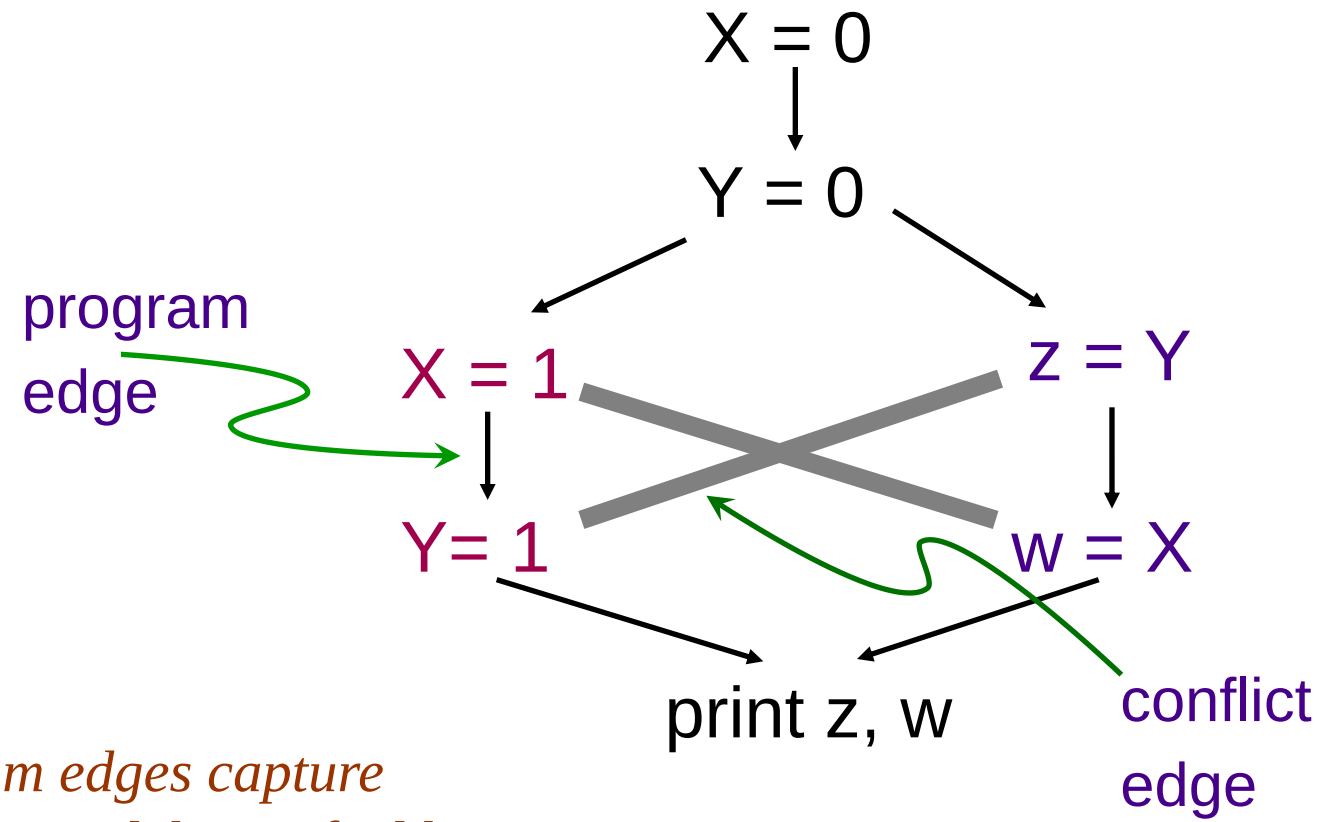
Question: Is it legal for ***z == 1*** and ***w == 0***?

Answer: ***NO***

Delay set analysis

Shasha and Snir, TOPLAS 1988

What program edges need to be enforced for the execution to give the appearance that all are enforced?

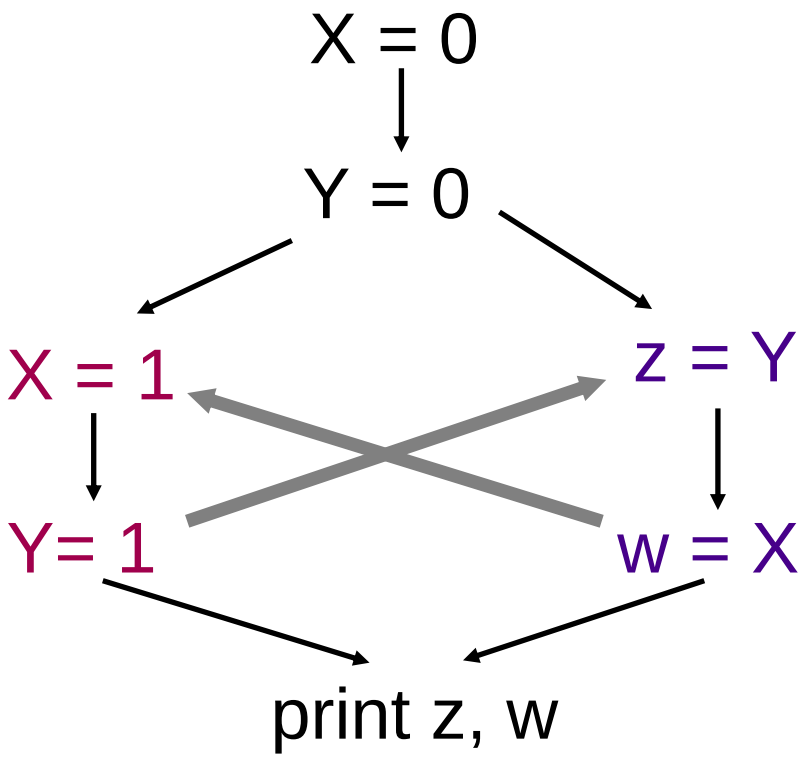


Program edges capture memory model specified by program

Delay set analysis

These two orientations of the conflict edges exist in the illegal execution

(z = 1, w = 0)

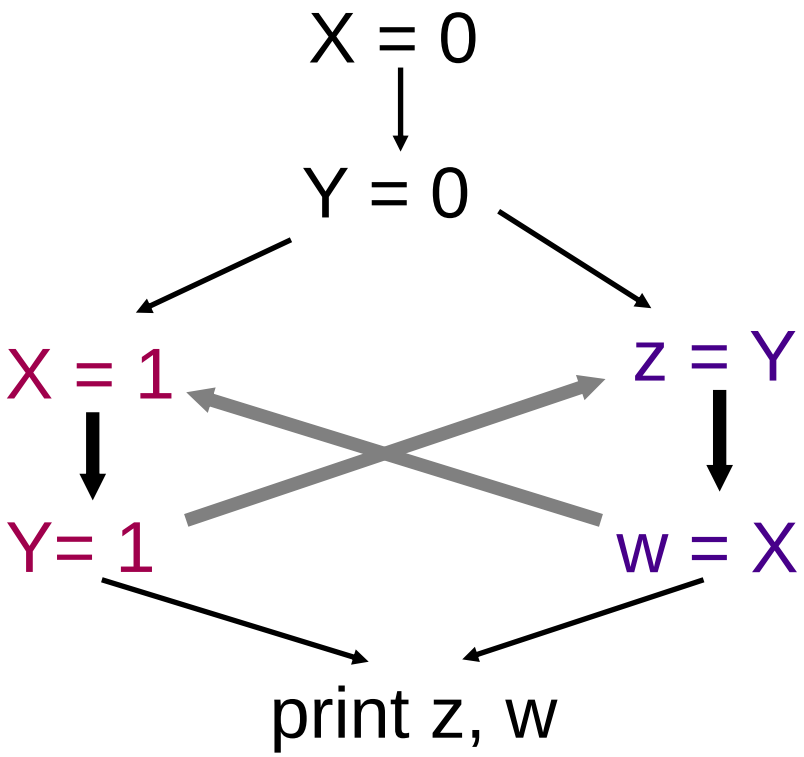


Delay set analysis

Program edges requiring enforcement are embedded in *minimal, mixed cycles*

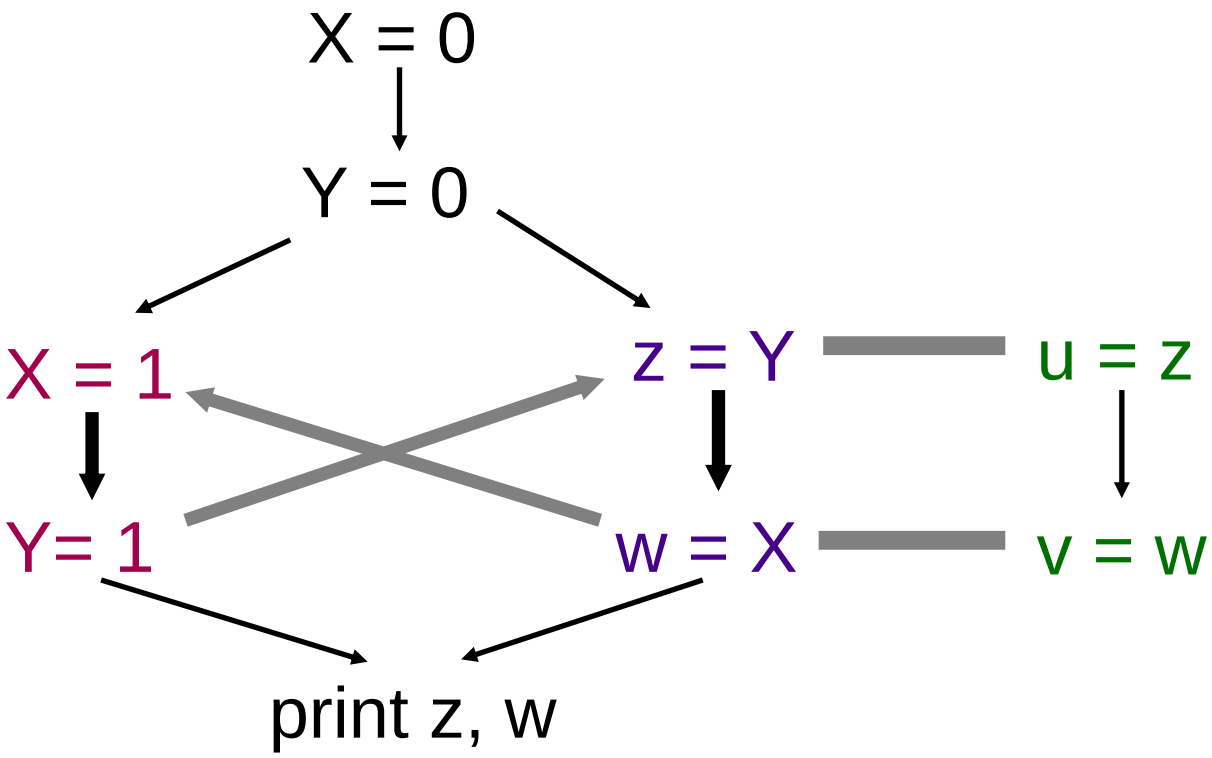
Minimal – contains no smaller cycle that spans two or more threads

Mixed – has program and conflict edges



Delay set analysis

Program edges needing to be enforced are embedded in minimal, mixed cycles



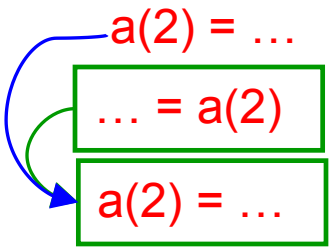
So, the problem is solved …

- Not really – Yelick & Krishnamurthy (JPDC '96, LCPC '03) showed that finding the cycles is exponential
- Has a solution that targets SPMD programs
  - Represent program as a P_{SPMD} graph
  - Fold all processes/threads into two "threads"
  - Complexity is v^n, $n=2$, which is quadratic
  - Uses synchronization information to improves solution
- Sura, et al. (PPoPP '05) have non-SPMD solution that finds paths, assumes a cycle exists

Eliminating dependences

- In theory, only flow or true dependences necessary
- Privatization (e.g., using array or scalar expansion) can be used to remove anti and output dependences
- In practice, it can be complicated to figure out how to create the new storage, and new storage is not free

Eliminating anti and output dependence



Anti-dependence – write to a location cannot occur before a previous read is finished

The original program is:

a(2) = …
… = a(2)
a(2) = …
= … a(2)

Create additional storage to eliminate the *anti* and *output* dependence

The new program is:

$a_0(2)$ = …
… = $a_0(2)$
$a_1(2)$ = …
= … $a_1(2)$

No more anti-dependence!

Eliminating dependences

- In theory, only flow or true dependences are necessary
- Privatization (e.g., using array or scalar expansion) can be used to remove anti and output dependences
- In practice, it can be complicated to figure out how to create the new storage, and new storage is not free

Symbolic Range Analysis

- Computes symbolic ranges for integer variables
  - Performs fine-grain value-flow analysis
  - Returns map from statements to sets of ranges
  - Ranges are valid before each statement
- Results are used for symbolic value comparison
  - Comparison under a given set of ranges
  - Useful for compile-time symbolic analysis
- Limitations
  - Conservative handling of side effects
  - Does not model integer overflow

Array Data Flow Analysis Using Symbolic Range Analysis

```
int foo(…)
{
 …
 for (tstep=1; tstep<=NSTEP; tstep++)
 {
  for (i=1; i<N-1; i++)
    for (j=1; j<M-1; j++)
    { /* range:{ 1<=i<=(N-2), 1<=j<=(M-2), 1<=tstep<=NSTEP} */
      A[i][j] = (B[i][j-1]+B[i][j+1]+B[i-1][j]+B[i+1][j])/4.0;
    }

  for (i=1; i<N-1; i++)
    for (j=1; j<M-1; j++)
    { /* range:{ 1<=i<=(N-2), 1<=j<=(M-2), 1<=tstep<=NSTEP} */
      B[i][j] = A[i][j];
    }
 }
}
```

DEF: A[1:N-2][1:M-2]
USE: B[0:N-1][0:M-1]

DEF: B[1:N-2][1:M-2]
USE: A[1:N-2][1:M-2]

Symbolic Range Analysis

```
   int foo(…)
   {
    …
    for (i=0; i<n; i++) {
S1   if ( i>0 ) {
S2    a[i] = …
     } else {
S3    a[i] = …
     }
    }
S4  a[i] = …
    …
   }
```

RangeMap(foo)

```
    …
S1  {0<=i<=n-1, n>=i+1}
S2  {1<=i<=n-1, n>=i+1}
S3  {i=0, n>=i+1}
S4  {i>=0, n<=i}
    …
```

compare(S4,n,i+1) → n < i+1

compare(S3,i,10) → i < 10

compare(S3,n,0) → n > 0

Symbolic Expression Manipulators

- Cetus provides expression normalizers
  - Simplifies a given expression
    - `1+2*a+4-a` → `5+a` (folding)
    - `a*(b+c)` → `a*b+a*c` (distribution)
    - `(a*2)/(8*c)` → `a/(4*c)` (division)
  - Symbolic operations
    - add(), subtract(), multiply(), divide(), …
- Where is this used?
  - Passes can operate on expressions in standard form
  - Reduces corner cases in analysis passes
  - Array subscript normalization for DD test
- See cetus.hir.Symbolic for more features

Array Privatization

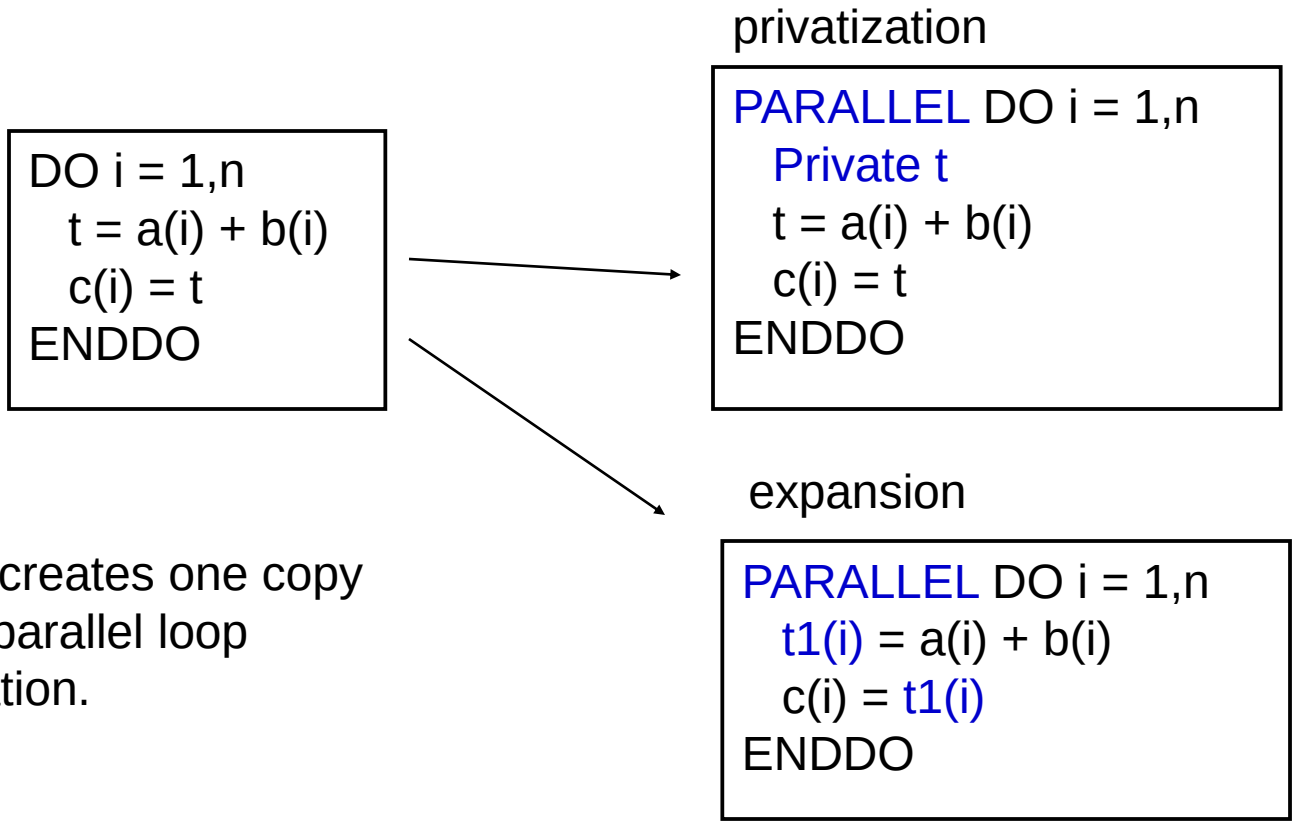
The Concept

```
#pragma omp parallel for private(work)
for (i=1; i<n; i++){
   work[1:n] = …
   … = …
   … = work[1:n]
}
```

Dependency: Elements of **`work`** read in iteration **`i`** were also written in iteration **`i'`**

Solution: Each processor is given a separate version of the data, so there is no sharing conflict

Scalar Expansion/Privatization



Private creates one copy per parallel loop iteration.

Analysis and Transformation for Scalar Expansion/Privatization

Loop Analysis:

- find variables that are used in the loop body but dead on entry. i.e., the variables are written (on all paths) through the loop before being used.
- determine if the variables are live out of the loop (make sure the variable is defined in the last loop iteration).

Transformation (variable *t*)

- Privatization:
  - put *t* on private list. Mark as *last-value* if necessary.
- Expansion:
  - declare an array *t0(n),* with *n=#loop_iterations.*
  - replace all occurrences of *t* in the loop body with *t0(i),* where *i* is the loop variable.
  - live-out variables: create the assignment *t=t0(n)* after the loop.

Array Privatization Algorithm

L: Loop, LIVE: Live-out variables
EXIT : Loop exits
PRED: Predecessors
KILL: Killed set due to modified variables
in the section representation
DEF : Defined set
USE: Used set
UEU : Upward-exposed uses
PRI: Private variables
LPRI: Live-out private variables

```
procedure Privatization(L, LIVE)
  // input : L, LIVE
  // output : DEF[L], UEU[L], PRI[L]
  // side effect : L is annotated with PRI[L] and LPRI[L]
  // 1. Privatize inner loops
  foreach direct inner loop l in L
    (DEF[l], USE[l], PRI[l]) = Privatization(l, LIVE)
  // 2. Create CFG of loop body w. collapsed inner loops
  G(N, E) = BuildCFG(L)
  // 3. Compute must-defined set DEF prior to each node
  Iteratively solve data flow equation DEF for node n ∈ N .
    DEFin[n] = ∩ p∈P RED[n] DEFout[p]
    DEFout[n] = (DEFin[n] − KILL[n]) ∪ DEF[n]
  // 4. Compute DEF[L], UEU[L], PRI[L]
  DEF[L] = ∩ n∈EXIT[L] DEFout[n]
  UEU[L] = ∪ n∈N (USE[n] − DEFin[n])
  PRI[L] = CollectCandidates(L, DEF, PRI)
  PRI[L] = PRI[L] − UEU[L]
  LPRI[L] = PRI[L] ∩ LIVE[L]
  PRI[L] = PRI[L] − LPRI[L]
  AnnotatePrivate(L, PRI[L], LPRI[L])
  // 5. Aggregate DEF and USE over the iteration space
  DEF[L] = AggregateDEF(DEF[L])
  UEU[L] = AggregateUSE(UEU[L])
  return (DEF[L], UEU[L], PRI[L])
end procedure
```

Reduction Parallelization

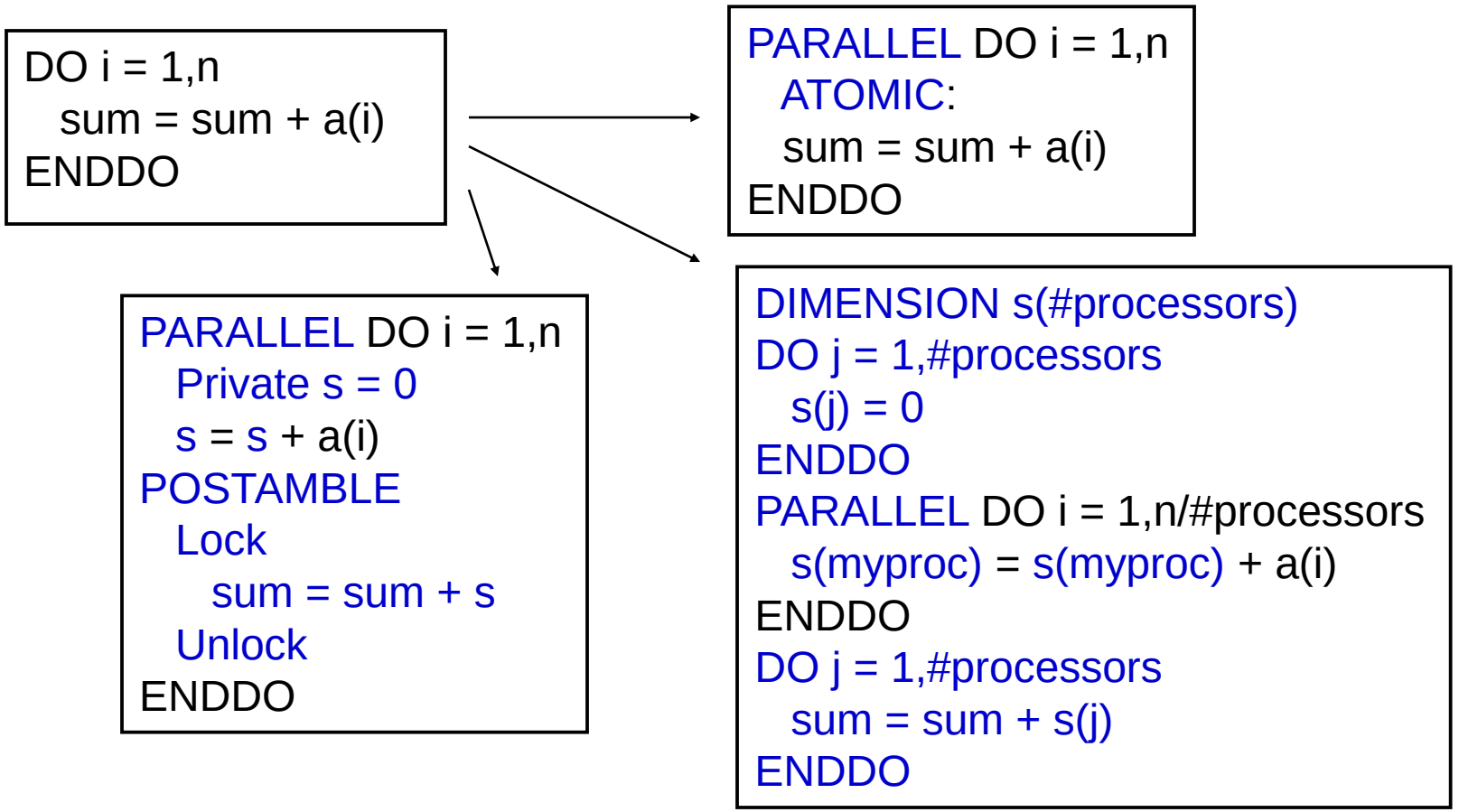
The Concept

```
#pragma omp parallel for reduction(+:sum)
for (i=1; i<n; i++){
   …
   sum = sum + A[i]
   …
}
```

Dependency: Value of sum written in iteration **`i`** is read in iteration **`i'`**

Solution: Each processor will accumulate partial sums, followed by combining these parts at the end of the loop

Parallelization of Reduction Operations



Analysis and Transformation for (Sum) Reduction Operations

- Loop Analysis:
  - find reduction statements of the form *s* = *s* + *expr* where expr does not use *s*.
  - discard *s* as a reduction variable if it is used in non-reduction statements.
- Transformation:

  (as shown on previous slide)
  - create private or expanded variable and replace all occurrences of reduction variable in loop body.
  - update original variable with sum over all partial sums, using a *critical section* in a loop *postamble* or a summation after the loop, respectively.

Reduction Recognition Algorithm

```
procedure RecognizeReductions(L)
  Input : Loop L
  Side-effect : adds reduction annotations for L in the IR
  REDUCTION = {} // set of candidate reduction expressions
  REF = {} // set of non-reduction variables referenced in L
  foreach expr in L
    localREFs = findREF(expr)
    if (expr is AssignmentExpression)
      candidate = lhse(expr)
      increment = rhse(expr) – candidate
      if ( !(getSymbol(candidate) in findREF(increment)) )
        // criterion1 is satisfied
        REDUCTION = REDUCTION ∪ candidate
        localREFs = findREF(increment)
    REF = REF ∪ localREFs // collect non-reduction references for criterion 2
  foreach expr in REDUCTION
    if ( ! (getSymbol(expr) in REF) )
      // criterion 2 is satisfied
      if (expr is ArrayAccess AND expr.subscript is loop-variant)
        CreateAnnotation(sum-reduction, ARRAY, expr)
      else
        CreateAnnotation(sum-reduction, SCALAR, expr)
end procedure
```

lhse/rhse extracts the left/right-hand side expression of an assignment expr. (note, assignments are referred to as expressions rather than statements)

findREF(X) returns the set of USE/DEF references in a given expression X.

"–" is a symbolic subtraction operator.

getSymbol(X) returns the base symbol of expression X (a scalar or an array)

criterion 1: the loop contains one or several assignment expressions of the form sum=sum+increment, where increment is typically a real-valued, loop-variant expression.

criterion 2: sum does not appear anywhere else in the loop.

Induction Variable Substitution

The Concept

```
ind = k
for (i=1; i<n; i++){
   ind  = ind + 2
   A(ind) = B(i)
}
```

simple induction variables →

```
ind = k
for (i=1; i<n; i++){
   A(k+2*i) = B(i)
}
```

```
for (j=1; j<n; j++){
   ind  = ind + j
   A(ind) = B(i)
}
```

generalized induction variables →

```
for (j=1; j<n; j++){
   A(k+(j**2+j)/2) = B(i)
}
```

Induction Variable Substitution Algorithm

```
procedure SubstituteInductionVariable(iv, L, LIVE)
  input : iv, // induction variable to be substituted
          L // loop to be processed
          LIVE // set of variables that are live-out of L
  side effect : iv is substituted in IR
  inc = FindIncrement(L0) // L0 is the outermost loop of the nest
  Replace(L0 , iv)
  if (iv ∈ LIVE) InsertStatement("iv = inc") // at the end of L
end procedure
```

```
procedure FindIncrement(L)
  // Find the increments incurred by iv from the beginning of L:
  // inc_after[s] is the increment from beginning of loop body
  //     to statement s
  // - inc_into_loop[L] is the increment from beginning of L to beginning
  //   oj  j th iteration (j counts L's iterations from 1 to ub, in steps of 1)
  // - the subroutine returns the total increment added by L
  inc = 0
  foreach statement stmt in L of type Ind, Loop
    if stmt has type Loop
      inc += FindIncrement(stmt)
    else // statement has the form iv = iv + exp
      inc += exp
    inc_after[stmt] = inc
    inc_into_loop[L] = Σ_1^(j-1) (inc) // inc may depend on j
    return Σ_1^ub (inc)
end procedure
```

```
procedure Replace(L, initialval)
  // Substitute v with the closed-form expression
  val = initialval + inc_into_loop[L]
  foreach statement stmt in L of type Ind, Loop, Use
    if stmt has type Loop
      Replace(stmt, val)
    if stmt has type Loop, Ind
      val += inc_after[stmt]
    if stmt has type Use
      Substitute(stmt, iv, val)
        // replace occurrences of iv in stmt
end procedure
```

Benchmarks and Performance

- Parsing
  - SPEC CPU 2006: 14 codes, 990k lines, 4 ½ minutes
  - SPEC OMP 2001: 3 codes, 17k lines, 15 seconds
  - NAS Parallel Benchmark: 8 codes, 15k lines, 17 seconds
- Runtime validation
  - Parse/Print → Compile → Run → Result Validation
  - All of the 25 codes validates
- Loop parallelization (#loop, #manual, #auto, #manual and auto)
  - SPEC OMP 2001: 408, 23, 99, 8
  - NAS Parallel Benchmark: 910, 195, 653, 156

Cetus Roadmap

- Cetus 1.2 release: May 21, 2010
  - New passes
    - Alias analysis on top of points-to analysis
    - Inline expansion (experimental)
  - Improved robustness of the IR package
  - Improved automatic parallelization
  - First release with OpenMP-to-CUDA translator package
- Future enhancements

  Improved parallelization
  - Enhancements to existing passes
  - Locality enhancement techniques

  Cetus application passes
  - OpenMP to MPI translator
  - Autotuning system

We want community contributions!

Cetus Intermediate Representation (IR)

Code Traversal

Major Parts of Class Hierarchy

Base Classes Sub Classes

Program
TranslationUnit
Declaration
- Procedure
- VariableDeclaration
- ClassDeclaration
- AnnotationDeclaration
- …

Statement
- IfStatement
- ForLoop
- CompoundStatement
- AnnotationStatement
- …

Expression
- BinaryExpression
- UnaryExpression
- FunctionCall
- …

Base Classes Sub Classes

IRIterator
- DepthFirstIterator
- BreadthFirstIterator
- FlatIterator

Declarator
- VariableDeclarator
- ProcedureDeclarator
- …

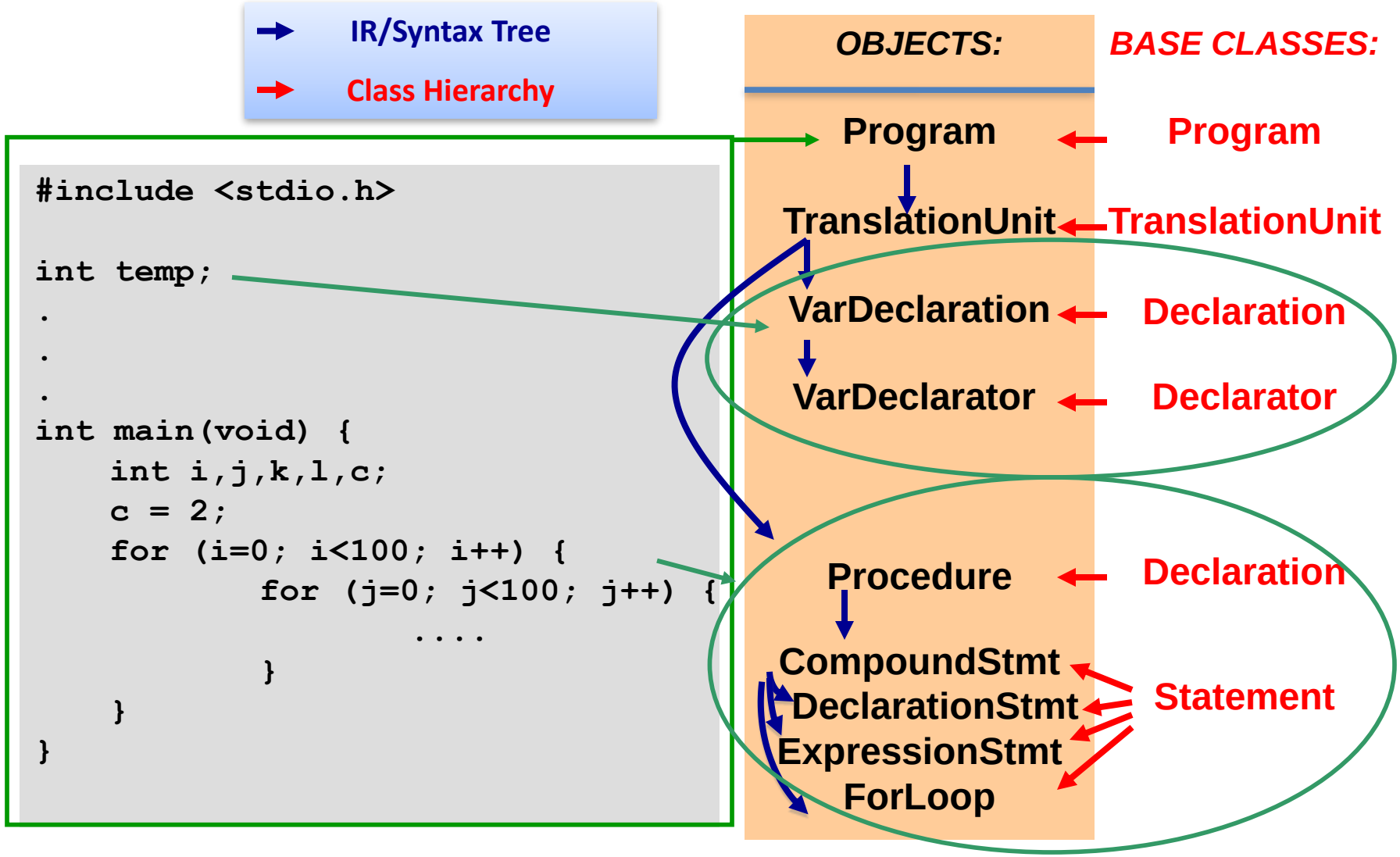
Annotation
- CommentAnnotation
- PragmaAnnotation
- CodeAnnotation

BinaryOperator
- AssignmentOperator
- AccessOperator

UnaryOperator
Specifier
- PointerSpecifier
- …

Traversable
Not traversable

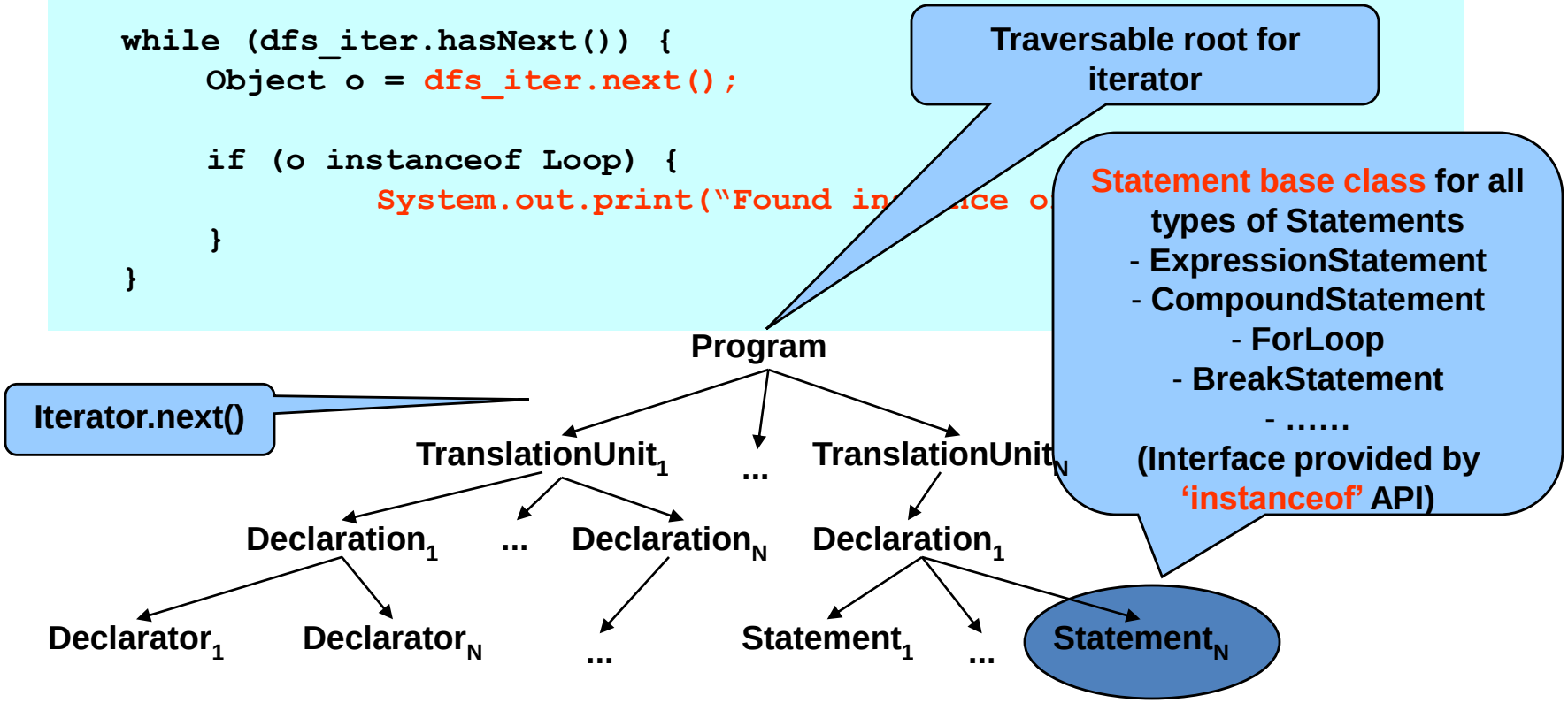
Cetus IR Tree versus Class Hierarchy



IR Iterators

- DepthFirst Iteration

```
/* Iterate depth-first over program which is instance of Program */
DepthFirstIterator dfs_iter = new DepthFirstIterator(program);

while (dfs_iter.hasNext()) {
    Object o = dfs_iter.next();

    if (o instanceof Loop) {
        System.out.print("Found in ce o
    }
}
```

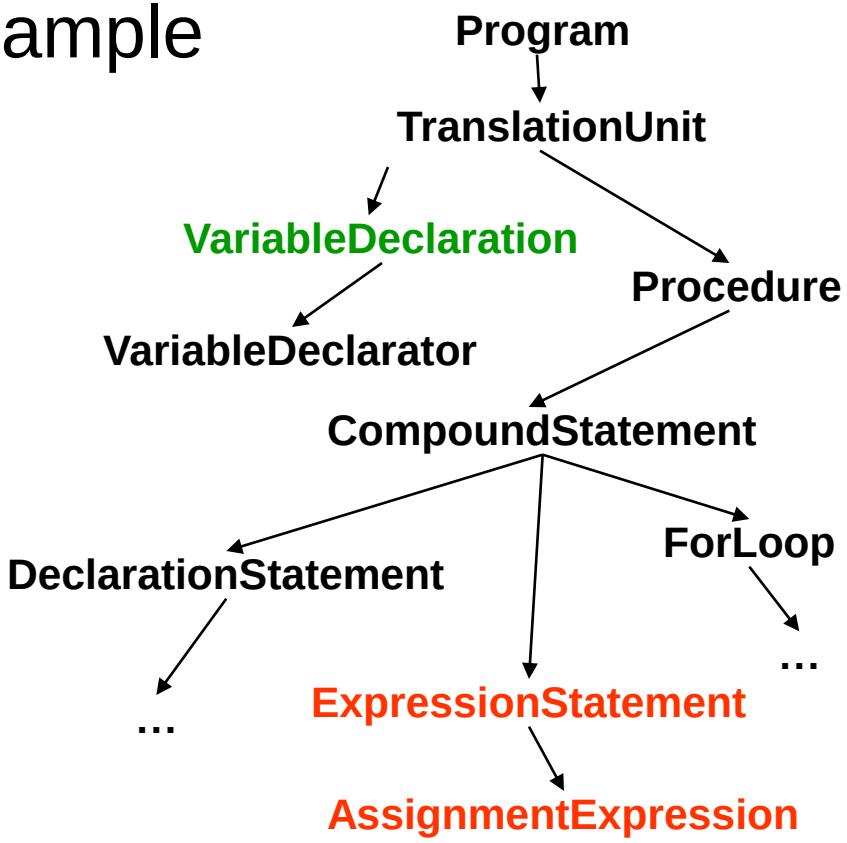


IR Iterators

- IR Tree DepthFirst Example

```
int temp;
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
    }
}
```

```
int temp;
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
    }
}
```

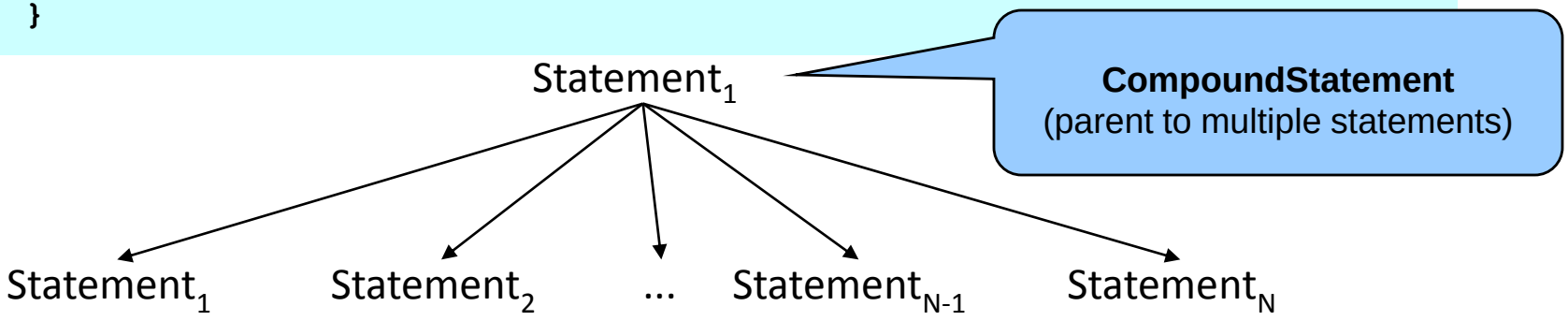


DepthFirst traversal is most commonly used for IR traversal.

IR Iterators

- Flat Iteration (Single level iteration)

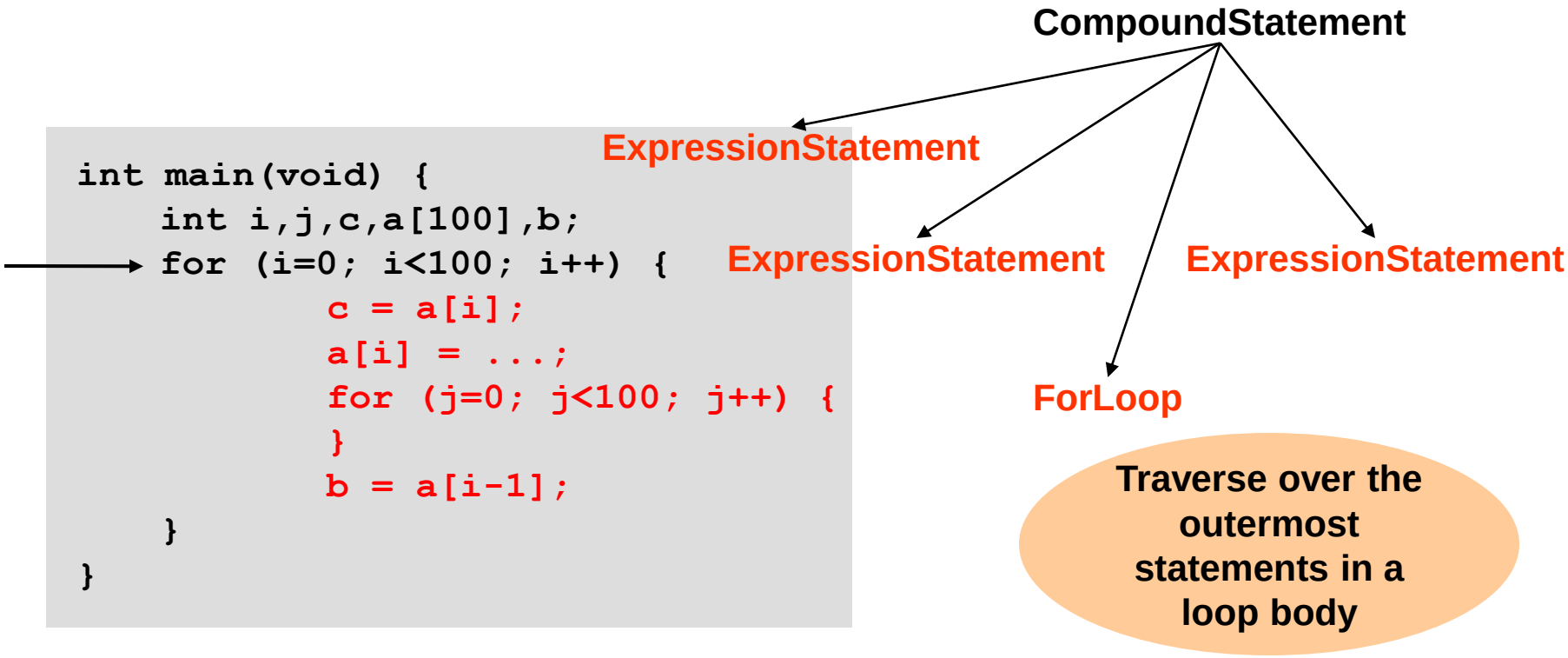
```
/* Iterate single level on compound statement */
CompoundStatement loop_body = (CompoundStatement)loop.getBody();
FlatIterator flat_iter = new FlatIterator(loop_body);

while (flat_iter.hasNext()) {
    Object o = flat_iter.next();

    if(o instanceof ExpressionStatement) {
            Tools.println("Found Expression Statement in Loop", 2);
    }
}
```



Flat iteration is the first level of breadth-first iteration. (Breadth-first iteration is available as well, but used rarely)

IR Iterators

- IR Tree Flat Iteration Example



Compound Statement – ForLoop, ExpressionStatement (collection of Statement objects)

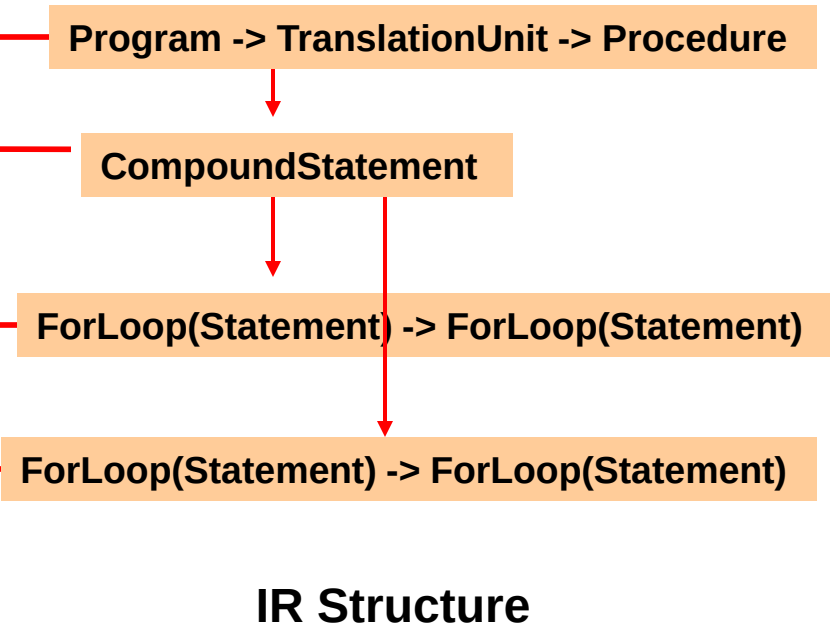
IR Iterators

- Additional functional interface for traversal
  - next(Class c) returns the next object of Class c
    - `iter.next(Statement.class);`
  - next(Set s) returns the next object belonging to any of the classes in Set s
    - `HashSet<Class> of_interest = new HashSet<Class>();`
    - `of_interest.add(AssignmentExpression.class);`
    - `of_interest.add(UnaryExpression.class);`
    - `iter.next(of_interest);`
  - pruneOn(Class c) forces the iterator to skip all descendants of objects of Class c in the IR tree
    - `iter.pruneOn(AssignmentExpression.class);`

IR Traversal – Example

- Loop Nest Identification – Traverse over all *outermost loops* in the program i.e. all loops at the outermost level of a loop nest in the program

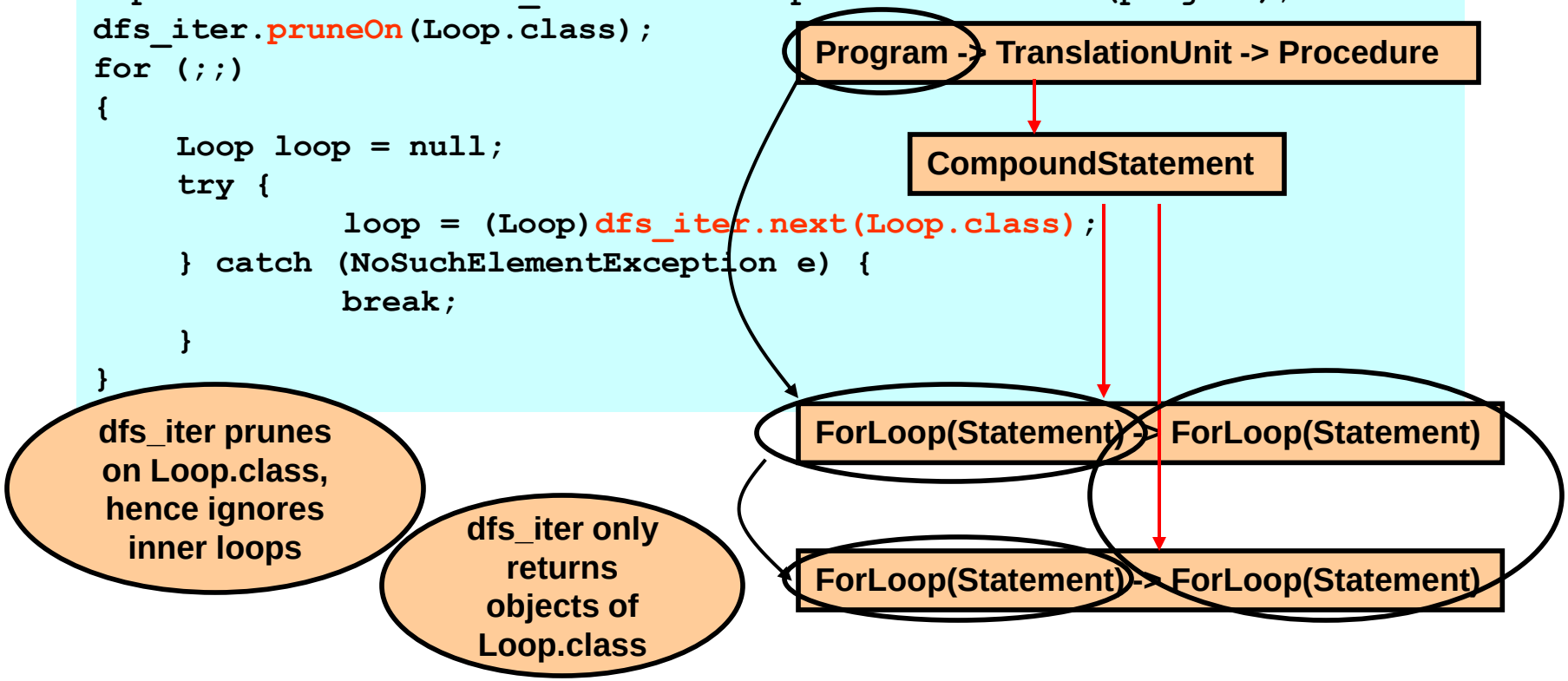
```
int main(void) {
    int i,j,k,l;
    float a[100][100],b[100][100], c;

    c = 2.0;
    for (i=0; i<100; i++) {
        for (j=0; j<100; j++) {
            ....
        }
    }
    for (k=0; k<100; k++) {
        for (l=0; l<100; l++) {
            ....
        }
    }
}
```



IR Traversal - Example

- Loop nest identification – Iterator interface

```
/* Iterate depth-first over outermost loops in the program */
DepthFirstIterator dfs_iter = new DepthFirstIterator(program);
dfs_iter.pruneOn(Loop.class);
for (;;)
{
    Loop loop = null;
    try {
        loop = (Loop)dfs_iter.next(Loop.class);
    } catch (NoSuchElementException e) {
        break;
    }
}
```



Cetus IR Traversal

- Call Tree Traversal
  - Call Tree is HashMap indexed by Procedure
  - Tree Node contains Procedure and list of callers and callees

// Interface includes
callsSelf(Procedure p)
isLeaf(Procedure p)
isRecursive(Procedure p)

Get Call Graph

```
/* Generate a call graph from
the program IR */
CallGraph cg_generator =
    new CallGraph(program);
HashMap cg =
    cg_generator.getCallGraph();
```

```
/* Obtain the call graph node for
Procedure proc and access its
callers and callees */
...
Node p = cg.get(proc);
```

Get Proc Node

```
ArrayList<Caller> proc_callers =
                        p.getCallers();
ArrayList<Procedure> proc_callees =
                        p.getCallees();
for(Caller c : proc_callers) {
    Statement call_site =
            c.getCallSite();
    Procedure calling_proc =
            c.getCallingProc();
}
```

Access callers and callees

Symbol Table Interface

Accessing Symbol Table

SymbolTable interface

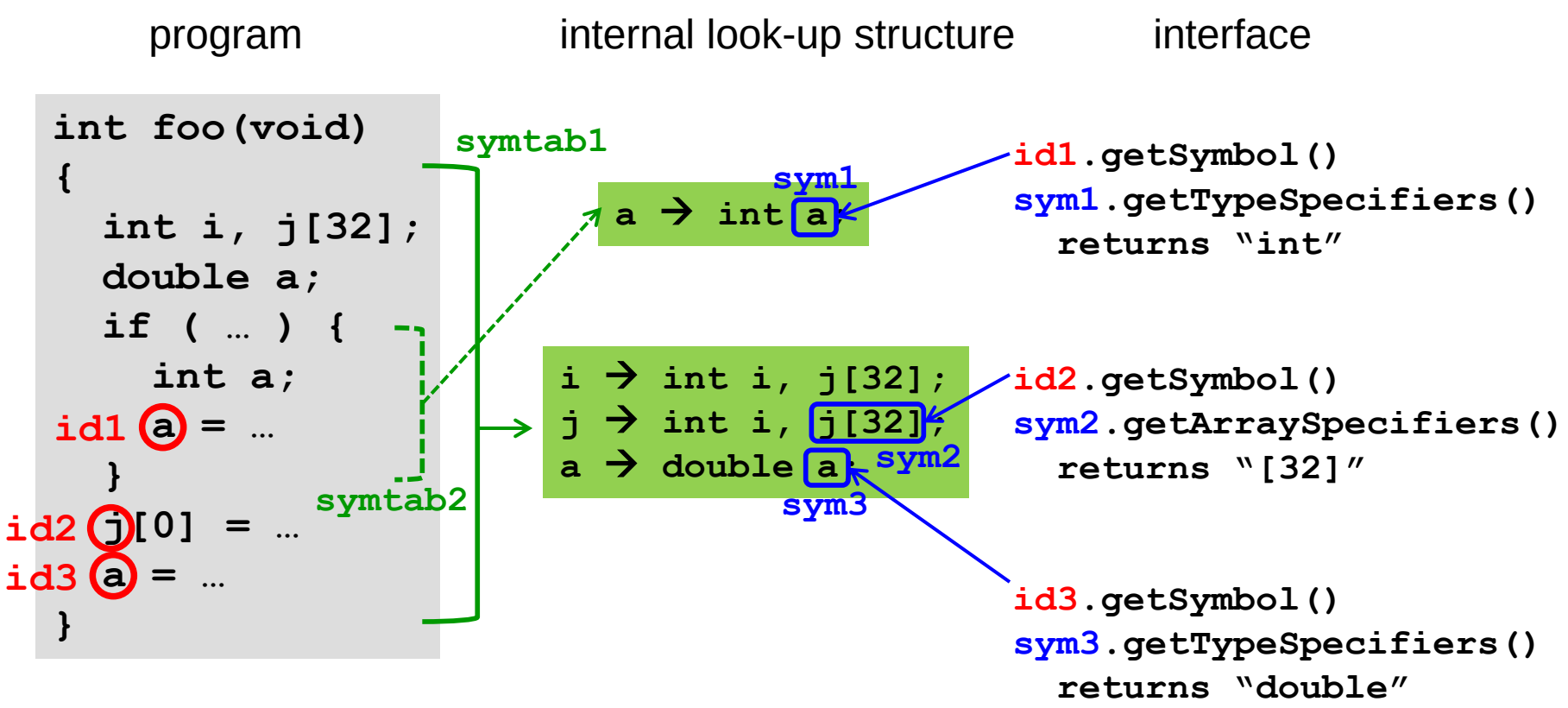
- IR with scope inherits SymbolTable interface
  - TranslationUnit, CompoundStatement, …
- Such an IR object stores a look-up structure
  - From identifier to its declaration
  - Interface functions search/modify this structure
- Methods provided by SymbolTable interface
  - Adding declaration
    - addDeclaration(), addDeclarationBefore(), …
  - Finding declaration of an identifier
    - findSymbol()

Accessing Symbol Table

Symbol interface

- IR representing declarator inherits Symbol interface
  - VariableDeclarator, ProcedureDeclarator, …
- Defines a unique entity for every variable
- Provides fast access to symbol attributes
  - Symbol look-up is preprocessed after parsing
  - Every Identifier expression points to its symbol
- Methods provided by Symbol interface
  - Accessing variables' specifiers
    - getTypeSpecifiers(), getArraySpecifiers()
  - Accessing the symbol name
    - getSymbolName()

Accessing Symbol Table



Declaring Variables

- Find the scope for a new variable
- Create an identifier with a non-conflicting name
  - Use SymbolTable interface
- Prepare specifiers (type, array, pointer, …)
  - Predefined intrinsic types from Specifier class
  - Use UserSpecifier to create user-defined types
- Create variable declaration
- Insert the declaration to the scope

Declaring Variables

Inserting a temporary integer variable in current scope

```
/* Inserts a temporary integer */
void InsertTemp(Traversable t)
{
  while (!(t instanceof SymbolTable))
    t = t.getParent();
  SymbolTable symtab = (SymbolTable)t;
  NameID temp = new NameID("temp");
  int i = 0;
  while (symtab.findSymbol(temp) != null)
    temp = new NameID("temp"+(i++));
  Declarator d = new VariableDeclarator(temp);
  Declaration decl =
    new VariableDeclaration(Specifier.INT,temp);
  symtab.addDeclaration(dd);
}
```

find scope

decide name

create declaration

insert declaration

```
for (i=0; i<100; i++) {
 for (j=0; j<100; j++) {
  double temp;
  …
  a[i][j] = b;
  …
 }
}
```

t

symtab =

temp = "temp0"

decl = "int temp0;"

```
for (i=0; i<100; i++) {
 for (j=0; j<100; j++) {
  double temp;
  int temp0;
  …
  a[i][j] = b;
  …
 }
}
```

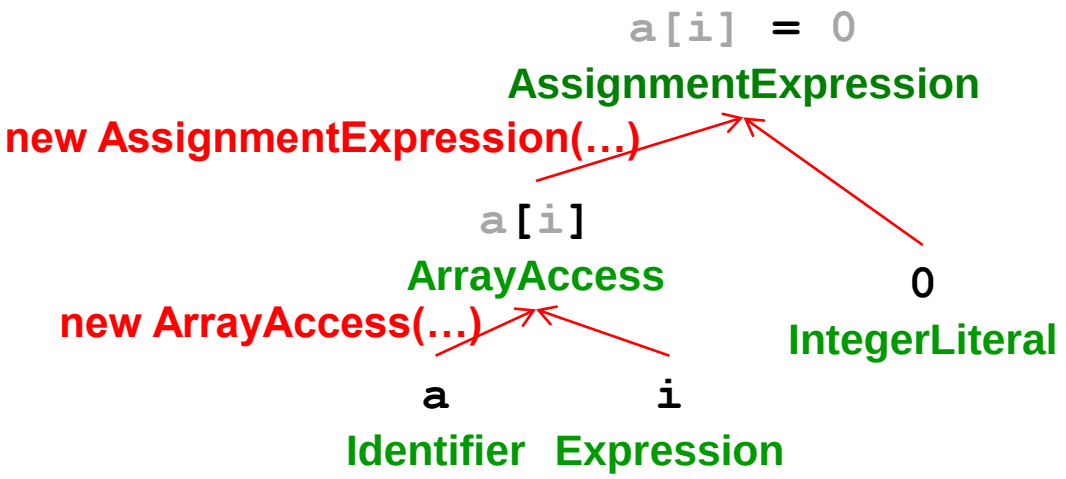
hir.SymbolTools class contains methods that provide these functionalities

Working on the Program IR

Creating Cetus IR objects

- Equivalent to building a tree from leaves
  - Some IRs are created first then populated,
    e.g., CompoundStatement
- IR constructors perform the tasks
- Use cloning if creating from an existing IR object

Creating
AssignmentExpression
"a[i] = 0"



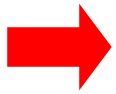
Modifying Cetus IR objects

- Equivalent to inserting/replacing a node in a tree
- IR methods handle most modifications
- Other modifications are usually search/replace
  - Create a new expression or statement
  - Traverse the IR to locate the position for the new node to be inserted or replaced with
  - Perform insertion or replacement
- We will explore the details with examples
  - Loop unrolling
  - Loop instrumentation

Working Example – Loop Unrolling

Loop unrolling by factor of 2

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```



```
for (i=0; i<100; i+=2) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
  fx[i+1] = fx[i+1]+x;
  fy[i+1] = fy[i+1]+y;
  fz[i+1] = fz[i+1]+z;
}
```

Find loop index
Create expression i+=2
Replace expression i++ → i+=2

Create 3 Statements
Find loop index
Search/Replace Expression i → i+1
Insert 3 Statements

```
UnrollLoopBy2(ForLoop L)
{
  /* we will compose assuming
     L is a Fortran-style loop */
}
```

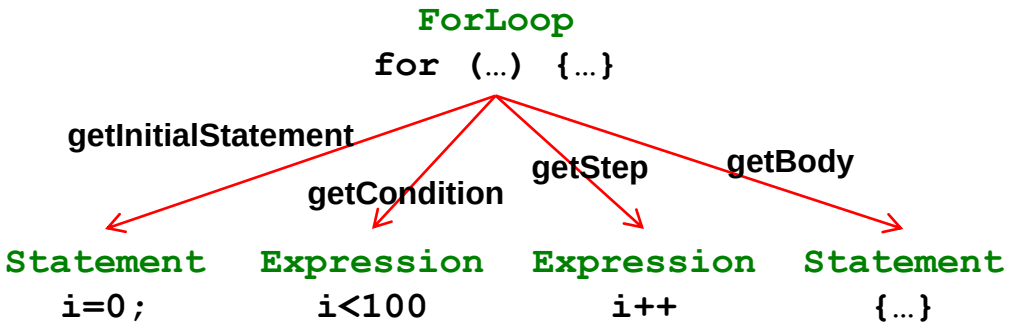
Working on the Program IR

Expressions

Accessing Expressions with IR methods

- IR methods provide direct access to child expressions
  - **ForLoop**: getInitialStatement(), getCondition(), getStep(), …
  - **IfStatement**: getControlExpression, getThenStatement(), …
  - **ExpressionStatement**: getExpression()
  - **ArrayAccess**: getIndex(), getArrayName()
  - **FunctionCall**: getArguments(), getName(), …
  - **BinaryExpression**: getLHS(), getRHS()

```
for (i=0; i<100; i++) {
  …
}
```



Accessing Expressions with IR methods

Finding loop index in UnrollLoopBy2(L)

```
Expression FindLoopIndex(ForLoop L)
{
(1)Statement stmt = L.getInitialStatement();
(2)Expression expr = stmt.getExpression();
(3)Expression index = expr.getLHS();
   return index;
}
```

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

(1) stmt = "i=0;"

(2) expr = "i=0"

(3) index = "i"

Creating Expressions

- Most modifications start from creation
  - Replacing an expression with a new one
  - Inserting a new expression
- Use IR constructors
  - **BinaryExpression**(lhs, op, rhs)
  - **AssignmentExpression**(lhs, op, rhs)
  - **FunctionCall**(name, arguments)
- Children are usually created first
- Do not reuse existing IR objects – use clone()

Creating Expressions

Creating new step expression in UnrollLoopBy2(L)

```
Expression CreateStepBy2(Expression index)
{
(1)Expression index0 = (Expression)index.clone();
(2)Expression step = new IntegerLiteral(2);
(3)Expression expr = new AssignmentExpression(
     index0, AssignmentOperator.ADD, step);
   return expr;
}
```

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

FindLoopIndex(L) = "i"

(1) index0 = "i"

(2) step = "2"

(3) expr = "i+=2"

Creating Expressions

Creating new indices for unrolled statements in UnrollLoopBy2(L)

```
Expression CreateIndexPlus1(Expression index)
{
(1)Expression index0 = (Expression)index.clone();
(2)Expression one = new IntegerLiteral(1);
(3)Expression expr = new BinaryExpression(
     index0, BinaryOperator.ADD, one);
   return expr;
}
```

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

FindLoopIndex(L) = "i"

(1) index0 = "i"

(2) one = "1"

(3) expr = "i+1"

Modifying Expressions with IR methods

- IR methods provide high-level modifiers
  - **ForLoop**: setCondition(), setStep(), …
  - **IfStatement**: setControlExpression(), setThen(), ...
  - **ArrayAccess**: setIndex(), …
  - **FunctionCall**: setArguments(), addArgument(), …
  - **BinaryExpression**: setLHS(), setRHS()
- Replacing step expression In UnrollLoopBy2(L)

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

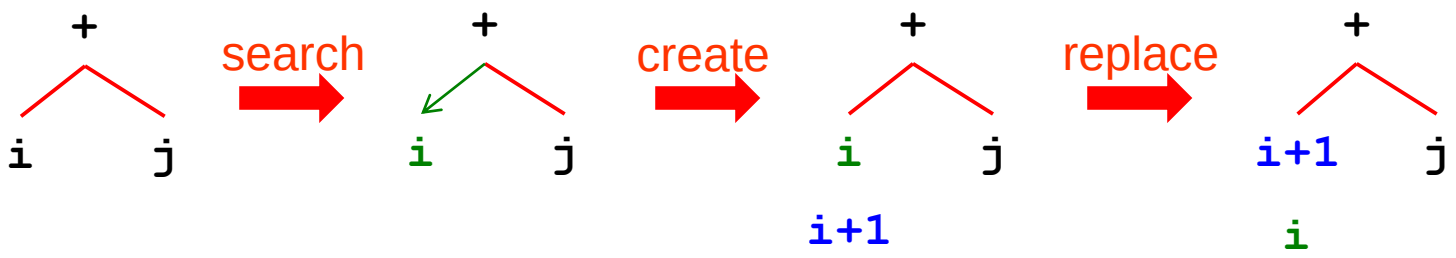
new_step = "i+=2"

L.setStep(new_step);

```
for (i=0; i<100; i+=2) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

Searching/Replacing Expressions

- Frequently used in program transformation
  - Replacing identifiers with constants: `foo(i,j)→foo(10,100)`
  - Replacing identifiers with other expressions: `i+j→(i+1)+j`
  - Replacing common sub expressions with temporaries: `i+j→t`
- Steps
  - Locate the expression to be replaced (traversal/compare)
  - Create a new expression with a constructor
  - Replace the old expression with the new one



Searching/Replacing Expressions

Modifying array indices in unrolled statements in UnrollLoopBy2(L)

```
void ReplaceExpr(Expression a, Expression b, Traversable t)
{
  List old_exprs = new ArrayList();
  DepthFirstIterator iter = new DepthFirstIterator(t);
  while ( iter.hasNext() ) {
    Object child = iter.next();
    if ( child.equals(a) ) old_exprs.add(child);
  }                                                    search
  for ( Object old_expr : old_exprs ) {
    Expression new_expr = (Expression)b.clone();
    ((Expression)old_expr).swapWith(new_expr);        replace
  }
}
```

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

```
a: i, b: i+1
t: fx[i] = fx[i]+x;

old_exprs: [i, i] (search)

t: fx[i+1] = fx[i+1]+x; (replace)
```

Working on the Program IR

Statements

Creating Statements

- Use IR constructors
  - **ExpressionStatement**(expr)
  - **ForLoop**(init, condition, step, body)
  - **IfStatement**(condition, thenstmt, elsestmt)
  - **CompoundStatement**() – child statements are added later
- Use clone() from existing statements
- Creating new unrolled statements in UnrollLoopBy2(L)

```
List CreateNewStatements(ForLoop L)
{
  List stmts = new ArrayList();
  FlatIterator iter =
    new FlatIterator(L.getBody());
  while ( iter.hasNext() )
    stmts.add(iter.next().clone());
  return stmts;
}
```

statements from cloning

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

```
stmts: [fx[i]=fx[i]+x;,
  fy[i]=fy[i]+y;,
  fz[i] = fz[i]+z;]
```

Inserting Statements

- Steps
  - Locate the reference statement (before or after)
  - Locate the parent **CompoundStatement**
  - Use appropriate methods in **CompoundStatement**
    - addStatement(new_stmt): at the end
    - addStatementBefore(ref_stmt, new_stmt): before ref_stmt
    - addStatementAfter(ref_stmt, new_stmt): after ref_stmt
- Inserting unrolled statements in UnrollLoopBy2(L)

```
void InsertNewStatements(ForLoop L, List stmts)
{
  CompoundStatement parent =
    (CompoundStatement)L.getBody();
  for ( Object stmt : stmts )
    parent.addStatement((Statement)stmt));
}
```

Composing UnrollLoopBy2(ForLoop L)

```
void UnrollLoopBy2(ForLoop L)
{
  Expression index = FindLoopIndex(L);
  Expression new_step = CreateStepBy2(index);
  L.setStep(new_step);
  List new_stmts = CreateNewStatements(L);
  Expression index1 = CreateIndexPlus1(index);
  for ( Object stmt : new_stmts )
    ReplaceExpr(index, index1, (Statement)stmt);
  InsertNewStatements(L, new_stmts);
}
```

index: `i`

new_step: `i+=2`

`for (…; i+=2) {…}`

new_stmts:
[`fx[i]=fx[i]+x, …`]

index1: `i+1`

new_stmts:
[`fx[i+1]=fx[i+1]+x,…`]

ForLoop L

```
for (i=0; i<100; i++) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
}
```

UnrollLoopBy2(L)

```
for (i=0; i<100; i+=2) {
  fx[i] = fx[i]+x;
  fy[i] = fy[i]+y;
  fz[i] = fz[i]+z;
  fx[i+1] = fx[i+1]+x;
  fy[i+1] = fy[i+1]+y;
  fz[i+1] = fz[i+1]+z;
}
```

Working on the Program IR

Annotations

Creating/Inserting Annotations (v1.1)

- Annotations are used in Cetus for
  - Information exchange between passes
    e.g., set of modified / used variables
  - Information for backend compilers – pragmas
    e.g., OpenMP pragmas
  - Information for code readers – comments
- Annotations can be either
  - Stand-alone annotation or
  - Attached annotation: associated with a specific statement or a declaration

Creating/Inserting Annotations (v1.1)

- Steps
  - Create an annotation with an appropriate type
    - CommentAnnotation, PragmaAnnotation, …
  - Locate the position for the annotation
    - Attached annotation: the associated statement/declaration
    - Stand-alone annotation: the reference statement/declaration
  - Insert the annotation
    - Attached annotation: insert directly to the statement/declaration
    - Stand-alone annotation: encapsulate in a statement/declaration and insert around the reference
  - Insert code in the driver to call the annotation tool

Cetus driver to create a loop instrumentation tool

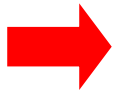
- Modify Driver.java to run MyTool

```
public void run_modified(String[] args) {
    parseCommandLine(args);

    parseFiles();

    runPasses();

    program.print();
  }
```

```
public void runPasses() {
    ...
    MyTool my_loop_instrumenter = new MyTool();
    my_loop_instrumenter.InstrumentLoops(program);
    ...
  }
```

Working Example – Loop Instrumentation

```
int compute(…)
{
 for (i=is; i<ie; i++) {
  …
 }
 …
 for (j=js; j<je; j++) {
  …
 }
}
```



Inserts loop name and timing calls

Find loop
Create function call
Insert function call

Find loop
Create annotation
Insert annotation

```
int compute(…)
{
 /* compute#0 */
 cetus_tic(0);
 for (i=is; i<ie; i++) {
  …
 }
 cetus_toc(0);
 …
 /* compute#1 */
 cetus_tic(1);
 for (j=js; j<je; j++) {
  …
 }
 cetus_toc(1);
}
```

Loop Instrumentation Code

```
public static void InstrumentLoops(Program P) {     Pseudo code
  Create Identifiers "cetus_tic" and "cetus_toc"
  foreach Procedure proc in P {    → see DepthFirstIterator
    List<Statement> loops = new ArrayList<Statement>();
    foreach Loop loop in proc      → see the next slide
      loops.add(loop);                                  find loops
    String name = proc.getName();
    int i=0;
    for ( Statement loop : loops ) {
                                                  create annotation
      Annotation loopname = new CommentAnnotation(name+"#"+i);
      Statement ticstmt = CreateTimingCall(cetus_tic, i);
create timing calls
      Statement tocstmt = CreateTimingCall(cetus_toc, i++);
      AddInstrumentation(loop, loopname, ticstmt, tocstmt);
                                                  insert statements
    }
  }
}
```

Loop Instrumentation Code

Finding loops

```
void InstrumentLoops(Program P)
{
  /* foreach proc in P */
  Procedure proc = …
  List<Statement> loops =
    new ArrayList<Statement>();
  /* foreach loop in proc */
  DepthFirstIterator iter =
    new DepthFirstIterator(proc);
  while ( iter.hasNext() ) {
    Object o = iter.next();
    if ( o instanceof Loop )
      loops.add((Statement)o);
  }
  …
}
```

proc

```
int compute(…)
{
 for (i=is; i<ie; i++) {
  …
 }
 …
 for (j=js; j<je; j++) {
  …
 }
}
```

loops: [
  for(i=is…){…},
  for(j=js;…){…}
]

Loop Instrumentation Code

Creating loop naming annotations as comments

```
public static void InstrumentLoops(Program P) {
  …
  /* proc: current procedure, loops: loops in proc */
  String name = proc.getName();
  int i=0;
  for ( Statement loop : loops ) {
    Annotation loopname = new CommentAnnotation(name+"#"+i);
    … = … i++ …;
  }
}
```

proc

```
int compute(…)
{
 for (i=is; i<ie; i++) {
  …
 }
 for (j=js; j<je; j++) {
  …
 }
}
```

name: **"compute"**

For the first loop:
loopname: **/* compute#0 */**

For the second loop:
loopname: **/* compute#1 */**

Loop Instrumentation Code

Creating timing calls

```
/* Creates a function call "fcname(num);" */
Statement CreateTimingCall(Identifier fcname, int num)
{
(1)FunctionCall fc = new FunctionCall(fcname.clone());
(2)fc.addArgument(new IntegerLiteral(num));
(3)Statement fcstmt = new ExpressionStatement(fc);
   return fcstmt;
}
```

```
int compute(…)
{
 for (i=is; i<ie; i++) {
  …
 }
 …
 for (j=js; j<je; j++) {
  …
 }
}
```

```
(1) fc: cetus_tic()
(2) fc: cetus_tic(0)
(3) fcstmt: cetus_tic(0);
```

Similarly,

```
cetus_toc(0);, cetus_tic(1);, cetus_toc(1);
```

Loop Instrumentation Code

Inserting statements

```
/* Inserts name and tic before loop,
 * and toc after loop */
AddInstrumentation(Statement loop,
Annotation loopname, Statement tic, Statement toc)
{
  CompoundStatement parent =
    (CompoundStatement)loop.getParent();
  loop.annotate(loopname);                    (1)
  parent.addStatementBefore(loop, tic);       (2)
  parent.addStatementAfter(loop, toc);        (3)
}
```

locate parent compound statement

```
int compute(…)
{
 for (i=is; i<ie; i++) {
  …
 }
 …
 for (j=js; j<je; j++) {
  …
 }
}
```

```
int compute(…)                    (1)
{
 /* compute#0 */
 for (i=is; i<ie; i++) {
  …
 }
 …
}
```

```
int compute(…)                    (2) (3)
{
 /* compute#0 */
 cetus_tic(0);
 for (i=is; i<ie; i++) {
  …
 }
 cetus_toc(0);
 …
}
```

```
int compute(…)
{
 /* compute#0 */
 cetus_tic(0);
 for (i=is; i<ie; i++) {…}
 cetus_toc(0);
 /* compute#1 */
 cetus_tic(1);
 for (j=js; j<je; j++) {…}
 cetus_toc(1);
}
```

Checking Aliases

- Cetus' alias analysis performs on top of the interprocedural points-to analysis which was introduced since v1.2. It provides flow-sensitive and context-insensitive information

Example

```
int i, sum=0;
int *p=0, *q=0;
int A[N], B[N];

q = A;
p = &sum;
for (i=0; i<N; i++)
{
  sum += A[i];
  B[i] = *p + *q;
}
```

Points-to relations before each statement

```
p:NULL, q:NULL
p:NULL, q:A
p:sum, q:A

p:sum, q:A
p:sum, q:A
```

Alias set before the for loop

```
p   : {sum}
sum: {p}
q   : {A}
A   : {q}
```

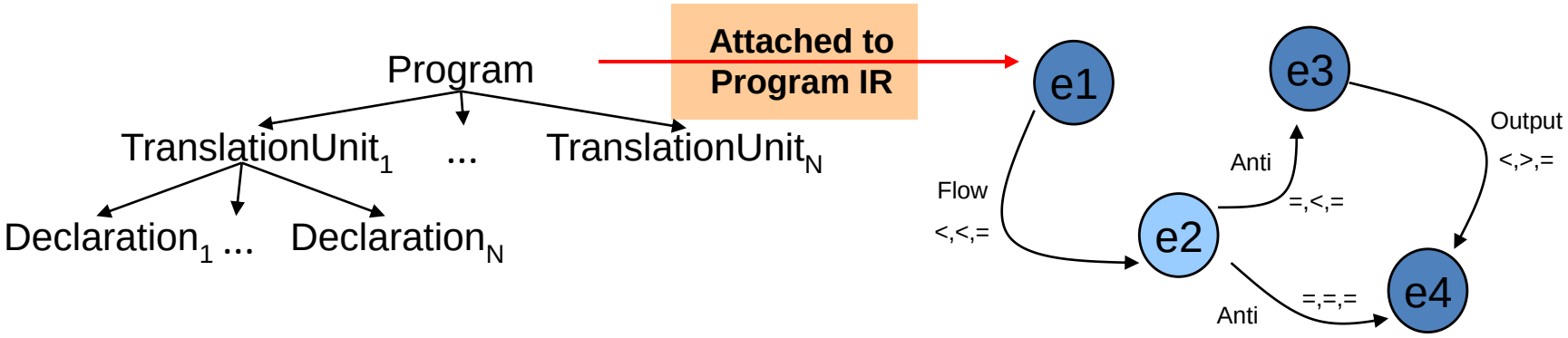
Using Alias Analysis

```
AliasAnalysis alias = new AliasAnalysis(program);
alias.start();

/* 1. returns true if Symbols A and B are aliased at STMT */
if (alias.isAliased(STMT, A, B)) {
  …
}

/* 2. returns Symbols that are aliased to A at STMT */
Set alias_set = alias.get_alias_set(STMT, A);
if (alias_set.contains(A)) {
  …
}
```

Data Dependence Test Interface

Data Dependence Interface – v1.1

- Getting Data Dependence Information
  - DependenceAnalysis run once per IR configuration and output DD Graph attached to Program IR
  - Program IR related DD graph common to all optimization/analysis passes
  - Loop Parallelization pass and other analysis passes only query dependence graph, do not run dependence analysis
  - DD graph must be regenerated explicitly by Pass writer following IR transformation and changes visible to all compiler passes



Data Dependence Interface – v1.1

- Using dependence information for parallelization

```
boolean is_parallel;
DDGraph pdg = program.getDDGraph();
// Check eligibility for dependence testing
if (LoopTools.checkDataDependenceEligibility(loop)==true) {
        /* check if scalar dependences are present using
         * privatization and reduction variable information */
        if (LoopTools.scalarDependencePossible(loop)==true)
                is_parallel = false;
        // check if array loop carried dependences exist
        else if (pdg.checkLoopCarriedDependence(loop)==true)
                is_parallel=false;
        // No dependences
        else
                is_parallel=true;
}
else
        is_parallel=false;

return is_parallel;
```

```
/* After transformation pass such as
Loop-distribution, generate new
dependence graph that is
automatically attached to Program IR,
overwriting the old dependence graph
*/
DDTDriver ddt_driver =
        new DDTDriver(program);
ddt_driver.start();
```

Frequently Asked Questions (& Answers)

- **Having problems compiling Cetus?**
  The source code for Cetus is located in the /src/ directory of your download version. Several options are provided to compile Cetus, these are provided in the Cetus release notes (included in your download), and will be made available via the manual that is under development. One of the most important issues to look out for is the java classpath environment variable. While using each of the build scripts provided with the download, please ensure that you are setting the environment variables correctly to point to the right locations that contain your libraries.

- **How do I run Cetus?**
  Cetus can be run from the command line by invoking cetus.exec.Driver. Details of this command can be found in the release notes, as well as the manual.

- **What tools do I require in order to run Cetus?**
  You will need to have the following on your computer
  - JAVA 2 SDK, SE 1.5.x (or later)
  - ANTLRv2
  - GCC

- **How do I learn more about the Cetus infrastructure?**
  Please refer to the section "Papers describing the infrastructure" on the documentation page.

- **How do I reference Cetus in my work?**
  We would recommend you use the publication that is most relevant to your work from the section on "Papers describing the infrastructure“ on the Cetus website.

Frequently Asked Questions (& Answers)

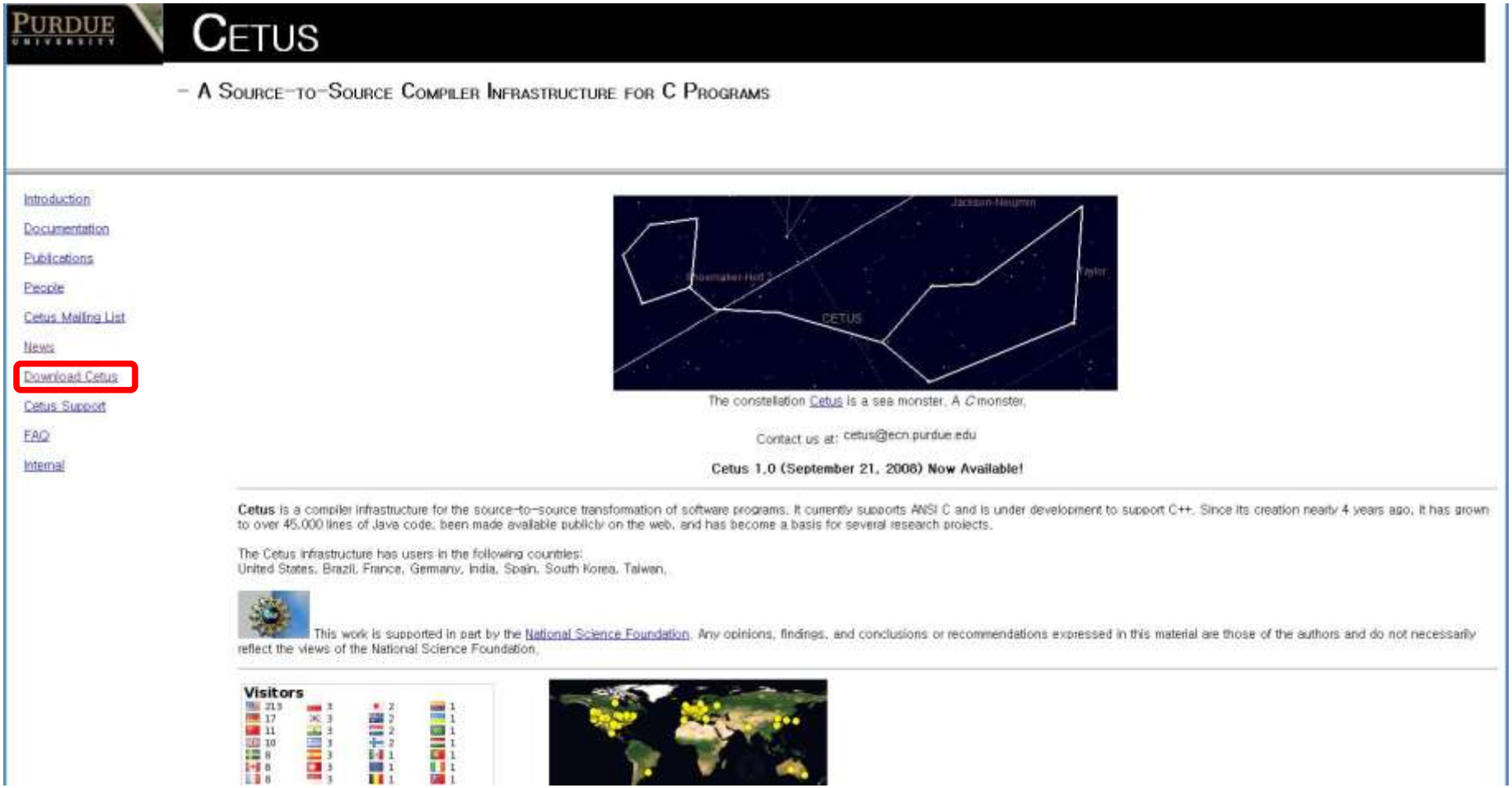
- **Does Cetus include a C++ frontend?**
  The current release does not support C++. We have a mostly functional C++ parser that is separate from Cetus. The cctreewalker directory in your source directory for Cetus provides dump routines to interface with external C++ parser tools such as Flex/Bison and generate IR. We cannot provide additional information regarding the correctness or scope of this implementation due to insufficient testing. Look out for more information in later releases.

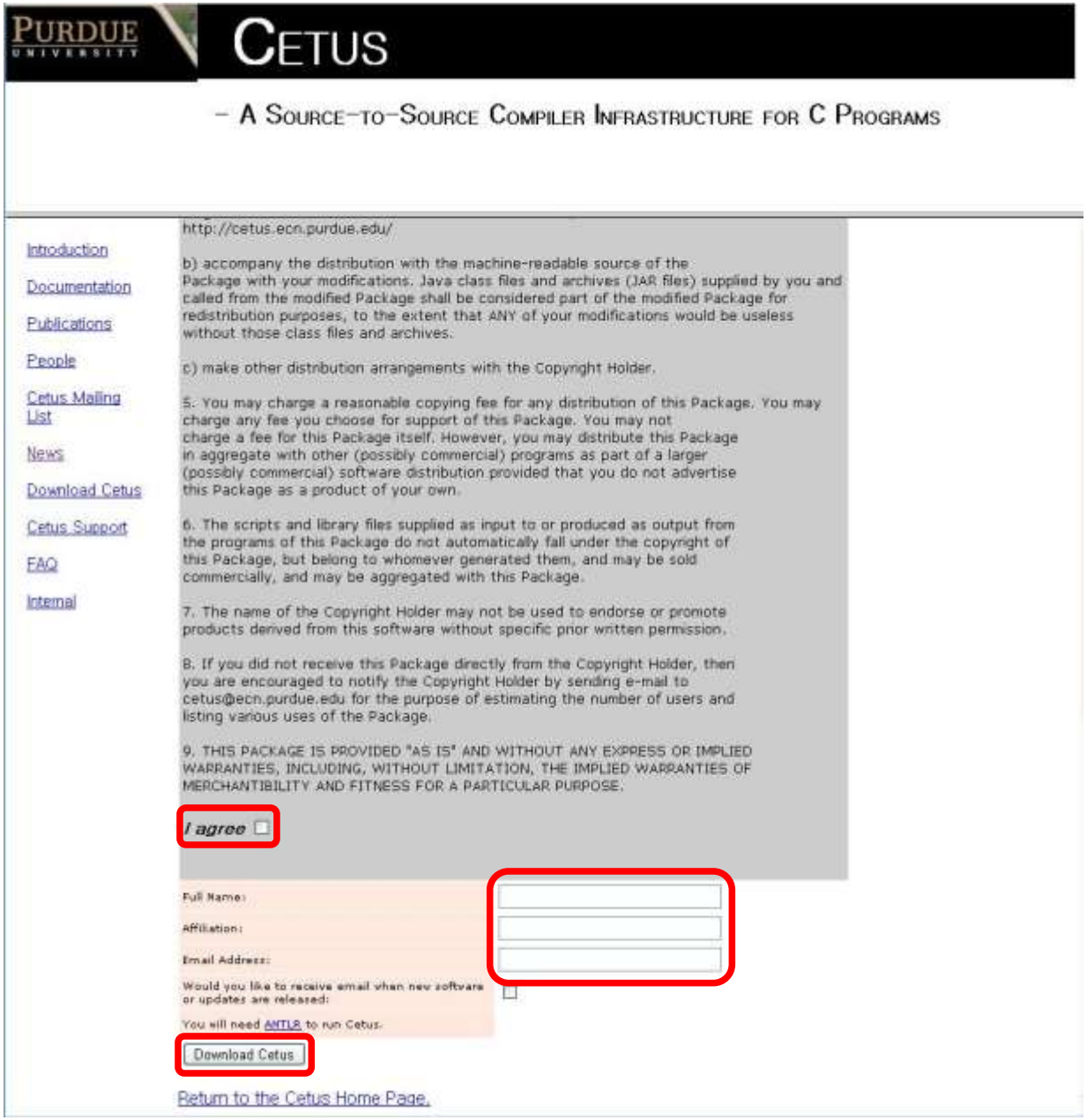
- **Which SDK/JRE is needed?**
  We are using the Java 2 SDK, SE 1.5.0. The documentation and SDK are available from Sun's website.

- **How do I inform the developers of Cetus about bugs that I have come across?**
  We are always looking forward to receiving feedback (good or bad!) from the users of Cetus. Either provide us feedback through the Bugzilla support system or by sending us email through our contact email provided on the homepage.

- **What is Cetus currently being tested on?**
  We are currently working with the following benchmarks on Cetus.
  - SPECCPU 2006
  - SPECOMP 2001
  More information about these suites can be found on the SPEC website at www.spec.org.
  - NPB3.0
  More information about the NAS Parallel Benchmark suite can be found at their website.

Download Cetus (cetus.ecn.purdue.edu)




PURDUE UNIVERSITY
CETUS
– A SOURCE-TO-SOURCE COMPILER INFRASTRUCTURE FOR C PROGRAMS
Introduction
Documentation
Publications
People
Cetus Mailing List
News
Download Cetus
Cetus Support
FAQ
Internal
http://cetus.ecn.purdue.edu/
b) accompany the distribution with the machine-readable source of the Package with your modifications. Java class files and archives (JAR files) supplied by you and called from the modified Package shall be considered part of the modified Package for redistribution purposes, to the extent that ANY of your modifications would be useless without those class files and archives.
c) make other distribution arrangements with the Copyright Holder.
5. You may charge a reasonable copying fee for any distribution of this Package. You may charge any fee you choose for support of this Package. You may not charge a fee for this Package itself. However, you may distribute this Package in aggregate with other (possibly commercial) programs as part of a larger (possibly commercial) software distribution provided that you do not advertise this Package as a product of your own.
6. The scripts and library files supplied as input to or produced as output from the programs of this Package do not automatically fall under the copyright of this Package, but belong to whomever generated them, and may be sold commercially, and may be aggregated with this Package.
7. The name of the Copyright Holder may not be used to endorse or promote products derived from this software without specific prior written permission.
8. If you did not receive this Package directly from the Copyright Holder, then you are encouraged to notify the Copyright Holder by sending e-mail to cetus@ecn.purdue.edu for the purpose of estimating the number of users and listing various uses of the Package.
9. THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
I agree
Full Name:
Affiliation:
Email Address:
Would you like to receive email when new software or updates are released:
You will need ANTLR to run Cetus.
Download Cetus
Return to the Cetus Home Page.

Questions?

Loop Parallelization Results with Cetus

NPB	#LOOPS	#MANUAL	#AUTO	#MANUAL & AUTO
BT	223	54	158	35
CG	41	22	24	21
EP	11	1	6	0
FT	51	6	22	2
IS	14	1	7	0
LU	171	29	146	21
MG	77	12	25	6
SP	314	70	264	70
TOTAL	910	195	653	156

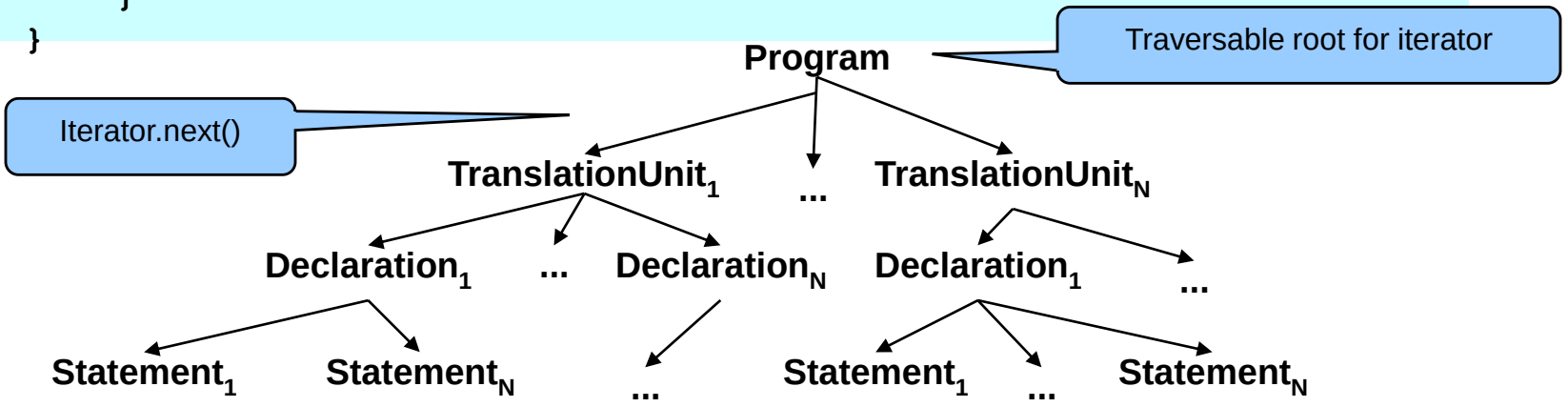
SPEC OMP 2001	#LOOPS	#MANUAL	#AUTO	#MANUAL & AUTO
320.equake	83	11	48	7
330.art	75	5	36	0
332.ammp	250	7	27	2
TOTAL	408	23	99	8

Cetus – IR Iterators

All IR objects implement the Traversable interface

- BreadthFirst Iteration

```
/* Iterate breadth-first program */
BreadthFirstIterator bfs_iter = new BreadthFirstIterator(program);

for (;;) {
    Object o = null;
    try {
            o = bfs_iter.next();
            System.out.print(o.getclass().getname());
    } catch (NoSuchElementException e) {
            break;
    }
}
```

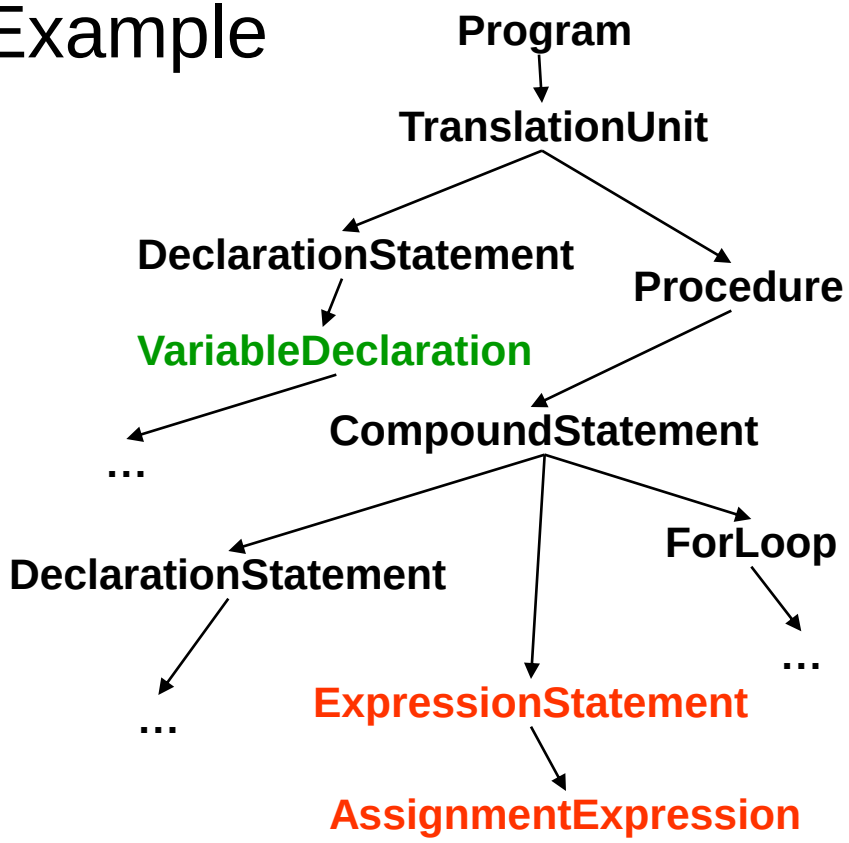


IR Iterators

- IR Tree BreadthFirst Example

```
int temp;
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
    }
}
```

```
int temp;
int main(void) {
    int i,j,k,l,c;
    c = 2;
    for (i=0; i<100; i++) {
    }
}
```



BreadthFirst traversal can be used to traverse upper-level objects in the IR before proceeding to lower-level objects

Diophantine equations

- An equation whose coefficients and solutions are all integers is a Diophantine equation
- Determining if a Diophantine equation has a solution requires a slight detour into elementary number theory
- Let $f(i) = a_1*i_1 + c_1$ and $g(i) = b_1*i_1 + c_2$, then
  - $f(i) = g(i) \Rightarrow a_1*i_1 - b_1*i'_1 = c_2 - c_1$
  - fits general form of Diophantine equation of $\boldsymbol{a_1*i_1 + a_2*i_2 = c}$

Does f(i) = g(i) have a solution?

- The Diophantine equation

$$a_1*i_1 + a_2*i_2 = c$$

has a solution *iff* gcd(a_1,b_1) evenly divides c

Examples:

15*i +6*j -9*k = 12	has a solution	gcd=3
2*i + 7*j = 3	has a solution	gcd=1
9*i + 3*j + 6*k = 5	has no solution	gcd=3

Euclid Algorithm: find gcd(a,b)

Repeat

a ← a mod b

swap a,b

Until b=0

→*The resulting a is the gcd*

for more than two numbers:
gcd(a,b,c) = (gcd(a,gcd(b,c))

Finding GCDs

Euclid Algorithm: find gcd(a,b)

Repeat

a ← a mod b

swap a,b

Until b=0

→*The resulting a is the gcd*

for more than two numbers:
gcd(a,b,c) = (gcd(a,gcd(b,c))

a = 16, b = 6

a ← 16 mod 6

b ← 4, a ← 6

a ← 6 mod 4

b ← 2, a ← 4

a ← 4 mod 2

a ← 2, b ← 0

Determining if a Diophantine equation has a solution

Let g = gcd(a_1,a_2), can rewrite the equation as:

$$g*a'_1*i_1 + g*a'_2*i_2 = c \rightarrow g*(a'_1*i_1+a'_2*i_2) = c$$

Because a'_1 and a'_2 are relatively prime, all integers can be expressed as a *linear combination* of a'_1 and a'_2.

$a'_1*i_1+a'_2*i_2$ is just such a linear combination and therefore $a'_1*i_1–a'_2*i_2$ generates all integers $i \in I$, (assuming a'_1,a'_2 can range over the integers.)

If remainder(c/g) = 0, c is a solution since c = g*c', and $g*(a'_1*i_1–a'_2*i_2)$ generates all multiples of g.

If remainder(c/g) != 0, c cannot be a solution, since all values generated by $g*(a'_1*i_1–a'_2*i_2)$ are (trivially) divisible by g, and cannot equal any c that is not divisible by g.

More information on gcd's and dependence analysis

- General books on number theory
- Books by Utpal Banerjee (Kluwer Academic Publishers), (Illinois, now Intel) who developed the GCD test in late 70's, Mike Wolfe, (Illinois, now Portland Group) "High Performance Compilers for Parallel Computing
- Randy Allen's thesis, Rice University
- Work by Eigenman & Blume Purdue (range test)
- Work by Pugh (Omega test) Maryland
- Work by Hoeflinger, etc. Illinois (LMAD)

What do dependence tests do?

- Some tests, and Banerjee's in some situations (affine subscripts, rectangular loops) are precise
  - Definitively proves existence or lack of a dependence
- Most of the time tests are conservative
  - Always indicate a dependence if one may exist
  - May indicate a dependence if it does not exist
- In the case of "may" dependence, run-time test or speculation can prove or disprove the existence of a dependence
- Short answer: tests disprove dependences for some dependences

Banerjee's Inequalities

If $a*i_1 - b*i'_1 = c$ has a solution, does it have a solution within the loop bounds, and for a given direction vector?

```
do i = 1, 100
   x(i) =
        = x(i-1)
end do
```

Note: there is a (<) dependence.

Let's test for (=) and (<) dependence.

By the mean value theorem, c can be a solution to the equation $f(i) = c,\ i \in [lb,ub]$ iff

- $f(lb) <= c$
- $f(ub) >= c$

(assumes $f(i)$ is monotonically increasing over the range $[lb,ub]$)

The idea behind ***Banerjee's Inequalities*** is to find the maximum and minimum values the dependence equation can take on for a given direction vector, and see if these bound *c*. ***This is done in the real domain since integer solution requires integer programming (in NP).***

Example of where the direction vector makes a difference

```
do i = 1, 100
   x(i) =
        = x(i-1)
end do
```

Note: there is a (<) dependence.

Let's test for (=) and (<) dependence.

Dependence equation is **i-i' = -1**

If i = i', then **i-i' = 0**, $\forall$ i, i'

If i < i', then i-i' $\neq$ 0, and when **i'=i+1**, the equation has a solution.

Banerjee test

If $a*i_1 - b*i'_1 = c$ has a solution, does it have a solution within the loop bounds for a given direction vector (<) or (=) in this case)?

For our problem, does $i_1 - i'_1 = -1$ have a solution

- For $i_1 = i'_1$, then it does not (no (=) dependence).
- For $i_1 < i'_1$, then it does ((<) dependence).