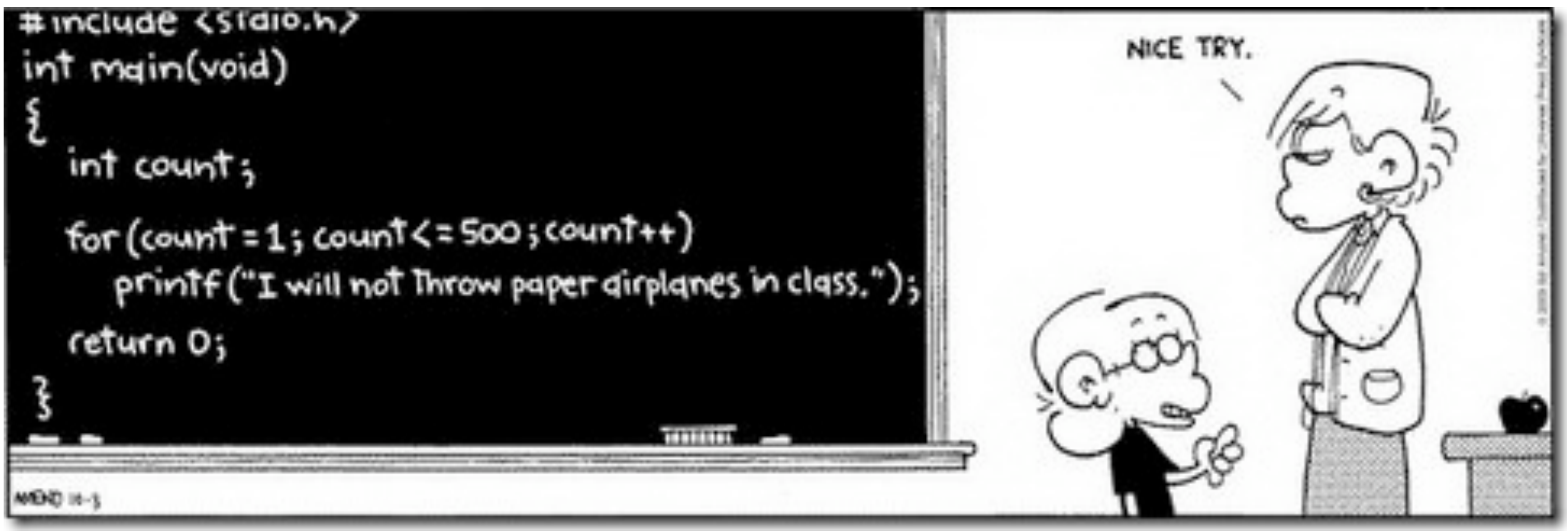
Formal Methods for Program Analysis and Generation

Robert van Engelen

First, a little story…

Step 0: School

We learned to program in school…



Step 1: College

… then told to forget what we learned and start over…

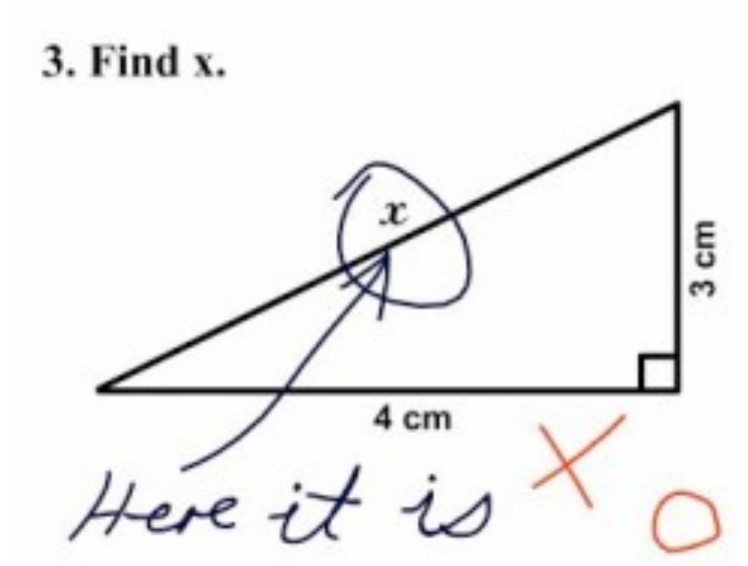
```
// Assignment 1:   cupsof.java
// Submitted by:   * bucks
// Shows good Java coding style and commenting

import java.lang.*;

public class cupsof
{
	public static void main(String[] arg)
	{
		// print 500 times something to cheer about
		for (int count = 0; count < 500; count++)
			System.out.println(count + " cups of java on the wall");
	}
}
```

Step 2: Graduation

… all the while doing our best to impress professors…



Step 3: Business

… to find our dream job!

The Experts Told Us…

Carefully design your programs!

"Controlling complexity is the essence of computer programming."

(Brian Kernigan)

Don't hack!

"If debugging is the process of removing bugs,
then programming must be the process of putting them in."

(Edsger W. Dijkstra)

But don't feel too bad about mistakes?

"The best thing about a boolean is even if you are wrong, you are only off by a bit."

(Anonymous)

Other programming languages may offer salvation, but we don't use them

"There are only two kinds of programming languages:
those people always bitch about and those nobody uses."

(Bjarne Stroustrup)

Programming = ?

= Solving problems?

- Specify the problem and find a (software) tool to solve it…
- … if only we had tools that powerful to solve anything!
  - For certain domains: *Excel, R, Mathematica, Maple, MATLAB, TurboTax, Garmin/TomTom, Blackboard*, etc.
- … otherwise, if we can't use a tool or library we *design an algorithm*

Programming = ?

= Writing code?

- No one really writes code…
- … we write *abstract specifications* that we usually call *programs*
- … only the compiler/interpreter writes code by translating our program into *machine instructions*
- The compiler complains when your specification has *syntactic errors* or *static semantic errors*

Programming = ?

= Documenting, testing, and debugging?

- Run-time errors and logical errors are not caught by compilers
- We must specify unit and regression tests
- Unless we use Dilbert's agile programming ;-)

Programming = Specifying?

1. Specify an algorithm (design)
2. Specify a program (implementation)
3. Specify tests (testing)

Lather, rinse, repeat…

Programming = Specifying?

1. Specify an algorithm (design)
2. Specify a program (implementation)
3. Specify tests (testing)

Lather, rinse, repeat…

Questions

- How can *well-designed programming languages* prevent programming mistakes?

- How can we use *static analysis* and *formal methods* to analyze programs for errors?

- How can we *formally specify an algorithm* and generate efficient code for it?

Some Comments on Programming Language Design

- *Principle of least surprise* (POLS)
- *Uniform Access Principle* (UAP) for OOP
- No *pointer arithmetic*, references are OK
- *Static typing*
- *Orthogonality* of programming constructs
- *Exception handling*
- Support for *assertions* and *invariants* (as in Eiffel)
- And also … (this depends on the target apps)
  - *Referential transparency* (functional programming)
  - *Immutable objects* (functional programming)

There are Lots of Tools out There to Check Your Code

Most tools target C, C++, C#, and/or Java

Static analysis of source code (bug sniffing):

Lint and splint	GNU tool for bug-sniffing C
PC-lint	by Gimpel for bug-sniffing C++

Model checking (steps through every possible execution path):

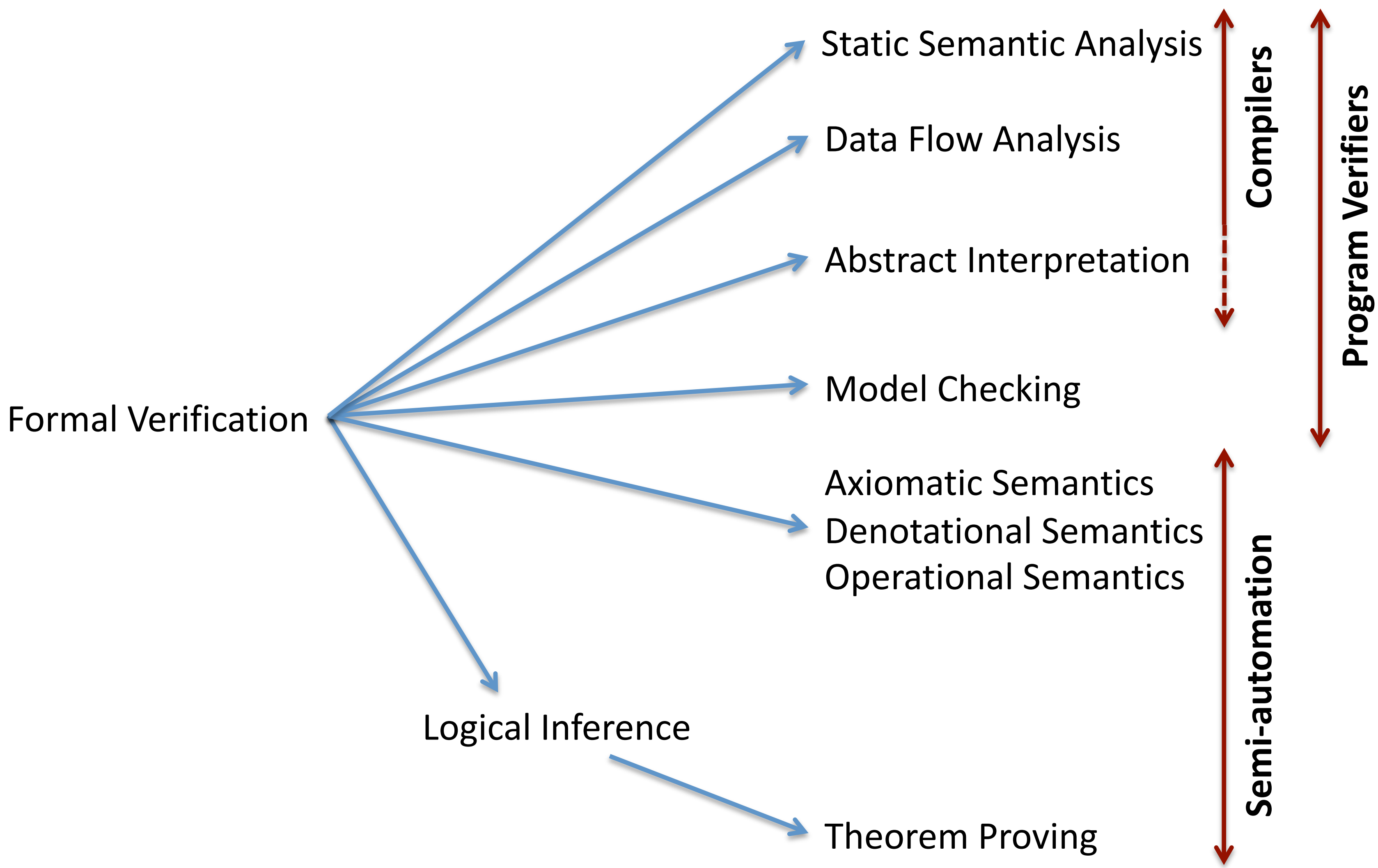
Klocwork	C/C++, C#, and Java analysis
Coverity	C/C+ analysis

Dynamic analysis (detecting memory leaks and/or race conditions)

Valgrind, Dmalloc, Insure++, TotalView

… and many more

Static Analysis



Better Programming with Tools?

A programmer once wrote

```
for( int i = 0; i < 5; i++ )
    p++;
```

He probably meant to increase the pointer p by 5

Better Programming with Tools?

```
for( int i = 0; i < 5; i++ )
    p++;
```

Fortunately, a compiler with good static analysis will optimize this to p += 5

Can it do the same for the following loop? If so, how?

```
for( int i = 0; i < n; i++ )
    p++;
```

Better Programming with Tools?

Let's rewrite

```
for( int i = 0; i < n; i++ )
    p++;
```

into the de-sugared form

```
int i = 0;
while( i < n ) {
    p = p + 1;
    i = i + 1;
}
```

is this the same as p = p + n ?

Formal Proof: Axiomatic Semantics

$\{ 0 \leq n \wedge p = q \}$	*weakest precondition*
$\{ 0 \leq n \wedge p - q = 0 \}$	
i = 0;	*apply assignment rule*
$\{ i \leq n \wedge p - q = i \}$	*loop invariant*
while(i < n) {	
$\{ i < n \wedge p - q = i \}$	*$i < n \wedge$ loop invariant*
$\{ i{+}1 \leq n \wedge p{+}1 - q = i{+}1 \}$	
p = p + 1;	*apply assignment rule*
$\{ i{+}1 \leq n \wedge p - q = i{+}1 \}$	
i = i + 1;	*apply assignment rule*
$\{ i \leq n \wedge p - q = i \}$	*loop invariant*
}	
$\{ i \geq n \wedge i \leq n \wedge p - q = i \}$	*$i \geq n \wedge$ loop invariant*
$\{ p = q + n \}$	*postcondition*

Formal Proof: Axiomatic Semantics

{ Q[V\E] }	*weakest precondition*
V* := *E	
{ Q }	*postcondition*

$\{ (!C \vee P_1) \wedge (C \vee P_2) \}$	*weakest precondition*
if (*C*) {	
$\{ P_1 \}$	*precondition of* S_1
$\mathbf{S_1}$	
{ Q }	*postcondition*
} else {	
$\{ P_2 \}$	*precondition of* S_2
$\mathbf{S_2}$	
{ Q }	*postcondition*
}	
{ Q }	*postcondition*

Formal Proof: Axiomatic Semantics

$\{ P_1 \}$	*precondition*
S_1;	
$\{ Q_1 \}$	*postcondition s.t. Q_1 implies P_2*
$\{ P_2 \}$	*precondition*
S_2;	
$\{ Q_2 \}$	*postcondition*

$\{ Inv \}$	*weakest precondition (Inv = the loop invariant)*
while (*C*) {	
$\{C \wedge Inv \}$	*precondition of S*
S	
$\{ Inv \}$	*postcondition of S*
}	
$\{ !C \wedge Inv \}$	*postcondition*

The Good, the Bad, and the Ugly

```
for( int i = 0; i < 5; i++ )
    p++;
```

is optimized by a good compiler to `p += 5;`

When we prefer elegant code, we should not have to optimize it by hand to ugly fast code: the compiler does this for you in most (but not all) cases

```
int gcd( int a, int b )
{   if (0 == b)
        return a;
    return gcd(b, a % b);
}
```

```
int gcd( int a, int b )
{   while (b != 0)
    {   register int t = b;
        b = a % b;
        a = t;
    }
    return a;
}
```

The Good, the Bad, and the Ugly

Many inefficiencies can be optimized away by compilers

But compilers optimize without regard to parallel execution!

```
x = 1;   // compiler removes this dead code
x = 0;
```

The Good, the Bad, and the Ugly

Many inefficiencies can be optimized away by compilers

But compilers optimize without regard to parallel execution!

Process 0

```
x = 1;   // removed
x = 0;
```

Process 1

```
if (x == 1)
    exit(0);
```

Syntactic Mistakes are Easy to Detect by Compilers/Static Checking Tools

```
int a[10000];

void f()
{
    int i;

    for( i = 0; i < 10000; i++ );
        a[i] = i;
}
```

Syntactic Mistakes are Easy to Detect by Compilers/Static Checking Tools

```
if( x != 0 )
    if( p ) *p = *p/x;
else
    if ( p ) *p = 0;
```

Compilers/Static Checking Tools Warn About Data Type Usage Mistakes

```
unsigned a[100] = {0};

int main()
{
   char buf[200];
   unsigned n = 0;

    while( fgets( buf, 200, stdin ) )
    {
       if( n < 100 ) a[n++] =  strlen(buf);
    }
    while( --n >= 0 )
    {
       printf( "%d\n", a[n] );
    }
    return 0;
 }
```

Static Checking Tools Warn About Data Compatibility Mistakes

```
x = 4;
if( x >= 0 )
    x = (x > 0);
else
    x = -1;
x = x % 2;
```

Static Checking Tools Warn About Execution Order Mistakes

```
void out( int n )
{
   cout << n << "\n";
}

void show( int a, int b, int c )
{
    out( a ); out( b ); out( c );
 }

 int main()
 {
    int i = 1;
    show( i++, i++, i++ );
    return 0;
 }
```

Static Checking Tools Warn About Arithmetic/Logic Mistakes

```
void print_mod( int i, int n )
{
    if( n == 0 && i == 0 ) return;
    printf( "%d mod %d == %d\n", i, n, i % n );
}

int main()
{
    for( int i = 0; i < 10; i++ )
        for( int j = 0; j < 10; j++ )
            print_mod( i, j );
    return 0;
}
```

Static Checking Tools Warn About Loop Index Mistakes

```
int a[10];

i = 0;
for( i = 0; i < 10; i++ )
    sum = sum + a[i];
weighted = a[i] * sum;
```

Static Checking Tools Warn About Data Flow Mistakes

```
int shamrock_count( int leaves, double leavesPerShamrock )
{
    double shamrocks = leaves;
    shamrocks /= leavesPerShamrock;
    return leaves;
}

int main()
{
   printf( "%d\n", shamrock_count( 314159, 3.14159 ) );
   return 0;
}
```

More Difficult: Incorrect API Logic

```
int main()
{
   FILE *fd = fopen( "data", "r" );
   char buf[100] = "";
   if( getline( fd, buf ) )
      fclose( fd );
   printf( "%s\n", buf );
}

int getline( FILE *fd, char *buf )
{
   if( fd )
   {
      fgets( buf, 100, fd );
      return 0;
   }
   return 1;
}
```

More Difficult: Dynamic Typing/Data Flow Mistakes

```
class BankAccount

  def accountName
      @accountName = "John Smith"
  end

  def deposit
      @deposit
  end

   def deposit=(dollars)
       @deposit = dollars
   end

   def initialize ()
       @deposet = 100.00
   end

   def test_method
       puts "The class is working"
       puts accountName
   end
end
```

Abstract Interpretation

```
int[] a = new int[10];
i = 0;

while( i < 10 ) {

   ... a[i] ...

   i = i + 1;
}
```

What is the range of i at each program point?

Is the range of i safe to index a[i]?

Abstract Interpretation

Define a lattice:
$[a,b] \sqcup [a',b'] = [\min(a,a'), \max(b,b')]$
$[a,b] \sqcap [a',b'] = [\max(a,a'), \min(b,b')]$

```
    int[] a = new int[10];
    i = 0;
p0:
    while( i < 10 ) {
p1:
      ... a[i] ...
p2:
      i = i + 1;
    }
p3:
```

p_0: i = [0,0]

p_1:
i = $[0,0] \sqcap [-\infty,9] = [0,0]$
i = $[0,0] \sqcup [1,1] \sqcap [-\infty,9] = [0,1]$
… use *acceleration* to determine finite convergence
i = [0,9]

p_2:
i = $[1,1] \sqcap [-\infty,9] = [1,1]$
i = $[1,1] \sqcup [2,2] \sqcap [-\infty,9] = [1,2]$
… use *acceleration* to determine finite convergence
i = [1,10]

p_3: i = $[1,10] \sqcap [10,+\infty] = [10,10]$

Model Checking

Process 1

p_0:while(x > 0) {

p_1: *use resource*

x = x + 1;

}

p_2:sleep;

Process 2

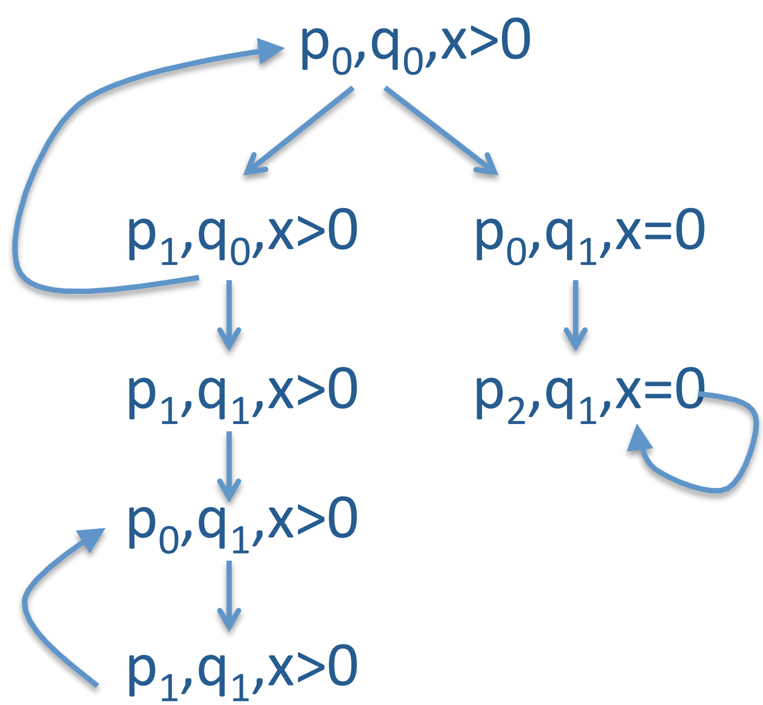
q_0:x = 0;

q_1:*use resource forever*

Q: starting with x > 1, will p_1 ever concurrently execute with q_1?

Q: will execution reach a state where x stays 0?

Model (using abstract traces where x is positive or 0)



Related Examples

C with assertion

```
for (i = 1; i < N/2; i++) {
  k = 2*i – 1;
  assert(k >= 0 && k < N);
  a[k] = …
```

Programmer moved assertion

```
for (i = 1; i < N/2; i++) {
  assert(i > 0 && 2*i < N+1);
  k = 2*i+1;
  a[k] = …
```

Can we move the assertion before the loop? If so, how?

Related Examples

Eiffel "design by contract"

```
indexing ... class
      COUNTER
feature
      ...
invariant

      item >= 0

end
```

Methods must obey invariant

```
decrement is
         -- Decrease counter by one.
   require
         item > 0
   do
        item := item - 1
   ensure
        item = old item - 1
   end
```

Generating Correct and Efficient Code from High-Level Specifications

$$p = \int_0^1 \int_y^1 \frac{\partial u}{\partial x} \, dy \, dz \qquad \forall (x, y) \in \Omega_{x,y} \qquad (1)$$

Ctadel

parallel HPF code

```
      PROGRAM P3
      REAL u(0:n+1,m,l),g(0:n+1),h,p(0:n+1,m),q(0:n+1)
      REAL s(0:n+1),t(0:n+1,m)
      ...
CMIC$ PARALLEL ...
CMIC$ CASE
      DO 2300 j = 1,m
       FORALL(i=0:n+1) s(i)=s(i)+u(i,j,l)
2300  CONTINUE
CMIC$ CASE
      DO 2310 j = m,2,-1
       FORALL(i=1:n+1) t(i,j-1)=t(i,j)
       DO 2320 k = 1,l
        FORALL(i=1:n+1) t(i,j-1)=t(i,j-1)+u(i,j,k)
2320   CONTINUE
2310  CONTINUE
CMIC$ END CASE
CMIC$ END PARALLEL
CMIC$ PARALLEL ...
CMIC$ CASE
      FORALL(i=1:n) q(i)=g(i)*(s(i)-s(i-1))/h
CMIC$ CASE
      FORALL(i=1:n+1,j=1:m) t(i,j)=t(i,j)/h
CMIC$ END CASE
CMIC$ END PARALLEL
      FORALL(i=1:n,j=1:m) p(i,j)=t(i+1,j)-t(i,j)
```

Generating Correct and Efficient Code from High-Level Specifications

Algorithm: $[A] := \mathrm{LU\_BLK\_VAR1}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$A_{01} := U_{01} = L_{00}^{-1} A_{01}$

$A_{10} := L_{10} = A_{10} U_{00}^{-1}$

$A_{11} := \mathrm{LU}(A_{11} - L_{10} U_{01})$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

FLAME → library code

Some Concluding Remarks

- Static checking requires all compiler stages except the back-end
- Static checkers find deviations from “best practices”
- Static checkers may find many false alarms
- There are very few checkers for scripting languages (Perl, Python, Ruby, …)
- Functional languages (Haskell, ML, …) are generally safer, but do not prevent logical programming errors