Automatic verification of textbook programs that use comprehensions.

<u>Rosemary Monahan</u>

National University of Ireland, Maynooth, Ireland

K. Rustan M. Leino,

Microsoft Research, Redmond, USA

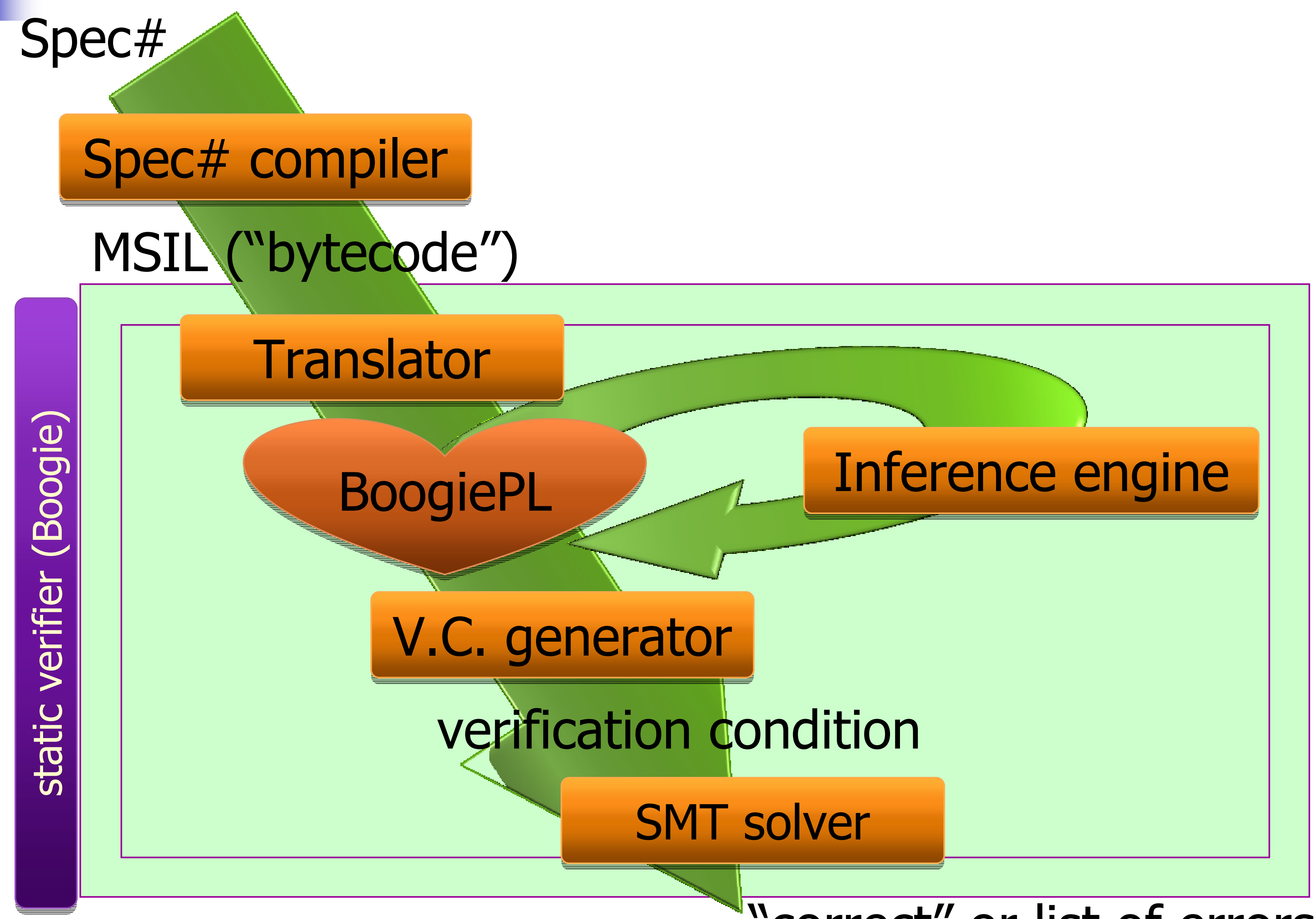
Presentation Overview

- Supporting comprehensions in Spec#
- Encoding comprehensions as first-order expressions
  - Comprehension Functions
  - Matching Triggers
  - Axioms and their Adequacy
- Verification of examples from *A Method of Programming* by Dijkstra and Feijen.
- Evaluation & Conclusions

Spec# Programming System

- Mix of contracts and tool support
- Superset of C#
  - non-null types, pre- and postconditions, object invariants
- Tool support
  - more type checking
  - compiler-emitted run-time checks
  - static program verification
    - sound modular verification
    - focus on automation of verification rather than full functional correctness of specifications

Spec# Verifier Architecture



With thanks to Microsoft Research

Supporting Comprehensions in the Spec# Language

Spec# Example

```
public static int SegSum(int[] a, int i, int j)
requires 0<=i  &&  i <= j  &&  j <= a.Length;
ensures result == sum{int k in (i:j); a[k]};
 {     int s = 0;
       for (int n = i; n < j; n++)
       invariant i <= n && n <= j;
       invariant s == sum{int k in (i:n); a[k]};
       {      s += a[n];
       }
        return s;
}
```

Comprehensions in Spec#

Q{ K k **in** E, F; T }

- sum {int k in (i:n); a[k]};
- product {int k in (1..n); k};
- min {int k in (0:a.Length); a[k]};
- sum {int k in (0:a.Length), i<=k && k <j ; a[k]};
- count {int k in (0: n); ((a[k] % 2)== 0)};
- max {int k in (0:a.Length), Even(a[k]); a[k]};

(or forall, exists or exists-unique but those forms have counterparts in first-order logic)

Boogie

The Spec# static program verifier

- Translates compiled Spec# programs into the intermediate verification language BoogiePL
  - Includes functions and axioms
  - Its expressions include logical quantifiers and arithmetic
- Generates verification conditions for Satisfiability Modulo Theories (SMT) solvers
  - Maps core language into first-order formulae using wp calculus
- Does not supply direct support for comprehensions, so the translation from Spec# to BoogiePL must use some suitable encoding

Encoding Comprehensions as First Order Expressions

Mathematical properties

Empty range for sum

$\forall$ lo, hi • hi <= lo $\Rightarrow$ sum {int k in(lo:hi); a[k]} = 0

Induction for sum

$\forall$ lo, hi • lo <= hi $\Rightarrow$

sum {int k in(lo:hi+1); a[k]}

= sum {int k in(lo:hi); a[k]} + a[hi]

Comprehension Translation

Introduce and axiomatise one BoogiePL function for each different *comprehension template* occurring in the Spec# program.

Example:

```
ensures result == sum{int k in (i:j), true; a[k]};
```

The BoogiePL translations of:

int k in (i:j), true, a[k] are

i, j, true, ArrayGet($Heap[a, $elements], k)]

Example:

sum{int k in (i:j); a[k]}

- Comprehension template

  (sum, ☐, ArrayGet(☐, k))

- Comprehension function

```
function sum#0(i:int, j : int, a0 :bool, a1:Elements)
   returns (int);
```

- Translate to BoogiePL

```
sum#0(i, j, true, $Heap[a, $elements])
```

Axioms

- For each comprehension function, our translation also generates a number of axioms.
- Quantifier instantiation via e-graph matching
- A *matching pattern (trigger)* is a set of terms that together mention all the bound variables, none of which is just a bound variable by itself
- Examples:
  - $(\forall x :: \{ f(x) \}\ \ 0 \leq f(x))$
  - $(\forall x,y :: \{ g(x,y) \}\ \ f(x) < g(x,y))$

Triggers

- Fragile e.g +

  $\forall x{:}int \bullet \{g(x+1)\}\ h(x) = g(x+1)$

  doesn't match g(2+y-1) or g(1+y)
- Not limiting enough

  $\forall x{:}int \bullet \{h(x)\}\ h(x) < h(k(x))$
  - matches any argument of h
  - the instantiation produces a term with another argument of h
  - if h(x) occurs in the e-graph, then this quantifier will be instantiated with x, k(x), k(k(x)), … causing a matching loop

Axioms

- For every comprehension template, our encoding introduces not one, but two function symbols sum#n and s#n.

- We axiomatise these to be synonyms of each other

  (∀ lo:int, hi :int, aa:T • {sum#n(lo, hi, aa)} sum#n(lo, hi, aa) = s#n(lo, hi, aa))

Unit Axiom

$\forall$ lo: int, hi : int, aa:T • {s#n(lo, hi, aa)}

($\forall$ k: int • lo <= k $\wedge$ k < hi $\Rightarrow$ $\neg$Filter [aa, k])

$\Rightarrow$ s#n(lo, hi, aa) = 0

- Empty range property is a special case
- Trigger says for the outer quantifier to be instantiated for every occurrence of s#n
- The inner quantifier appears in a negative position so we need not worry about triggers for it.

Induction

- Susceptible to matching loops
- Limit each sum#n expression in the input to one instantiation of each induction axiom
- Achieved by mentioning sum#n, not s#n, in the triggers
- We provide four induction axioms altogether
  - **induction below** relates
    s#n(lo, hi, aa) and s#n(lo + 1, hi, aa)
  - **induction above** relates
    s#n(lo, hi, aa) and s#n(lo, hi – 1, aa)

Induction Below Axiom

$\forall$ lo: int, hi : int, aa:T • {sum#n(lo, hi, aa)}

lo < hi $\wedge$ Filter [aa, lo]

$\Rightarrow$

s#n(lo, hi, aa) =

s#n(lo + 1, hi, aa) + Term[aa, lo]

For 2nd part negate Filter[aa, lo]) and drop + Term[aa, lo]

Induction Above Axiom

$\forall$ lo: int, hi : int, aa:T • {sum#n(lo, hi, aa)}

lo < hi $\wedge$ Filter [aa, hi-1]

$\Rightarrow$ s#n(lo, hi , aa) = s#n(lo, hi –1, aa) + Term[aa, hi -1]

For 2nd part negate Filter[aa, hi -1]) & drop + Term[aa, hi -1]

Alternative triggers avoid matching loops but are fragile

- s#n(lo + 1, hi, aa)
- s#n(lo, hi – 1, aa)

Split Range Axiom

$\forall$ lo:int, mid :int, hi :int, aa:T •

{sum#n(lo, mid, aa), sum#n(mid, hi, aa)}

{sum#n(lo, mid, aa), sum#n(lo, hi, aa)}

lo <= mid $\wedge$ mid <= hi

$\Rightarrow$

s#n(lo, mid, aa) + s#n(mid, hi, aa) = s#n(lo, hi, aa)

Comments on Triggers

- Each trigger mentions two terms, because there is no single term that covers all bound variables

- The trigger {sum#n(lo, hi, aa), sum#n(mid, hi, aa)} is omitted due to its impact on performance

- The triggers use sum#n, despite the fact that using s#n would not lead to any matching loop.
  - Using s#n has a detrimental impact on performance (by as much as a factor of 10 for our examples)

Same Term Axiom

$\forall$ lo:int, hi:int, aa:T, bb:T •

{sum#n(lo, hi, aa), s#n(lo, hi, bb)}
($\forall$k: int • lo <= k < hi $\Rightarrow$
Filter[aa, k] $\equiv$ Filter[bb, k] $\wedge$
Filter[aa, k] $\Rightarrow$ Term[aa, k] = Term[bb, k])

$\Rightarrow$ s#n(lo, hi, aa) = s#n(lo, hi, bb))

Same Term Axiom …

- The inner quantifier appears in a negative position
  - so we need not worry about a trigger for it
- For the outer quantifier, we could have chosen the trigger {s#n(lo, hi , aa), s#n(lo, hi , bb)}.
  - the trigger with two s#n terms gave rise to unacceptable performance
  - so we chose to use sum#n in one of the terms
- We also tried the trigger {sum#n(lo, hi , aa), sum#n(lo, hi , bb)}
  - but that was too restrictive for our example programs

Distribution (of plus over min/max)

$\forall$ lo: int, hi: int, aa:T, bb:T, D: int $\bullet$

{min#n(lo, hi, aa) + D, m#n(lo, hi, bb)}

($\forall$ k: int $\bullet$ lo <= k $\wedge$ k < hi $\Rightarrow$

(Filter [aa, k] $\equiv$ Filter [bb, k]) $\wedge$

(Filter [aa, k] $\Rightarrow$ Term[aa, k] + D = Term[bb, k]))

$\wedge$

($\exists$ k: int $\bullet$ lo <= k $\wedge$ k < hi $\wedge$ Filter [aa, k] $\wedge$

Term[aa, k] + D = Term[bb, k])

$\Rightarrow$ m#n(lo, hi, aa) + D = m#n(lo, hi, bb)

Triggers

- The nested universal quantifier appears in a negative position
  - so we need not worry about a trigger for it
- The trigger for the existential quantifier matters
  - what makes a good trigger for it depends on the comprehension template - we specify no trigger but include Term[aa, k] + D = Term[bb, k] to give the SMT solver a chance of finding a trigger
- The trigger of the outer quantifier is problematic
  - it mentions + and is therefore fragile rendering the axiom useless for Z3.

Adequacy of Axiomatisation 1

- All axioms concern just one comprehension function
- No axiom relates two different comprehension functions
  - sum{int k in (i:j); a[k]};

    sum#0(i, j, true, $Heap[a, $elements])
  - sum{int k in (0:a.Length), i<= k && k<j ; a[k]};

    sum#1(0, $ArrayLength(a), i, j,
    $Heap[a,$elements])

Adequacy of Axiomatisation 2

- Using sum#n instead of s#n in some triggers limits the number of quantifier instantiations.
  - However, the instantiations are adequate for all of the examples we tried.
- Using Simplify as the SMT solver, we have not experienced any problems with the fragile trigger of the **distribution** axiom.
- The lack of the **distribution** axiom for Z3 means that it cannot verify examples like Minimal Segment Sum.

Adequacy of Axiomatisation 3

- Ranges of size 0 or 1 can be addressed by the **unit** and **induction** axioms
- All larger ranges can be addressed by decomposing them into smaller ranges with the **split range** axiom
- An induction axiom that enlarges the range at the lower end, as in (lo-1:h) is not needed
  - reason about the ranges

    (lo: lo+1) and (lo +1:hi)
  - use the **split range** axiom

Triggers are an issue.

```
public int ReverseSum(int[] a)
ensures result == sum{int i in (0: a.Length); a[i]};
{  int s = 0;
   for (int n = a.Length; 0 < = --n; )
   invariant 0 <= n && n <= a.Length;
   invariant s == sum{int i in (n: a.Length); a[i]};
   {
        s += a[n];
   }
   return s;
}
```

Triggers are an issue!

```
public int ReverseSum(int[] a)
ensures result == sum{int i in (0: a.Length); a[i]};
{  int s = 0;
   for (int n = a.Length; 0 < = --n; )
   invariant 0 <= n && n <= a.Length;
   invariant s == sum{int i in (n: a.Length); a[i]};
   {
      assert a[n] == sum{int i in (n: n+1); a[i]};
      s += a[n];
   }
   return s;
}
```

Prover directive to trigger instantiation of the **induction** axiom

Some More Difficult Examples

Loop Iterations

Coincidence Count

Minimal Segment Sum

...

Loop Iterations

```
public static int Sum0(int[ ] a)
ensures result == sum{int i in (0 : a.Length); a[i ]};
{   int s = 0;
    for (int n = 0; n < a.Length; n++)
    invariant n <= a.Length && s == sum{int i in (0 : n); a[i ]};
    {
        s += a[n];
    }
    return s;
}
```

Loop Iterations

```
public static int Sum1(int[ ] a)
ensures result == sum{int i in (0 : a.Length); a[i ]};
{   int s = 0;
    for (int n = 0; n < a.Length; n++)
    invariant n <= a.Length &&
            s + sum{int i in (n : a.Length); a[i ]}
                            == sum{int i in (0: a.Length); a[i ]}
    {
            s += a[n];
    }
    return s;
}
```

Loop Iterations

```
public static int Sum2(int[ ] a)
ensures result == sum{int i in (0 : a.Length); a[i ]};
{   int s = 0;
    for (int n = a.Length;0 <= --n;)
    invariant 0<= n && n <= a.Length &&
                        s == sum{int i in (n: a.Length); a[i ]};
    {
            s += a[n];
    }
    return s;
}
```

Loop Iterations

```
public static int Sum3(int[ ] a)
ensures result == sum{int i in (0 : a.Length); a[i ]};
{   int s = 0;
    for (int n = a.Length; 0<= --n;)
    invariant 0<= n && n<= a.Length &&
          s + sum{int i in (0 : n); a[i ]}
                          == sum{int i in (0: a.Length); a[i ]}
    {
          s += a[n];
    }
    return s;
}
```

Coincidence Count

```
public int CoincidenceCount(int[] f, int[] g)

requires
   forall{int i in (0:f.Length),
                    int j in (i+1:f.Length), i <j; f[i] < f[j]};
   forall{int i in (0: g.Length),
                    int j in (i+1:g.Length ), i <j; g[i] < g[j]};
ensures
   result == count{int i in (0:f.Length),
                    int j in (0:g.Length); f[i] == g[j]};
```

Coincidence Count

- Inefficient version
- Efficient version
  - Initial attempts required many Spec# assertions
  - Using two triggers for the split range axiom eliminates the need for Spec# assertions

    {sum#n(lo,mid, aa), sum#n(mid, hi, aa)}

    {sum#n(lo, mid, aa), sum#n(lo, hi, aa)}
- Efficient version using an alternative invariant

Inefficient Version:Invariant

```
m <= f.Length || n <= g.Length;

ct ==
   count {int i in (0:m), int j in (0:n); f[i] == g[j]};

m == f.Length || forall {int j in (0:n); g[j] < f[m]}
n == g.Length || forall {int j in (0:m); f[i] < g[n]}
```

Inefficient Version:Program

```
int ct = 0; int m = 0; int n = 0;
while (m < f.Length || n < g.Length)
{
   if (n == g.Length) ||(m < f.Length && f[m] < g[n])
        m++;
   else if (m == f.Length) || ( n < g.Length && g[n] < f[m])
        n++;
   else // (g[n] == f[m])
   {
        ct++;m++;n++;
   }
   return ct;
 }
```

Efficient Version:Invariant

m <= f.Length && n <= g.Length;

Change from || to &&

ct ==

count {int i in (0:m), int j in (0:n); f[i] == g[j]};

m == f.Length || forall {int j in (0:n); g[j] < f[m]}

n == g.Length || forall {int j in (0:m); f[i] < g[n]}

Efficient Version:Program

```
int ct = 0; int m = 0; int n = 0;
while (m < f.Length || && n < g.Length)
{
   if (n == g.Length) ||(m < f.Length && f[m] < g[n])
         m++;
   else if (m == f.Length) || ( n < g.Length && g[n] < f[m])
         n++;
   else // (g[n] == f[m])
   {
         ct++;m++;n++;
   }
   return ct;
}
```

Alternative Invariant

```
ct + count{int i in (m:f.Length),
           int j in (n:g.Length); f[i] == g[j]}
==
count{int i in (0:f.Length),
      int j in (0:g.Length); f[i] == g[j]};
```

Using Spec#

➔

Demonstration of invoking the compiler and Boogie to verify a program that uses comprehensions

Evaluation: Performance

- Acceptable with the two first order SMT solvers, Simplify and Z3.
- In most cases, the Z3 solver verifies the programs slightly faster than Simplify.
- Z3 cannot verify our Factorial or MinSegmentSum examples
  - multiplications by non-constants
  - distribution of + over the min comprehension
- Z3 cannot verify CoincidenceCount1
  - If we remove the first of the two triggers for the **split range** axiom for the outer count comprehension, Z3 verifies the program in less than 2 seconds.
  - The problem therefore seems related to the first of these triggers setting off a chain of instantiations that prevent Z3 from completing the verification.

Performance

Program	Simplify	Z3
Sum0	0.219s	0.172s
Sum1	0.063s	0.016s
Sum2	0.047s	0.016s
Sum3	0.110s	0.016s
Factorial	0.172s	
MinSegmentSum	16.063s	
CoincidenceCount0	6.017s	1.815s
CoincidenceCount1	18.970s	
CoincidenceCount2	12.907s	1.16s

Measurements (in seconds) of verification performance on a Core 2 Duo laptop, running at 2.33GHz with a 4 MB L2 cache and the current version of Spec#.

Conclusions

- Implemented support for summation-like comprehensions in an automatic program verifier
- We need (and welcome help with)
  - More informative error messages
  - More case studies & examples
  - Support for mathematical data structures and abstraction
- http://research.microsoft.com/specsharp
- http://www.cs.nuim.ie/~rosemary/