Language Tools for Distributed Computing and Program Generation (III)

Morphing:
Bringing Discipline to Meta-Programming

Yannis Smaragdakis
University of Oregon

These Lectures

- NRMI: middleware offering a natural programming model for distributed computing
  - solves a long standing, well-known open problem!
- J-Orchestra: execute unsuspecting programs over a network, using program rewriting
  - led to key enhancements of a major open-source software project (JBoss)
- Morphing: a high-level language facility for safe program transformation

Program Generation

- The kinds of techniques used in J-Orchestra, JBoss AOP, etc. are an instance of program generation
  - program generators = programs that generate other programs
- This is a research area that I have worked on for a long time
- Next, I'll give a taste of why the area inspires me and what research problems are being solved

Why Do Research on Program Generators?

- intellectual fascination
  - "If you are a Computer Scientist, you probably think computations are interesting. What then can be more interesting than computing about computations?"
- practical benefit
  - many software engineering tasks can be substantially automated

Sensationalist Program Generation

- You know what I mean if you feel anything when you look at a self-generating program

- ```
  ((lambda (x) (list x (list (quote quote) x)))
  (quote (lambda (x) (list x (list (quote quote) x)))))
  ```

- ```
  main(a){a="main(a){a=%c%s%c;printf(a,34,a,34);}";
  printf(a,34,a,34);}
  ```

Why Write a Generator?

- Any approach to automating programming tasks may need to generate programs
- Two main reasons (if we get to the bottom)
  - performance (code specialization)
  - conformance (generate code that interfaces with existing code)
    - e.g., generating code for J2EE protocols in JBoss
    - widespread pattern of generation today: generators that take programs as input and inspect them

A (Big) Problem

- Program generation is viewed as an inherently complex, “dirty”, low-level trick
- Hard to gain the same confidence in a generated program as in a hand-written program
  - even for the generator writer: the inputs to the generator are unknown
- Much of my work is on offering support for ensuring generators work correctly
  - necessary, if we want to move program generation to the mainstream
  - make sure generated program free of typical static “semantic” errors (i.e., it compiles)

Meta-Programming Introduction

- Tools for writing **program generators**: programs that generate other programs
- `` `[…] `` (quote)

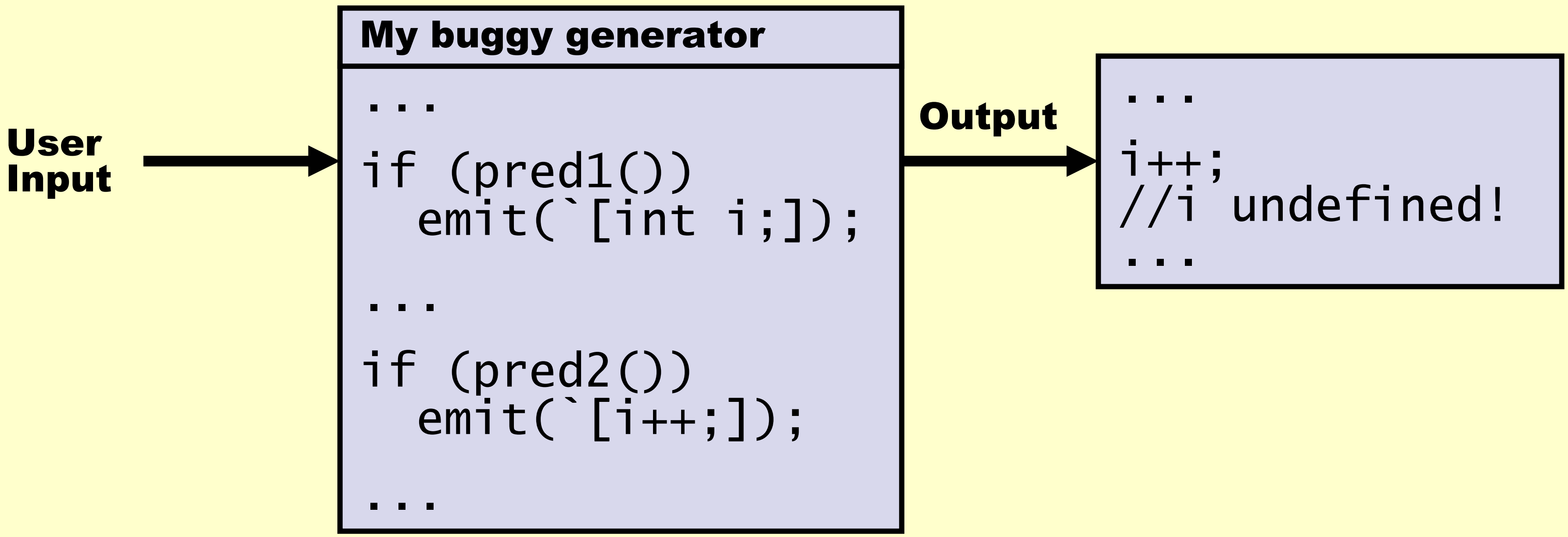
```
expr = `[7 + i];
```

- `#[…]` (unquote)

```
stmt = `[if (i > 0) return #[expr]; ];
```

```
stmt <= `[if (i > 0) return 7 + i;]
```

An Unsafe Generator



- Error in the generator: `pred2()` does not imply `pred1()` under ALL inputs.

Statically Safe Program Generation

- Statically check the ***generator*** to determine the safety of any ***generated program***, under **ALL** inputs***.***
- Specifically, check the generator to ensure that generated programs compile

Why Catch Errors Statically?

- "After all, the generated program will be checked statically before it runs"
  - Errors in generated programs are really errors in the generator.
  - compile-time for the generated program is *run-time* for the generator!
- Statically checking the generator is analogous to static typing for regular programs

The Problem:

- Asking whether generated program is well-formed is equivalent to asking any hard program analysis question (generally undecidable).
- Control Flow

```
if (pred1()) emit (`[int i;]);
if (pred2()) emit (`[i++;]);
```

- Data Flow

```
emit ( `[ int #[name1];
          int #[name2]; );
```

Early Approach: SafeGen

- A language + verification system for writing program generators
  - generator Input/Output: legal Java programs
- Describe everything in first-order logic
  - Java well-formedness semantics: `axioms`
  - structure of generator/generated code: `facts`
  - type property to check: `test`
  - conjecture: `(axioms ∧ facts) → test`
- Prove conjecture valid using automatic theorem prover: SPASS
  - a great way to catch bugs in the generator that only appear under specific inputs

SafeGen

- Input/Output: legal Java programs.
- Controlled language primitives for control flow, iteration, and name generation.

Example:

"Iterate over all the methods from the input class that have one argument that is public and a return type, such that it has at least one method with an argument that implements java.io.Serializable"

Generator: Signature

```
#defgen makeInterface (Interface i) {
   public interface Foo extends #[i.Name] {
      ...
   }
}
```

- keyword #defgen, name
- input: a single entity, or a set or entities.

Inside the Generator

```
#defgen makeInterface (Interface i) {
   public interface Foo extends #[i.Name] {
      ...
   }
}
```

- Between the `{ … }`
  - Any legal Java syntax
  - "escapes":
    - `#[…], #foreach, #when, #name["…"]`

#[...]

- Splice a fragment of Java code into the generator.

```
interface Bar {
    ... }
```

input

```
#defgen makeInterface ( Interface i ) {
   public interface Foo extends #[i.Name] {
     ...
   }
}
```

output

```
public interface Foo extends Bar {
      ...
}
```

#foreach

- Takes a set of values, and a code fragment. Generate the code fragment for each value in the set.

```
interface Bar {
   int A (float i);
   float B (int i);
}
```

input

```
rface ( Interface i ) {
   public interface Foo extends #[i.Name] {
     #foreach (Method m : MethodOf(m, i) ) {
       void #[m.Name] ();
     }
   } }
```

output

```
public interface Foo extends Bar {
   void A ();
   void B ();
}
```

Cursors

- A variable ranging over all entities satisfying a first-order logic formula.
  - predicates and functions correspond to Java reflective methods: `Public(m), MethodOf(m, i), m.RetType,` etc.
  - FOL keywords: `forall, exists, &, |,`etc.

```
#foreach (Method m : MethodOf(m, i)) {
   ...
}
```

A Conjecture – In English

Given that all legal Java classes have unique method signatures, ***(axiom)***

given that we generate a class with method signatures isomorphic to the method signatures of the input class ***(fact)***

can we prove that the generated class has unique method signatures? ***(test)***

Phase I: Gathering Facts

```
#defgen makeInterface ( Interface i ) {
  public interface Foo extends #[i.Name] {
    #foreach (Method m : MethodOf(m, i)) {
       void #[m.Name] ();
     } } }
```

$\exists i\,($ Interface(i) $\wedge$

$\exists i'$ (Interface(i') $\wedge$ name(i') = Foo $\wedge$ SuperClass(i') = i $\wedge$

$(\forall m$ (MethodOf(m, i) $\leftrightarrow$

$(\exists m'$ (MethodOf(m', i') $\wedge$ RetType(m') = void $\wedge$

name (m') = name(m) $\wedge$

$\neg(\exists t$ ArgTypeOf(t, m'))))))))

Phase II: Constructing Test

```
#defgen makeInterface ( Interface i ) {
  public interface Foo extends #[i.Name] {
    #foreach (Method m : MethodOf(m, i)) {
       void #[m.Name] ();
     } } }
```

$\exists$i (Interface (i) $\wedge$
$\exists$i' ((Interface(i') $\wedge$ name(i') = Foo $\wedge$
$\forall$m (MethodOf(m, i') $\rightarrow$
$\neg$($\exists$m' MethodOf(m', i') $\wedge$ **$\neg$(m = m') $\wedge$**
name(m') = name(m) $\wedge$
ArgTypes(m') = ArgTypes(m)))))

When Does It Fail?

```
interface Bar {
    int A (float i);
    float A (int i);
}
```

input

```
#defgen makeInterface ( Interface i ) {
    public interface Foo extends #[i.Name] {
      #foreach (Method m : MethodOf(m, i)) {
        void #[m.Name] ();
      }
    } }
```

output

```
public interface Foo extends Bar {
    void A ();
    void A ();
}
```

SafeGen Safety

- Checks the following properties:
  - A declared super class exists
  - A declared super class is not `final`
  - Method argument types are valid
  - A returned value's type is compatible with method return type
  - Return statement for a `void`-returning method has no argument

Experience w/ Theorem Provers

- We tried several theorem provers:
  - Hand-constructed axioms, facts, and tests for common bugs and generation patterns.
  - Criteria: ability to reason without human guidance and terminate.
  - SPASS became the clear choice.

Overall Experience

- We had predefined a set of ~25 program generation tasks
  - pre-selected *before* SafeGen was even designed
- SafeGen reported all errors correctly, found proofs for correct generators
  - all proofs in under 1 second
- SafeGen terminated 50% of the time with a proof of error, when one existed
  - it could conceivably fail to prove a true property and issue a false warning

Do We Really Want Theorem Provers for This?

- The SafeGen approach is effective
- But the whole point was to offer certainty to the programmer
- Theorem proving is an incomplete approach, which is not intuitively satisfying
  - no clear boundary of incompleteness: just that theorem prover ran out of time
- Can we get most of the benefit with a type system?

Morphing: Shaping Classes in the Image of Other Classes

The MJ Language

WARNING: The examples are important. Keep me honest!

Morphing: MJ

- Static reflection over members of type params

- ```
  class MethodLogger<class X> extends X {
    <Y*>[meth] for(public int meth (Y) : X.methods)
    int meth (Y a) {
      int i = super.meth(a);
      System.out.println("Returned: " + i);
      return i;
    }
  }
  ```
  - Other extensions (over Java) in this example?
  ```

Real-World Example (JCF)

- ```
  public class MakeSynchronized<X> {
    X x;
    public MakeSynchronized(X x) { this.x = x; }

    <R,A*>[m] for(public R m(A) : X.methods)
    public synchronized R m (A a) {
      return x.m(a);
    }

    <A*>[m] for(public void m(A) : X.methods)
    public synchronized void m(A a) {
      x.m(a);
    }
  }
  ```
  - 600 LOC in class Collections, just to do this
  ```

More Morphing / MJ

```
public class ArrayList<E> extends AbstractList<E> ...  {
  ...
  <F extends Comparable<F>>[f]for(public F f : E.fields)
  public ArrayList<E> sortBy#f () {
    public void sortBy#f () {
      Collections.sort(this,
        new Comparator<E> () {
          public int compare(E e1, E e2) {
            return e1.f.compareTo(e2.f);
          }
        });
    }
  }
}
```

Modular Type Safety

- Our theorem of generator safety *for all inputs*, is *modular type safety* in MJ
  - the generic class is verified on its own (not when type-instantiated)
  - type error if *any* type parameter can cause an error
  - can distribute generic code with high confidence

Type Errors?

- ```
  class CallWithMax<class X> extends X
  {
    <Y>[meth]for(public int meth(Y) : X.methods)
    int meth(Y a1, Y a2) {
      if (a1.compareTo(a2) > 0)
        return super.meth(a1);
      else
        return super.meth(a2);
    }
  }
  ```
  - Where is the bug?
    - where is the other bug?
  ```

Once More...

- ```
  public class AddGetSet<class X> extends X
  {
    <T>[f] for(T f : X.fields) {|
      public T get#f () { return f; }
      public void set#f (T nf) { f = nf; }
    |}
  }
  ```

  - Where is the bug?
  ```

Filter Patterns

- ```
  public class AddGetSet2<class X> extends X
  {
    <T>[f] for(    T f : X.fields ;
            no get#f() : X.methods)
    public T get#f () { return f; }

    <T>[f] for(    T f  : X.fields ;
            no set#f(T) : X.methods)
    public void set#f (T nf) { f = nf; }
  }
  ```

  - keywords “some”, “no”
  ```

Type Checking in More Detail

Validity and Well-definedness without Filter Patterns

Well-Definedness (Single Range)

- ```
  class CopyMethods<X> {
    <R,A*>[m] for( R m (A) : X.methods)
    R m (A a) { ... }
  }
  ```
  - Uniqueness implies uniqueness
    - what if I am mangling signatures?

- ```
  class ChangeArgType<X> {
    <R,A>[m] for ( R m (A) : X.methods)
    R m ( List<A> a ) { ... }
  }
  ```
    - example of problems?

Validity

- ```
  class InvalidReference<X> {
    Foo f; ... // code to set f field
    [n] for( void n (int) : X.methods )
    void n (int a) { f.n(a); }
  }

  class Foo {
    void foo(int a) { ... }
  }
  ```
  - Any problems?
  ```

Easy-to-Show Validity

- ```
  class EasyReflection<X> {
    X x; ... // code to set x field

    [n] for( void n (int) : X.methods )
    void n (int i) { x.n(i); }
  }
  ```
  ```

Validity in Full Glory

- ```
  class Reference<X> {
    Declaration<X> dx; ... //code to set dx
    <A*>[n] for( String n (A) : X.methods )
    void n (A a) { dx.n(a); }
  }
  class Declaration<Y> {
    <R,B*>[m] for( R m (B) : Y.methods )
    void m (B b) { ... }
  }
  ```

  - type checking: range subsumption
  - range R1 subsumes R2 if patterns unify (one way)
  - what are the patterns above?
  ```

Well-Definedness

- ```
  class StaticName<X> {
    int foo () { ... }

    <R,A*>[m]for (R m (A) : X.methods)
    R m (A a) { ... }
  }
  ```

  - Ok?
  ```

Less Clear When Doing Type Manipulation

- ```
  class ManipulationError<X> {
    <R>[m] for (R m (List<X>) : X.methods)
    R m (List<X> a) { ... }

    <P>[n] for (P n (X) : X.methods)
    P n (List<X> a) { ... }
  }
  ```

  - Any problems?
  ```

Fixing Previous Example

- ```
  class Manipulation<X> {
    <R>[m] for (R m (List<X>) : X.methods)
    R list#m (List<X> a) { ... }

    <P>[n] for ( P n (X) : X.methods)
    P nolist#n (List<X> a) { ... }
  }
  ```
  ```

Two Way Unification?

- ```
  class WhyTwoWay<X> {
    <A1,R1> for ( R1 foo (A1) : X.methods)
    void foo (A1 a, List<R1> r) { ... }

    <A2,R2> for ( R2 foo (A2) : X.methods)
    void foo (List<A2> a, R2 r) { ... }
  }
  ```

  - Any problems?
  ```

Now Add Filters...

Positive Filter Patterns

```
• public class DoBoth<X,Y> {
    <A*>[m] for(static void m(A):X.methods;
               some static void m(A):Y.methods)
    public static void m(A args) {
      X.m(args);
      Y.m(args);
    }
  }
```

Rules

- ⟨P1,+F1⟩ subsumes ⟨P2,+F2⟩ if P1 subsumes P2, and F1 subsumes F2.
- ⟨P1,−F1⟩ subsumes ⟨P2,−F2⟩ if P1 subsumes P2, and F2 subsumes F1.

- ⟨P1, ?F1, G1⟩ is disjoint from ⟨P2, ?F2, G2⟩ if G1 is disjoint from G2.
- ⟨P1, ?F1, G1⟩ is disjoint from ⟨P2, −F2, G2⟩ if F2 subsumes P1.
- ⟨P1, +F1, G1⟩ is disjoint from ⟨P2, −F2, G2⟩ if F2 subsumes F1.

Comprehensive Example

- ```
  public class UnionOfStatic<X,Y> {
    <A*>[m] for(static void m (A):X.methods)
    static void m(A args) { X.m(args); }

    <B*>[n] for(static void n (B):Y.methods;
          no static void n(int,B):X.methods)
    static void n(int count, B args) {
      for (int i = 0; i < count; i++)
        Y.n(args);
    }
  }
  ```
  - First unify primary, then substitute, then unify filter
  ```

So What?

Lots of power *and* modular type safety?

Fill in Interface Methods

- ```
  class MakeImplement<X, interface I> implements I {
    X x;
    MakeImplement(X x) { this.x = x; }

    // for all methods in I, but not in X, provide default impl.
    <R,A*>[m]for( R m (A) : I.methods; no R m (A) : X.methods)
    R m (A a) { return null; }

    // for X methods that correctly override I methods, copy them
    <R,A*>[m]for ( R m (A) : I.methods; some R m (A) : X.methods)
    R m (A a) { return x.m(a); }

    // for X methods with no conflicting I method, copy them.
    <R,A*>[m]for(R m (A) : X.methods; no m (A) : I.methods)
    R m (A a) { return x.m(a); }
  }
  ```
  ```

MJ in the Universe

- “Write code once, apply it to many program sites”
  - so far the privilege of MOPs, AOP, meta-programming
  - modular type safety only with MJ

In Summary

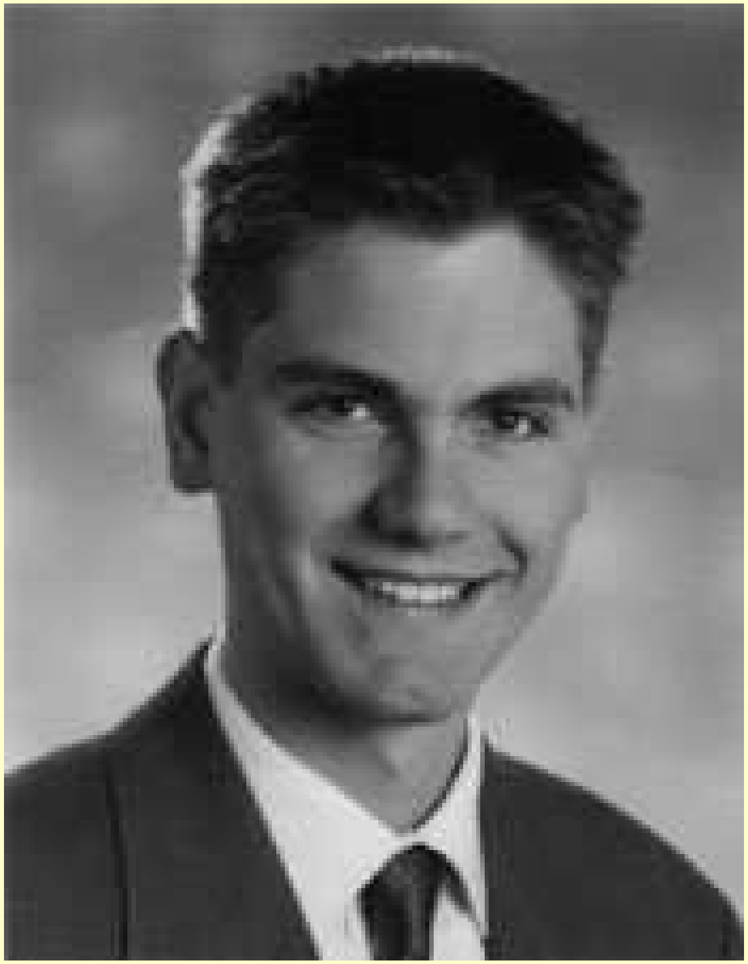
What did I talk about?

These Lectures

- NRMI: middleware offering a natural programming model for distributed computing
  - solves a long standing, well-known open problem!
- J-Orchestra: execute unsuspecting programs over a network, using program rewriting
  - led to key enhancements of a major open-source software project (JBoss)
- Morphing: a high-level language facility for safe program transformation
  - “bringing discipline to meta-programming”

Credits: My Students

- Christoph Csallner
  - automatic testing
    - JCrasher
    - Check-n-Crash (CnC)
    - DSD-Crasher
  - tools used at NC State, MIT, MS Research, Utrecht, UWashington
  - about to intern at MS Research



Credits: My Students

- Shan Shan Huang
  - program generators and domain-specific languages
    - Meta-AspectJ (MAJ)
    - SafeGen
    - cJ
    - MJ
  - Intel Fellowship
  - NSF Graduate Fellowship

Credits: My Students

- Brian McNamara
  - multiparadigm programming
    - FC++
    - LC++
- now at Microsoft

Credits: My Students

- Eli Tilevich
  - language tools for distributed computing
    - NRMI
    - J-Orchestra
    - GOTECH
  - binary refactoring

- now an Assistant Professor at Virginia Tech

Credits: My Students

- David Zook
  - program generators and domain-specific languages
    - Meta-AspectJ (MAJ)
    - SafeGen

Credits: My Students

- Ranjith Subramanian (M.Sc.)
  - Adaptive replacement algorithms
  - hardware caching

Credits: My Students

- Austin Chau (M.Sc.)
  - language tools for distributed computing
    - J-Orchestra

Credits: My Students

- Marcus Handte (M.Sc.)
  - language tools for distributed computing
    - J-Orchestra
- now a Ph.D. student at Stuttgart

Credits: My Students

- Nikitas Liogkas (M.Sc.)
  - language tools for distributed computing
    - J-Orchestra
- now a Ph.D. student at UCLA

Credits: My Students

- Stephan Urbanski (M.Sc.)
  - language tools for distributed computing
    - GOTECH
  - now a Ph.D. student at Stuttgart

Thank you!