Compilation 2012

Program Transformations

Jan Midtgaard
Michael I. Schwartzbach
Aarhus University
Program Transformations

- A special case of a compiler
- Source and target language are identical
- Must still preserve the semantics

- Why do we want to do this?
- Because we like the new version better!

- A pretty printer is only an identity transformation
Examples of Transformations

- Normalization
  - translate to a subset of the language
- Refactoring
  - change the design of the program
- Optimization
  - make the program more efficient
- Obfuscation
  - make the program harder to reverse engineer
- Watermarking
  - make the program harder to pirate
Linguistic Normalization

- **Desugaring**
  - translates constructions into simpler forms
  - simplifies the compilation process
  - makes formal reasoning easier

- **Constant folding**
  - eliminates compile-time constant computations
  - may possibly involve algebraic rules
An observer class should have the same behavior

- No operators or constants (use library methods)
- No control structures except if and recursion
- No nested expressions, only variables
- No blocks
- Only assignments like: \[ x = y \cdot m(z_1, \ldots, z_k); \]
How Small Could "Core OCaml" Be?

- No pattern matching (encode as nested `ifs`)
- No polymorphism (eliminate by duplication)
- No nested expressions, only variables

One can even eliminate let-bindings by encoding them as immediate applications:

```
let x = e in e' ~ (fun x -> e') e
```
Stylistic Normalization

- **Clean up import statements**
  - the subset of Java where every imported class is also used somewhere

- **Upgrade deprecated methods**
  - the subset of Java where no "deprecated" warnings are issued during compilation

- **Minimize access rights**
  - the subset of Java where any weakened access modifier would cause a compile-time error
Software Evolution

- As programs evolve, so must their designs
  - change the class hierarchy
  - split a single method into several methods
  - add extra arguments to methods
- Major changes are difficult to perform by hand
  - does the program still work correctly?
  - are the desired improvements really obtained?
  - has the client code been systematically updated?
- Automatic support is a real practical benefit
A refactoring is a program transformation that
- changes some aspect of the design
- does not change the overall observable behavior

Programmer insight is required here
- there is no canonical best design
- a refactoring often has an inverse refactoring

Refactorings are part of most modern IDEs
Examples of Refactorings

- Rename a variable, method, class, or package
  - make systematic changes in all client code
- Extract a method from a sequence of statements
  - determine arguments, result, and exceptions
- Extract a superclass or interface
  - pull up shared behavior
- Encapsulate a field
  - introduce `getX` and `setX` methods
- Change a method signature
  - add, remove, or reorder arguments
Optimization

- Source-to-source optimizers improve a program
  - makes it smaller
  - makes it run faster
  - makes it use less memory
  - makes it use less power

- Optimizations may work against each other
  - time vs. space

- Optimization may work against other goals
  - prevent modularity
  - inhibit abstraction
  - obscure readability
Dead Code

- Usually the class library is huge
- A given `main` method may only use a fraction
- Code that can never be executed is called dead

- Why ship dead code with an application?
Dead Code Removal

- Dead code can be identified through a transitive closure of the call graph, starting at `main`.
- The analysis may consider different granularities:
  - classes
  - methods and fields
  - statements and locals
- Application sizes may be reduced by 90%.
- Example: Google’s Closure Compiler for JavaScript.
Supercompilation

- When parts of a program's input are known, the computations may be specialized and optimized
- Constant folding is a weak version of this

- Supercompilation is an aggressive technique for optimizing programs in this manner
- The program is symbolically executed by a metaprogram that analyzes its behavior
public class BinarySearch {
    private final float[] array;
    protected BinarySearch() { array = null; };
    protected BinarySearch(float[] array) { this.array = array; }
    public int getIndex(float x) {
        return getIndexB(x, 0, this.array.length);
    }
    private int getIndexB(float x, int lower, int upper) {
        int mid;
        while (upper - lower > 1) {
            mid = (upper + lower) / 2;
            if (x < this.array[mid]) upper = mid;
            else lower = mid;
        }
        return lower;
    }
}
public class Test {
    static final float[] array = { 0.5f; 1.5f, 2.5f, 3.5f };  
    static final BinarySearch bs = new BinarySearch(array);

    public static int test(float x) { return bs.getIndex(x); } 

    public static void main(String args[]) {  
        System.out.println( test(0f) );  
        System.out.println( test(1f) );  
        System.out.println( test(2f) );  
        System.out.println( test(3f) );  
        System.out.println( test(4f) );  
    }
}
public static int test(final float x) {
    if (x < array[2]) {
        if (x < array[1]) return 0;
        return 1;
    } else {
        if (x < array[3]) return 2;
        return 3;
    }
}

- The optimal code for searching an array of size 4
Secret Code

- Sometimes, source code is meant to be secret
  - proprietary algorithms
  - national security issues
  - Windows XP / Vista / 7 / …
  - Joos 2 compilers

- Even so, it may occasionally become public
  - snooping
  - legal requirements
  - accidents
  - decompiling
Obfuscation

- Obfuscation is a program transformation that
  - protects source code by destroying information
  - still preserves its semantics

- What kind of information may be destroyed?
  - documentation
  - names
  - structured control flow
  - general legibility
public class IntegerToString {

    public IntegerToString() throws Exception {
        this.printNumber(this.parseNumber(), 8);
    }

    public static void main(String[] args) throws Exception {
        new IntegerToString();
    }

    public int parseNumber() throws Exception {
        System.out.print("Enter a number: ");
        int c = 0;
        int number = 0;
        while ((c = System.in.read()) >= 48)
            number = 10*number + (c-48);
        return number;
    }
}
public void printNumber(int number, int base) throws Exception {
    System.out.print("In base "+base+" that is: ");
    String ns = "";
    if (number == 0) {
        ns = "0";
    } else {
        while (number > 0) {
            ns = (number % base) + ns;
            number = number / base;
        }
    }
    System.out.print(ns);
}
Name Obfuscation

- Systematically rename all identifiers
- Use meaningless names
- Apply maximal overload induction

- Keep a map to reconstruct debug information
- Not sound if reflection is allowed
public class foobar {

    public foobar() throws Exception {
        this.foobar(this.foobar(), 8);
    }

    public static void main(String[] foobar) throws Exception {
        new foobar();
    }

    public int foobar() throws Exception {
        System.out.print("Enter a number: ");
        int foobar = 0;
        int fo0bar = 0;
        while ((foobar = System.in.read()) >= 48) {
            fo0bar = 10*fo0bar + (foobar-48);
        }
        return fo0bar;
    }
}
public void foobar(int foobar, int fo0bar) throws Exception {
    System.out.print("In base "+fo0bar+" that is: ");
    String f0obar = "";
    if (foobar == 0) {
        f0obar = "0";
    } else {
        while (foobar > 0) {
            f0obar = (foobar % fo0bar) + f0obar;
            foobar = foobar / fo0bar;
        }
    }
    System.out.print(f0obar);
}
Flow Obfuscation

- Also known as control mangling
- Rewrite control structures to less obvious ones
- Introduce unnecessary complexity
- Example:

```java
while B { S }
int i = 0;
while (i<K) {
    i++;
    boolean b = B;
    if (!b && i<N) break;
    for (;i>0;i--) S
}
```
public class foobar {

    public foobar() throws Exception {
        this.foobar(this.foobar(), 8);
    }

    public static void main(String[] foobar) throws Exception {
        new foobar();
    }

    public int foobar() throws Exception {
        System.out.print("Enter a number: ");
        int foobar = 0;
        int fo0bar = 0;
        int f0obar=0;
        while (f0obar<123) {
            f0obar++;
            boolean f00bar = (foobar = System.in.read()) >= 48;
            if (!f00bar && f0obar<42) break;
            for (;f0obar>0;f0obar--) {
                fo0bar = 10*fo0bar + (foobar-48);
            }
        }
        return fo0bar;
    }
}
public void foobar(int foobar, int fo0bar) throws Exception {
    System.out.print("In base "+fo0bar+" that is: ");
    String f0obar = "";
    if (foobar == 0) {
        f0obar = "0";
    } else {
        int f00bar=0;
        while (f00bar<937) {
            f00bar++;
            boolean f0obar = foobar > 0;
            if (!f0obar && f00bar<11) break;
            for (;f00bar>0;f00bar--) {
                f0obar = (foobar % fo0bar) + f0obar;
                foobar = foobar / fo0bar;
            }
        }
        System.out.print(f0obar);
    }
}
Design Obfuscation

- In the general case:
  - perform random refactorings of the code

- In the typical case:
  - make random statement sequences into methods
  - inline methods at some invocation sites
public class foobar {

    public foobar() throws Exception {
        this.foobar(this.foobar(), 8);
    }

    public static void main(String[] foobar) throws Exception {
        new foobar();
    }

    public int foobar() throws Exception {
        System.out.print("Enter a number: ");
        int foobar = 0;
        int fo0obar = 0;
        int f0obar=0;
        while (f0obar<123) {
            f0obar++;
            boolean f00bar=(foobar = System.in.read()) >= 48;
            if (!f00bar && f0obar<42) break;
            for (;f0obar>0;f0obar--) { fo0obar = 10*fo0obar + (foobar-48); }
        }
        return fo0obar;
    }
}
public String f0Obar(int fO0bar, int foobar, String f0obar) {
    return (fO0bar % foobar) + f0obar;
}

public void foobar(int foobar, int fo0bar) throws Exception {
    System.out.print("In base "+fo0bar+" that is: ");
    String f0obar = "";
    if (foobar == 0) { f0obar = "0"; }
    else {
        int f00bar=0;
        while (f00bar<937) {
            f00bar++;
            boolean fOobar=foobar > 0;
            if (!fOobar && f00bar<11) break;
            for (;f00bar>0;f00bar--) {
                f0obar = f0Obar(foobar,fo0bar,f0obar);
               foobar = foobar / fo0bar;
            }
        }
    }
    System.out.print(f0obar);
}

String Obfuscation

- String constants contain documentation hints
- They are readily available in the constant pool

- Split strings into tiny fragments
  - each fragment will not be meaningful

- Use cryptographic techniques
  - compile-time encryption
  - runtime decryption
public class foobar {

    String f00bar = "er a n";
    String f00bar = "mbe";

    String fO0bar(String foobar) {
        String fo0bar = "";
        for (int fOObar=0; fOObar<foobar.length(); fOObar++) {
            fo0bar=fo0bar+(char)(foobar.charAt(fOObar)-7);
        }
        return fo0bar;
    }

    public foobar() throws Exception {
        this.foobar(this.foobar(), 8);
    }

    public static void main(String[] foobar) throws Exception {
        new foobar();
    }
}
public int foobar() throws Exception {
    System.out.print("Ent"+f00bar+"u"+fO0bar+"r: ");
    int foobar = 0;
    int fo0obar = 0;
    int f0obar=0;
    while (f0obar<123) {
        f0obar++;
        boolean f00bar=(foobar = System.in.read()) >= 48;
        if (!f00bar && f0obar<42) break;
        for (;f0obar>0;f0obar--) { fo0obar = 10*fo0obar + (foobar-48); }
    }
    return fo0obar;
}

public String f0Oobar(int fOObar, int foobar, String f0obar) {
    return (fOObar % foobar) + f0obar;
}
public void foobar(int foobar, int fo0bar) throws Exception {
    System.out.print(fOobar("Pu'ihzl'")+fo0bar+fOobar("'{oh{'pzA'"));
    String f0obar = "";
    if (foobar == 0) {
        f0obar = f0Obar("7");
    } else {
        int f00bar=0;
        while (f00bar<937) {
            f00bar++;
            boolean f0obar=foobar > 0;
            if (!f0obar && f00bar<11) break;
            for (;f00bar>0;f00bar--) {
                f0obar = fOobar(foobar,fo0bar,f0obar);
                foobar = foobar / fo0bar;
            }
        }
        System.out.print(f0obar);
    }
}
Software watermarking protects against piracy

Each copy of a program has a unique embedded number (the fingerprint)

Ownership is proved by making the fingerprint the product of two large prime numbers

A watermarking scheme has two components
  - the embedder, which is a program transformation
  - the recognizer, which extracts the fingerprint again

By extracting the fingerprint, a pirated copy can be traced back to an original, registered owner
A Classical Arms Race

- Pirates will want to destroy the fingerprint without changing the behavior of the program
- Watermarking must be robust and hard to detect

- Obvious attacks on a watermarking scheme:
  - obfuscators
  - optimizers
  - refactorings
  - any old program transformation

- The response is to use more complex schemes
Simple Watermarking Schemes

- Encode the fingerprint in comments
  - attack: delete all comments

- Encode the fingerprint in identifiers
  - attack: rename all identifiers

- Encode the fingerprint in strings
  - attack: split and encrypt all strings
Better Watermarking Schemes

- Encode the fingerprint in dead code
  - attack: remove all dead code

- Encode the fingerprint in the choice and nesting of control structures
  - attack: mangle the control flow

- Encode the fingerprint in the frequencies of bytecodes in the generated code
  - attack: optimize the code
A Losing Game

- Any scheme based on the program structure is doomed to be successfully attacked.

- Every aspect of the program structure may be changed while preserving the semantics.

- Instead, encode the fingerprint in the semantics of the program.
A Simple Semantic Scheme

- When a special input is supplied, the program outputs the fingerprint

- This is vulnerable, since this behavior can be detected through debugging and code inspection

- Easter eggs are always discovered and published

- A watermarking scheme must be stealthy
State of The Art Scheme

- The fingerprint is encoded as a graph of a particular shape
- On a special input, this shape will appear as a subgraph of the graph of objects in the heap
- Be careful not to have the subgraph too loosely connected to the rest of the heap:
State of The Art Attack

- Add extra fields to all classes
- Permute and rename all fields
- Change all constructors to add random edges from the extra fields to all other heap objects:
Why is This an Attack?

- Almost every fingerprint can now be recovered from the heap
- Too many fingerprints are just as bad as none
- But the price of defeating a watermark is becoming higher and higher
- And the next step is to add tamperproofing techniques to the scheme
Implementing Program Transformations

- Add an extra phase transforming the AST:

```ocaml
let compile filenames =
    let apply phase ast msg =
      begin
        print_string "*** ";
        print_endline msg;
        let ast' = phase ast in
        flush stdout;
        ast'
      end in
    let () = print_endline "Applying phases:" in
    let prog = apply (List.map parse_file) filenames "parsing" in
    let tenv = apply Environment.env_program prog "environment building " in
    let last = apply (Linking.link_program prog) tenv "linking/name resolving" in
    let tast = apply (Typechecking.tcheck_program last) tenv "type checking" in
    let cast = apply Constants.const_program tast "constant folding" in
    let tast = apply Transformation.trans_program cast "transforming program" in
    exit 0
```