Lab Lessons

- What are you leaning from the lab?

Lab Lessons for you

- Yes we can! You can build your own language processors. basta.
- Programming languages are complex animals, but not so difficult.
- Master level requires you to understand to the bone, and be precise (e.g. CC complexity, find out what it really means)
- Software metrics can be very valuable, but they have a dark side: aggregation and interpretation.
- Metrics are valuable when you compare metrics between comparable systems (why two SQL interpreters in this lab?)
- (almost) never use averages/means to aggregate. why not?
- (almost) never use a metric to pass immediate judgment, build in the human assessment in your metrics program, like SIG does (also check out Tudor Girba's work)
- Software assessment using metrics works "in the large", but be very careful when judging quality of units using metrics.

Lab Lessons for us

- We are replacing regular expressions by something different (and hopefully faster)
- We will change the exercise to use lexer/parser generation for dealing with comments and strings
- We will emphasize teaching relational calculus such that you have more plans for solving the exercises
- We will change series 0 to an explicit exercise
- We will (probably) change series 1 to include finding trends over versions

Refactoring

Software Evolution 2012/2013

Jurgen Vinju

- What is refactoring?
- When and why do we do it?
- How do refactoring tools work?

- Who has experience with refactoring?
- Who has experience using refactoring tools?
- Who has experience building refactoring tools?
- Name a few “refactorings”

Software Refactoring

- "factor something in/out" = include/exclude something as a relevant element when making a calculation or decision.
- $a.n + a.m = a.(n + m)$ "factoring"
- $20 = 2^2.5$ "prime factorization"
- No change of externally observable behavior

Software Refactoring

- Refactoring is changing design decisions, without changing functionality
- "Refactoring Improving the Design of Existing Code" by Martin Fowler, Kent Beck, John Brand, William Opdyke, Don Roberts.
- Refactoring is only about internal quality

What is Refactoring?

- Strongly related to agile programming!
  - http://agilemanifesto.org/
  - responding to change
  - small iterations, less up-front design
- Internal quality is improvable, let's do it!
- What is the risk?

Agile needs refactoring tools

- Change is the constant
- Test first, to help this change
- But still...
  - tests need to be updated too
  - who knows if the tests are good enough?
- What if we could prove that the refactoring does not change semantics? then yes! ergo, tools.

Continous vs Batch

- When do you refactor?
  - Always? Continuously?
  - Sometimes? Batch?
- What can you refactor, and what can you not?
  - Only what the tools provide, or
  - Manual refactoring

Refactoring tools

- A valuable component of IDEs
- Are complex source code processors
- Complex refactorings are sometimes “broken”
- Big potential for improvement:
  - more different refactorings
  - better support and feedback to users

Tools

- Rename
- Move
- Extract class/method
- Change X into Y
- Pull up
- Push down
- Infer type argument
- Organize imports
- Introduce type parameter
- Introduce parallelism
- public to private
- visitor to interpreter
- switch to dynamic dispatch
- ...

Why are tools necessary

- What is the reason that a tool does better than a human being?
- Co-evolution: scattering, duplication, dependency
- Complexity of language semantics
  - overloading, scopes, sub-typing, synchronization, exception handling, static initialization
- Prevent tedious, boring, repetitive and error-prone work
- What is the risk?

What are the limits?

- A computer can not design a system, right?
- These are ongoing research topics
  - Can we find heuristics to propose refactorings?
  - Can we describe an intended design and let the system find the refactorings towards it?
  - Can we provide more and more useful refactoring tools?
- Tools are making a big difference here, but
- Refactoring remains mostly a human task (design)
- Like design, it requires practice and expertise

Tools tools tools

- There is away out of a source code mess, and this could be incremental refactoring.
- We need tools that support this incremental refactoring, or we have no guarantees of quality
- The tools we have are used a lot
- But such tools do not exist in enough quantity:
  - to support all kinds of incremental change
  - to support all kinds of languages and extensions

Creating your own refactoring tools

Refactoring tools

1. User selects point of interest
2. User selects refactoring transformation
3. User provides parameters
4. Tools computes preconditions for change
5. Tool executes transformation
6. User recompiles: no new warnings or errors
7. User runs unit tests: no extra failures

Or, we reuse the compiler

1. User selects point of interest
2. User selects refactoring transformation
3. User provides parameters
4. Tool executes transformation
5. Tool runs compiler and:
    1. finds errors, and aborts
    2. does not find errors and terminates
6. User runs unit tests: no extra failures

Language Processors

- Are very much like compilers or interpreters
  - they parse the source code
  - the form abstract models of the source code
  - they analyze these models
- Yet, they are different
  - can abstract from certain details
  - do not consider optimization
  - need better/clearer feedback to the programmer

Open Compilers

- Java, C# have open compiler frameworks
- Eclipse has its own open Java compiler
- It means you can reuse parts of the compiler
- And build your own language processor
- Open compilers are the exception, not the rule

Example

```
int f(int y) {
  int x = 0;
  return x + y;
}
```

Let's rename x.

What do you need to check?

What do you need to transform?

Example

```
class X extends Y {
  public int f(int x) { return 2 * x; }
}
```

Let's rename f to twice:

what to check?

what to transform?

Example

```
class Vector<X> { ...}

interface Iterable<X> { ... }

class FriendList implements Iterable {
  private Vector friends = new Vector();
  public void addFriend(String name) {
    this.friends.add(name);
  }
  public Iterator iterator() {
    return friends.iterator();
  }
}
```

Now I want to change this code to use type parameters!

Example

```
class Vector<X> { ...}
interface Iterable<X> { ... }
class FriendList implements Iterable<String> {
  private Vector<String> friends = new Vector();
  public void addFriend(String name) {
    this.friends.add(name);
  }
  public Iterator<String> iterator() {
    return friends.iterator();
  }
}
```

** this adds the actual parameters, correctly

** it can remove unnecessary casts

** but how to get this complex refactoring correct?

How does this work?

- Refactoring with constraint solving
  - Assume: a correct compilable input program
  - Extract: all the conditions under which the program is correct
  - Solve the resulting constraint system
  - The solutions are parts of refactored programs
- Check out papers by Frank Tip, Friedrich Steinman, Max Schaeffer

Constraints

- "x > 0 && x < 10"
  - solutions: {1,2,3,4,5,6,7,8,9}
  - solver understands arithmetic
- "X subtype of Integer && X subtype of String", "X <= Integer && Y <= String"
  - solution: {java.lang.Object}
  - solver understands sub-typing in Java
  - If X was a type parameter, we now know that it should be "Object".
- Does this remind you of anything?

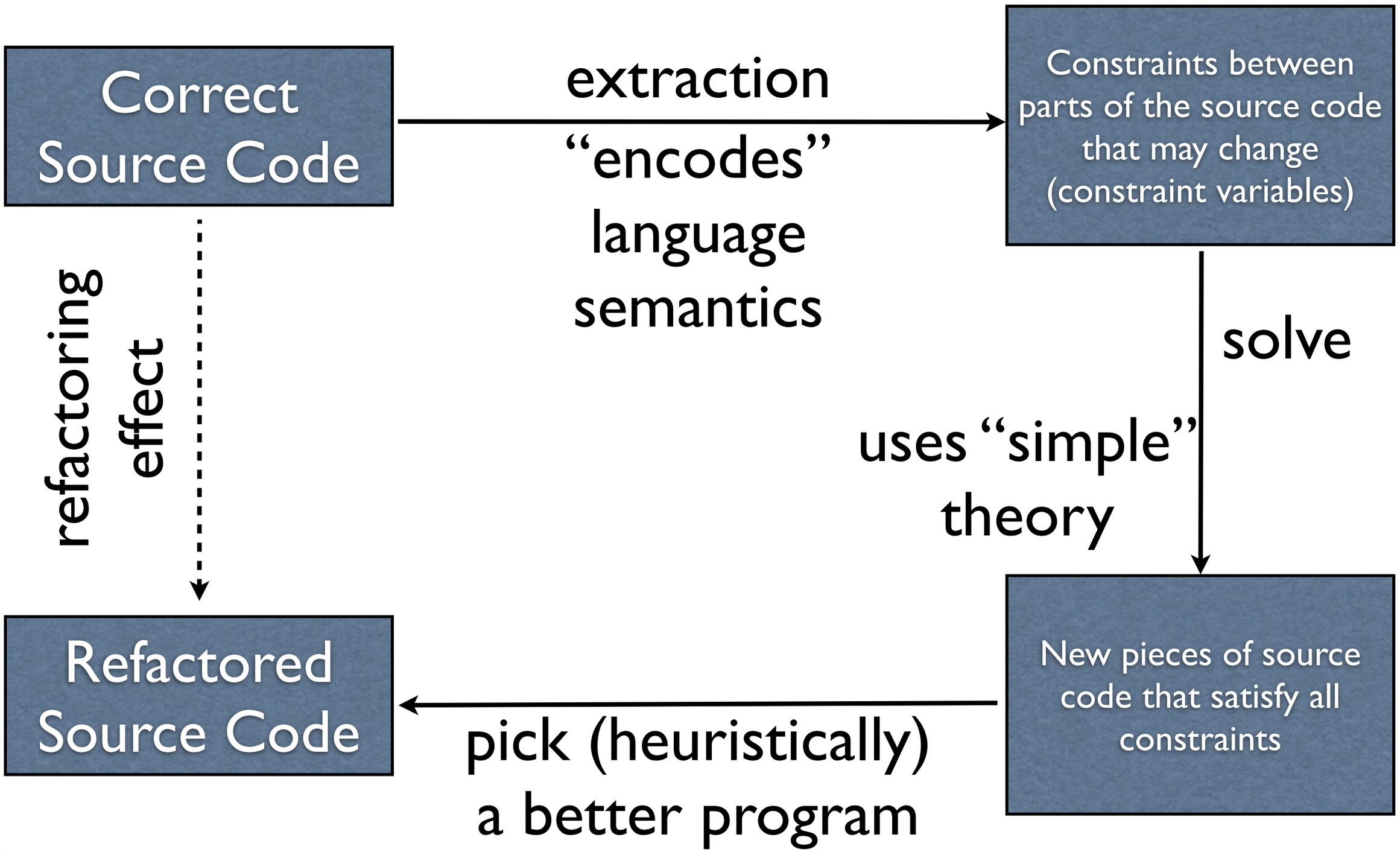
Example

```
class Vector<X> { ...}

interface Iterable<X> { ... }

class FriendList implements Iterable {
  private Vector friends = new Vector();
  public void addFriend(String name) {
    this.friends.add(name);
  }
  public Iterator iterator() {
    return friends.iterator();
  }
}
// This generates a huge set of constraints
```

Refactoring using constraints

- What can/should change is a "constraint variable"
- A "constraint" makes a condition under which the program is correct explicit
- A constraint "solver" computes all alternative correct solutions for the constraint variable(s)
- If there are no solutions, the refactoring precondition fails
- If there are, a unique one has to be selected
- Then the tool just substitutes the new solution for the old code in the source

Architecture



Evaluation

- There are alternatives, but this is the most principled way of building refactoring tools, IMHO
- What is attractive in this design?
- What can go wrong with this design?
- What are the limits of this design?