Mining Jungloids to Cure Programmer Headaches

Dave Mandelin, Lin Xu, Ras Bodik UC Berkeley
Doug Kimelman IBM

Motivation: the price of code reuse

- Inherently, reusable code has complex APIs. Why?
  - Many classes and methods
  - Indirection
  - Many options
- Simple tasks often require arcane code — *jungloids*
  - *Example*. In Eclipse IDE, parsing a Java file into an AST
  - simple: a handle for the Java file (object of type **`IFile`**)
  - simple: what we want (object of type **`CompilationUnit`**)
  - hard: finding the parser
    - took hours of documentation/code browsing

```
ICompilationUnit cu = JavaCore.createCompilationUnitFrom(javaFile);
CompilationUnit ASTroot = AST.parseCompilationUnit(cu, false);
```

First key observation

- *Part 1:* Headache task requirements can usually be described by a *1-1 query*:

  "What code will transform a (single) object of (static) type A into a (single) object of (static) type B?"
- Our experiments:
  - 12 out of 16 queries are of such single-source, single-target, static-type nature
- Same example:
  - type A: **`IFile`**, type B: **`CompilationUnit`**

```
ICompilationUnit cu = JavaCore.createCompilationUnitFrom(javaFile);
CompilationUnit ASTroot = AST.parseCompilationUnit(cu, false);
```

First key observation (cont'd)

- *Part 2:* Most 1-1 queries are correctly answered with *1-1 jungloids*
- 1-1 jungloid: an expression with single-input, single-output operations:
  - field access; instance method calls with 0 arguments; static method and constructor calls with one argument ; array element access.
- Our experiments:
  - 9 out of 12 such 1-1 queries are 1-1 jungloids
  - Others require operations with k inputs

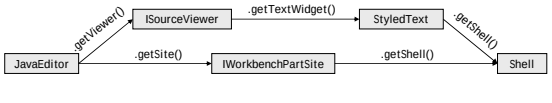
```
ICompilationUnit cu = JavaCore.createCompilationUnitFrom(javaFile);
CompilationUnit ASTroot = AST.parseCompilationUnit(cu, false);
```

Prospector: a jungloid assistant tool

- Prospector: a programmer's "search engine"
  - *mine* API implementation and sample client code
  - *search* a jungloid "database"
  - *paste* the result into programmers code
- User experience:
  - similar to code assist in Eclipse or .Net
  - editor cursor position specifies both target type B and context from which the source type A is drawn
- Soundness guarantees?
  - such as "does the mined jungloid do the work I intend?"
  - no such guarantees, of course (because the query doesn't specify the full intention)

Program representation

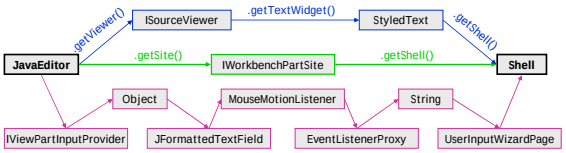
- The representation is defined to support 1-1 jungloid mining
  - A directed graph where each path is a 1-1 jungloid
  - Vertices: pointer types (instances and arrays)
  - Edges: well-typed expressions with single pointer-typed input and single pointer-typed output
- A small part of our representation:



Second key observation

The jungloid that answers a 1-1 query "How do I get from *A* to *B*?" typically corresponds to the shortest path from *A* to *B*.

- Fewer steps are fewer chances to
  - throw an exception
  - return semantically unrelated objects
  - confuse the programmer



Experiment (shortest-path jungloids)

Result:

- in 10 out of 10 queries, shortest path produced correct code

Breakdown:

- 9 found best code (in 3, path length = 1, but code non-trivial)
- 1 found correct code, but the graph contains a subjectively better jungloid of equal length

Conclusions:

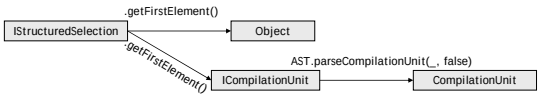
- shortest path a very good heuristic for finding correct jungloids
- offering k shortest jungloids likely to find the best jungloid

The downcast problem

- Problem: Java code is full of downcasts
  - containers return Objects
  - type depends on configuration files or other input

```
IStructuredSelection ssel = (IStructuredSelection) sel;
ICompilationUnit cu = (ICompilationUnit) ssel.getFirstElement();
CompilationUnit ast = AST.parseCompilationUnit(cu, false);
```

~~ActionContext ac = new ActionContext(sel);~~
~~char[] char_ary = ac.toString().toCharArray();~~
~~CompilationUnit resultVar = AST.parseCompilationUnit(char_ary);~~



The subtype mining algorithm

- Mining a code base
  - Mine sample API client code base to find valid casts
  - Assumption: Code base contains the scenario the user wants
- Goal: for `A.f()` declared to return object of T, find a superset of possible dynamic subtypes
  - Superset ensures that the correct jungloid is in the graph
- Idea: mine invocation sites of `A.f()`, find casts reached by return value
- Algorithm: flow insensitive, interprocedural inference
  - `(T)` $e_1 \rightarrow T \in types[e_1]$
  - e_1 `instanceof T` $\rightarrow T \in types[e_1]$
  - $types[e_1] \in types[(e_0 \; ? \; e_1 \; : \; e_2)]$
  - $T\ x = e_1 \rightarrow types[x] \subseteq types[e_1]$

The big picture

- Prospector architecture
  - cast miner
  - shortest path searcher
  - representation
  - UI

TODO (goal of slide is to use the architecture to recap how prospector works)

Summary

- Two key observations
  - many headache scenarios are searches for 1-1 jungloids
  - most jungloids can be found with "k-shortest paths" over a simple program representation based on declared types + static cast mining
- Under the hood
  - new program representation
  - cast mining
  - memory footprint reduction (graph clustering)
- Prospector status
  - for Java under Eclipse
  - to be available … Summer 2004

Future work

- Semantics
  - Q: Is this jungloid semantically valid?
  - A: Model checking
- Types
  - Q: Can we mine more kinds of jungloids?
  - A: Java 1.5 generics
  - A: Inferring polymorphic types
  - A: Inferring input types
  - A: Typestates
- Plenty more…

Future work

- Graph-theoretic considerations:
  - breaking 2-cycles (conversion from A to B and back)
  - high-degree nodes may need special handling.
  - assign weights regarding probabilities that expressions will succeed, and use weighted SP.
- Generalize the downcast problem
  - into a more general inference of narrower types.
- Modeling and inferring generics/polymorphic types
  - legacy code
- k-shortest path results
  - ranking
  - clustering
- Dynamic techniques
  - finding downcasts controlled by configuration data