Java Class Hierarchies with Maximal Sharing

Jurgen Vinju

December 12, 2002
Context

- **Goal:**
  
  Automatic implementation of tree-like data-structures
  Compilers, Transformers, Analyzers, XML Processors
  Memory Efficiency seems to be a problem

- **Maximal Sharing**
  
  Works well in functional programming, rewriting
  (Where all *data representation* is automatic)
  So let’s carry this over to Java data-structures

- **For our purposes:**
  
  Meta-Environment tools in Java (AsFix, ATerms)
  DocGen-like tools
TOC

- Requirements
- Java ATerm
- Generators
  - SharedObjectFactory
  - Composite subclasses of ATermAppl
- Example
- Experience
- Sales Talk
- Discussion
Requirements

- Fast access time, and
- Memory efficient
- Data-hiding:
  - Documentation
  - Use of types prevents programming errors
  - Profiling
  - Consistency checks
- Extensibility
- Linearization {Debugging, Communication}
The Java ATerm library

- Redundancy + Maximal Sharing \(\equiv\) Memory Efficiency

- Types of redundancy
  - Natural (e.g. layout nodes in a parse tree)
  - Artificial: use redundancy to store data close to use sites

- Annotations \(\equiv\) Extensibility

- Textual (Shared) representation

- Data-hiding? Not quite enough.
  - Lessons learned in the GLT project \(\rightarrow\) ApiGen
Algebraic data-types

- Syntax definitions, Tree Grammars, Signatures, ...

- Generate implementations of tree-like data-structures

- Many types $\equiv$ compile-time feedback $\equiv$ safe code

- Refactoring the representation itself becomes feasible

- Extra (hard,boring,expensive) features for free:
  - Linearization
  - Traversal
  - Profiling
  - Consistency checks
  - And why not... *Maximal Sharing*??
Generate classes based on the ATerm library

- Just like ApiGen for C...
- Type-system of Java prevents simple aliasing like in C
- Composition → we lose maximal sharing
- Inheritance
  - Not possible for trivial reasons
  - Not practical due to code duplication
- Recipe:
  - Refactoring the ATerm library for extensibility
  - Try to keep the efficiency
  - Generate extensions of PureFactory and ATermAppl
ApiGen organization
Shared Object Factory

- Maximal Sharing is implemented using hash-consing
- The SOF implements only hash-consing, **nothing** more
- Design Patterns:
  - Factory - a class that builds objects from other objects
  - Prototype - minimize allocations
- HashFunctions - a practical collection of hash functions
public class SharedObjectFactory {
    public SharedObject build(SharedObject prototype);
    public String toString(); // profile report
}

public interface SharedObject {
    SharedObject duplicate(); // clone
    boolean equivalent(SharedObject o);
    int hashCode();
}

public class HashFunctions {
    static int simple(Object[] o);
    ...
}
Inheriting from SharedObjectFactory

```java
public class PureFactory extends SharedObjectFactory implements ATermFactory {
    static ATermApplImpl protoAppl;
    ...

    public PureFactory() {
        protoAppl = new ATermApplImpl();
        ...
    }

    public ATermAppl makeAppl(AFun fun, ATermList args) {
        protoAppl.init(fun, args);
        return build(protoAppl);
    }
}
```
Composite SubClasses of ATermAppl

- **Factory** - subclass PureFactory (prototypes and make methods)
- **Constructor** - subclass ATermAppl to deal with patterns
  - implements `toTerm()` and `toString()`
  - adds a field: `static private` `ATerm` `pattern`;
- **Types** - Abstract subclasses of Constructor
  - delegates `fromTerm(ATerm trm)` to subclasses
  - provides default `false` value for all kinds of static properties
- **Operators** - subclasses of abstract types
  - implements `fromTerm(ATerm trm)` using the pattern
  - overrides values for static properties
  - specializes `hashFunction()` as an optimization
Example

```java
{ 
    BooleansFactory f = new BooleansFactory();
    Bool b = f.makeBoolAnd(f.makeBoolTrue(),
                           f.makeBoolFalse());

    if (b.hasLeftHandSide()) {
        Bool l = b.getLeftHandSide();
        System.out.println("This is the lhs:" + l);
    }

    Bool c = Bool.fromTerm(f.readFromFile("input.trm"));
    if (c.isAnd()) {
        System.out.println("It's a conjunction");
    }
}
```
Experience

- Benchmarks: EvalTree, EvalExp, Fibonacci, etc...
  Compared to (new) ATerm code: slightly faster
  Compared to (old) ATerm code: 10-20% slower
  Compared to non-shared API: much faster and smaller
  Significant reduction in LOC (sometimes 50%)

- Refactoring the Tom compiler
  Incremental reverse engineering of TomFix
  Detected bugs and design flaws
  Reduction in LOC
  Bootstrapped (but without support for list matching)
  Compared to ATerm code: much faster
Sales Department

- Only dependencies: shared-objects and aterm-java
- Coupled with SDF2 via ADT files
  Hierarchy provides abstract view,
  but linearized representation can be anything (AsFix!)
- Coupled with Tom via generated signature definition
  Simple pattern matching for free
- Coupled with JJTraveler (W.I.P.)
  Safe Traversal for free
- Generates empty class files
  Implementation is hidden in separate package
  Empty classes can be filled with user-code
Discussion

• Conclusion
  Class Hierarchies with Maximal Sharing

• Future work
  Specialized support for list types
  Applications:
    Meta-Environment, Analysis, Reengineering

• Questions?