Enforcing and Validating User-Defined Programming Disciplines

Todd Millstein

Computer Science Department
University of California, Los Angeles

todd@cs.ucla.edu
http://www.cs.ucla.edu/~todd

Too Much to Think About

void doit(Stuff s) { ... s.payload ... }

♦ can s be null?
♦ is s possibly accessed concurrently?
♦ is s potentially aliased?

How do programmers manage this complexity?
Programming Disciplines

- syntactic rules on program entities
  - naming conventions
  - design patterns
  - framework usage rules
- programmer understandable way to...
  - prevent common programming errors
  - ensure important run-time invariants
  - enforce desired programming styles

An Endless Supply

- locking disciplines for managing concurrency
- confinement disciplines for object encapsulation
- ownership disciplines for managing pointers
- singleton design pattern
- finite-state protocols
Enforcing and Validating User-Defined Programming Disciplines

**Better, But Still Too Much to Think About!**

- programming disciplines are typically unchecked
  - programmer must manually ensure conformance

- often they are not even fully specified
  - informally in comments, if you’re lucky

- easy to forget or misuse

---

**A Management Guru to the Rescue**

[Book cover: *Getting Things Done* by David Allen]
Getting Things Done

“GTD rests on the principle that a person needs to move tasks out of the mind by recording them externally. That way, the mind is freed from the job of remembering everything that needs to be done, and can concentrate on actually performing those tasks.”

-- Wikipedia article on GTD

A GTD Approach to Software Development

let programmers easily define their own programming disciplines
- provide declarative rules that are automatically enforced statically
- possibly employing new program annotations
- help programmers ensure the correctness of their programming disciplines
Todd Millstein
Enforcing and Validating User-Defined Programming Disciplines

**Our Approach**

- Annotated Programs
- Discipline Rules
- Discipline Semantics

**Discipline Users**
- Discipline Enforcer
- Discipline Designer
- Discipline Validator

**Instantiations of Our Approach**

- **Clarity:** type qualifiers for C programs
  - with Brian Chin, Shane Markstrum, Jens Palsberg
  - pos/neg/nonzero, nonnull, tainted/untainted
  - qualifier inference; automated validation; formalization

- **A theory of user-defined effect systems**
  - with Dan Marino
  - memory, locking, exceptions, “user-level” effects
  - modular, mechanized metatheory

- **JavaCOP:** “pluggable type systems” for Java
  - with Shane Markstrum, Dan Marino, Matt Esquivel, Chris Andreae, James Noble
  - nonnull, unique, race conditions, EJB, Polyglot, Junit
  - flow sensitivity, rule validation via testing
Todd Millstein

Enforcing and Validating User-Defined Programming Disciplines

JavaCOP: Pluggable Type Systems for Java

Java Programs

Typing Rules

Runtime Checks

Pluggable Typechecker

Rule Testing Framework

JavaCOP: Pluggable Type Systems for Java

Java Programs

Typing Rules

Pluggable Typechecker
Enforcing and Validating User-Defined Programming Disciplines

JavaCOP Demo

Example: Polyglot Compiler Framework

- Java compiler written in Java [Nystrom et al. 03]
- widely used in the research community
- designed to be extensible
  - no existing code need be modified
  - all code for an extension in its own package
  - relies on several unchecked programming disciplines...
Todd Millstein
Enforcing and Validating User-Defined Programming Disciplines

**Polyglot Programming Disciplines**
- **factory pattern** for AST nodes
  - a NodeFactory has methods to create each kind of node
  - only the NodeFactory can directly instantiate nodes
  - allows extensions to easily substitute new AST nodes
- **visitor pattern** for compiler passes
  - each node class must override the visitChildren method
  - specifies how passes should traverse its subtree
- **delegates** for AST nodes
  - must invoke compiler passes through a node’s delegate
  - allows in-place update of existing AST nodes

---

**Case Study: Polyglot Compiler Framework**

used JavaCOP to specify these Polyglot disciplines

80 lines of (non-blank, non-comment) JavaCOP code

<table>
<thead>
<tr>
<th>Compiler</th>
<th>LOC</th>
<th>Errors signaled</th>
<th>Errors actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polyglot-1.3.4</td>
<td>20910</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Polyglot-2.3.0</td>
<td>25154</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>JL5</td>
<td>7800</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>JPred</td>
<td>3343</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>eJava</td>
<td>2458</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>jet-0.9.0</td>
<td>921</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jif-3.1.1</td>
<td>22020</td>
<td>14</td>
<td>12</td>
</tr>
</tbody>
</table>
**Challenge: Balancing Expressiveness and Simplicity**

- Programming disciplines may depend on control-flow info.

- Example: Conditional nonnullness
  ```java
  void insert(T t) {
      if (head != null) { head.insert(t); } else { ... }
  }
  ```

- Flow-sensitive type systems are expressive but complex:
  - Require alias info.
  - Require special features for handling data structures precisely.

- Simple idea: Allow programmers to define dataflow passes that annotate the AST with dataflow facts:
  - JavaCOP rules can access these facts.

**Incorporating Dataflow Information**

```java
interface FlowFacts {
    FlowFacts genSet(JCTree t);
    FlowFacts killSet(JCTree t);

    FlowFacts genSetTrue(JCTree t);
    FlowFacts genSetFalse(JCTree t);
    FlowFacts killSetTrue(JCTree t);
    FlowFacts killSetFalse(JCTree t);

    FlowFacts mergeWith(FlowFacts f);
}
```
Todd Millstein
Enforcing and Validating User-Defined Programming Disciplines

Three Example Uses

- **conditional nonnullness**
  - “var is definitely nonnull here” facts
  - for locals and formals

- **raw types for initialization of @Nonnull fields**
  - Fahndrich & Leino, OOPSLA 2003
  - “field is definitely assigned” facts

- **transfer of @Unique-ness**
  - “must be dead” facts
  - JavaCOP rules verify deadness on reads

JavaCOP: Pluggable Type Systems for Java

- **Java Programs**
- **Typing Rules**
- **Runtime Checks**
- **Pluggable Typechecker**
- **Rule Testing Framework**
Challenge: Rule Validation

- problem: programming disciplines can be subtle
  - special cases
    - initialization of `@NonNull` fields
  - complex language features
    - multithreading, exceptions

- how to help JavaCOP users gain confidence in the correctness of their rules?
  - what does “correctness” mean?
  - how to practically check it?

Defining Correctness

- observation: many pluggable type systems have a natural run-time semantics
  - a `@NonNull` type system prevents null dereferences

- allow users to define run-time checkers for their type systems
  - often much simpler than an associated static checker
  - acts as a “spec” for the static type system

- simple API on top of the ASM bytecode rewriter
  - can optionally instrument each kind of bytecode instruction with an assertion method
**Rule Validation**

- A JavaCOP type system is “sound” if it is sufficient to prevent run-time assertion violations.

- Could try to prove this:
  - Our earlier work in Clarity did this for type qualifiers in C.
  - Fully automatic only for simple type systems and run-time invariants.
  - Interactive theorem provers still a black art.

**Testing for Soundness**

- Typing Rules
- Test Suite
- Runtime Checker
- Rule Testing Framework
Testing for Soundness

Typing Rules
Test Suite
Runtime Checker
Rule Testing Framework

Related Work: Type Qualifiers

- user-defined type annotations that refine existing types
  - Type Refinements for ML (Freeman and Pfenning 92)
  - CQual (Foster et al. 99, 02), JQual (Greenfield & Foster 08)

- limited expressiveness for specifying disciplines
  - subtypes of ML datatypes
  - uninterpreted tags with a partial order

- more expressive than our approach in other ways
  - qualifier inference, polymorphism, object sensitivity
**Related Work: Dependent Types**

- allow types to directly express run-time invariants
  - `{int*(p) | p != NULL}`
  - Dependent ML, Cayenne, Epigram, Ynot, ...

- very expressive
- no a priori restriction on programming style

- programmers must become provers
  - similar problem to that of program verification

- no explicit discipline for programmers to follow
- not appropriate for some program properties
  - e.g., design patterns

**Related Work: JSR308**

- proposal to extend Java to better support type annotations
  - led by Michael Ernst at MIT

- reference implementation and associated pluggable type system
  - Papi et al. 2008
  - no rule language; users write Java code
  - type systems can leverage JSR308’s more flexible annotations
  - adapts our approach to flow sensitivity
  - no rule validation
Todd Millstein

Enforcing and Validating User-Defined Programming Disciplines

**Related Work: Declarative Static Checkers**

- checking temporal protocols
  - SLAM, BLAST, Metal, ESP, ...
  - target a different class of program properties than JavaCOP
  - we add a notion of rule validation

- static program querying systems
  - CCEL, JTL, ASTLog, JQuery, CodeQuest, ...
  - expressive, optimized queries and constraints
  - whole-program checkers
  - no type information, no dataflow information, no validation

**Conclusion**

- an approach to allow programmers to specify, enforce, and validate their own programming disciplines

- helps bring the benefits of static type systems to the programming masses
  - researchers shouldn’t have all the fun

- current JavaCOP work
  - automated test suite generation for rule validation
  - lightweight dependent types