elements of software construction

semantics of Java

Daniel Jackson

september 12, 2007
review of last lecture
consider state machine $M$ with ops $op1$, $op2$, ... and states $S = \{S1, S2, \ldots\}$

machine as class ("singleton" pattern)

```java
class M {
    static S s;
    static void op1 () {
        if (s == S1) s = S2; ...
    }
}
```

usage
```
M.op1 (); ...
```

machine as object (standard OO idiom)

```java
class M {
    S s;
    void op1 () {
        if (s == S1) s = S2; ...
    }
}
```

usage
```
M m = new M(); m.op1 (); ...
```

state as object ("state" pattern)

```java
interface S {
    S op1 (); ...
}
class S1 implements S {
    S op1() {
        return new S2 ();
    }
}
```

usage
```
S state = new S1(); state = state.op1(); ...
```
# Pattern Comparison

<table>
<thead>
<tr>
<th></th>
<th>Singleton</th>
<th>Standard</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can instantiate multiple machines</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Adding orthogonal state components</td>
<td>easy</td>
<td>easy</td>
<td>awkward</td>
</tr>
<tr>
<td>Adding nested state components</td>
<td>awkward</td>
<td>awkward</td>
<td>easy</td>
</tr>
<tr>
<td>State-dependent behavior by</td>
<td>if-stmt</td>
<td>if-stmt</td>
<td>dynamic dispatch</td>
</tr>
<tr>
<td>Best use</td>
<td>resources limited to one machine</td>
<td>elsewhere</td>
<td>complex behavior but few modes</td>
</tr>
</tbody>
</table>
state invariants
safety properties

two kinds of property

‣ safety: the machine never does something bad
‣ liveness: the machine eventually does something good

in practice, liveness rarely useful

‣ eventually isn’t good enough
‣ happens before clock tick is a safety property (“no clock tick before op”)

how to formulate safety?

‣ can often express as reachability property
‣ a property true of every state
‣ example for traffic light: Safe = !(Green_EW && Green_NS)
property as state set

state property is equivalently

.predicate $P(s)$ applied to state $s$

 subset \{s: S | P(s)\} of states
how to check safety property?
- if diagram is small, just check every reachable state
- but if state machine is large, need better method

invariant reasoning
- check property holds in initial state
- check each operation “preserves” property
  property holds before --> property holds after
- if so, property is “invariant”

diagram
- initial state is green
- each takes green state to green state
example

a classic problem

• 8 x 8 chessboard can be filled with 32 dominos
• suppose we remove top-left and bottom-right squares
• can remaining 62 squares be tiled with 31 dominos?

invariant reasoning

• consider number of black, white squares covered
• invariant: \( \text{black} = \text{white} \)
• initially, \( \text{black} = \text{white} = 0 \)
• only operation is \texttt{placeDomino (loc)}
  always adds 1 to \texttt{black} and to \texttt{white}
  so it preserves the invariant
• board with corners removed has 32 black, 30 white
  this state does not satisfy the invariant, so it’s not reachable
strengthening

when property is not an invariant
- even though it holds for all reachable states
- need to strengthen the property
- typical feature of inductive reasoning

diagram (upper)
- `op2` in takes green `S6` to non-green
- but `S6` is not reachable!

diagram (lower)
- consider green-blue invariant
- now preserved, and green-blue => green
example: radiotherapy

**problem & approach**

- want to deliver dose and record on disk, but both may fail
- so break dose into small segments, and alternate deliver/record

**given specs**

- \( \text{op } \text{dose}: \text{post } d' = d + 1 \)
- \( \text{op } \text{write}: \text{post } r' = r + 1 \)

**prove** \( d - r \leq 1 \)
the Java interpreter as a state machine
idea

state machines for design modelling
  · use state machine to design software
  · can often represent aspect of program as a small machine

state machines for semantics
  · can define a language like Java using state machines
  · give a generic machine, with each statement as operation
  · can then use to prove properties

type checking
  · for each var or expr, assign a type that’s no smaller than type of value
  · called a “conservative” analysis
vars, classes and fields

• declare local vars, classes, fields (but no methods)

```java
c v; // in logic, write “v: C”
class C1 {C2 f;}// in logic, write “f: C1→C2”
class C2 extends C1 {} // in logic, write “C2 ≤ C1” (and “C2 ≤ C2” etc)
class C3 extends C2 {} // in logic, write “C3 ≤ C2” (and “C3 ≤ C1” etc)
```

statement

<table>
<thead>
<tr>
<th>statement</th>
<th>syntax</th>
<th>type checking rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>u = new C ();</td>
<td>u: cu ⇒ C ≤ cu</td>
</tr>
<tr>
<td>assignment</td>
<td>u = v;</td>
<td>u: cu ∧ v: cv ⇒ cv ≤ cu</td>
</tr>
<tr>
<td>field get</td>
<td>u = v.f;</td>
<td>u: cu ∧ v: cv ∧ f: c→cf ⇒ cv ≤ c ∧ cf ≤ cu</td>
</tr>
<tr>
<td>field set</td>
<td>u.f = v;</td>
<td>u: cu ∧ v: cv ∧ f: c→cf ⇒ cu ≤ c ∧ cv ≤ cf</td>
</tr>
</tbody>
</table>
public class List {List reverse;}
public class EmptyList extends List {}
public class NonEmptyList extends List {Object elt; List rest;}

List el, l; NonEmptyList l1, l2;
el = new EmptyList (); el.reverse = el;
l1 = new NonEmptyList (); l1.rest = el; l1.elt = l1;
l2 = new NonEmptyList (); l2.rest = el; l2.elt = l1;

l = l1.rest; // OK, accepted
l = l2.elt; // OK, rejected
l = el.rest; // bad, rejected
el.rest = el; // bad, rejected
state machine semantics

vars
  stack: Var \rightarrow (\text{Ref} + \text{Null})
  heap: \text{Ref} \rightarrow \text{Field} \rightarrow (\text{Ref} + \text{Null})
  tag: \text{Ref} \rightarrow \text{Class}

init stack = \emptyset \text{ \&\& heap} = \emptyset

op constructor (u: \text{Var}, C: \text{Class}) // u = \text{new C()}
post stack' = stack + (u \rightarrow r) \text{ and tag'} = tag + (r \rightarrow C) \text{ for some fresh } r

op assignment (u, v: \text{Var}) // u = v
post stack' = stack ++ (u \rightarrow \text{stack}[v])

op fieldGet (u, v: \text{Var}, f: \text{Field}) // u = v.f
pre stack[v] \neq \text{null}
post stack' = stack ++ (u \rightarrow \text{heap}[\text{stack}[v]][f])

op fieldSet (u, v: \text{Var}, f: \text{Field}) // u.f = v
pre stack[u] \neq \text{null}
post heap' = heap ++ (\text{stack}[u] \rightarrow f) \rightarrow \text{stack}[v]

KEY
p + q : \text{union of relations or sets } p \text{ and } q
a \rightarrow b : \text{relation mapping } a \text{ to } b
p ++ q : \text{override; like } p + q, \text{ but tuples in } q
\text{ that start with same elements as tuples in } p
\text{ replace } \text{the tuples in } p
r[x] : \text{image of } x \text{ under relation } r

notation based on Alloy language
see http://alloy.mit.edu
type safety
type safety

what can go wrong?

› might attempt to dereference non-existing field

dynamic typing

› types checked at runtime
› instead of garbage, stop and report failure

static typing

› types checked by compiler, no runtime failures
› Robin Milner: “well-typed programs never go wrong”

but how do we know that static typing really works?

› use invariant method
safety invariant

stack safety

• to ensure no bad field refs, just need to ensure that stack is safe
• if variable $x$ has declared type $c$, its reference is tagged with a subtype of $c$

$$\text{STACK-SAFE} \equiv \quad \forall x : c \Rightarrow (\text{stack}[x] = \text{null}) \lor (\text{tag}[\text{stack}[x]] \leq c)$$

• but not an invariant: heap must be safe too

heap safety

• if field $g$ of ref $r$ holds $s$, then $r$ and $s$ have tags compatible with $g$’s decl

$$\text{HEAP-SAFE} \equiv \quad \forall r, g, s : \text{heap}[r][g] = s \land g : c \Rightarrow (\text{tag}[r] \leq c \land \text{tag}[s] \leq c)$$

invariant

$$\text{INV} \equiv \text{HEAP-SAFE} \land \text{STACK-SAFE}$$
proving the invariant

initially
\- stack and heap are empty, so \texttt{INV} holds trivially

preservation
\- if \texttt{INV} holds before \texttt{op}, and if \texttt{op} type-checks, then \texttt{INV} holds after

conclusion
\- by induction, \texttt{INV} holds after any sequence of ops
preservation arguments

invariant

\[
\text{STACK-SAFE} \equiv \ x: c \Rightarrow (\text{stack}[x] = \text{null}) \lor (\text{tag}[\text{stack}[x]] \leq c)
\]

\[
\text{HEAP-SAFE} \equiv \ \text{heap}[r][g] = s \land g: c \rightarrow cg \Rightarrow \text{tag}[r] \leq c \land \text{tag}[s] \leq cg
\]

sample argument

`given`

- \text{op constructor} (u: Var, C: Class) // \text{u = new C()}

- \text{post stack'} =  \text{stack + (u -> r) and tag'} = \text{tag + (r -> C)} for some fresh r

\[
\text{TYPECHECKS} \equiv u: \text{cu} \Rightarrow C \leq \text{cu}
\]

`we argue`

by hypothesis, \text{STACK-SAFE} holds in prestate, so stack’ is safe everywhere except \text{u} at \text{u}, we have tag’[stack'[u]] = C, and C \leq \text{cu} by \text{TYPECHECKS}

so \text{STACK-SAFE} holds in poststate
preservation, another

another argument

\[\text{given}\]

\begin{align*}
\text{op} & \text{ fieldGet} \ (u, v : \text{Var}, f : \text{Field}) \ // u = v.f \\
\text{pre} & \ v \neq \text{null} \\
\text{post} & \ \text{stack'} = \text{stack} ++ (u \rightarrow \text{heap [stack[v]][f]})
\end{align*}

\text{TYPECHECKS} \equiv u : c_u \land v : c_v \land f : c ightarrow c_f \Rightarrow c_v \leq c \land c_f \leq c_u

\[\text{we argue}\]

\begin{align*}
\text{at } u, \ &\text{tag'[stack'[u]] = tag[heap [stack[v]][f]]]} \\
\text{by HEAP-SAFE, tag[heap[stack[v]][f]]} \leq c_f, \text{ and } c_f \leq c_u \text{ by TYPECHECKS} \\
\text{so STACK-SAFE holds in poststate}
\end{align*}

\textbf{note}

\textbf{HEAP-SAFE} is trivially maintained for ops that only modify stack
type safety in practice
null dereferences

recall

• op had precondition to prevent null deref:

\textbf{op} \text{fieldGet} (u, v: \text{Var}, f: \text{Field}) \quad // \quad u = v.f
\textbf{pre} v \neq \text{null}
\textbf{post} \text{stack}’ = \text{stack} ++ (u \rightarrow \text{heap}[\text{stack}[v]][f])

in practice

• statement throws exception
• variable is not bound
• control continues at point at which exception is caught
casts

recall

• statement fails to typecheck, but no runtime error
  
  ```java
  List l = l2.elt; // OK, rejected
  ```

solution

• programmer inserts `cast`
  
  ```java
  List l = (List) l2.elt;
  ```

• runtime checks that expression has type indicated by cast
• if not, throws `ClassCastException`
primitives

not all values are object references or null
- also have primitives (int, char, boolean)
- corresponding object types (Integer, Character, Boolean)
- ugly part of Java, but improved in Java 5 with automatic “un/boxing”
methods

method calls
• same issue as field access
• type checking ensures that methods exist
• same principles

semantically
• method call adds a new “frame” to the stack
• special variable “this” bound to receiver object
real proofs of type safety

real proofs
  • use same method we used
  • but fuller semantics
  • and details checked mechanically

for example, see
  Tobias Nipkow and David von Oheimb
  Java Light is Typesafe -- Definitely
  Proc. 25th ACM Symp. Principles of Programming Languages
  http://david.von-oheimb.de/cs/papers/POPL98.html
summary

invariants
  • powerful way to reason about state machines
  • use induction on operations to prove property of all reachable states

state machine semantics
  • called “small step semantics”
  • many applications in design and theory of languages

type safety
  • ensures integrity: no “buffer overflows”
  • ensures modularity: can’t access data by address -- need name
  • becoming especially important with increasing security concerns