how to design a type system

Daniel Jackson
February 25, 2008
announcements

none today

reminders

• project 1B due Thursday
• project lab on Friday
plan for today

topics

• response to feedback
• how to design a traffic signal, continued
• fault tolerance
• how to design a type system
response to feedback

a student asked: when is a model too simple?

two views of modelling

‣ ontological: trying to represent truth, what exists
‣ operational: build a model for some purpose

taking operational view

‣ what’s model for? exploring problem? preempting implementation issues? ensuring correctness?
‣ you need to model the things that matter for the purpose at hand

Doug Ross (Structured Analysis, 1974)
‣ M models A to the extent that M correctly answers questions about A
asynchronous traffic lights
problem

road works
• road narrows to one lane
• workers have flags but can’t see each other

a strategy
• start out one green, one waiting
• worker gives last car to get through a message and shows red flag
• worker at other end gets message and shows green flag
state machine model

**events**

- **r0**: worker 0 raises red flag
- **send0**: worker 0 sends message to worker 1
- **recv0**: worker 0 receives message from worker 1
product machine

showing reachable states only

- still deterministic
- two more states than before, for messages in transit
- easy to see that invariant holds
strengthening the invariant

the desired invariant is

not GGx // x is informal way of saying that other process can have any state

exercise

• is this inductive?
• give example of violating transition
• show how to strengthen

solution

• bad transition
  recv0 from WGB on GGI
• can’t happen: when second process is green, no message waiting for first
• strengthen invariant: if there’s a message, both are waiting
  if message in A or B, then flags in WW
textual form on right

- same meaning as diagrams
- (but gave buffer process richer state)
- text generally more flexible & scalable

notational issues

- multiset: unordered collection that allows duplicate elements
- {0,1}: type used to index workers

key difference

- buffer now unbounded
- was just 3 states in diagram version

state

f[0], f[1]: {G, R, W}
buf: MultiSet<{0,1}>

init


op send (i: {0,1})
   when f[i] = R
   do f[i] = W; buf = buf + {1-i}

op recv (i: {0,1})
   when f[i] = W and i in buf
   do f[i] = G; buf = buf - {i}

op r (i: {0,1})
   when f[i] = G
   do f[i] = R
proof by induction

to check that op preserves invariant
  \• assume invariant holds before
  \• ... and firing condition (\textit{when}) is true
  \• ... and transition occurs (\textit{do})
  \• ... then show invariant holds after

example
  \• shown on right
  \• can you do others yourself?

key property of this analysis
  \• one operation at a time
  \• it's \textit{modular}

\begin{verbatim}
\textbf{invariant}
  \textbf{not} (f[0] = f[1] = G) \quad 1
  f[0] = W \textbf{ or } f[1] = W \quad 2
  buf = {} \textbf{ or } f[0] = f[1] = W \quad 3
  \#buf \leq 1 \quad 4

\textbf{initially}
  -- holds trivially

\textbf{checking send}
  \textbf{op} send (i: \{0,1\})
    \textbf{when} f[i] = R \quad 5
    \textbf{do} f[i] = W; buf = buf + \{1-i\} \quad 6
  -- 1 follows from 6
  -- 2 follows from 6
  -- 3 follows from 2, 5, 6
  -- 4 follows from 3, 5, 6
\end{verbatim}
modelling faults

possible faults

• dropped message (driver forgets, or veers off road)
• duplicated message (forged by mischievous driver)

can we model these?

• yes, just make ops non-deterministic (eg, send may insert dups)
• or add new ops (eg, duplicate message)

```
op send (i: {0,1})
  when f[i] = R
  do f[i] = W; add {1-i} zero or more times to buf
```
fault tolerant systems

system = nodes + network

what can go wrong?
• code is buggy, so all nodes can do the wrong thing
• node gets stuck or dies
• node misbehaves ("byzantine fault")
• network duplicates or drops messages

standard interventions
• to avoid bugs: good testing, reasoning & testing
• to avoid runtime faults: replication + special algorithms

example of node misbehaving
• http://en.wikipedia.org/wiki/Clayton_Tunnel_rail_crash
how to design a type system
the problem

what's a type error?

• misapplying operation to data, eg: \texttt{23+"5"}

approaches

• dynamic typing: types checked at runtime (eg, Python, Scheme)
• static typing: types checking at compile-time (eg, C, Pascal, ML)
• strongly-typed ("safe"): no type errors missed (Scheme, ML, Java)

Java

• is strongly-typed
• mostly static, but some dynamic checks
  (casts, array stores, reflection, casts inserted by generic typing)
how to do static typing

in object-oriented language

‣ basic operations are method calls
‣ objects carry their own methods
‣ so no-type-error = called-method-exists

simplest approach

‣ no subtyping (interfaces, inheritance)
‣ in class, declare methods and types of their args
‣ declare type of every variable
‣ rules

\[
\begin{align*}
T & \ u; \quad \text{// set } \ declType(u) = T \\
u & = v; \quad \text{// check } \ declType(u) = declType(v) \\
... \ u.m() ... & \text{// check } \ m \in \text{methods}(declType(u))
\end{align*}
\]
class IntList {
    IntList () {...}
    void add (Integer elt) {...}
    void set (Integer index, Integer elt) {...}
    boolean contains (Integer elt) {...}
}

class IntSet {
    IntSet () {...}
    void add (Integer elt) {...}
    boolean contains (Integer elt) {...}
}

class Main {
    public static void main (String[] args) {
        IntList l = new IntList ();
        IntSet s = new IntSet ();
        IntSet t = new IntSet ();
        ...
        if (s.contains(3)) {
            t = s;
            t.set (1, 4);
        }
        else {
            l.set (1, 4)
            t = l;
            t.add (3);
        }
    }
}

can you find
    • a type error that would result in a runtime failure?
    • a type error that wouldn’t?
introducing subtyping

this scheme is too restrictive
• can’t make plugins
• can’t design generic collections

so, organize types in hierarchy
• classes and interfaces declared with extends and implements
  
  class S extends T {...}
  class S implements T {...}

• define subtyping relation
  
  $S \leq T$ when $S = T$, $S$ extends $T$, $S$ implements $T$, or $S \leq X$ and $X \leq T$

• idea: if $S \leq T$, then $S$ has at least the methods of $T$

• so can accept an $S$ when expecting a $T$
example

```java
interface Collection {
    boolean contains (Object e);
    boolean add (Object e);
    boolean retainAll (Collection c);
}

interface List extends Collection {
    Object set (int i, Object e);
}

interface Set extends Collection {
}

class ArrayList implements List {
    ArrayList () {...}
    void ensureCapacity (int cap) {...}
    ...
}

class HashSet implements Set {
    HashSet () {...}
    boolean add (Object e) {...}
    ...
}
```

exercises

- draw the subtype relation
- why does retainAll take Collection?
- what are the methods of List?
class Main {
    public static void main (String[] args) {
        List l = new ArrayList();
        Set s = new HashSet();
        l.add (3);
        s.add (3);
        l.ensureCapacity (10);
        s.ensureCapacity (10);
        Collection c;
        c = l;
        c.set (1, 4);
    }
}

consider each statement
	- which are rejected?
	- which would result in failures?
making sense of subtypes

runtime vs. compile-time types

• no longer need to match
• distinguish these
  \[\text{declType}\ (x): \text{the declared type of variable } x\]
  \[\text{runType}\ (x): \text{the type of the object } x \text{ refers to}\]
• note that runtime type is always a \textbf{class}
• declaration type may be an interface or abstract class

new rules

\[
\begin{align*}
T \ u; & \quad \text{// set } \text{declType}\ (u) = T \\
u = v; & \quad \text{// check } \text{declType}\ (v) \leq \text{declType}\ (u) \\
... \ u.m() ... & \quad \text{// check } m \in \text{methods(declType}\ (u))
\end{align*}
\]

why does this work?

• subtype always has at least the supertype’s methods
type soundness

but are we sure?
• is the type system really sound?
• how would you prove it?

what are we trying to establish?
• that at every point
  \( \text{runType} (u) \leq \text{decltype} (u) \)
• looks like an ... invariant!

proving preservation
• one statement kind at a time
• argument involves type rules and execution semantics

sample invariant reasoning
for \( u = v \)
assume that before execution
\( \text{runType} (v) \leq \text{decltype} (v) \)
then by type checking rule
\( \text{declType} (v) \leq \text{decltype} (u) \)
and so
\( \text{runType} (v) \leq \text{decltype} (u) \)
and by execution semantics
\( \text{runType} (u) = \text{runType} (v) \)
and therefore after
\( \text{runType} (u) \leq \text{DECLTYPE} (u) \)
strengthening invariant

consider inductive step for

\[ u = v.f \]

suppose we do a proof as before

\[ \text{what fails?} \]
\[ \text{need to know something about runType}(v.f) \]
\[ \text{so, stronger invariant needed} \]
\[ \text{must also claim that fields are well typed} \]
summary
what did we do?

designed a traffic signalling scheme

• using invariants -- very powerful technique

invariants

• an invariant describes a state set
• all reachable states should be in set
• can reason inductively, one operation at a time
• may need to strengthen invariant

fault tolerance

• noted that our scheme is not fault-tolerant

type soundness for programming languages

• same technique, where state machine is executing program
lecture exercise

by next lecture (Wednesday) in your LNBs

traffic signals

• [easy] do the inductive proofs for the other operations
• [moderate] consider both faults (message drop and duplication) and see what effects they have on the proof

type soundness

• [easy] prove preservation of the type soundness invariant for statements with type casts of the form $u = (t) v$

clayton railway accident

• [hard, optional] model the protocol and explain what went wrong