Statically Type-Safe Virtual Types in Object-Oriented Languages

Kim Bruce
Williams College
Joint work with Joe Vanderwaart
Outline

- OO type system limitations
- Systems of classes
  - Patterns
- MyType adds flexibility
- Systems & Inheritance
Static OO Type Systems generally restrictive

- Java, C++, Object Pascal
  - No changes to parameter or instance variable types in subclasses
  - C++ allows covariant changes to return types.

- Restrictions get in the way of using inheritance.
Subject-Observer Pattern

- Reflects Java event-driven programming
- Subject (GUI component) generates events
  - E.g., button, scrollbar, etc.
- Observer (Listener) is notified of events and reacts
Subject-Observer in Java

class Subject {
    Observer[] observers;  ...
    void addObserver(Observer newObs) {...}
    void notifyObservers(Event e) {
        int len = observers.length;
        for (int i = 0; i < len; ; i++)
            observers[i].notify(this,e);
    }
}

class Observer {
    ...
    void notify (Subject subj, Event e) {
        {... // make approp response }
    }
}
Can we specialize?

- Suppose want Choice menus as subjects
- Observers expect specialized events
  - Get item selected in menu.
- Need to specialize all simultaneously
  - But Java doesn’t allow specializing types!
Specialized Subject-Observer

class Choice {
    ItemObserver [] observers;  ...
    void addItemObserver(ItemObserver newObs) {
        ...
    }
    void notifyItemObservers(ItemEvent e) {...}
    String getItem() {...}
}

class ItemObserver {
    ...
    void itemStateChanged (Choice subj,
                            ItemEvent e) {
        {... e.getItem()  ... subj.getItem() ...};
    }
}
Why not use inheritance?

class Choice extends Subject {
    ItemObserver [] observers; ...
    void addObserver(ItemObserver newObs) {
        ...
    }
    void notifyObservers(ItemEvent e) {
        ...
    }
    String getItem() {...}
}

class ItemObserver extends Observer {
    ...
    void notify (Choice subj, ItemEvent e)
    {... e.getItem() ... subj.getItem() ...};
}
Type changes illegal in Java!

- Java won’t allow changes to types of
  - instance variables
  - methods
- Example made **covariant** changes to:
  - type of instance variable
  - parameter types of methods
- Now what?
Virtual classes - Beta

class Subject {
    typedef ObType as Observer;
    typedef EventType as Event;
    ObType[] observers; ...
    void notifyObservers(EventType e) {
        int len = observers.length;
        for (int i = 0; i < len; i++)
            observers[i].notify(this, e);
    }
}

class Observer {
    typedef SubType as Subject;
    typedef EventType as Event;
    void notify (SubType subj, EventType e) {
        ...;
    }
}
Inheritance & Virtual Classes

class Choice extends Subject {
    typedef ObType as ItemObserver;
    typedef EventType as ItemEvent;
    String getItem() {...}
}
class ItemObserver extends Observer {
    typedef SubType as Choice;
    typedef EventType as ItemEvent;
    void notify (SubType subj, EventType e) {
        ... e.getItem() ... subj.getItem();
    }
}
Not Statically Type-Safe!

Observer obs = new ItemObserver();
Subject subj = new Subject();
Event evt = new Event();

obs.notify( subj, evt );

**Crashes** when send `getItem` to `subj`

- Because `notify` body of `obs` sends `getItem` to `subj` & `evt`. 
What went wrong?

- Covariant change of parameter type in subclass.
- Requires run-time check.
- Can we catch the error statically?
- Problem shows up in other examples,
  - E.g., interpreters using Visitor pattern
LOOM & LOOJ

- Expressive, yet statically type-safe.
- **LOOM**
  - Introduce *MyType* as type (`interface`) of `self`.
  - T means “exactly T”, #T means T or extension.
- **LOOJ - Java extension w/similar type system**
  - *ThisType* as type of `this`, @T means exact type.
- Developed at Williams:
  - ....., Leaf Petersen, Joe Vanderwaart, Jon Burstein, Nate Foster, Doug Thunen, Rob Gonzalez, Diane Bennett
Singly and doubly-linked nodes in LOOM

Node = ObjectType{
    getValue : func():integer;
    setValue : proc(integer);
    setnext : proc(MyType);
    getnext : func():MyType;
}

DblNode = ObjectType include Node {
    getprev : func():MyType;
    setprev : proc(MyType); }
Singly-linked nodes

class NodeClass {
  var
    value = 0: integer;
    next = nil: MyType;
  methods
    setValue = procedure(newValue: integer)
    { value = newValue; } 
    getValue = function():integer
    { return value;}
    setNext = procedure(newNext: MyType)
    { next = newNext;}
    getNext = function():MyType
    { return next; } }

Doubly-linked nodes

class DbleNodeClass inherits NodeClass
  overrides setNext {
    var
      prev = nil: MyType;
  methods
    setNext = procedure(newNext: MyType)
      { next = newNext;
        if newNext != null then
          next.setPrev(self); }
    setPrev = procedure(newPrev: MyType)
      { prev = newPrev; }
    getPrev = function(): MyType
      { return prev; }  
  }
Type safe?

Watch out for subtyping!

```javascript
breakIt = function(first, second: Node) {
    first.next = second;
}

If singleNode: Node, dbleNode: DblNode
    breakIt(dbleNode, singleNode)
causes run-time error!
```
Type safe?

- Problem is subtyping!
  - No problems if use exact types.
- $x : T$ means $x$ must be $T$ only
- $x : \#T$ means $x$ must be $T$ or extension.
- Write $T <\# U$ to mean $T$ extends $U$.
- Subsumption:
  - $x : T$ implies $x : \#T$
  - $T <\# U$ and $x : \#T$ implies $x : \#U$
Binary Methods

- Method w/ parameter of type `MyType`
- Restriction on binary methods:
  - if `m` has parameter w/type `MyType`, then can only send to exact type.
- Thus if `x: #Node` then `x.setNext(y)` illegal
- Thus,

```javascript
breakIt = function(first, second: #Node) {
  first.setNext(second);
}
```

Illegal!
MyType => Homogeneity

- Use of MyType requires homogeneity
- Generalized “breakIt” fine:
  
  ```javascript
  gb //< breakIt = function(N <# Node,
  
  first, second: N) {
    first.setNext(second);
  }
  ```
- Use #-types where want flexibility!
- Type check under weak assumptions on
Virtual types generalize MyType!

- Can think of MyType as recursive type.
- Semantics of object types with method suite given by $M(MyType)$ is:
  $$\mu MT. \exists Y. Y \times (Y \rightarrow M(MT))$$
- Virtual types like mutually recursive types.
- Extend LOOM to support groups of types.
Type groups

SubjObsGrpTp = TypeGroup {
  ObjectType(MySubject) {
    addObserver: proc(MyObserver);
    notifyObservers: proc(MyEvent);
  }

  ObjectType(MyObserver) {
    notify: proc(MySubject, MyEvent);
  }

  ObjectType(MyEvent) {
    ...
  }
}
class SubjectClass generates MySubject in SubjObsGrpTp {

    var
        observers: MyObserver[ ];  ...

    methods
        addObserver = proc(newObserver: MyObserver)
            {   ...   }
        notifyObservers = proc(evt: MyEvent) {   ...   }

}
Extending type groups

ChoiceSubjObsGrpTp = TypeGroup extends SubjObsGrpTp
{
    ObjectType(MySubject) with ObjectType {
        getItem: func(): String;
    }

    ObjectType(MyEvent) with ObjectType {
        getItem: func(): String;
    }
}
class ChoiceSubjectClass generates MySubject in ChoiceSubjObsGrpTp inherit SubjectClass {

  var
  ...

  methods
  getItem: func(): String {...}

}
Same restrictions as before!

- Important not to mix components from different groups!
- If method has My... parameter, then can only send to object with exact type.
- Type checking based on weak assumptions on My... types:
  - Guarantees extensions type-safe!
Can use type groups with polymorphism

class doSomething(TG <# SubjObsGrpTp)
{
    ... TG.MySubject ...
    ... TG.MyObserver ...
}
Related Work

- Virtual classes introduced in Beta.
- Proposed as Java extension to support generics by Thorup and Torgersen.
- Igarashi & Pierce: Foundations for Virtual Types.
  - Captured different aspects of virtual types.
  - Use for replacing generics.
  - Uses exotic type theory & limited to functional languages.
- Odersky et al: A Nominal Theory of Objects with Dependent Types - *undecidable type theory*
- Remy & Vouillon: O’Caml
Summary

- Type groups allow definition (and extension) of mutually referential types and classes in statically type-safe way.
- Simple generalization of MyType.
- System can be proved safe by writing formal semantics.
- Extension suggested by formal semantics:
  - Replace recursive type for MyType by mutual recursion.
Type-checking rules

\[
C', E \vdash o: \text{TPG'.MT}_i, \quad C' \vdash \text{TPG'} \prec \# \text{TPG}
\]

\[
C, E \vdash o.m: U[\text{TPG'.MT} / \text{MTG.MT}]
\]

\text{where} \quad \text{TPG.MT}_i = \{\ldots, m: U, \ldots\}
Type-checking rules

\[ C', E \vdash iv : IV(MTG.MT), \]
\[ C', E' \vdash m : M(MTG.MT) \]

\[ \frac{C, E \vdash \text{class generates } TG.MT_i(iv,m) : \text{ClassType}(TG, IV(MT), M(MT))}{\text{where } C' = C \cup \{ MTG <# TG \}, \quad E' = E \cup \{ \text{self: MTG.MT}_i \}} \]