Extending Dynamic Constraint Detection with Polymorphic Analysis

Nadya Kuzmina and Ruben Gamboa

May 22, 2007

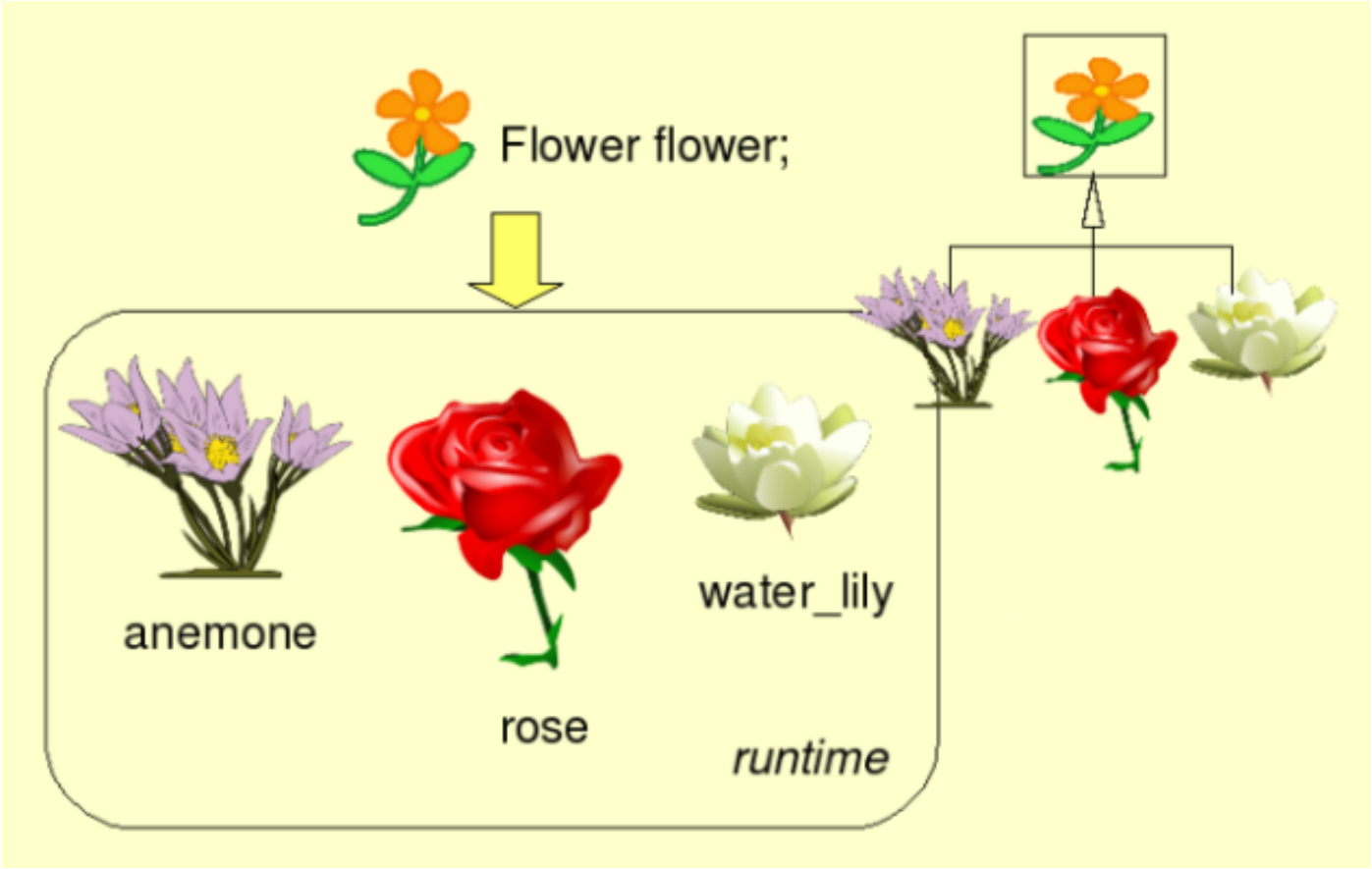
Department of Computer Science
University of Wyoming

Goal

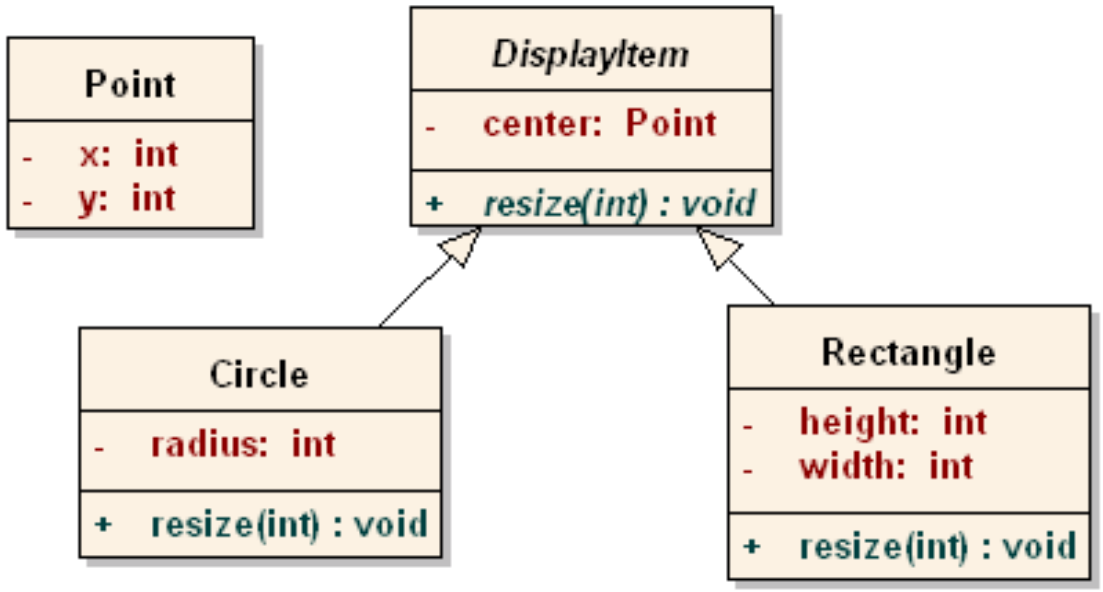
- Recover polymorphic constraints that provide insight into the runtime behavior of an object-oriented system.
- Turnip, our prototype implementation for polymorphic constraint detection, is based on Daikon.

Polymorphism in Object-Oriented Languages

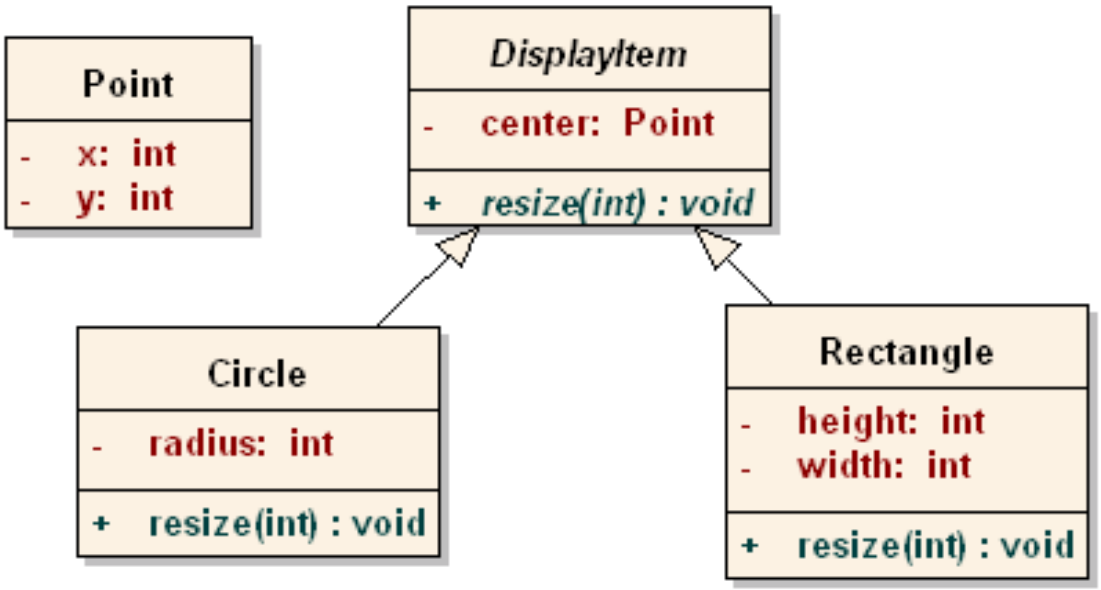
- Object-oriented languages make extensive use of polymorphism.
- Polymorphism means that the actual runtime type of a program variable may be different from its declared type.



Polymorphic Behavior: `DisplayItem` Example



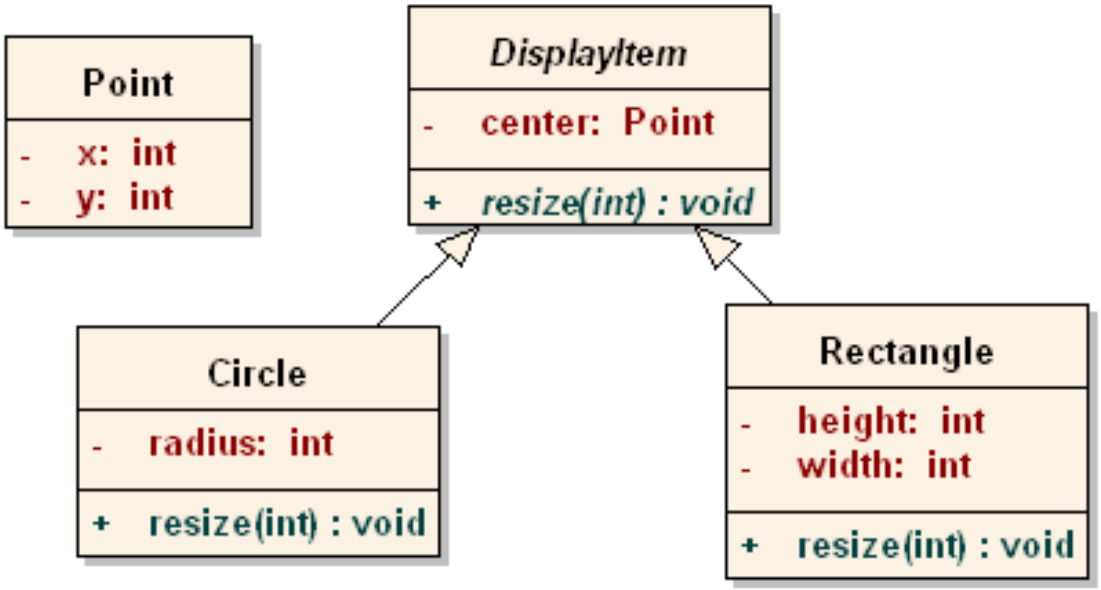
Polymorphic Behavior: `DisplayItem` Example (Cont'd)




```
// graphical component
DisplayItem figure;
// resizing factor
int amount;

// user adjusted the size
// of the component by amount
void redraw() {
    ...
    figure.resize(amount);
    ...
}
```

Polymorphic Behavior: `DisplayItem` Example (Cont'd)




```
// graphical component
DisplayItem figure;
// resizing factor
int amount;

// user adjusted the size
// of the component by amount
void redraw() {
    ...
    figure.resize(amount);
    ...
}
```

redraw():::EXIT

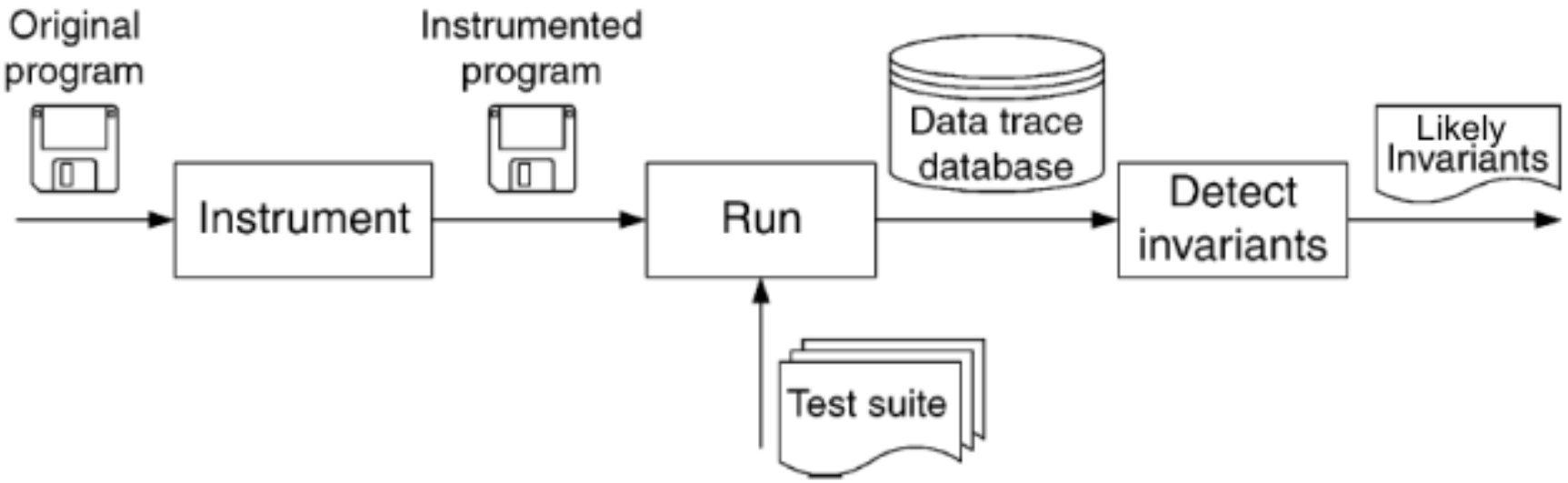
```
( figure.class == Circle ) ==>
 ( figure.radius == figure.radius@pre * amount )
( figure.class == Rectangle ) ==>
 ( figure.width == figure.width@pre * amount )
( figure.class == Rectangle ) ==>
 ( figure.height == figure.height@pre * amount )
```

The Challenges of Polymorphism

- The declared type of a polymorphic variable may not fully characterize the variable's behavior.
- Polymorphic behavior requires examining the actual runtime type of a program variable to grasp meaningful constraints on it.

Daikon

Daikon is a general and publicly available implementation for dynamic invariant detection, which was developed by Michael Ernst at the University of Washington and is maintained by Michael Ernst and his research group at MIT. [Ernst et al.]



Daikon and Polymorphism

- Daikon considers only the declared type of a variable when instantiating properties.
- `dfej` used to provide a solution to capturing the runtime behavior:
  - Assume that a variable's runtime value can be guaranteed to be of one specific type.
  - Specify the refined type via an annotation:
    */*refined_type: Rectangle*/* `DisplayItem figure;`

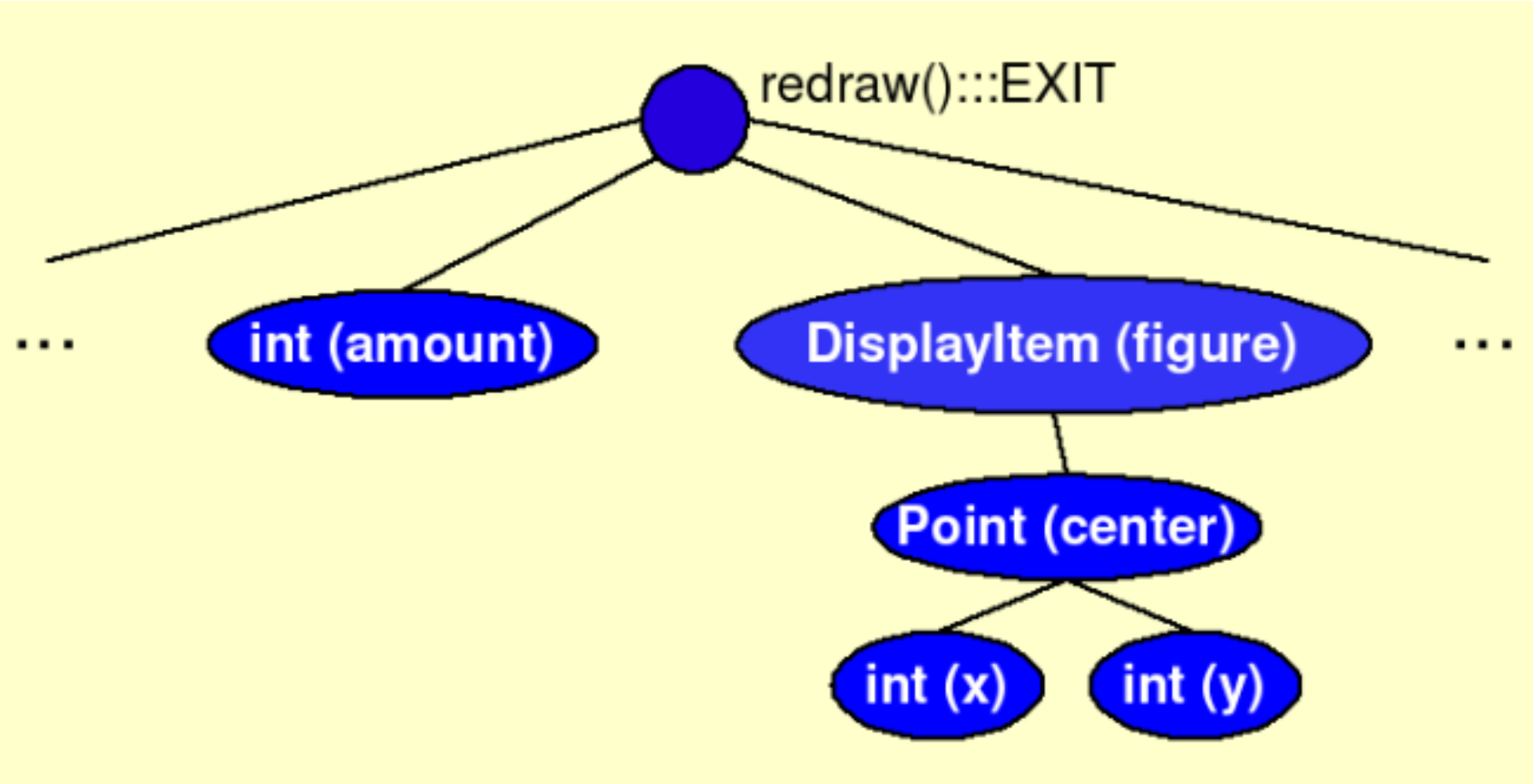
Meet Turnip

- Turnip extends Daikon to examine the fields of runtime objects to infer the constraints that likely hold between the variables.
- Turnip examines the actual runtime class of each polymorphic program variable to infer the properties that likely hold for the fields of the examined runtime class, which we call *runtime-refined constraints*.
  Example:

```
( figure.class == Circle ) ==>
  ( figure.radius == figure.radius@pre * amount )
```

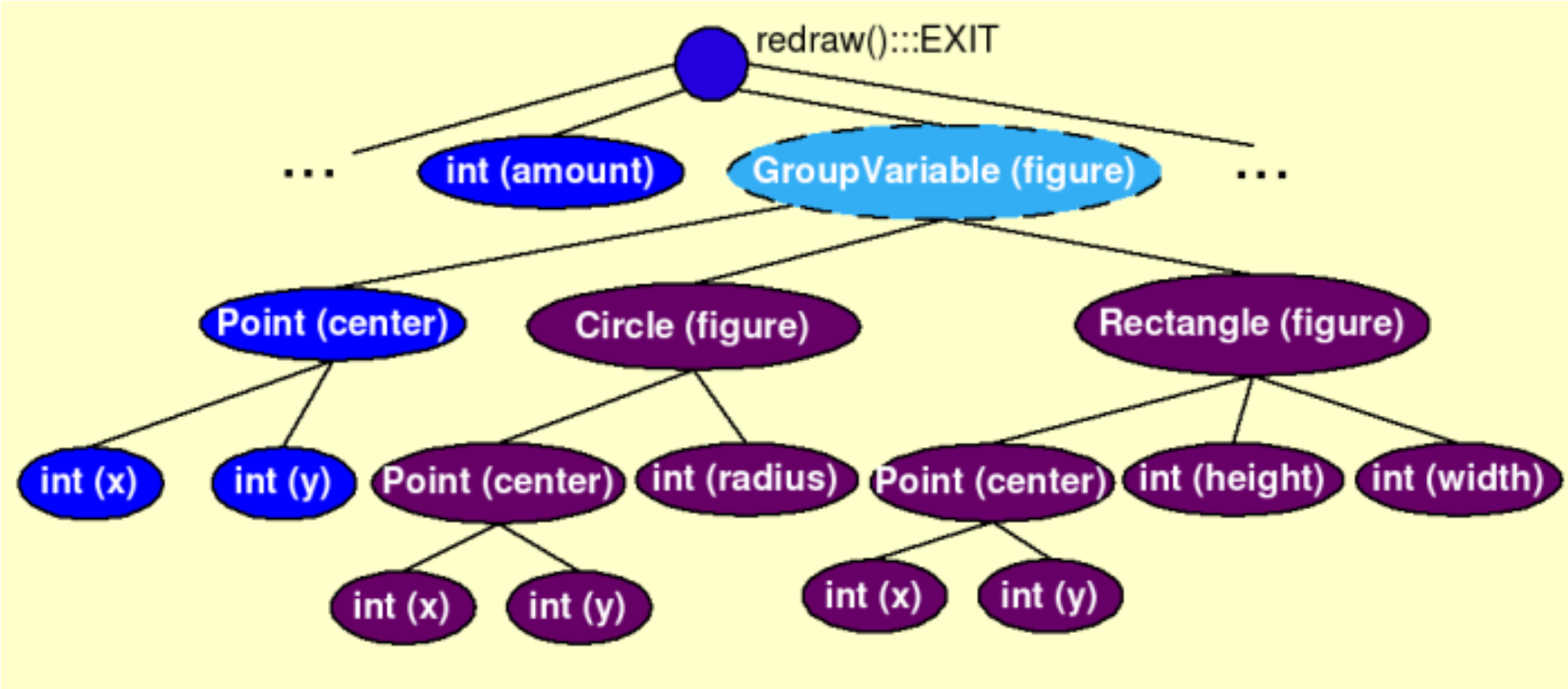
- Turnip discovers the same constraints on common variables as Daikon because Turnip analyzes a superset of Daikon's variables.

Implementation: Chicory Variable Tree



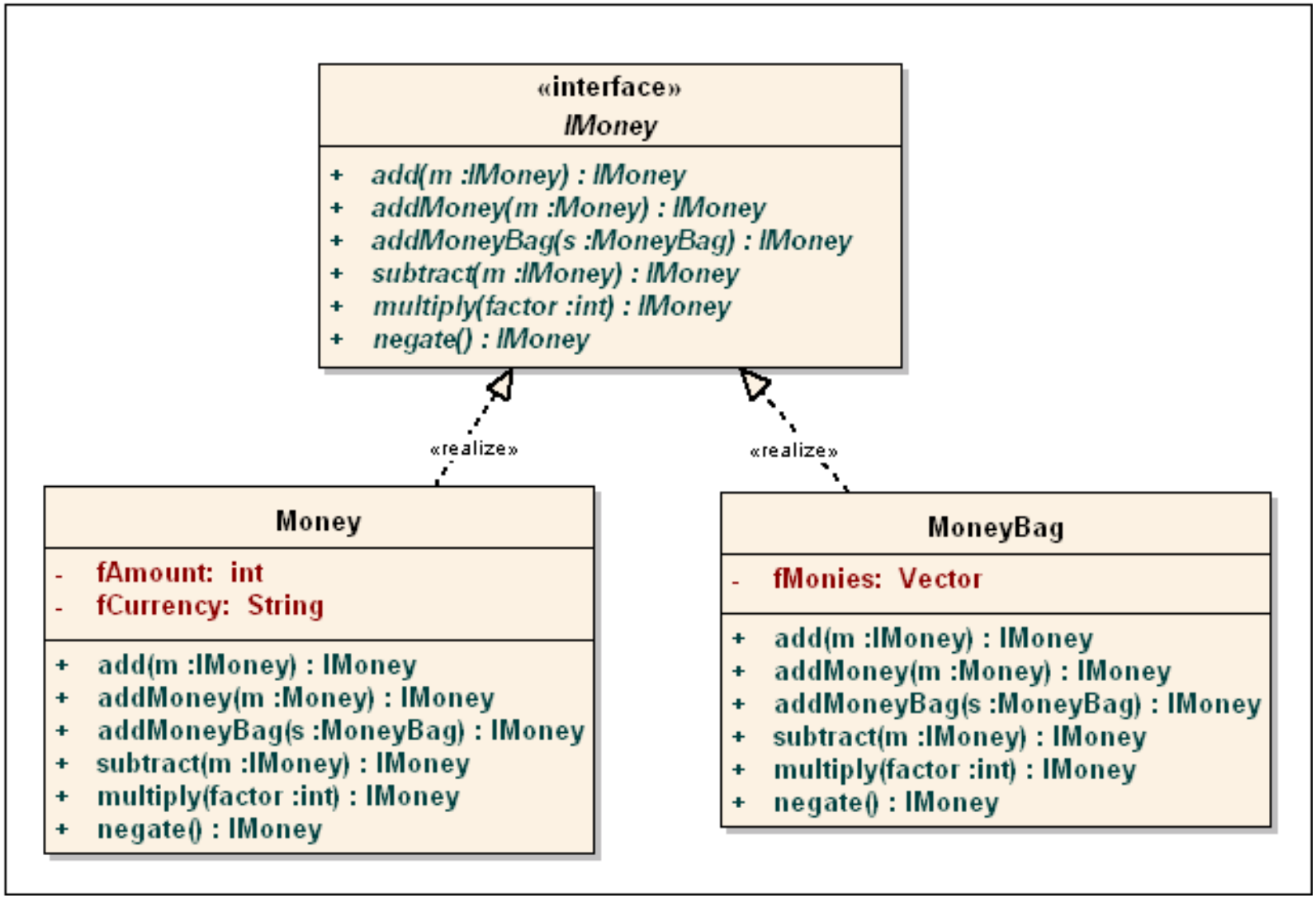

```
DisplayItem figure;
int amount;
void redraw() { ...
    figure.resize(amount);
... }
```

Implementation: Turnip Variable Tree




```
DisplayItem figure;
int amount;
void redraw() { ...
    figure.resize(amount);
... }
```

Results: Introducing the `Money` Example



The `Money` class represents a quantity of money in a particular currency. The `MoneyBag` class represents a collection of monies in different currencies. [Beck and Gamma, 1998]

Constraints by both Daikon and Turnip

Turnip discovers the same constraints on common variables as Daikon:

```
IMoney MoneyBag.addMoney(Money m):::EXIT
m.fAmount == orig(m.fAmount)
m.fCurrency == orig(m.fCurrency)
m.fAmount != 0
this.fMonies[].sum == orig(this.fMonies[].sum)
return != null
```

Constraints by Turnip but Not Daikon

Turnip (but not Daikon) discovered the following runtime-refined constraints as postconditions:

```
IMoney MoneyBag.addMoney(Money m):::EXIT
( return.class == MoneyBag ) ==>
    ( m.fAmount - return.fMonies[].sum + this.fMonies[].sum == 0 )
( return.class == Money ) ==>
    ( m.fAmount - return.fAmount + this.fMonies[].sum == 0 )
```

Polymorphic Constraints for `MoneyBag`

Method	*Postcondition*	*Daikon*	*Turnip*
IMoney add(IMoney m)	(m.class == MoneyBag & return.class == MoneyBag) ⇒ (m.fMonies[].sum - return.fMonies[].sum + this.fMonies[].sum == 0)	–	√
	(m.class == MoneyBag & return.class == Money) ⇒ (m.fMonies[].sum - return.fAmount + this.fMonies[].sum == 0)	–	√
	(m.class == Money & return.class == MoneyBag) ⇒ (m.fAmount - return.fMonies[].sum + this.fMonies[].sum == 0)	–	√
	(m.class == Money & return.class == Money) ⇒ (m.fAmount - return.fAmount + this.fMonies[].sum == 0)	–	√
IMoney addMoney(Money m)	(return.class == MoneyBag) ⇒ (m.fAmount - return.fMonies[].sum + this.fMonies[].sum == 0)	–	√
	(return.class == Money) ⇒ (m.fAmount - return.fAmount + this.fMonies[].sum == 0)	–	√
IMoney addMoneyBag(MoneyBag s)	(return.class == MoneyBag) ⇒ (s.fMonies[].sum - return.fMonies[].sum + this.fMonies[].sum == 0)	–	√
	(return.class == Money) ⇒ (s.fMonies[].sum - return.fAmount + this.fMonies[].sum == 0)	–	√
IMoney subtract(IMoney m)	(m.class == MoneyBag & return.class == MoneyBag) ⇒ (m.fMonies[].sum + return.fMonies[].sum - this.fMonies[].sum == 0)	–	√
	(m.class == MoneyBag & return.class == Money) ⇒ (m.fMonies[].sum + return.fAmount - this.fMonies[].sum == 0)	–	√
	(m.class == Money & return.class == Money) ⇒ (m.fAmount + return.fAmount - this.fMonies[].sum == 0)	–	√
IMoney multiply(int factor)	(return.class == MoneyBag) ⇒ (return.fMonies[].sum == (this.fMonies[].sum * factor))	–	√
IMoney negate()	(return.class == MoneyBag) ⇒ (return.fMonies[].sum == – this.fMonies[].sum)	–	√

Polymorphic Constraints for `Money`

Method	*Postcondition*	*Daikon*	*Turnip*
IMoney add(IMoney m)	(m.class == MoneyBag & return.class == MoneyBag) ⇒ m.fMonies[].sum - return.fMonies[].sum + this.fAmount == 0	–	√
	(m.class == MoneyBag & return.class == Money) ⇒ (m.fMonies[].sum - return.fAmount + this.fAmount == 0)	–	√
	(m.class == Money & return.class == MoneyBag) ⇒ (m.fAmount - return.fMonies[].sum + this.fAmount == 0)	–	√
	(m.class == Money & return.class == Money) ⇒ (m.fAmount - return.fAmount + this.fAmount == 0)	–	√
IMoney addMoney(Money m)	(return.class == Money) ⇒ (m.fCurrency == return.fCurrency)	√	√
	(return.class == Money) ⇒ (m.fAmount - return.fAmount + this.fAmount == 0)	√	√
	(return.class == MoneyBag) ⇒ (m.fAmount - return.fMonies[].sum + this.fAmount == 0)	√	√
IMoney addMoneyBag(MoneyBag s)	(return.class == MoneyBag) ⇒ (s.fMonies[].sum - return.fMonies[].sum + this.fAmount == 0)	–	√
	(return.class == Money) ⇒ (s.fMonies[].sum - return.fAmount + this.fAmount == 0)	–	√
IMoney subtract(IMoney m)	(m.class == MoneyBag & return.class == Money) ⇒ (m.fMonies[].sum + return.fAmount - this.fAmount == 0)	–	√
	(m.class == Money & return.class == Money) ⇒ (m.fAmount + return.fAmount - this.fAmount == 0)	–	√
IMoney multiply(int factor)	(return.class == Money) ⇒ (return.fAmount == (this.fAmount * factor))	–	√
IMoney negate()	(return.class == Money) ⇒ (return.fAmount == – this.fAmount)	–	√

Daikon's Success with `Money.addMoney` Method

Daikon successfully inferred polymorphic constraints for the `addMoney` method in the `Money` class.

```
IMoney Money.addMoney(Money m):::EXIT
(return.class == Money) ==>
   (m.fCurrency == return.fCurrency)
(return.class == Money)  ==>
   (m.fAmount - return.fAmount + this.fAmount == 0)
(return.class == MoneyBag)  ==>
   (m.fAmount - return.fMonies[].sum + this.fAmount == 0)
```

Daikon's Success with `Money.addMoney` Method (Cont'd)

Daikon successfully inferred polymorphic constraints for the `addMoney` method in the `Money` class.

```
IMoney Money.addMoney(Money m):::EXIT
(return.class == Money) ==>
   (m.fCurrency == return.fCurrency)
(return.class == Money)  ==>
   (m.fAmount - return.fAmount + this.fAmount == 0)
(return.class == MoneyBag)  ==>
   (m.fAmount - return.fMonies[].sum + this.fAmount == 0)
```

```
public class Money implements IMoney {
    ...
    public IMoney addMoney(Money m) {
        if (m.currency().equals(currency()) )
            return new Money(amount()+m.amount(), currency());
        return MoneyBag.create(this, m);
    }
    ...
}
```

Limitations

- Turnip considers only user-defined hierarchies of classes.
- More variables per program point result in slower performance and more irrelevant properties than Daikon.
- Turnip assumes that all inherited fields are used for specialized purposes in the subclasses.

Conclusions and Future Work

- Our results suggest that examining polymorphic behavior results in better accuracy of dynamically inferred specifications for object-oriented systems.
- Our future work will explore the use of static program analysis to improve the quality of the reported constraints.

Thank you!

Questions?

Daikon and Turnip: Comparative Analysis

?