

# Safe Dynamic Multiple Inheritance

Erik Ernst

TOOL – May 2005



# Overview

- Idea
- How does it look?
- It can be made safe – overview
- Basic entities
- Results
- Conclusion



# Overview

- Idea
- How does it look?
- It can be made safe – overview
- Basic entities
- Results
- Conclusion



## The idea behind this work

- Dynamic multiple inheritance is useful
- It can be made safe
- This is about how to do it



## Assume basic functionality + $N$ features

- E.g., Entry in a calendar (alarm, privacy, group, repeat . . .)
- Could be expressed as  $2^N$  classes using multiple inheritance
- If MI is dynamic, only  $N$  classes needed
- Problem: requires dynamic MI, and safety is non-trivial
- Relevant at multiple levels: class, class family, system . . .



## Dynamic MI as a typing issue

- Type systems establish compile-time guarantees
- Typical: “this object is typable as  $T$ ”
- But also: “the value of these two expressions can be added”
- Consider: “. . . can be divided”
- Here: “the value of these expressions can be used for MI”
- Result: automated proof of certain kinds of safety



# Overview

- Idea
- How does it look?
- It can be made safe – overview
- Basic entities
- Results
- Conclusion



## How does it look?

```
Point: (# x,y: @integer #);  
ColorPoint: Point(# color: @string #);  
  
addColor:  
  (# inClass,outClass: ##Point;  
  enter inClass##  
  do inClass & ColorPoint ## -> outClass##  
  exit outClass##  
  #)
```



## How does it look? (cont'd)

```
(# myClass: ##Point;  
  myPoint: ^Point;  
do ClickablePoint## -> addColor -> myClass##;  
  myClass [] -> myPoint []  
#)
```



# Overview

- Idea
- How does it look?
- It can be made safe – overview
- Basic entities
- Results
- Conclusion



## It can be made safe

- Basic entities: Mixins and patterns
- Mixins are partially ordered, globally
- Patterns are ordered sets of mixins
- Some patterns are totally determined by the global order
- Such a pattern is safe to combine with any pattern



# Overview

- Idea
- How does it look?
- It can be made safe – overview
- **Basic entities**
- Results
- Conclusion



## Basic entities: Mixins

- A mixin is a *MainPart*, ( $\# \dots \#$ ), with enclosing part object
- Here, we just consider mixins primitive, as a set  $\mathcal{M}$
- There is a partial order, ' $\preceq$ ', on  $\mathcal{M}$
- $m_1 \preceq m_2$  iff  $m_1$  is able to depend on  $m_2$
- Ex:  $(\# \text{color}: @\text{string}\#) \preceq (\# x,y: @\text{integer}\#)$

```
Point: (# x,y: @integer #);  
ColorPoint: Point(# color: @string #);
```



## Basic entities: Patterns

### Definition (Pattern).

The set of patterns,  $\mathcal{P}$ , is defined to be the set of ordering relations on finite subsets of  $\mathcal{M}$  such that each pattern  $P \in \mathcal{P}$  satisfies the following criteria:

$$\forall x \in \text{supp}(P), y \in \mathcal{M} \quad . \quad x \preceq y \Rightarrow y \in \text{supp}(P) \quad (1)$$

$$\forall x, y \in \text{supp}(P) \quad . \quad x \preceq y \Rightarrow x \preceq_P y \quad (2)$$

$$\forall x, y \in \text{supp}(P) \quad . \quad x \preceq_P y \vee y \preceq_P x \quad (3)$$



## Pattern combination, '&', as an algorithm

```
fun merge ([]: int list) (ys: int list) = ys
  | merge xs [] = xs
  | merge (xxs as x::xs) (yys as y::ys) =
    if x=y then x::(merge xs ys)
    else if not (member y xs) then y::(merge xxs ys)
    else if not (member x ys) then x::(merge xs yys)
    else raise Inconsistent;

fun member x [] = false
  | member x (y::ys) =
    if x=y then true else member x ys;
```



## Pattern combination, formally

### Definition ('&').

Given two total order relations  $R_1$  and  $R_2$ , the combination of them is defined as follows:

$$R_1 \& R_2 \triangleq R \cup (R_{21} \setminus R^{-1}) \quad (4)$$

where  $R \triangleq (R_1 \cup R_2)^*$ , i.e., the transitive closure of the union of  $R_1$  and  $R_2$ ,  $R_{21} \triangleq \text{supp}(R_2) \times \text{supp}(R_1)$ , i.e., all edges from an element in  $R_2$  to an element in  $R_1$ , and  $R^{-1}$  is the inverse relation of  $R$ , i.e.,  $\{(y, x) \mid (x, y) \in R\}$ .



# Pattern combination, examples & properties

$[1, 2] \& [\text{sub}] = [1, 2, \text{sub}]$	ord. specialization
$[\text{wrap}] \& [1, 2] = [\text{wrap}, 1, 2]$	wrapping
$[1, 2, 3] \& [a, b, c] = [1, 2, 3, a, b, c]$	concatenation
$[1, 2, 3] \& [\text{ins}, 2] = [1, \text{ins}, 2, 3]$	insertion
$[1, \text{ins}] \& [1, 2, 3] = [1, \text{ins}, 2, 3]$	rev. insertion
$\forall P, Q \text{ mergeable} . P \& Q \leq P \wedge P \& Q \leq Q$	$\sim$ min
$\forall P \leq Q . P \& Q = Q \& P = P$	$\sim$ min
$\forall P, Q \text{ mergeable} . P \& Q \& Q = P \& Q$	idempotency
$\forall P, x . P \& [x] \neq \perp \wedge [x] \& P \neq \perp$	singleton safety
$\forall P, Q . \text{rigid}(P) \Rightarrow P \& Q \neq \perp \wedge Q \& P \neq \perp$	<b>rigidity safety</b>



# Overview

- Idea
- How does it look?
- It can be made safe – overview
- Basic entities
- Results
- Conclusion



## Preliminary results

**Lemma (Preservation of total orders).**

*Given two total orders  $P$  and  $Q$ . If  $P \cup Q$  does not contain cycles then  $P \& Q$  and  $Q \& P$  are both total orders.*

**Lemma (Partial closure property of ‘&’).**

*Given two patterns  $P$  and  $Q$ . If  $P \cup Q$  does not contain cycles then  $P \& Q$  and  $Q \& P$  are both patterns.*



## Rigidity as a good thing :-)

### Definition (Rigid patterns).

*A pattern  $P$  is rigid iff the restriction of the global ordering ' $\preceq$ ' to  $\text{supp}(P)$  is a total order.*

### Lemma (Everybody agrees with a rigid pattern).

*Assume  $P$  is a rigid pattern and  $Q$  an arbitrary pattern. If  $x, y \in \text{supp}(P) \cap \text{supp}(Q)$ , then*

$$(x \preceq_P y \wedge x \preceq_Q y) \quad \vee \quad (y \preceq_P x \wedge y \preceq_Q x)$$



## Main result

**Theorem (It is safe to merge with a rigid pattern).**

*Let  $P$  be a rigid pattern and  $Q$  an arbitrary pattern. Then  $P&Q$  and  $Q&P$  are both patterns.*



# Overview

- Idea
- How does it look?
- It can be made safe – overview
- Basic entities
- Results
- Conclusion



## Conclusion

- Dynamic multiple inheritance is not only possible and useful, it is also provably safe in a significant and useful category of cases
- Questions?