Genericity

Parameterizing by Type

Generic Class

- One that is parameterized by type

  » **Works when feature semantics is common to a set of types**

- On object declaration the parameter is assigned a type

  » **For example**

    rowList : ARRAY [MATRIX_ELEMENT]

  » **We want an array of pointers to matrix elements**

  » **All the array operations for rowList are customized to use matrix elements**

Common Generic Classes

- Collection classes – classes that are collections of objects

  » **Strong typing requires specifying a type**

  » **But feature semantics is independent of type**

- Examples

  » **Sets, Stacks, Arrays, Queues, Sequences**

   rowList : ARRAY [MATRIX_ELEMENT]

   rowList : ARRAY [INTEGER]

   rowList : ARRAY [STACK [ELEPHANTS]]

Your Generic Classes

- You can write generic classes

- Why is this useful?

  » **Reuse**

    > **The basic operations (e.g. extend) are the same**

    > **Do not have to re-write the same program text over and over again**

  » **Reliability**

    > **Only write the program text once**

Generic Stack

```
class STACK [ G ] feature
    count : INTEGER -- number of elements

    empty : BOOLEAN do ... end

    full : BOOLEAN do ... end

    item : G do ... end

    put ( x : G ) do ... end

    remove do ... end
end
```

- Can use parameter **G** where ever a type is expected

Generic Array

```
class ARRAY [ P ]
create make
feature
    make ( minIndex , maxIndex : INTEGER ) do ... end

    lower, upper, count : INTEGER

    put ( value : P ; index : INTEGER ) do ... end

    infix alias "@" , item ( index : INTEGER ) : P  do ... end

end
```

Using the Generic Array

circus : ARRAY [STACK [ELEPHANTS]]
create circus . make (10 , 200)

st_el : STACK [ELEPHANTS]   -- element to put in the array
create st_el

circus . put (st_el , 30)   -- put an element into the array

st_el2 : STACK [ELEPHANTS]

st_el2 := circus @ 101   -- get an element from the array
st_el2 := circus[101]

The Type Rule – no Genericity

- Assume class **C** has the feature **f (a : T) : U do … end**

- A call of the form **x . f (d)** appearing in an arbitrary class **B** where **x** is of type **C** is type-wise correct if and only if

  » **f is available to B**

    > **exported to B**

  » **d is of type T**

    > **With inheritance d can be a descendent of T**

  » **The result is of type U**

© Gunnar Gotshalks

The Type Rule – with Genericity

- Assume **C** is generic, with **G** as its parameter and has the feature   **h (a : G) : G do ... end**

- A call to **h,** appearing in an arbitrary class **B,** will be of the form **y . h (e)** where **y** has been declared of type **C [V]**

- Then

  » **h is available to B**

    > **exported to B**

  » **e is a of type V**

    > **With inheritance d can be a descendent of V**

  » **The result is of type V**

Types of Genericity

- Types

  » **Unconstrained**

  » **Constrained**

- The previous examples showed unconstrained genericity

  » **Any type could be passed as a parameter**

Constrained Genericity

- Used when the generic type parameters must satisfy some conditions

- The following makes sense only if **P** has the feature **≥**

```
class RHINO [ P ]
feature

    ...
    minimum ( x , y : P ) : P  do
        if  x  ≥  y  then
            Result := y
        else
            Result := x
        end
    ...
end
```

Constrained Genericity – 2

- The class COMPARABLE declares the feature **≥**

  » **The generic parameter P must be a descendent of COMPARABLE**

```
class RHINO [ P -> COMPARABLE ]
 feature
    ...
    minimum ( x , y : P ) : P  do
       if  x  ≥  y  then
          Result := y
       else
          Result := x
       end
    ...
 end
```

Constrained Genericity – 2

- Use the following syntax for constrained genericity

  » **CLASS_NAME [TYPE –> CONSTRAINING_TYPE]**

    > **DICTIONARY [G , H –> HASHABLE]**

- The –> indicates inheritance

  » **H must be a type that inherits from HASHABLE**

- Inheritance guarantees the type passed has all the features one needs in the context of its use

- Unconstrained genericity is really written as follows

    > **STACK [G –> ANY]**

Discussion on Genericity

- What programming languages offer genericity that you know of? Java? C++? Other?

- What is the effect of genericity on

  » **compile time**

  » **size of the generated code**

  » **execution time**

  » **execution space**

    > **Generics are cheap in Eiffel**

© Gunnar Gotshalks

Does run-time vs compile time matter?

- When flying a plane, **run-time** is too late to find out that you don't have landing gear!

- Always better to catch errors at compile time

- This is the main purpose of Strong Typing

- Genericity helps to enforce Strong Typing

  » **No run-time typing errors**

    LIST [INTEGER]

    LIST [BOOK]

    LIST [STRING]