

Parametric Polymorphism (Generics)

Tomáš Matoušek

tm.matfyz.cz/teaching/clr.htm

Terminology

- generic/polymorphic type declaration
 - C is generic type
 - “template” for type *instantiations* created at run-time
- type parameters (S, T)
 - can be custom-attributed
- nested generic type
 - nested types has implicitly type parameters of all classes in which they are recursively nested

```
class C<T>
{
    C<int> x;
    C<T>[] y;
}
```

```
class D<S> { Pair<S, T> u; }
```

- generic/polymorphic method declaration

```
bool m<A>(A a, object o)
{
    if (!(a is C<C<int>>>))
        m<C<A>>(new C<A>(), o);
    return o is C<C<T>>.D<A>;
}
```

- constraints (on T)

– T : I, T : struct, T : class, T : new()

- constructed types

– open

- C<T>, C<C<S>>, C<T[]>
- some type parameters remain
- closed
 - C<int>, X<string, int>, X<int, C<bool>>
 - no type parameters

```
class E<S, T>
where S : class, I, new()
where T : struct
{
}
```

Notes

- C# allows to “overload” declarations by the number of type parameters
 - type with n generic parameters named 'T`n' in metadata
- type parameters
 - formal placeholders
 - identifiers
- type arguments
 - make-up constructed types
 - can be any type
 - including type parameters or constructed types

Restrictions (1)

- it's illegal to extend from type parameter
 - `class C<T> : T { }`
- it's illegal to extend from the type itself
 - `class C<T> : C<int> { }`
 - `class C<T> : C<C<T>> { }`
- it's illegal to access static fields or nested types thru type parameter
 - `T.field`, `T.method()`, `C.T.field`
- implemented interfaces must be unique for all possible constructed types
 - `interface I<T> { }`
 - `class C<S,T> : I<S>, I<T> { }`

Restrictions (2)

- overloads must be unique in signature among all possible closed constructed types disregarding any type parameter constrains
 - valid

```
class C<T>
{
    void f(T a, T b);
    void f(int a, bool b);
}
```

- invalid

```
class C<U, V>
{
    void f(U a, V b);
    void f(V a, U b);
}
```

Restrictions (3)

- no covariant conversions
 - contrary to arrays
 - valid
 - `object [] o = new string[10];`
 - invalid
 - `List<object> o = new List<string>(10);`

Static Fields and Constructors

- stored per instantiation
 - static fields of C<int> have different values from C<string>
- static constructor
 - called when a new instantiation is created
 - i.e. first time that either
 - an instance of the closed constructed type is created
 - a static member of the closed constructed type is referenced
 - useful for imposing restrictions on the type parameters

```
class Gen<T> where T: struct
{
    static Gen()
    {
        if (!typeof(T).IsEnum)
            throw new ArgumentException("T must be an enum");
    }
}
```

Implementation

- goals
 - exact runtime types
 - is and as operators
 - safe type casts
 - code sharing
 - share code among instantiations where possible
 - dynamic loading
 - create instantiations only when necessary (fast type load)
 - enable polymorphic recursion
- non-uniform instantiations
 - contrary to treating all types as objects
 - value types boxing

```
class List<T> { ... List<List<T>> ... }  
void m<T>() { ... m<List<T>>>(); ... }
```

Sharing

- JITted code shared among compatible instantiations
 - instantiations whose actual parameters are compatible
- compatibility
 - all reference types are compatible with each other
 - value types are compatible iff their layout is the same
- e.g.
 - `Dictionary<string, int>` and
 - `Dictionary<object, int>` shares the compiled code
- issue
 - How to get an exact type handle of open constructed type into the code shared for several instantiations?

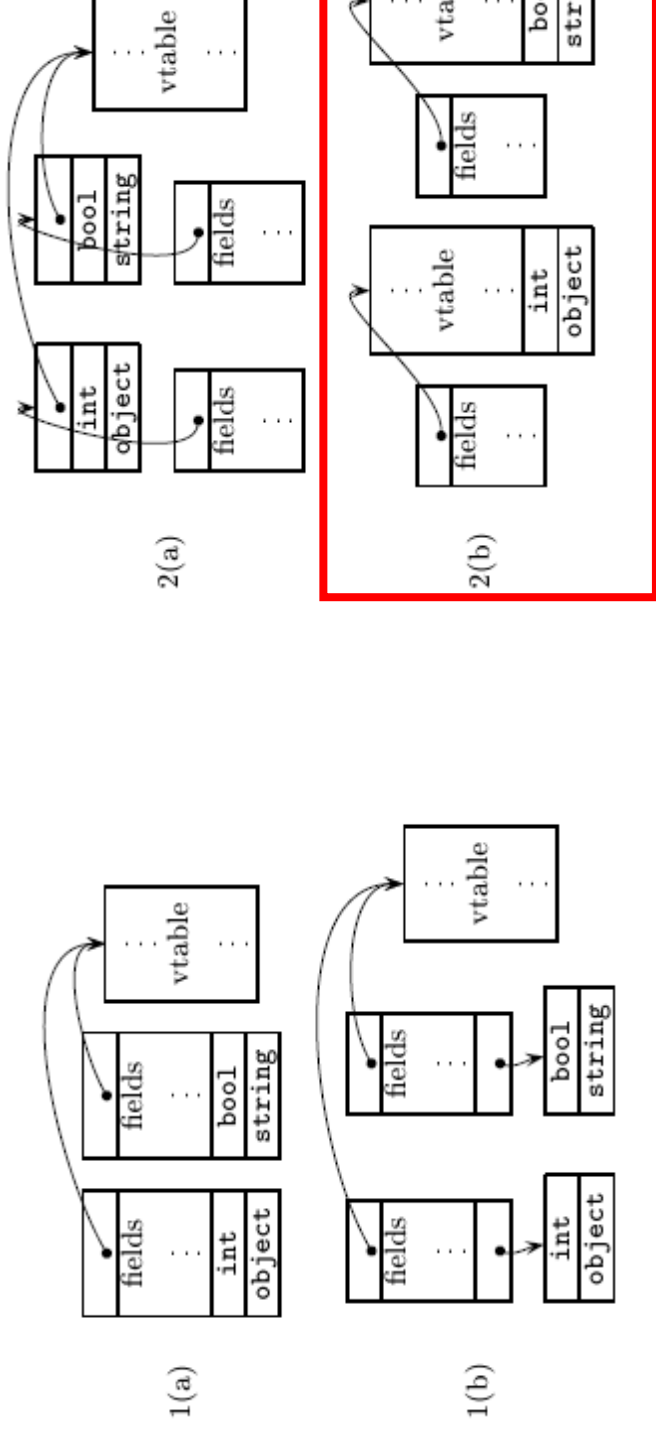
Generic Type Instantiations

- type instantiation
 - represented by **MethodTable**
 - virtual method slots
 - actual type parameters
 - tables are replicated
 - one per each instantiation
 - shared
 - **EEClasses** are shared among compatible instantiations
 - content of the v-table is also shared

Storing Runtime Type Information

Dictionary<int, string>

Dictionary<bool, string>



| Technique | Space (words) | | Time (indirections) | |
|-----------|---------------|----------|---------------------|------|
| | per-object | per-inst | vtable | inst |
| 1(a) | n | 0 | 1 | 0 |
| 1(b) | 1 | n | 1 | 1 |
| 2(a) | 0 | $n+1$ | 2 | 2 |
| 2(b) | 0 | $n+s$ | 1 | 2 |

Dictionaries

- global instantiations dictionary
 - maps TypeSpec tokens to type handles (MethodTables)
 - slot for each instantiation
 - populated incrementally by the JIT
 - access is slow
 - needs to compute hash from TypeSpec and look-up the table
- local instantiations dictionary
 - stored in MethodTable bound to an instantiation
 - contains one slot per each open constructed type used
 - by fields (filled by the type loader)
 - by the JITted methods (filled by the JITter)
 - JITted code contains an offset in this dictionary when the exact open constructed type handle is needed
- similarly for method instantiations
 - “global table” is per generic type
 - local tables passed as arguments to the calls

Lazy Population of Dictionaries

- type loader looks-up the dictionary
 - for all constructed types used by fields
 - look-ups for methods postponed to compilation time
 - instantiation exists
 - the field size is calculated
 - instantiation doesn't exist
 - new slot added
- JITter looks-up the dictionary
 - for all constructed types the method refers to
 - look-up
 - instantiation exists
 - the method table pointer is encoded to the generated code
 - instantiation doesn't exist
 - new slot added

Generic Method Instantiations

- local instantiations dictionary
 - type handles of constructed types occurring in the method and containing method type parameters
 - empty slots for instantiations of other generic methods invoked by the method
 - lazy creation, enables polymorphic recursion
 - stored along v-table
 - index passed to the method by a hidden argument
- issue
 - virtual polymorphic methods (first-class polymorphism)
 - How to pass the dictionary?
 - the caller doesn't know which method gets eventually invoked

Getting Exact Type Information at Runtime

- constructed types comprising of type parameters
 - in instance methods
 - we have “this”
 - we can access the local dictionary in MethodTable
 - in static methods
 - hidden argument needed (MethodTable pointer)
- constructed types including method type parameters
 - local dictionaries passed to the method by the caller

Rotor

- VM/generics
- VM/genericdict
- VM/genmethod
- VM/instmethash
- VM/methodtable

References

- Andrew Kennedy, Don Syme: *Design and Implementation of Generics for the .NET Common Language Runtime*
- Dachuan Yu, Andrew Kennedy, Don Syme: *Formalization of Generics for the .NET Common Language Runtime*
- Andrew Kennedy, Don Syme: *Combining Generics, Precompilation and Sharing Between SoftwareBased Processes*
- Andrew Kennedy, Don Syme: *Transposing F to C#: Expressivity of parametric polymorphism in an object-oriented language*