Virtual Class Support at the Virtual Machine Level

Anders Bach Nielsen Erik Ernst

Department of Computer Science
Aarhus University
Denmark

3rd Workshop
Virtual Machines and Intermediate Languages
Orlando, Florida – October 25th, 2009

Outline

Outline

Virtual Classes

Reminder: What is a Virtual Method?

- A method whose behavior can be overridden within an inheriting class by a method with the same signature.
- On method invocation, the virtual method is looked up in the object at run time.

Virtual Classes

Reminder: What is a Virtual Method?

- A method whose behavior can be overridden within an inheriting class by a method with the same signature.
- On method invocation, the virtual method is looked up in the object at run time.

Here: What is a Virtual Class?

Virtual Classes

Reminder: What is a Virtual Method?

- A method whose behavior can be overridden within an inheriting class by a method with the same signature.
- On method invocation, the virtual method is looked up in the object at run time.

Here: What is a Virtual Class?

- A class whose state and behavior can be extended within an inheriting class by a class with the same name.
- On class access, the virtual class is looked up in the object at run time.

The Origin and Evolution of Virtual Classes

BETA

- Designed in the 1970s; the first language to mention Virtual Classes
- Unified classes and methods into patterns
- Restricted use of virtual patterns
- Limited by compilation strategy, both type system and code generation

The Origin and Evolution of Virtual Classes

BETA

gbeta

- Generalized version of BETA
- Fully general support for virtual classes
  - Required total reconstruction of the language, both type system and code generation
- Translated to bytecode, executed on a specialized VM

The Origin and Evolution of Virtual Classes

BETA

gbeta

- Generalized version of BETA
- Fully general support for virtual classes
  - Required total reconstruction of the language, both type system and code generation
- Translated to bytecode, executed on a specialized VM

CaesarJ, Object Teams & Scala

- Translated to Java byte code; executed on the JVM
- Restricted support for virtual classes: A location where the class is fully known at compile time.
- Scala: no native support for virtual classes; they can be partially emulated using traits and abstract types.

The Programming Language of Discourse

gbeta

Ideas behind gbeta

Design Criteria

- Generalization of many BETA features, especially Virtual Patterns (“Superpattern”)

Mechanisms

- Linearization-based semantics & deep mixin composition
  ⇒ higher-order hierarchies
- Simple dependent types
  ⇒ family polymorphism

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Program Syntax

The Expression Problem

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:
6      };
7      LangPr                                    string)}};
8      LangPr
9        Lit:. { print:. { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Pattern declaration (class):
<ident>: <kind> <type>;

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }};
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Pattern declaration (method):

```
<ident>:  %( {<ident>:<type>} | {<ident>:<type>} ) <type>;
```

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ La
16       lit: ^F
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Reference:
<ident>: ^ <kind> <type>;

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit;
19       lit.eval | int
20     }
21   }
```

Assignment (left-to-right):
<expression> | <expression>

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: Lan
9        Lit:: { print::
10     };
11     LangVar1: ^#=LangP
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Declaration of kind object:

```
<ident>:  @ <type>;
```

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Program Intension

The Expression Problem

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Program Intension

The Expression Problem

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Program Intension

The Expression Problem

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Program Intension

The Expression Problem

A Brief Taste of gbeta Code – Running Example

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPr                                        string)}};
8      LangPr
9        Lit:                                          }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

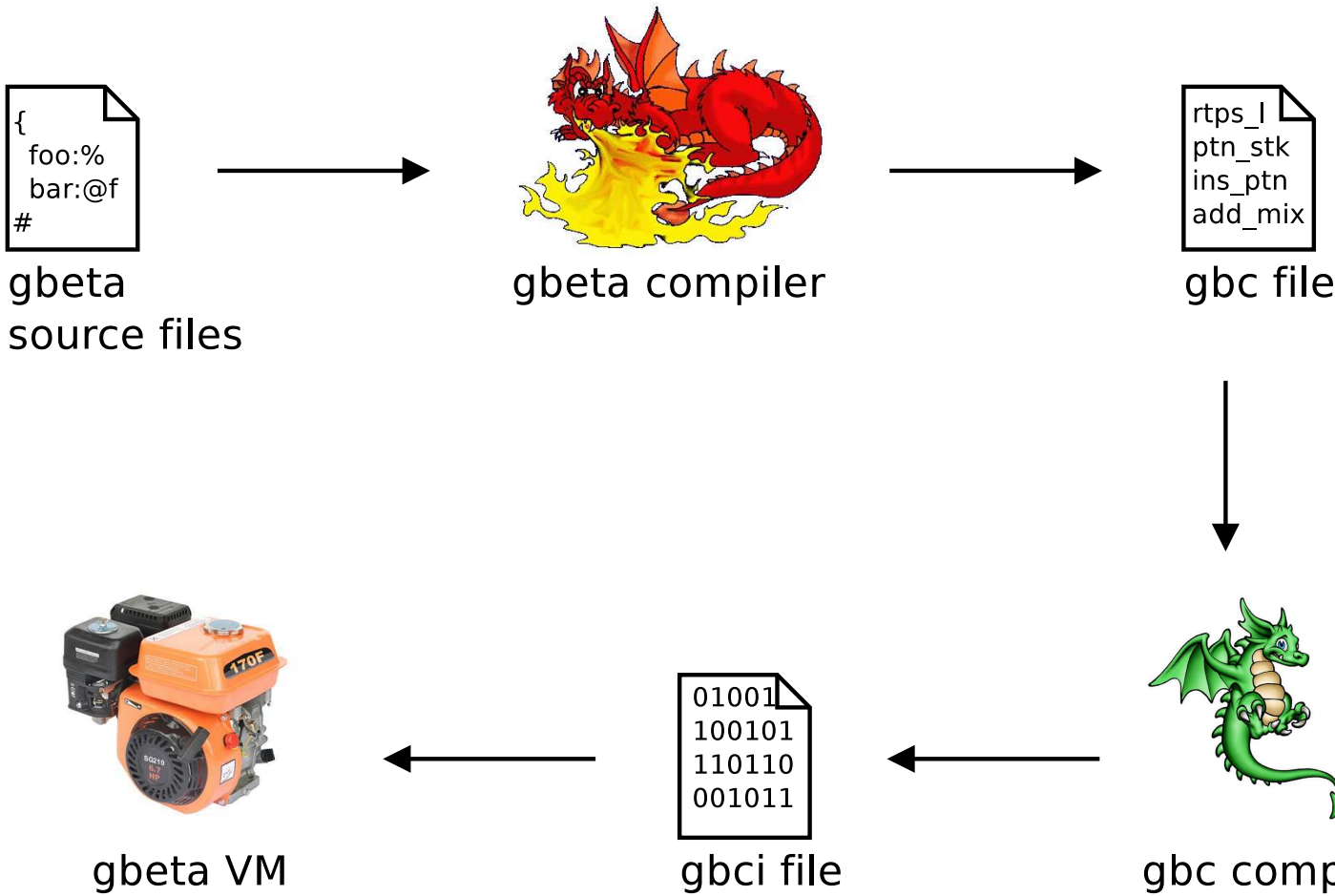
Motivation:
Dynamic merge and Object creation

Program Intension

The Expression Problem

Outline

Overview of the gbeta Run-time System



The gbeta Virtual Machine

The gvm

- Implemented in 6700 lines of C++
- Standard Cheney garbage collector
- Uses a direct threaded interpreter
  - Over 200 byte-code instructions
  - In progress: A JIT compiler for native-code execution
- Two memory spaces
  1. Static space; items from input file (not recyclable)
  2. Heap space: items created at run time (recyclable)

The Layout of the gbci File

The gbci input file is divided into three parts

Tables

- Mainpart names
- Symbols
- Strings
- Floats

Mainparts

- Smallest compile-time entity

Staticpatterns

- Generated when the compiler can calculate all mixins in a pattern

Compile-time Entities

```
2       Lang: %{1 Exp:< object; Lit:< Exp %{2 value: int } };
3       LangEval: Lang %{3 Exp:: %{4 eval: %(|i:int)}};
4       LangEvalImpl: LangEval {5
5         Lit:: {6 eval:: {7 value | i }}
6       };
```

Compile-time Entities

```
2       Lang: %{1 Exp:< object; Lit:< Exp %{2 value: int } };
3       LangEval: Lang %{3 Exp:: %{4 eval: %(|i:int)}};
4       LangEvalImpl: LangEval {5
5         Lit:: {6 eval:: {7 value | i }}
6       };
```

Mainpart

- Fields map, virtual fields map and more
- Initialization code for each field
- Action part
- Stack and temp space requirements

Compile-time Entities

```
2       Lang: %{1 Exp:< object; Lit:< Exp %{2 value: int } };
3       LangEval: Lang %{3 Exp:: %{4 eval: %(|i:int)}};
4       LangEvalImpl: LangEval {5
5         Lit:: {6 eval:: {7 value | i }}
6       };
```

Mainpart

- Fields map, virtual fields map and more
- Initialization code for each field
- Action part
- Stack and temp space requirements

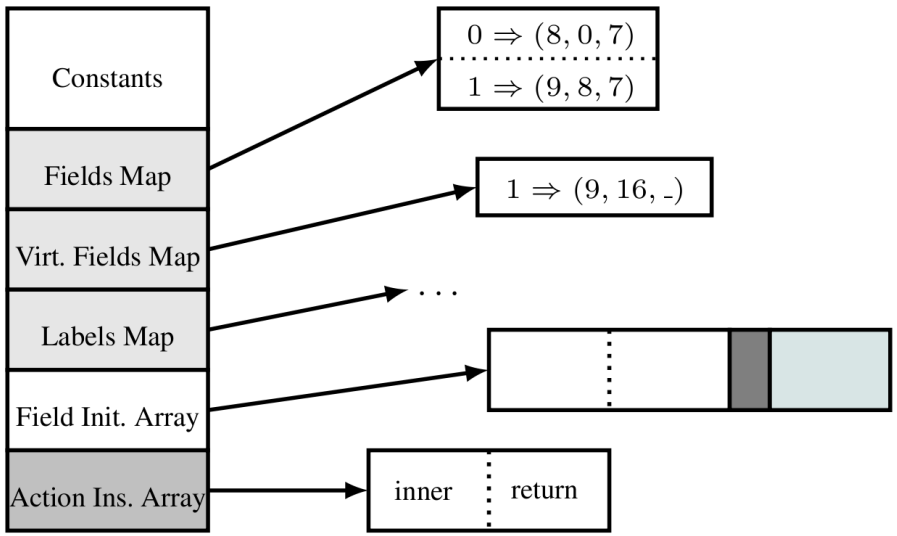
Staticpattern

- Static description of the mainparts that form a pattern
- Code for evaluating the contexts for each mainpart

Compile-time Entity – Mainpart

Mainpart

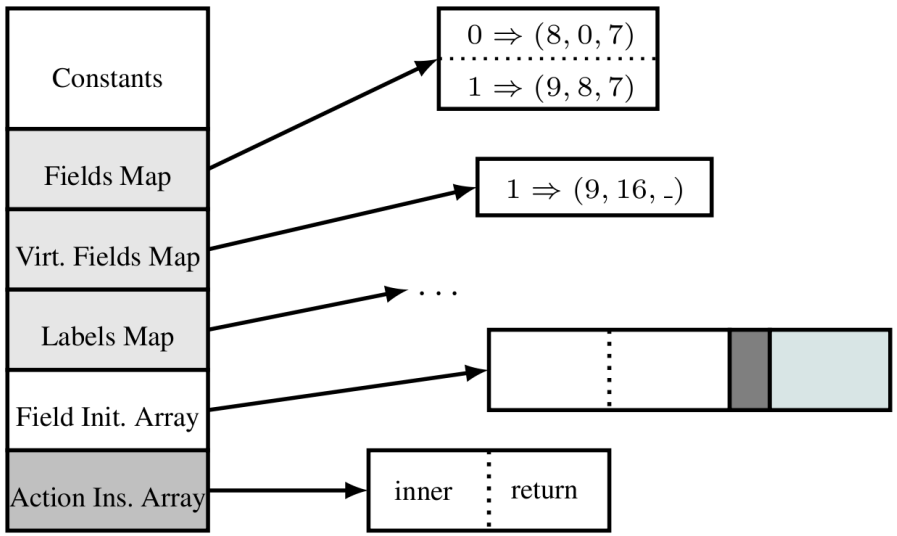
- Fields map, virtual fields map and more
- Initialization code for each field
- Action part
- Stack and temp space requirements



Compile-time Entity – Mainpart

Mainpart

- Fields map, virtual fields map and more
- Initialization code for each field
- Action part
- Stack and temp space requirements



- A mainpart only “works” together with a context

$$(\text{ Context, Mainpart }) \Rightarrow \text{Mixin}$$

Run-time Entities

The Mixin

- The smallest building block at run-time
- A list of mixins $\Rightarrow$ a pattern

Run-time Entities

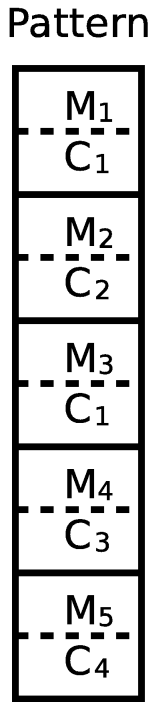
The Mixin

- The smallest building block at run-time
- A list of mixins $\Rightarrow$ a pattern
- But we can not address a mixin directly, only patterns

Run-time Entities

The Pattern

- Represents classes and methods
- A pattern is a array of mixins (fixed size elements)



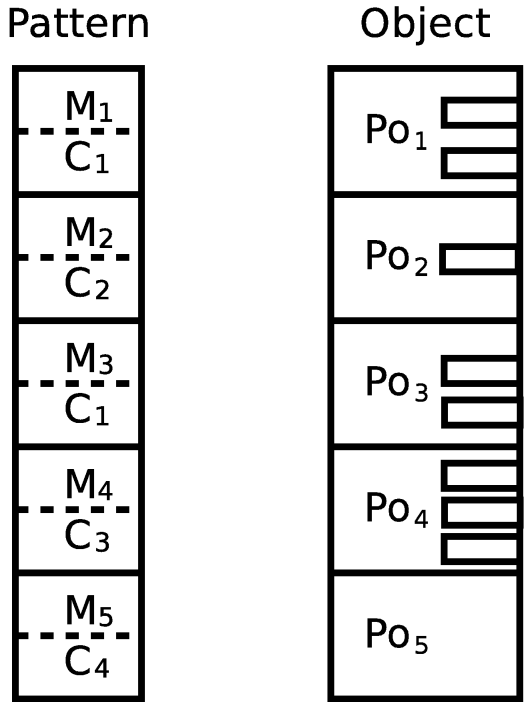
Run-time Entities

The Pattern

- Represents classes and methods
- A pattern is a array of mixins (fixed size elements)

The Object

- Created from a pattern
- Objects are list of part objects (part object size may vary)



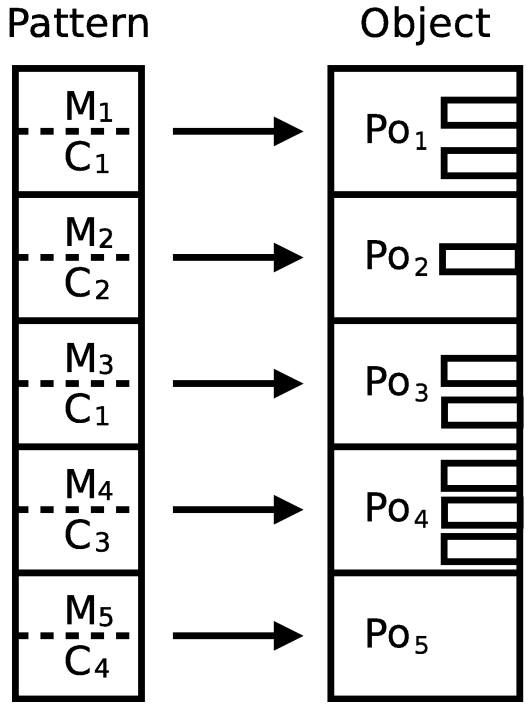
Run-time Entities

The Pattern

- Represents classes and methods
- A pattern is a array of mixins (fixed size elements)

The Object

- Created from a pattern
- Objects are list of part objects (part object size may vary)
- One-to-one correspondence between mixin and part object
- Object are contiguously allocated in memory



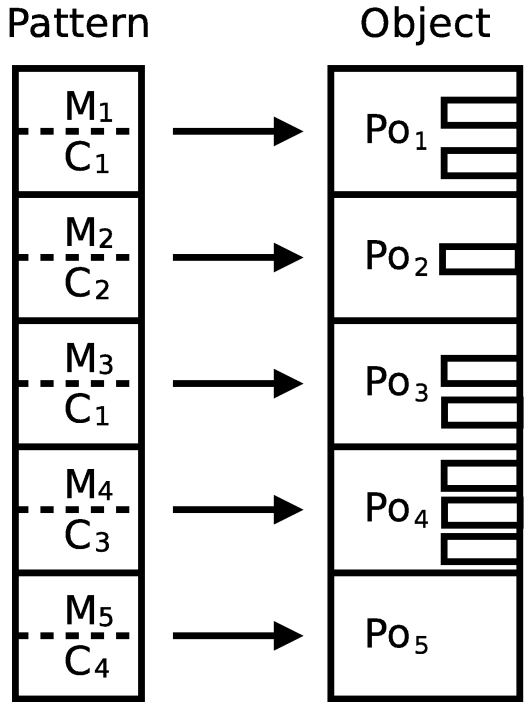
Run-time Entities

The Pattern

- Represents classes and methods
- A pattern is a array of mixins (fixed size elements)
- The context of a mixin is a part object

The Object

- Created from a pattern
- Objects are list of part objects (part object size may vary)
- One-to-one correspondence between mixin and part object
- Object are contiguously allocated in memory



What About the Virtual Patterns?

Are virtual patterns special?

What About the Virtual Patterns?

Are virtual patterns special? No!

- A virtual pattern is not a special pattern
  – It just has special run-time semantics
- There are two kinds of virtual pattern declarations: initial bindings and further bindings of virtual patterns

What About the Virtual Patterns?

Are virtual patterns special? No!

- A virtual pattern is not a special pattern
  – It just has special run-time semantics
- There are two kinds of virtual pattern declarations: initial bindings and further bindings of virtual patterns

Example

```
2     Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3     LangEval: Lang %{ Exp:: %{ eval: %(|i:int)} };
```

What About the Virtual Patterns?

Are virtual patterns special? No!

Example

```
2     Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3     LangEval: Lang %{ Exp:: %{ eval: %(|i:int)} };
```

Compilation

- Initial bindings produce an initialization block with a search instruction
- Further bindings produce an extension block and an initialization block.
  - The extension block will add the addition and searches for more extensions
  - The initialization block will install the complete pattern

Outline

Family Combination – The Example Program

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Family Combination – The Example Program

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Family Combination – The Example Program

```
1    {
2      Lang: %{ Exp:< object; Lit:< Exp %{ value: int } };
3      LangEval: Lang %{ Exp:: %{ eval: %(|i:int)}};
4      LangEvalImpl: LangEval {
5        Lit:: { eval:: { value | i }}
6      };
7      LangPrint: Lang %{ Exp:: %{ print:%(|s:string)}};
8      LangPrintImpl: LangPrint {
9        Lit:: { print:: { value | int2str | s }}
10     };
11     LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
12   #
13     LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
14     {
15       F: @ LangVar1 & LangVar2;
16       lit: ^F.Lit;
17     #
18       F.Lit^ | lit; 3 | lit.value;
19       lit.eval | int2str | stdio
20     }
21   }
```

Family Combination – The Magic Line

```
15          F: @ LangVar1 & LangVar2;
```

Family Combination – The Magic Line

```
15          F: @ LangVar1 & LangVar2;
```

1. Get the pattern from `LangVar1`
2. Get the pattern from `LangVar2`
3. Merge these two patterns to create a larger pattern
4. Create an object from the larger pattern
5. Initialize the object
6. Install the object into the field `F`

Family Combination – The Magic Line

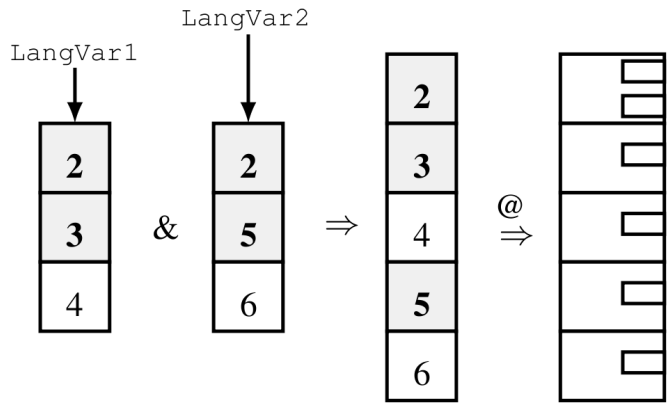
```
15          F: @ LangVar1 & LangVar2;
```

1. Get the pattern from `LangVar1`
2. Get the pattern from `LangVar2`
3. Merge these two patterns to create a larger pattern
4. Create an object from the larger pattern
5. Initialize the object
6. Install the object into the field `F`

Family Combination – The Magic Line

```
15          F: @ LangVar1 & LangVar2;
```

1. Get the pattern from `LangVar1`
2. Get the pattern from `LangVar2`
3. Merge these two patterns to create a larger pattern
4. Create an object from the larger pattern
5. Initialize the object
6. Install the object into the field `F`

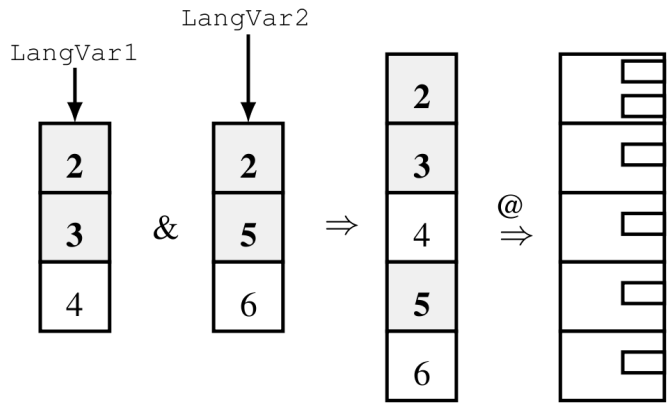


Family Combination – The Magic Line

```
15          F: @ LangVar1 & LangVar2;
```

1. Get the pattern from `LangVar1`
2. Get the pattern from `LangVar2`
3. Merge these two patterns to create a larger pattern

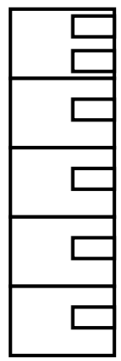
   Nielsen, A.B. & Ernst, E. Optimizing dynamic class composition in a statically typed language. Tools Europe 2008
4. Create an object from the larger pattern
5. Initialize the object
6. Install the object into the field `F`



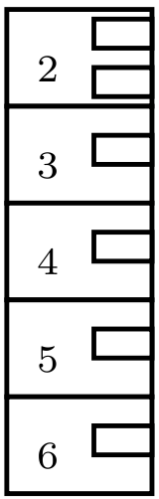
Family Combination – The Magic Line

```
15          F: @ LangVar1 & LangVar2;
```

1. Get the pattern from `LangVar1`
2. Get the pattern from `LangVar2`
3. Merge these two patterns to create a larger pattern
4. Create an object from the larger pattern
5. Initialize the object
6. Install the object into the field `F`

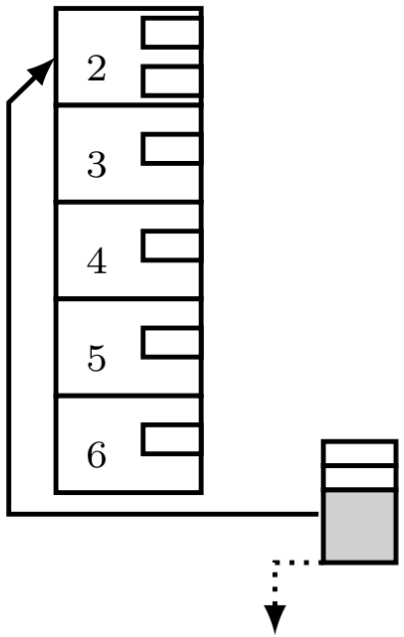


Family Combination – Initialize Object




```
{1
  Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
  LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
  LangEvalImpl: LangEval {4
    Lit:: {11 eval:: { value | i }}
  };
  LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
  LangPrintImpl: LangPrint {6
    Lit:: {12 print:: { value | int2str | s }}
  };
  LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
#
  LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
  {7
    F: @ LangVar1 & LangVar2;
    lit: ^F.Lit;
  #
    F.Lit^ | lit; 3 | lit.value;
    lit.eval | int2str | stdio
  }
}
```

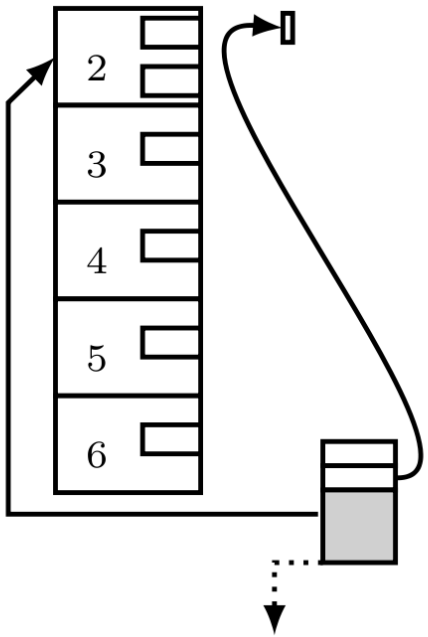
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Evaluation frame created to initialize object
- Initialization starts at the most general part object

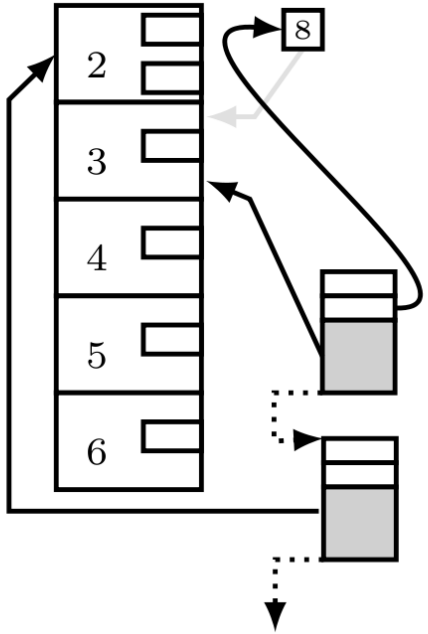
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Pushing onto stack the initial pattern of `Exp`
- Searching for extensions of `Exp`

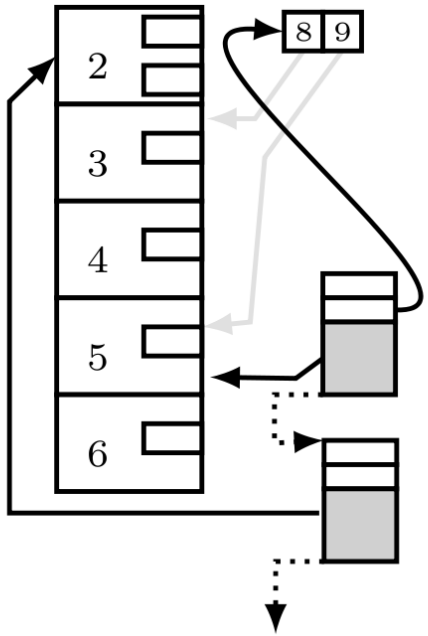
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Extension found in part object 3
- New evaluation frame created
- Merge initial pattern and the extension; push result onto stack
- Searching for more extensions of `Exp`

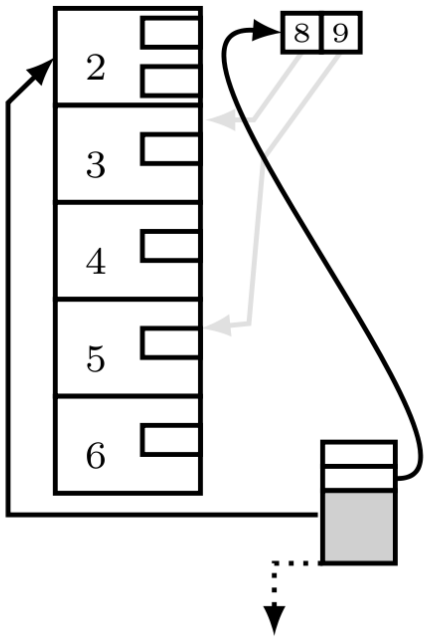
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Extension found in part object 5
- Merge pattern on stack with extension; push result onto stack
- Searching for more extensions of `Exp`

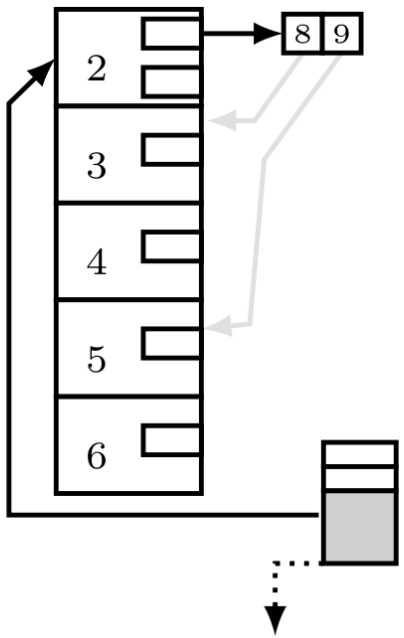
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- No extensions of `Exp` found
- Pop pattern from stack;
  push the pattern onto the stack of old evaluation frame
- Remove the evaluation frame

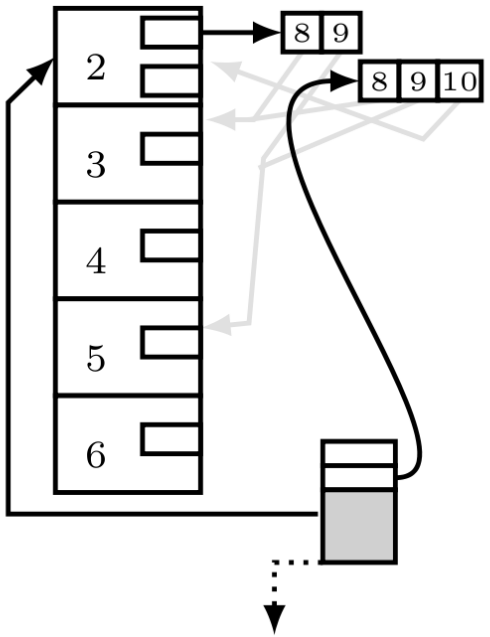
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Install pattern from stack into first field

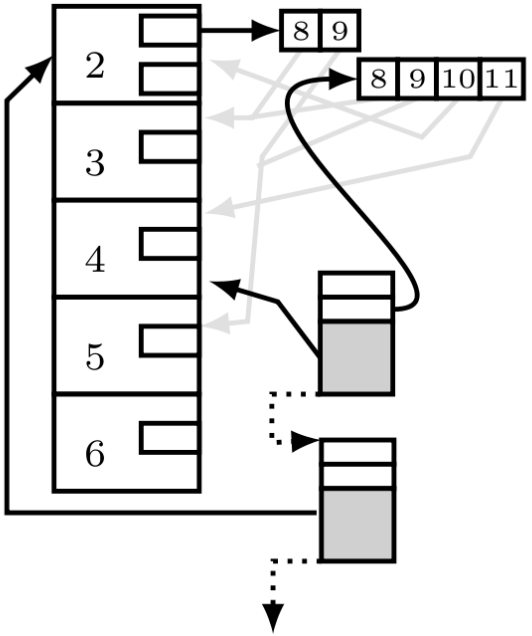
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Initial pattern of `Lit` created; pushed onto stack
- Searching for extensions of `Lit`

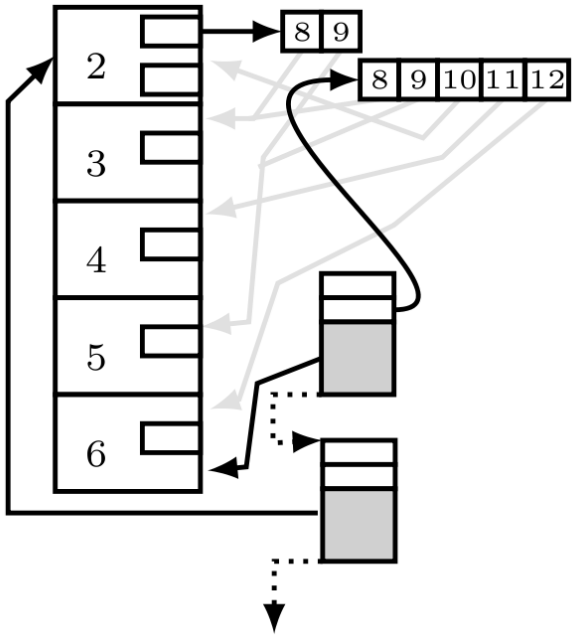
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Extension found in part object 4
- New evaluation frame created
- Merge initial pattern and extension; push result onto stack
- Searching for more extensions of `Lit`

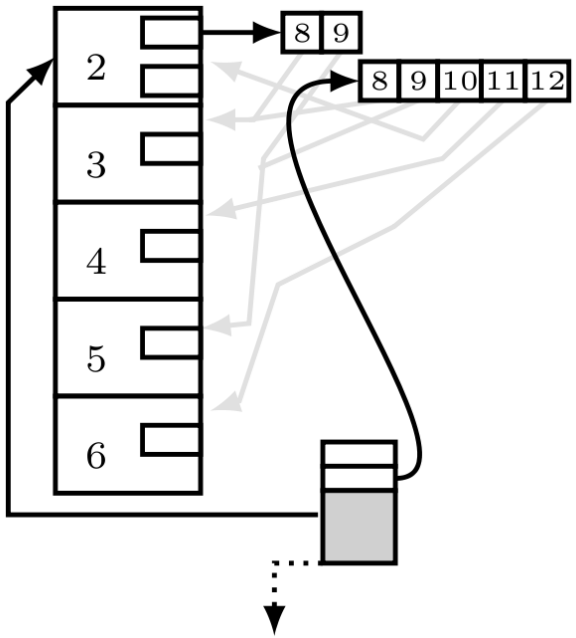
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Extension found in part object 6
- Merge pattern on stack with extension; push result onto stack
- Searching for more extensions of `Lit`

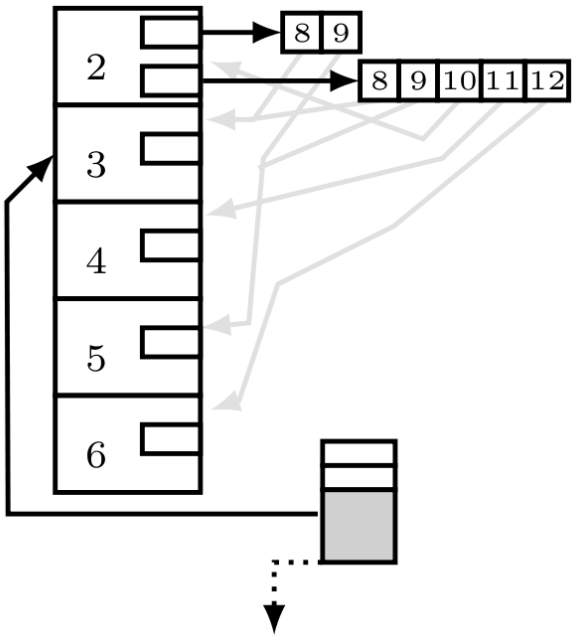
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- No extensions of `Lit` found
- Pop pattern from stack;
  push the pattern onto stack of the old evaluation frame
- Remove the evaluation frame

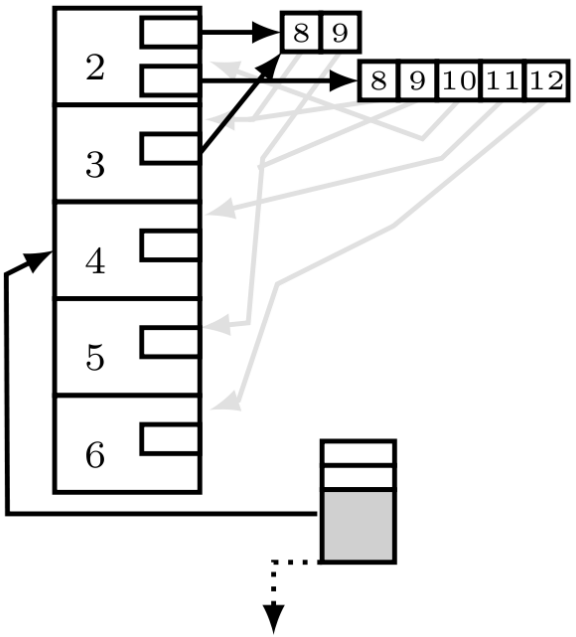
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Install pattern from stack into second field
- Concludes initialization of part object 2
- Proceeds with initialization of part object 3

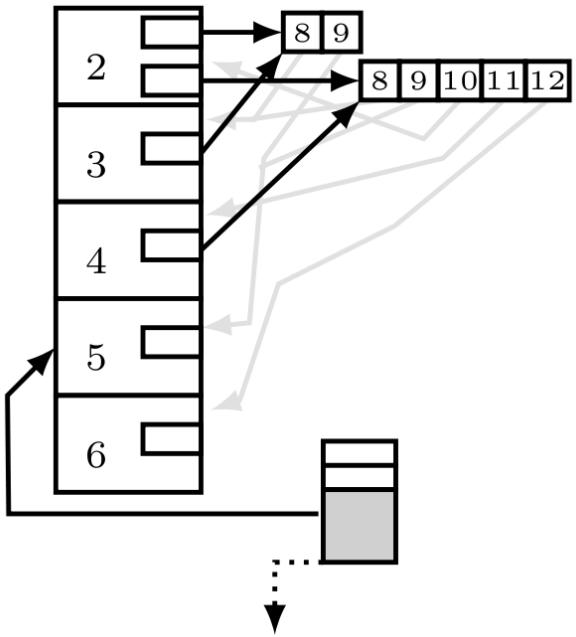
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Field was a further binding of `Exp`
- Install `Exp` pattern into field
- Proceeds with initialization of part object 4

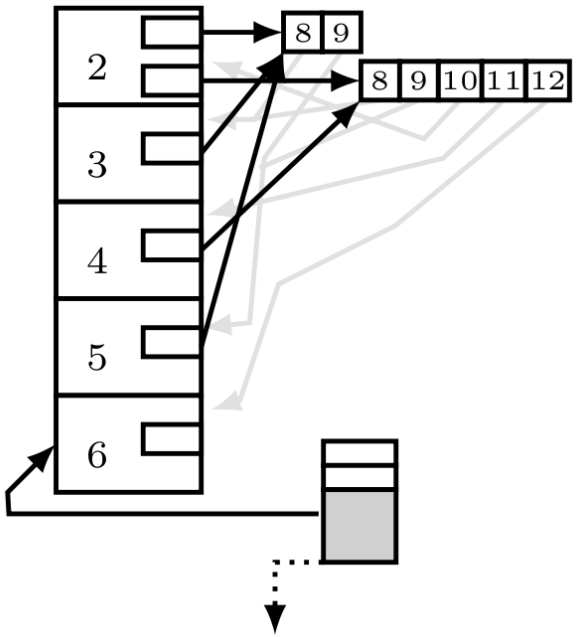
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Field was a further binding of `Lit`
- Install `Lit` pattern into field
- Proceeds with initialization of part object 5

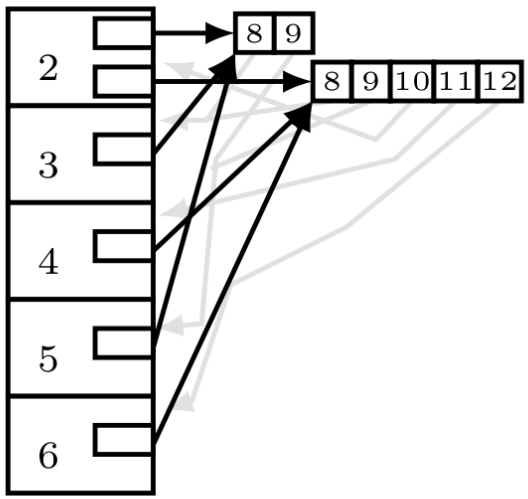
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Field was a further binding of `Exp`
- Install `Exp` pattern into field
- Proceeds with initialization of part object 6

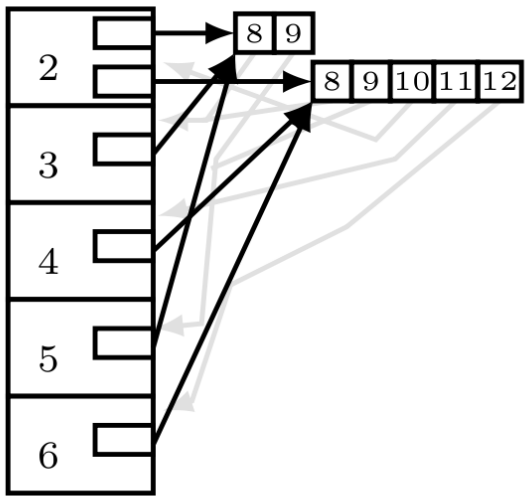
Family Combination – Initialize Object




```
Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
LangEvalImpl: LangEval {4
  Lit:: {11 eval:: { value | i }}
};
LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
LangPrintImpl: LangPrint {6
  Lit:: {12 print:: { value | int2str | s }}
};
```

- Field was a further binding of `Lit`
- Install `Lit` pattern into field
- Object is completely initialized

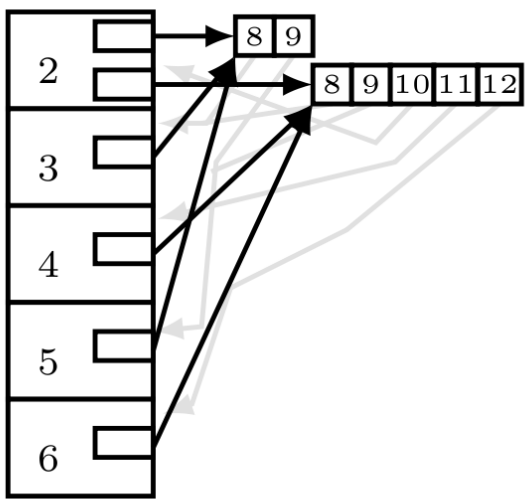
Family Combination – Initialize Object




```
{1
  Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
  LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
  LangEvalImpl: LangEval {4
    Lit:: {11 eval:: { value | i }}
  };
  LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
  LangPrintImpl: LangPrint {6
    Lit:: {12 print:: { value | int2str | s }}
  };
  LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
#
  LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
  {7
    F: @ LangVar1 & LangVar2;
    lit: ^F.Lit;
  #
    F.Lit^ | lit; 3 | lit.value;
    lit.eval | int2str | stdio
  }
}
```

Family Combination – Initialize Object

- Download the gbeta compiler and virtual machine at
  `http://www.cs.au.dk/~abachn/vmil-2009.tar.gz`




```
{1
  Lang: %{2 Exp:< object; Lit:< Exp %{10 value: int } };
  LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
  LangEvalImpl: LangEval {4
    Lit:: {11 eval:: { value | i }}
  };
  LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
  LangPrintImpl: LangPrint {6
    Lit:: {12 print:: { value | int2str | s }}
  };
  LangVar1: ^#=LangPrint; LangVar2: ^#=LangEval;
#
  LangPrintImpl# | LangVar1#; LangEvalImpl# | LangVar2#;
  {7
    F: @ LangVar1 & LangVar2;
    lit: ^F.Lit;
  #
    F.Lit^ | lit; 3 | lit.value;
    lit.eval | int2str | stdio
  }
}
```

Related Work

Standard Java VM

- CaesarJ
- Object Teams
- Scala - Partial emulation
- J& - Nested inheritance

Related Work

Standard Java VM

- CaesarJ
- Object Teams
- Scala - Partial emulation
- J& - Nested inheritance

Modified Squeak VM

- Newspeak
  - Dynamically typed language
  - Classes are features of Objects found by lookup
  - No deep mixin composition; no mechanism to ensure the overriding definition is a subclass
  - Dynamic class creation (image based languages)

Summary

Assumptions:

- Specialized virtual machine
- Mixins are atomic compile-time entities

Summary

Assumptions:

- Specialized virtual machine
- Mixins are atomic compile-time entities

Contributions:

- Run-time model
  - **Classes** build from **Mixins**
  - **Object** build from **Part Objects**
  - **Dynamic class composition** and **virtual classes**
- **Object initialization** with classes as members of objects
- The gbeta language and **the gbeta virtual machine**

Summary

Assumptions:

- Specialized virtual machine
- Mixins are atomic compile-time entities

Contributions:

- Run-time model
  - **Classes** build from **Mixins**
  - **Object** build from **Part Objects**
  - **Dynamic class composition** and **virtual classes**
- **Object initialization** with classes as members of objects
- The gbeta language and **the gbeta virtual machine**

Thank you for listening! Questions?