BUILDING EXTENDIBLE TRANSLATORS

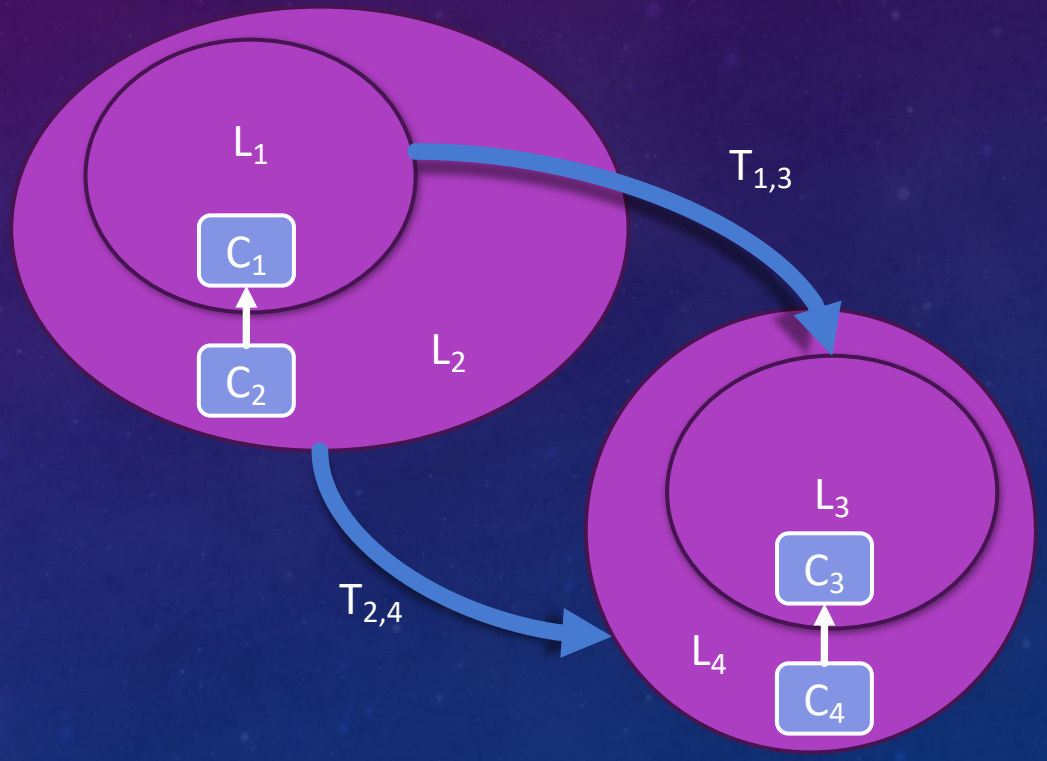
Gert Veldhuijzen van Zanten

MODULAR LANGUAGE DESIGN

- Languages can have layers
  - Physical units
  - Time
  - Optional features
  - Aspects
- Dependencies
  - Extension knows about base
  - Not the other way around
  - Base has something that is extendible (abstract, generic)

LANGUAGES AND TRANSLATORS

- L_2 extends L_1 if L_2 contains a concept C_2 that is a subconcept of some concept C_1 from L_1
- A translator $T_{1,3}$ from L_1 to L_3 needs to be extended for L_2:
  - $T_{2,4}$ should contain all mappings from L_1 to L_3 plus some mappings from $L_2 \backslash L_1$ to L_4 (some extension of L_3)



MODELS IN MPS

- Models consist of nodes
- Nodes have
  - Properties (basic values),
  - Children (also nodes) and
  - References (to other nodes)
- Projectional editing
- No name binding

- Languages
  - Structure (concepts)
  - Editors
  - Constraints
  - Types
  - Interpreter
  - Transformations

TRANSLATOR LANGUAGE

- Translators
  - Polymorphic dispatch
  - Memoization
  - Nesting
  - A language aspect

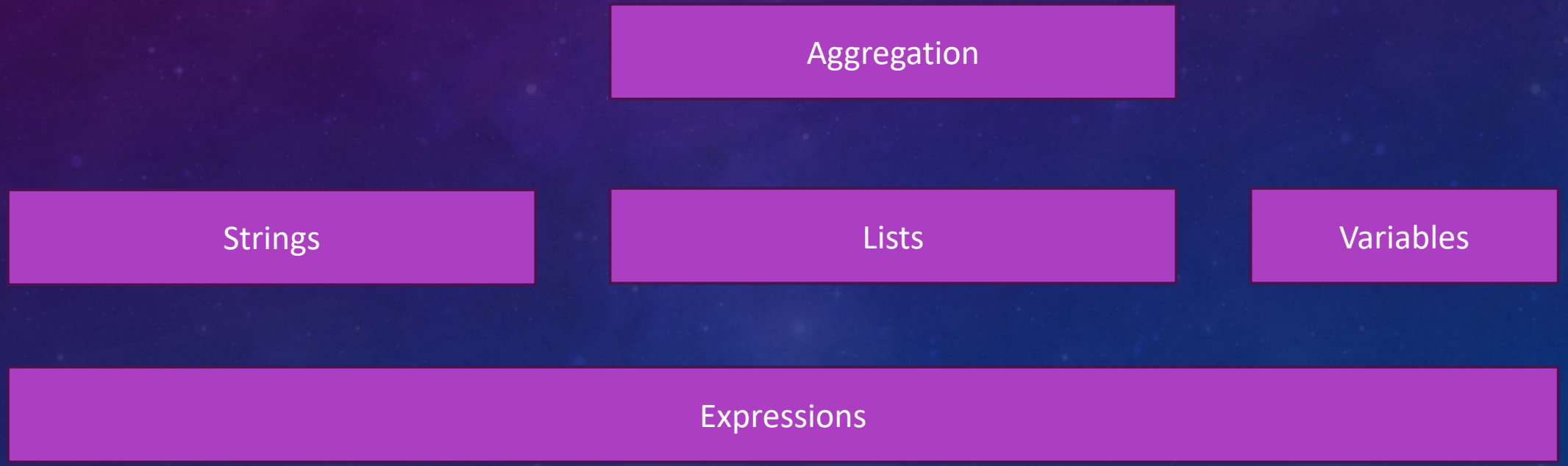
A LANGUAGE TO DESCRIBE TRANSLATORS: WHY?

- Model-to-model-transformation are everywhere
  - Lots of case analysis (.isInstanceOf)
  - Hashmaps for transforming references

MEMOIZATION

- Mapping calls with same parameters are executed once
- All such calls return exactly the same (==) result
- Crucial for generating references
- No need for additional hashmaps or ‘mapping labels’

LANGUAGE LAYERS



TRANSLATOR

- The translator language is an extension of baselanguage
- A translator is special kind of class
- A mapping is a kind of (nameless) method

- Most-specific mapping that matches runtime types is applied

```
translator countVars() {

  ⟦node<Expression> expr⟧ ⇨ int {
    0;
  }
  ⟦node<VariableReference> varRef⟧ ⇨ int {
    1;
  }
  ⟦node<BinaryOperation> binop⟧ ⇨ int {
    ⟦binop.leftExpression⟧ + ⟦binop.rightExpression⟧;
  }
}
```

MULTI-DISPATCH

- Dispatch to mapping with most-specific leftmost parameter
- If more than one: most specific next parameter
- etc

```
translator calculator() {

  ⟦node<Expression> expr⟧ ⇨ Object;
  ⟦node<IntegerConstant> lit⟧ ⇨ int {
    lit.value;
  }
  ⟦node<StringLiteral> s⟧ ⇨ string {
    s.value;
  }
  ⟦node<BinaryOperation> binop⟧ ⇨ Object {
    ⟦binop, ⟦binop.leftExpression⟧, ⟦binop.rightExpression⟧⟧;
  }

  ⟦node<BinaryOperation> binop, Object left, Object right⟧ ⇨ Object;
  ⟦node<PlusExpression> binop, int left, int right⟧ ⇨ Object {
    left + right;
  }
  ⟦node<PlusExpression> binop, string left, Object right⟧ ⇨ Object {
    left + right.toString();
  }
  ⟦node<MulExpression> binop, int left, int right⟧ ⇨ Object {
    left * right;
  }
  ⟦node<MulExpression> binop, string left, int right⟧ ⇨ Object {
    StringBuilder b = new StringBuilder();
    for (int i = 0; i < right; i++) {
      b.append(left);
    }
    return b.toString();
  }
```

AUTOMATIC LIFTING TO SEQUENCE

Even without a mapping for

```
⟦sequence<node<Expression>> seq⟧ ⇨ sequence<Integer>
```

```
⟦node<Summation> sum⟧ ⇨ Integer {
  ⟦sum.term⟧.foldLeft(0,{Integer a,~b => a + (Integer) b; });
}
```

is equivalent to

```
⟦node<Summation> sum⟧ ⇨ Integer {
  sum.term.select({~it => ⟦it⟧; }).foldLeft(0,{Integer a,~b => a + (Integer) b; });
}
```

GUARDS

- Mappings can have conditions that determine when they are applicable
- If guard fails: another equally or less specific mapping is tried
- Mapping with guard is considered more specific than mapping without guard with the same parameter types

```
⟦node<PlusExpression> binop, int left, int right⟧ ⇨ Object {
  left + right;
}
⟦node<PlusExpression> binop, string left, Object right⟧ ⇨ Object {
  left + right.toString();
}
⟦node<PlusExpression> binop, string left, Object right⟧ (right == null) ⇨ Object {
  left + "...";
}
⟦node<PlusExpression> binop, Object left, string right⟧ (left == null) ⇨ Object {
  "..." + right;
}
```

EXAMPLE: DESUGARING

```
translator translator() {

  copy {

    ⟦node<BaseConcept> source⟧ ⇨ node<BaseConcept> {
      if (source == null) { return null; }
      node<> nieuw = newEmptyInstance(source.concept);
      foreach p in source/.getProperties() {
        nieuw/.setProperty(p, source/.getProperty(p));
      }
      foreach child in source.children {
        nieuw/.addChild(child.containingLink, ⟦child⟧);
      }
      foreach r in source.references {
        nieuw/.setReferenceTarget(r.link, r.getTargetNode());
      }
      return nieuw;
    }
```

```
⟦node<DebugVariable> dVar⟧ ⇨ node<LocalVariableDeclaration> {
  return <LocalVariableDeclaration(
            name: dVar.name,
            type: # dVar.type,
            initializer: # ⟦dVar.initializer⟧:Expression
          )>;
}
⟦node<VariableReference> vRef⟧ (readsDebugVar(vRef)) ⇨ node<Expression> {
  final node<VariableDeclaration> vDecl = ⟦vRef.variableDeclaration⟧:VariableDeclaration;
  <{ =>
    System.out.println("$(  vRef.variableDeclaration.name )$" + " = " + ^( VariableReference vDecl )^);
    return ^( VariableReference vDecl )^;
  }.invoke()>;
}

⟦node<ExpressionStatement> stmt⟧ (stmt.expression.isInstanceOf(AssignmentExpression)) ⇨ node<BlockStatement
  node<AssignmentExpression> asgn = stmt.expression:AssignmentExpression;
  final node<VariableDeclaration> vDecl = ⟦asVar(asgn.lValue)⟧:VariableDeclaration;
  return <{
    ^( VariableReference vDecl )^ = %( ⟦asgn.rValue⟧:Expression)%;
    System.out.println("$(  vDecl.name )$" + " := " + ^( VariableReference vDecl )^);
  }>;
}
```

NESTED TRANSLATORS

```
⟦node<BinaryOperation> binop, Object left, Object right⟧ ⇨ Object;
⟦node<PlusExpression> plus, Object left, Object right⟧ ⇨ Object {
  plus⟦left, right⟧;
}
⟦node<MulExpression> plus, Object left, Object right⟧ ⇨ Object {
  multiply⟦left, right⟧;
}
```

```
plus {
  ⟦Object left, Object right⟧ ⇨ Object;
  ⟦int left, int right⟧ ⇨ Object {
    left + right;
  }
  ⟦string left, Object right⟧ ⇨ Object {
    left + right.toString();
  }
  ⟦string left, Object right⟧ (right == null) ⇨ Object {
    left + "...";
  }
  ⟦Object left, string right⟧ (left == null) ⇨ Object {
    "..." + right;
  }
} // end of plus
```

```
multiply {
  ⟦Object left, Object right⟧ ⇨ Object;
  ⟦int left, int right⟧ ⇨ Object {
    left * right;
  }
  ⟦string left, int right⟧ ⇨ Object {
    StringBuilder b = new StringBuilder();
    for (int i = 0; i < right; i++) {
      b.append(left);
    }
    return b.toString();
  }
} // end of multiply
```

COOPERATION BETWEEN TRANSLATORS

- Multiple Inheritance
- Extending nested translators

TRANSLATOR CONSTRUCTION

```
public static Object calculate(node<Expression> expr) {
  Calculator c = new Calculator(new hashmap<MemoKey, Object>);
  return c.⟦expr⟧;
}
```

- Adding an extended translator

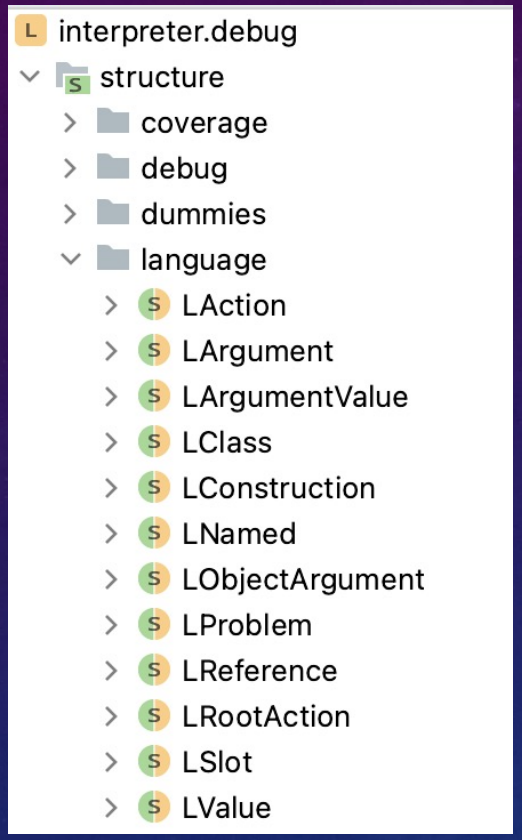
```
public static Object calculate(node<Expression> expr) {
  Calculator c = expr.model.new Calculator(new hashmap<MemoKey, Object>);
  return c.⟦expr⟧;
}
```

INTERPRETERS

- We can extend the base class for translators to intercept mapping calls

- Execution tracers
  - Coverage
  - Debugger

GENERIC INTERPRETERS INTERFACE



- Mapping calls with arguments that implement these interfaces get traced