New Typesystem Aspect

Path to Expressive and Natural Typesystems

Grigorii Kirgizov, MPS team

Outline of the talk

1. Introduction: existing aspect
2. CodeRules & its formal roots
3. Features of CodeRules
4. Status & roadmap

1. Introduction

Existing Typesystem Aspect

Features:

- Inference rules
- Weak & strong subtyping
- Engine for solving them
- Type variables & when_concrete blocks

Statement in *j.m.lang.typesystem language*:

```
infer typeof(assignment.init) :<=: typeof(assignment.decl);
```

Existing Typesystem Aspect

Problems:

- Opaque engine
- Restriction to "weak" & "strong" notions of subtyping
- Numerous *“when concrete is never concrete”* warnings

2. CodeRules & its formal roots

What is CodeRules?

It’s Language & Engine

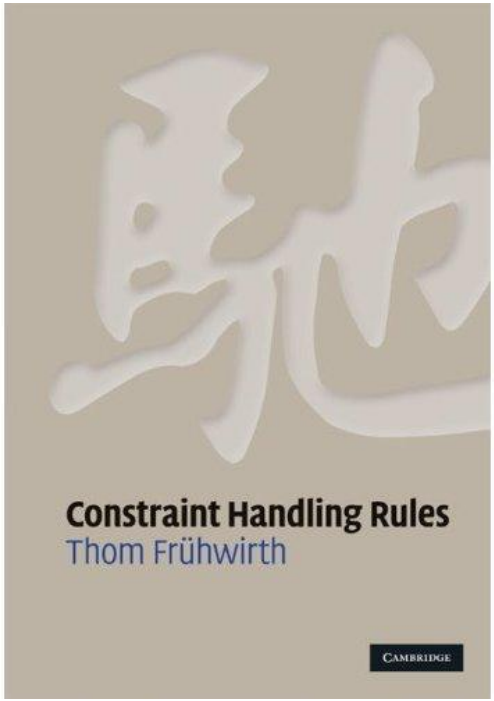
- Logic programming & Constraints processing
- Flexible: applicable for other model analyses
- Based on formalism *Constraints Handling Rules*

Constraint Handling Rules

CHR — formal basis for CodeRules

- Declarative rule-based language
- Well-defined semantics

Academical development with a strong theory behind it



Constraint Handling Rules

Basic notions:

- Constraints — some “facts” or relations
- Rules — specify processing of constraints

Rules consist of:

- Head — kind of “input”
- Guard (optional) — rule applicability condition
- Body — some “effect”

Program with 4 rules to infer siblings:

```
mother(X,Y) ==> parent(X,Y).
father(X,Y) ==> parent(X,Y).
parent(X,Z), parent(Y,Z) ==>
  sibling(X,Y).
sibling(X,Y) \ sibling(Y,X) <=> true.
```

Program to find minimum:

```
min(N) \ min(M) <=> N=<M | true.
```

Constraint Handling Rules Example

Given the information

```
mother(hans,mira), mother(sepp,mira),
father(sepp,john)
```

first two rules add parent relationships

```
parent(hans,mira), parent(sepp,mira),
parent(sepp,john)
```

third rule adds the sibling constraints:

```
sibling(hans,sepp), sibling(sepp,hans).
```

the last rule removes duplicate `sibling` constraint

Program with 4 rules to infer siblings:

```
mother(X,Y) ==> parent(X,Y).
father(X,Y) ==> parent(X,Y).
parent(X,Z), parent(Y,Z) ==>
  sibling(X,Y).
sibling(X,Y) \ sibling(Y,X) <=> true.
```

CodeRules Example

```
assignmentExpression matching AssignmentExpression ae <with subconcepts> <always apply>
{
  on <term LType, RType>
    typeOf(@ae.lValue, LType), typeOf(@ae.rValue, RType)
    activate
      convertsTo(RType, LType), typeOf(@ae, LType)
}
```

Why we chose CHR?

Benefits:

- Well-defined semantics
- Complete freedom over definition of a typesystem
- Type systems are naturally expressed as a set of declarative rules

3. Features of CodeRules

- Rule templates
- Logical variables & unification
- Dataforms & pattern matching
- Embedded predicates
- Evaluation trace

Rule templates

```
write_localVarDecl matching LocalVariableDeclaration lvd <with subconcepts> <always apply>
{
  on start
    activate
      %%
        <% loc(@lvd) %>
        if (lvd.initializer.isNotNull) {
          <% write(@lvd, @lvd) %>
        }
      %%
}
```

Logical variables & unification

Logical variables

- Two states: free (unassigned) or bound (assigned)
- Bound variables are immutable
- Unified variables share their state

Examples of unification

Simple unification:

```
Y=X, X=42 => Y=42
```

More interesting unification:

```
g(A,f(B))=g(h(C),f(D)) => A=h(C) & B=D
```

Dataforms & pattern matching

Dataform (or *term*):

- Simple immutable recursive data structure
- Contains values, other terms or term lists
- Suitable for unification
- Declared in a separate root “Dataform Table”

```
classifierType (
  value classifier
  value kind
  list parameter
)
```

```
longType : primType (
  final value kind = concept/LongType/
  value val = 0L
)
```

Dataforms & pattern matching

```
converts_any_lowerBoundType <no input> <always apply>
{
  on <term Type, LBType, LBnd>
    ~convertsTo(Type, LBType = lowerBoundType(parameter: LBnd))
    activate
      convertsTo(Type, LBnd)
}

converts_upperBoundType_bound <no input> <always apply>
{
  on <term Type, UBType, UBnd>
    ~convertsTo(Type, UBType = upperBoundType(parameter: UBnd))
    activate
      convertsTo(UBnd, Type)
}
```

Predicates

Predicates are used to

- *Check* conditions in *guards*
- *Assert* some properties in *bodies*

Embedded predicates:

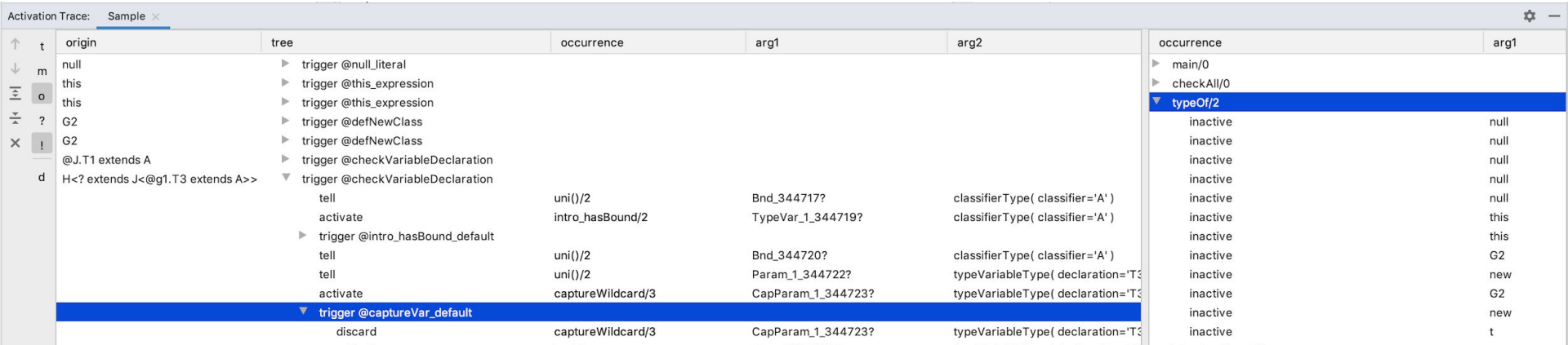
- Unification `X=Y`
- Equality `X==Y`
- Substitution `A[X->Y]`
- Evaluation `eval(...)`

```
converts_free_to_free <no input> <always apply>
{
  on <term S, T>
    ~convertsTo(S, T)
    when
      isFree(S), isFree(T)
    activate
      S = T
}
```

Evaluation trace

Indispensable for debugging:

- Contains complete information about evaluated program
- Can be accessed from highlighted errors
- Has “jump to failed rule” functionality



What you can do with CodeRules?

CodeRules is expressive:

- Turing-complete
- “Batteries included”: has necessary machinery for logical inference
- Advanced samples available

You can implement:

- Type inference
- Linear types
- Dependent types

4. Status & roadmap

Roadmap

Stages:

1. CodeRules as a plugin
2. CodeRules as a part of MPS core
3. CodeRules as *the* way to define a type system

Legacy typesystem aspect will be preserved for compatibility

Transitioning to the new aspect

Considerations:

- No migration — type systems must be rewritten
- Case of several languages
- Case of extending languages

Work in progress

- Type systems for BL extensions
- Incrementality
- Reporting of type checking errors
- Translation of SNode to Dataform

Where you can learn more & try

—

- Source code (with BL & other samples) — github.com/JetBrains/mps-coderules
- Documentation — jetbrains.github.io/mps-coderules/

Currently you need to build CodeRules from source
All instructions are in place

Thank you
for your attention

jetbrains.com