Light-Weight Object-FL-Programming in Java with “Paisley”

Baltasar Trancón y Widemann $^1$ $^2$ Markus Lepper$^2$

$^1$ Technische Universität Ilmenau, Baltasar.Trancon@tu-ilmenau.de

$^2$ <semantics/> GmbH, Berlin, post@markuslepper.eu

1 **Design Principles**
- Project Context
- Paisley Design Goals

2 **Paisley Implementation**
- Constructing and Applying Patterns

3 **Paisley and Logic Programming**
- Basic Principle
- Evaluation: Cryptarithmetic Puzzle
Authors in General

- Compiler Construction, Language Design
- OPAL, DSLs, Specification Languages (TTCN-3, TCI, Z), Temporal Logics, XML, Signal Processing

Project Context

- meta–tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established “object oriented” practical coding (currently: Java)
Authors in General

- Compiler Construction, Language Design
- OPAL, DSLs, Specification Languages (TTCN-3, TCI, Z), Temporal Logics, XML, Signal Processing

Project Context

- meta-tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established “object oriented” practical coding (currently: Java)
Authors in General

- Compiler Construction, Language Design
- OPAL, DSLs, Specification Languages (TTCN-3, TCI, Z), Temporal Logics, XML, Signal Processing

Project Context

- meta-tools — Collection of all our tools for generic programming, compiler construction and language processing
- Incorporating declarative techniques / ways of thinking into established “object oriented” practical coding (currently: Java)
Strategies

- **In the Large:**
  - Source code generation of **typed** models:
    - `tdom` — typed XML model
    - `umod` — data model plus processing infrastructure

- **In the Small:**
  - **Embedded DSLs**
    - `ops` — algebras of relations, finite maps, iterators, …
    - `paisley` — pattern matching algebra
**Strategies**

- **In the Large:**
  Source code generation of **typed** models:
  - `tdom` — typed XML model
  - `umod` — data model plus processing infra structure

- **In the Small:**
  Embedded DSLs
  - `ops` — algebras of relations, finite maps, iterators, ...
  - `paisley` — pattern matching algebra
Embedded Domain Specific Languages

- Here the “domain” is an area of applied mathematics
- Utmost smooth embedding, full reification
- Of course not comparable to large-scale implementations or dedicated machines w.r.t. transformation, optimization, etc.
- But immediately applicable in wide-spread practice since API based,
- and certain “pedagogical” effects possible
Embedded Domain Specific Languages

- Here the “domain” is an area of applied mathematics
- Utmost smooth embedding, full reification
  - Of course **not** comparable to large-scale implementations or dedicated machines w.r.t. transformation, optimization, etc.
  - But immediately applicable in wide-spread practice since API based,
- and certain “pedagogical” effects possible
Embedded Domain Specific Languages

- Here the “domain” is an area of applied mathematics
- Utmost smooth embedding, full reification
- Of course not comparable to large-scale implementations or dedicated machines w.r.t. transformation, optimization, etc.
- But immediately applicable in wide-spread practice since API based,
- and certain “pedagogical” effects possible
Embedded Domain Specific Languages

- Here the “domain” is an area of applied mathematics
- Utmost smooth embedding, full reification
- Of course not comparable to large-scale implementations or dedicated machines w.r.t. transformation, optimization, etc.
- But immediately applicable in wide-spread practice since API based,

- and certain “pedagogical” effects possible
Embedded Domain Specific Languages

- Here the “domain” is an area of applied mathematics
- Utmost smooth embedding, full reification
- Of course not comparable to large-scale implementations or dedicated machines w.r.t. transformation, optimization, etc.
- But immediately applicable in wide-spread practice since API based,
- and certain “pedagogical” effects possible
Paisley Design Goals

Seen from the OO programmers perspective:

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing or compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless used
Paisley Design Goals

Seen from the OO programmers perspective:

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing or compilation overhead at runtime
9. **Support for continuation-style nondeterminism**
10. Nondeterminism incurs no significant cost unless used
Paisley Design Goals

Seen from the OO programmers perspective:

1. Statically type-safe variables
2. Statically type-safe patterns
3. No language extension: independent of host compiler
4. No assumptions on host language beyond standard OOP
5. No adaptation of model datatypes required
6. Support for multiple views per type
7. Declarative, readable, writeable, customizable
8. Full reification: no parsing or compilation overhead at runtime
9. Support for continuation-style nondeterminism
10. Nondeterminism incurs no significant cost unless used
Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse.
  Object-oriented constructors *do not*
- Instead use getter patterns
- and pre-defined primitive type patterns
  eq, equal, less, NaN, …
- and class test/casting and other reflection based patterns
- Library of basic combinators
  – and of generic combinators for collections
- Arbitrary user defined classes adhering to the
  paisley.Pattern<A> interface
  (Up to random generator !-)
- Independent of all these:
  Explicit bindings of Variables
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse.
  Object-oriented constructors *do not*

- Instead use *getter* patterns

- and pre-defined *primitive type patterns*
  - `eq, equal, less, NaN, ...`

- and *class test/casting* and other *reflection based patterns*

- Library of *basic combinators*
  - and of *generic combinators for collections*

- Arbitrary *user defined* classes adhering to the *paisley.Pattern<A> interface*
  (Up to random generator !-)

- Independent of all these:
  Explicit bindings of *Variables*
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors **do not**
- Instead use **getter** patterns
  - and pre-defined **primitive type patterns**
    - eq, equal, less, NaN, …
  - and **class test/casting** and other **reflection based patterns**
- Library of **basic combinators**
  - and of generic combinators for collections
- Arbitrary **user defined** classes adhering to the paisley.Pattern<A> interface
  (Up to random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors *do not*
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  - eq, equal, less, NaN, ...  
- and **class test/casting** and other **reflection based patterns**
- Library of **basic combinators**
  - and of generic combinators for collections
- Arbitrary **user defined** classes adhering to the `paisley.Pattern<A>` interface
  (Up to random generator !-)
- Independent of all these:
  - Explicit bindings of **Variables**
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors do not
- Instead use getter patterns
- and pre-defined primitive type patterns eq, equal, less, NaN, ...
- and class test/casting and other reflection based patterns
- Library of basic combinators
  - and of generic combinators for collections
- Arbitrary user defined classes adhering to the paisley.Pattern<A> interface
  (Up to random generator !-)
- Independent of all these:
  Explicit bindings of Variables
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors *do not*
- Instead use *getter* patterns
- and pre-defined *primitive type patterns* (eq, equal, less, NaN, ...)
- and *class test/casting* and other *reflection based* patterns
- Library of *basic combinators*
  - and of generic combinators for collections
- Arbitrary *user defined* classes adhering to the paisley.Pattern<A> interface (Up to random generator !-)
- Independent of all these:
  Explicit bindings of *Variables*
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors do not.
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  - eq, equal, less, NaN, ...
- and **class test/casting** and other **reflection based** patterns
- Library of **basic combinators**
  - and of generic **combinators for collections**
- Arbitrary **user defined** classes adhering to the paisley.Pattern<A> interface
  (Up to random generator !-)
- Independent of all these:
  Explicit bindings of **Variables**
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors do not.
- Instead use getter patterns.
- and pre-defined primitive type patterns: eq, equal, less, NaN, ...
- and class test/casting and other reflection based patterns.
- Library of basic combinators
  - and of generic combinators for collections.
- Arbitrary user defined classes adhering to the paisley.Pattern<A> interface.
  (Up to random generator !-)
- Independent of all these: Explicit bindings of Variables.
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors *do not*
- Instead use **getter** patterns
- and pre-defined **primitive type patterns**
  - eq, equal, less, NaN, ...
- and **class test/casting** and other **reflection based** patterns
- Library of **basic combinators**
  - and of generic **combinators for collections**
- Arbitrary **user defined** classes adhering to the paisley.Pattern<A> interface
  (Up to random generator !-)
- Independent of all these:
  **Explicit bindings of Variables**
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors *do not*
- Instead use *getter* patterns
- and pre-defined *primitive type patterns*
  - eq, equal, less, NaN, ...
- and *class test/casting* and other *reflection based* patterns
- Library of *basic combinator*s
  - and of generic *combinators for collections*
- Arbitrary *user defined* classes adhering to the `paisley.Pattern<A>` interface
  (Up to random generator !-)
- Independent of all these: Explicit bindings of *Variables*
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors do not.
- Instead use getter patterns (← user must code)
- and pre-defined primitive type patterns
  eq, equal, less, NaN, ...
- and class test/casting and other reflection based patterns
- Library of basic combinators
  - and of generic combinators for collections
- Arbitrary user defined classes adhering to the paisley.Pattern<A> interface (← user may code)
  (Up to random generator !-)
- Independent of all these:
  Explicit bindings of Variables
Paisley Design Fundamentals

Seen from the pattern matching algebra perspective:

- Algebraic constructors do have an inverse. Object-oriented constructors do not.
- Instead use getter patterns (← user must code)
- and pre-defined primitive type patterns (eq, equal, less, NaN, ...)
- and class test/casting and other reflection based patterns
- Library of basic combinators (either() ⇒ nondet.)
  - and of generic combinators for collections (⇒ nondet.)
- Arbitrary user defined classes adhering to the paisley.Pattern<A> interface (← user may code)
  (Up to random generator !-) (may imply nondet.)
- Independent of all these: Explicit bindings of Variables
final D datum = ...
final D datum = ...
Variable<X> var1
= Pattern.<X>variable();
Variable<X> var2
= Pattern.<X>variable();
**Constructing and Applying Patterns**

```java
final D datum = ...
Variable<X> var1
    = Pattern.<X>variable();
Variable<X> var2
    = Pattern.<X>variable();
Pattern<D> p
    = Pattern.either(
        ...(
            ...)
    ...
    ...);
```
final D datum = ...
Variable<X> var1
= Pattern.<X>variable();
Variable<X> var2
= Pattern.<X>variable();
Pattern<D> p
    = Pattern.either((... (var1) ... ...
                     (var2) ...));
if (p.match(datum)) {...
     // maybe var1/var2 is meaningful
}
Simple Paisley Example

class D {
    public int f ;
    public List<D> subs = new ArrayList<D>();
}

Pattern<D> p0 = get_f(eq(17));

if (p0.match(d)) do {
    // do something
} while (p0.matchAgain())
Simple Paisley Example

class D {
    public int f;
    public List<D> subs = new ArrayList<D>();
}

Pattern<D> p0 = get_f(eq(17));
Pattern<D> p1 = and(p0, get_subs(any(p0)));

if (p1.match(d)) do {
    // do something
} while (p1.matchAgain())
```java
class D {
    public int f;
    public List<D> subs = new ArrayList<D>();
}
```

```java
Variable<D> v0 = new Variable<D>();
Pattern<D> p0 = get_f(eq(17));
Pattern<D> p1 = and(p0, get_subs(any(p0)));
```

```java
if (p1.match(d)) do {
    // do something
} while (p1.matchAgain())
```
Simple Paisley Example

```java
class D {
    public int f ;
    public List<D> subs = new ArrayList<D>();
}
Variable<D> v0 = new Variable<D>();
Pattern<D> p0 = get_f(eq(17));
Pattern<D> p1 = and(p0, get_subs(and(any(p0), v0)));

if (p1.match(d)) do {
    // do something
} while (p1.matchAgain())
```
Simple Paisley Example

class D {
    public int f;
    public List<D> subs = new ArrayList<D>();
}
Variable<D> v0 = new Variable<D>();
Pattern<D> p0 = get_f(eq(17));
Pattern<D> p1 = and(p0, get_subs(and(any(p0), v0)));

if (p1.match(d)) do {
    // do something with v0
} while (p1.matchAgain())
Simple Paisley Example

class D {
    public int f;
    public List<D> subs = new ArrayList<D>();
}
Variable<D> v0 = new Variable<D>(), v2 = new Variable<D>;
Pattern<D> p0 = get_f(eq(17));
Pattern<D> p1 = and(p0, get_subs(and(any(p0), v0)));
Pattern<D> p2 = and(v2.star(get_subs(any(v2))), p1);

if (p2.match(d)) do {
    // do something with v0
} while (p2.matchAgain())
Simple Paisley Example

class D {
    public int f ;
    public List<D> subs = new ArrayList<D>();
}

Variable<D> v0 = new Variable<D>(), v2 = new Variable<D>;

Pattern<D> p0 = get_f(eq(17));
Pattern<D> p1 = and(p0, getsubs(and(any(p0), v0)));
Pattern<D> p2 = and(v2.star(getsubs(any(v2))), p1);

Pattern<D> descend(Pattern<D> p){
    Variable<D> v = new Variable<D>();
    return v.star(getsubs(any(and(v, p))));
}
Paisley Practical Properties

- Fully reified
  Lifted to “object” level, usable as function argument, computation result, serializable...

- Compact notation
  Compare `star` closure to recursive function definition

- `static import` of construction functions

- Exploiting Java type inference

- Declarative and imperative construction can be mixed

- Declarative and imperative evaluation can be mixed

- These mixings bring advantages and jeopardies!
Paisley Practical Properties

- Fully reified
  Lifted to “object” level, usable as function argument, computation result, serializable...

- Compact notation
  Compare star closure to recursive function definition

  - static import of construction functions
  - Exploiting Java type inference
  - Declarative and imperative construction can be mixed
  - Declarative and imperative evaluation can be mixed
  - These mixings bring advantages and jeopardies!
Paisley Practical Properties

- Fully reified
  Lifted to “object” level, usable as function argument, computation result, serializable . . .

- Compact notation
  Compare star closure to recursive function definition

- static import of construction functions

- Exploiting Java type inference
  - Declarative and imperative construction can be mixed
  - Declarative and imperative evaluation can be mixed
  - These mixings bring advantages and jeopardies!
Paisley Practical Properties

- Fully reified
  Lifted to “object” level, usable as function argument, computation result, serializable . . .

- Compact notation
  Compare star closure to recursive function definition

- static import of construction functions

- Exploiting Java type inference

- Declarative and imperative construction can be mixed
  - Declarative and imperative evaluation can be mixed
  - These mixings bring advantages and jeopardies!
Paisley Practical Properties

- Fully reified
  Lifted to “object” level, usable as function argument, computation result, serializable . . .

- Compact notation
  Compare star closure to recursive function definition

- static import of construction functions

- Exploiting Java type inference

- Declarative and imperative construction can be mixed

- Declarative and imperative evaluation can be mixed

  These mixings bring advantages and jeopardies!
Paisley Practical Properties

- Fully reified
  Lifted to “object” level, usable as function argument, computation result, serializable ...
- Compact notation
  Compare star closure to recursive function definition
- static import of construction functions
- Exploiting Java type inference
- Declarative and imperative construction can be mixed
- Declarative and imperative evaluation can be mixed
- These mixings bring advantages and jeopardies!
Simple Paisley Example – Approaching Logic Programming

class D {
    public int f ;
    public List<D> subs = new ArrayList<D>();
}

Variable<D> v0 = new Variable<D>();
Pattern<D> p2 = and (get_f(v0),
    get_subs(any(get_f(eq(v0.value())))))
Implementation of the `both` operator

```java
public Pattern<A> both(Pattern<A> fst, Pattern<A> snd) {
    return new Both(fst, snd); }

class Both<A> {
    private Pattern<A> left, right;
    private A target_save;
    private boolean left_matched;
    public boolean match(A target) {
        if (left_matched = left.match(target)) {
            target_save = target;
            if (right.match(target)) return true;
            else while (left_matched = left.matchAgain())
                if (right.match(target_save)) return true;
        }
        return false;
    }
}
```
Implementation of the `both` operator

```java
public Pattern<A> both(Pattern<A> fst, Pattern<A> snd) {
    return new Both(fst, snd);
}

class Both<A> {
    private Pattern<A> left, right;
    private A target_save;
    private boolean left_matched;
    public boolean matchAgain() {
        if (left_matched) {
            if (right.matchAgain())
                return true;
            else while (left_matched = left.matchAgain())
                if (right.match(target_save))
                    return true;
        }
        return false;
    }
}
```
all nondeterminism/backtracking realized de-centrally in either() and both() combinators and their variants anyElement(), all(), ...

- fully free compositional
- no optimizing transformations
- no automated stack re-usage

Obvious question:

How does it perform?
all nondeterminism/backtracking realized de-centrally in `either()` and `both()` combinators and their variants `anyElement()`, `all()`, ...

+ fully free compositional

- no optimizing transformations
- no automated stack re-usage

Obvious question:

*How does it perform?*
all nondeterminism/backtracking realized de-centrally in \texttt{either()} and \texttt{both()} combinators and their variants \texttt{anyElement()}, \texttt{all()}, ...

+ fully free compositional

- no optimizing transformations

- no automated stack re-usage

Obvious question:

\textit{How does it perform?}
all nondeterminism/backtracking realized de-centrally in `either()` and `both()` combinators and their variants `anyElement()`, `all()`, ...

fully free compositional

no optimizing transformations

no automated stack re-usage

**Obvious question:**

*How does it perform?*
all nondeterminism/backtracking realized de-centrally in either() and both() combinators and their variants anyElement(), all(), ...

+ fully free compositional
- no optimizing transformations
- no automated stack re-usage

Obvious question:

How does it perform?
Matching against *generated* constellations

\[
\begin{array}{cccc}
S & E & N & D \\
\hline
M & O & R & E \\
\hline
M & O & N & E & Y
\end{array}
\]

\[
\text{size of search space } = 10^8
\]
Matching against *generated* constellations

\[
\begin{array}{cccc}
S & E & N & D \\
\downarrow & & & \\
M & O & R & E \\
\hline
M & O & N & E & Y
\end{array}
\]

size of search space $= 10^8$
Strategy 1 / Naïve

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
Strategy 1 / Naïve

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

\(s \in \mathbb{Z} \land e \in \mathbb{Z}\)
Strategy 1 / Naïve

\{0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 \}

\(s \in Z \land e \in Z \land n \in Z \land d \in Z \land m \in Z \land o \in Z \land r \in Z \land y \in Z\)
Strategy 1 / Naïve

\{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}

\( s \in \mathbb{Z} \land e \in \mathbb{Z} \land n \in \mathbb{Z} \land d \in \mathbb{Z} \land m \in \mathbb{Z} \land o \in \mathbb{Z} \land r \in \mathbb{Z} \land y \in \mathbb{Z} \land s \neq 0 \land m \neq 0 \)
Strategy 1 / Naïve

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

\(s \in \mathbb{Z} \land e \in \mathbb{Z} \land n \in \mathbb{Z} \land d \in \mathbb{Z} \land m \in \mathbb{Z} \land o \in \mathbb{Z} \land r \in \mathbb{Z} \land y \in \mathbb{Z}\)

\(\land s \neq 0 \land m \neq 0\)

\(\land s \neq e \land s \neq n \land s \neq d \land s \neq m \land s \neq o \land s \neq r \land s \neq y\)

\(\land e \neq n \land e \neq d \land e \neq m \land e \neq o \land e \neq r \land e \neq y\)

\(\land n \neq d \land n \neq m \land n \neq o \land n \neq r \land n \neq y\)

\(\land d \neq m \land d \neq o \land d \neq r \land d \neq y\)

\(\land m \neq o \land m \neq r \land m \neq y\)

\(\land o \neq r \land o \neq y\)

\(\land r \neq y\)
Strategy 1 / Naïve

\{ 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 \}

\( s \in \mathbb{Z} \land e \in \mathbb{Z} \land n \in \mathbb{Z} \land d \in \mathbb{Z} \land m \in \mathbb{Z} \land o \in \mathbb{Z} \land r \in \mathbb{Z} \land y \in \mathbb{Z} \)
\( \land s \neq 0 \land m \neq 0 \)
\( \land s \neq e \land s \neq n \land s \neq d \land s \neq m \land s \neq o \land s \neq r \land s \neq y \)
\( \land e \neq n \land e \neq d \land e \neq m \land e \neq o \land e \neq r \land e \neq y \)
\( \land n \neq d \land n \neq m \land n \neq o \land n \neq r \land n \neq y \)
\( \land d \neq m \land d \neq o \land d \neq r \land d \neq y \)
\( \land m \neq o \land m \neq r \land m \neq y \)
\( \land o \neq r \land o \neq y \)
\( \land r \neq y \)
\( \land \text{sum} \quad // \equiv \begin{pmatrix} 1000 * s + 100 * e + 10 * n + d \\ +1000 * m + 100 * o + 10 * r + e \\ = 10000 * m + 1000 * o + 100 * n + 10 * e + y \end{pmatrix} \)
Strategy 2 / Early tests

\[ \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \]

\[ s \in \mathbb{Z} \land s \neq 0 \]
Strategy 2 / Early tests

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

\[s \in \mathbb{Z} \land s \neq 0\]

\[\land e \in \mathbb{Z} \land e \neq s\]
Strategy 2 / Early tests

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

\(s \in \mathbb{Z} \land s \neq 0\)
\(\land e \in \mathbb{Z} \land e \neq s\)
\(\land n \in \mathbb{Z} \land n \neq s \land n \neq e\)
Strategy 2 / Early tests

\[ \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \]

\[ s \in \mathbb{Z} \land s \neq 0 \]
\[ \land e \in \mathbb{Z} \land e \neq s \]
\[ \land n \in \mathbb{Z} \land n \neq s \land n \neq e \]
\[ \land d \in \mathbb{Z} \land d \neq s \land d \neq e \land d \neq n \]
\[ \ldots \]
\[ \ldots \]
\[ \land y \in \mathbb{Z} \land y \neq s \land y \neq e \land y \neq n \land \ldots \]

\[ \land \text{sum} \]
Strategy 3 / Partial Sums

\[ \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \]

\[ d \in \mathbb{Z} \land e \in \mathbb{Z} \land y \in \mathbb{Z} \land d \neq e \land d \neq y \land e \neq y \]
**Strategy 3 / Partial Sums**

\[
\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
\]

\[d \in \mathbb{Z} \land e \in \mathbb{Z} \land y \in \mathbb{Z} \land d \neq e \land d \neq y \land e \neq y \land (d + e) \mod 10 = y\]
Strategy 3 / Partial Sums

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

\[d \in \mathbb{Z} \land e \in \mathbb{Z} \land y \in \mathbb{Z} \land d \neq e \land d \neq y \land e \neq y\]
\[(d + e) \mod 10 = y\]
\[n \in \mathbb{Z} \land r \in \mathbb{Z} \land n \neq d \land n \neq e \land n \neq y \land r \neq d \land r \neq e \land r \neq y\]
\[(d + e + 10 \times n + 10 \times r) \mod 100 = y + 10 \times e\]
Strategy 3 / Partial Sums

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

\[d \in \mathbb{Z} \land e \in \mathbb{Z} \land y \in \mathbb{Z} \land d \neq e \land d \neq y \land e \neq y \land (d + e) \mod 10 = y \]
\[\land n \in \mathbb{Z} \land r \in \mathbb{Z} \land n \neq d \land n \neq e \land n \neq y \land r \neq d \land r \neq e \land r \neq y \land (d + e + 10 \ast n + 10 \ast r) \mod 100 = y + 10 \ast e \]

\[\land \text{sum}\]
## Results

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy 1 – Naïve</td>
<td></td>
<td>5,470.24</td>
</tr>
<tr>
<td>Strategy 2 – Early Tests</td>
<td>simple re-arrangement of constraints</td>
<td>770.25</td>
</tr>
<tr>
<td>Strategy 3 – Partial Sums</td>
<td>elaborate auxiliary data structures</td>
<td>2.37</td>
</tr>
<tr>
<td>(specialized “C” code)</td>
<td></td>
<td>(0.17)</td>
</tr>
<tr>
<td>[Tamura2004]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More . . .

- ... in the proceedings
- ... meta-tools users’ guide at http://bandm.eu/metatools
- ... including Paisley demo download