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

Kiel, WFLP 2013, 11. Sept. 2013

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 infra structure
- 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**

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 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 (⟵ 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**

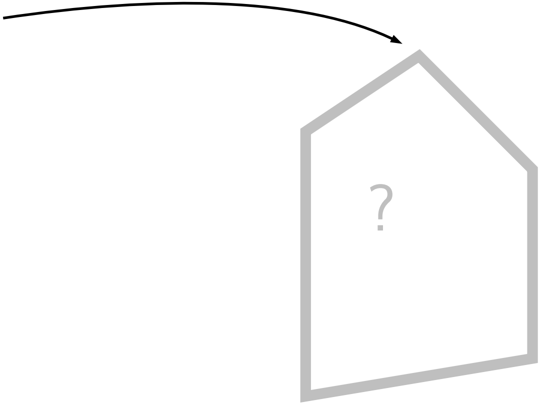
Paisley Design Fundamentals

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**

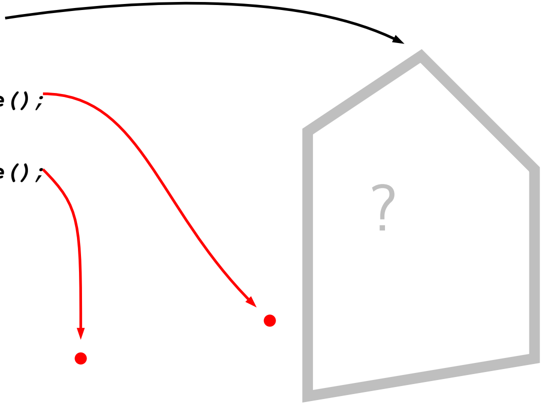
Constructing and Applying Patterns

```
final D datum = ...
```



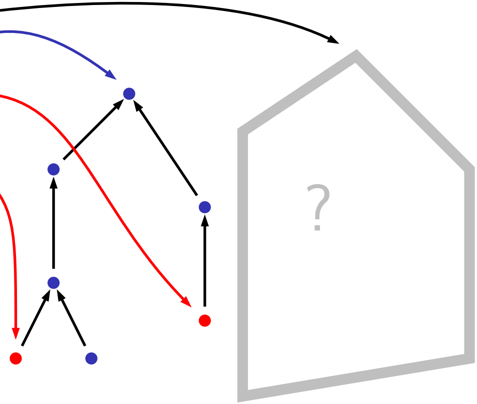
Constructing and Applying Patterns

```
final D datum = ...
Variable<X> var1
= Pattern.<X>variable();
Variable<X> var2
= Pattern.<X>variable();
```



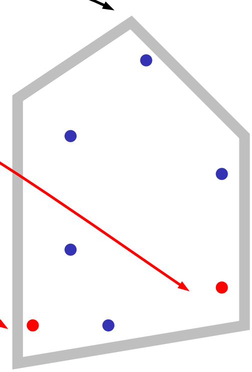
Constructing and Applying Patterns

```
final D datum = ...
Variable<X> var1
= Pattern.<X>variable();
Variable<X> var2
= Pattern.<X>variable();
Pattern<D> p
   = Pattern.either( ...
   ...(var1) ...
   ...(var2) ...);
```



Constructing and Applying Patterns

```
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())
```

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(       any(p0)        ));

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
} 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, get_subs(  and(any(p0), v0) ));
Pattern<D> p2 = and(v2.star(get_subs(any(v2))), p1);

Pattern<D> descend(Pattern<D> p){
  Variable<D> v = new Variable<D>();
  return v.star(get_subs(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

```
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

```
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 `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 `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

```
  S E N D
  M O R E
---------
M O N E Y
```

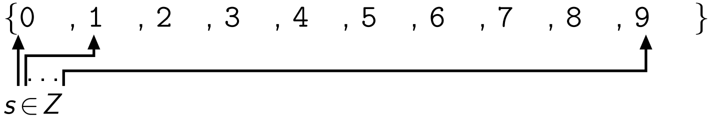
size of search space = 10^8

Matching against *generated* constellations

```
  S E N D
  M O R E
---------
M O N E Y
```

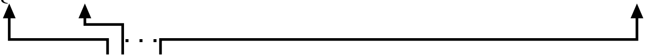
size of search space $= 10^8$

Strategy 1 / Naïve



Strategy 1 / Naïve

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



$s \in Z \ \wedge \ e \in Z$

Strategy 1 / Naïve

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

$s \in Z\ \wedge e \in Z \wedge n \in Z \wedge d \in Z \wedge m \in Z \wedge o \in Z \wedge r \in Z \wedge y \in Z$

Strategy 1 / Naïve

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

$s \in Z \ \wedge e \in Z \wedge n \in Z \wedge d \in Z \wedge m \in Z \wedge o \in Z \wedge r \in Z \wedge y \in Z$
$\wedge s \neq 0 \wedge m \neq 0$

Strategy 1 / Naïve

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

$$s \in Z \wedge e \in Z \wedge n \in Z \wedge d \in Z \wedge m \in Z \wedge o \in Z \wedge r \in Z \wedge y \in Z$$

$$\wedge s \neq 0 \wedge m \neq 0$$

$$\wedge s \neq e \wedge s \neq n \wedge s \neq d \wedge s \neq m \wedge s \neq o \wedge s \neq r \wedge s \neq y$$

$$\wedge e \neq n \wedge e \neq d \wedge e \neq m \wedge e \neq o \wedge e \neq r \wedge e \neq y$$

$$\wedge n \neq d \wedge n \neq m \wedge n \neq o \wedge n \neq r \wedge n \neq y$$

$$\wedge d \neq m \wedge d \neq o \wedge d \neq r \wedge d \neq y$$

$$\wedge m \neq o \wedge m \neq r \wedge m \neq y$$

$$\wedge o \neq r \wedge o \neq y$$

$$\wedge r \neq y$$

Strategy 1 / Naïve

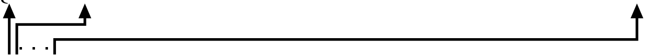
$$\{0\quad ,1\quad ,2\quad ,3\quad ,4\quad ,5\quad ,6\quad ,7\quad ,8\quad ,9\quad \}$$

$$
\begin{aligned}
& s \in Z \;\wedge\; e \in Z \wedge n \in Z \wedge d \in Z \wedge m \in Z \wedge o \in Z \wedge r \in Z \wedge y \in Z \\
& \wedge s \neq 0 \wedge m \neq 0 \\
& \qquad \wedge s \neq e \wedge s \neq n \wedge s \neq d \wedge s \neq m \wedge s \neq o \wedge s \neq r \wedge s \neq y \\
& \qquad \wedge e \neq n \wedge e \neq d \wedge e \neq m \wedge e \neq o \wedge e \neq r \wedge e \neq y \\
& \qquad \wedge n \neq d \wedge n \neq m \wedge n \neq o \wedge n \neq r \wedge n \neq y \\
& \qquad \wedge d \neq m \wedge d \neq o \wedge d \neq r \wedge d \neq y \\
& \qquad \wedge m \neq o \wedge m \neq r \wedge m \neq y \\
& \qquad \wedge o \neq r \wedge o \neq y \\
& \qquad \wedge r \neq y
\end{aligned}
$$

$$\wedge \quad \textbf{sum} \quad // \equiv \left(\begin{array}{r} 1000 * s + 100 * e + 10 * n + d \\ +1000 * m + 100 * o + 10 * r + e \\ = 10000 * m + 1000 * o + 100 * n + 10 * e + y \end{array} \right)$$

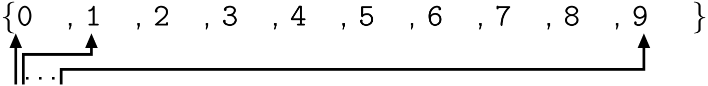
Strategy 2 / Early tests

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



$s \in Z \wedge s \neq 0$

Strategy 2 / Early tests

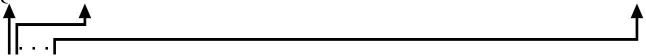


$s \in Z \wedge s \neq 0$

$\wedge\, e \in Z \wedge e \neq s$

Strategy 2 / Early tests

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



$s \in Z \wedge s \neq 0$
$\wedge\, e \in Z \wedge e \neq s$
$\wedge\, n \in Z \wedge n \neq s \wedge n \neq e$

Strategy 2 / Early tests

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

...

$s \in Z \wedge s \neq 0$

$\wedge\ e \in Z \wedge e \neq s$

$\wedge\ n \in Z \wedge n \neq s \wedge n \neq e$

$\wedge\ d \in Z \wedge d \neq s \wedge d \neq e \wedge d \neq n$

...

...

$\wedge\ y \in Z \wedge y \neq s \wedge y \neq e \wedge y \neq n \wedge$ ...

$\wedge$ **sum**

Strategy 3 / Partial Sums

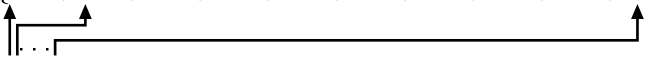
$\{0\ ,1\ ,2\ ,3\ ,4\ ,5\ ,6\ ,7\ ,8\ ,9\ \}$

. . .

$d \in Z \wedge e \in Z \wedge y \in Z \wedge d \neq e \wedge d \neq y \wedge e \neq y$

Strategy 3 / Partial Sums

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

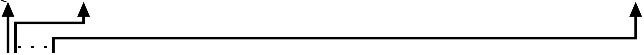


$d \in Z \wedge e \in Z \wedge y \in Z \wedge d \neq e \wedge d \neq y \wedge e \neq y$

$\wedge (d + e) \bmod 10 = y$

Strategy 3 / Partial Sums

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



$d \in Z \wedge e \in Z \wedge y \in Z \wedge d \neq e \wedge d \neq y \wedge e \neq y$

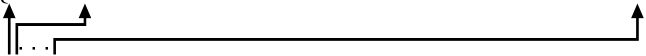
$\wedge (d + e) \bmod 10 = y$

$\wedge\ n \in Z \wedge r \in Z \wedge n \neq d \wedge n \neq e \wedge n \neq y \wedge r \neq d \wedge r \neq e \wedge r \neq y$

$\wedge (d + e + 10 * n + 10 * r) \bmod 100 = y + 10 * e$

Strategy 3 / Partial Sums

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



$d \in Z \wedge e \in Z \wedge y \in Z \wedge d \neq e \wedge d \neq y \wedge e \neq y$

$\wedge (d + e) \bmod 10 = y$

$\wedge\, n \in Z \wedge r \in Z \wedge n \neq d \wedge n \neq e \wedge n \neq y \wedge r \neq d \wedge r \neq e \wedge r \neq y$

$\wedge (d + e + 10 * n + 10 * r) \bmod 100 = y + 10 * e$

...

...

$\wedge$ **sum**

Results

(KiCS)		(7 490)
Strategy 1 – Naïve		5 470.24
Strategy 2 – Early Tests	simple re-arrangement of constraints	770.25
Strategy 3 – Partial Sums	elaborate auxiliary data structures	2.37
(specialized “C” code) [Tamura2004]		(0.17)

More . . .

- . . . in the proceedings
- . . . $^{\text{meta}}$–tools users' guide at `http://bandm.eu/metatools`
- . . . including Paisley demo download