Evolving unrestricted Java software with FINCH

Michael Orlov and Moshe Sipper

orlovm, sipper@cs.bgu.ac.il

Department of Computer Science
Ben-Gurion University, Israel

Dynamic Adaptive SBSE
The 26th CREST Open Workshop
April 2013, UCL

GP: Programs or Representations?

"While it is common to describe GP as evolving **programs**, GP is not typically used to evolve programs in the familiar Turing-complete languages humans normally use for software development."

A Field Guide to Genetic Programming
[Poli, Langdon, and McPhee, 2008]

GP: Programs or Representations?

"While it is common to describe GP as evolving **programs**, GP is not typically used to evolve programs in the familiar Turing-complete languages humans normally use for software development."

"It is instead more common to evolve programs
(or expressions or formulae)
in a **more constrained** and often **domain-specific language**."

A Field Guide to Genetic Programming
[Poli, Langdon, and McPhee, 2008]

Our Goals

From programs. . .

Evolve actual programs

written in Java

. . . to software!

Improve (existing) software

written in unrestricted Java

Our Goals

From programs. . .

Evolve actual programs
written in Java

. . . to software!

Improve (existing) software
written in unrestricted Java

Extending prior work

Existing work uses **restricted subsets** of Java bytecode as **representation language** for GP individuals

We evolve **unrestricted bytecode**

Let's Evolve Java Source Code

- Rely on the building blocks in the initial population
- Defining **genetic operators** is problematic
- How do we define **good** source-code crossover?

Factorial *(recursive)*

```
class F {
  int fact(int n) {
    int ans = 1;

    if (n > 0)
      ans = n *
        fact(n-1);

    return ans;
  }
}
```

$\Leftarrow$

Factorial *(iterative)*

```
class F {
  int fact(int n) {
    int ans = 1;

    for (;  n > 0;  n--)
      ans = ans * n;


    return ans;
  }
}
```

Oops

- **Source-level crossover** typically produces garbage

Factorial *(recursive* $\overset{\leftarrow}{\times}$ *iterative)*

```
class F {
  int fact(int n) {
    int ans = 1;

    if (n > = 1;
      for (;  n > 0;  n--)
        ans = ans * n;  n-1);

    return ans;
  }
}
```

Oops

- **Source-level crossover** typically produces garbage

Factorial *(recursive $\overleftarrow{\times}$ iterative)*

```
class F {
  int fact(int n) {
    int ans = 1;

    if (n > = 1;
      for (; n > 0; n--)
        ans = ans * n; n-1);

    return ans;
  }
}
```

Parse Trees

- Maybe we can design **better** genetic operators?

Parse Trees

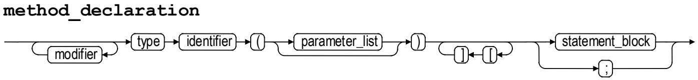
- Maybe we can design **better** genetic operators?
- Maybe. . . but too much harsh **syntax**
  Possibly use **parse tree**?

Parse Trees

- Maybe we can design **better** genetic operators?
- Maybe... but too much harsh **syntax**
  Possibly use **parse tree**?

Just one BNF rule *(of many)*

`method_declaration` ::$\Longrightarrow$
  `modifier`* `type` `identifier`
  "(" `parameter_list`$^?$ ")" "[ ]"*
  $\langle$ `statement_block` | ";" $\rangle$



Bytecode

Better than parse trees:

Let's use **bytecode**!

Java Virtual Machine *(JVM)*

- Source code is compiled to **platform-neutral bytecode**
- Bytecode is executed with **fast** just-in-time compiler
- High-order, **simple** yet powerful architecture
- **Stack-based**, supports hierarchical object **types**
- Not limited to Java!

  (Scala, Groovy, Jython, Kawa, Clojure, ...)

Bytecode (cont'd)
Some basic bytecode instructions

Stack ↔ Local variables

iconst 1	pushes `int` 1 onto operand stack
aload 5	pushes **object** in local variable 5 onto stack *(object* ***type*** *is deduced when class is loaded)*
dstore 6	pops two-word `double` to local variables 6–7

Bytecode (cont'd)
Some basic bytecode instructions

Stack ↔ Local variables

iconst 1 pushes `int` 1 onto operand stack
aload 5 pushes **object** in local variable 5 onto stack
*(object **type** is deduced when class is loaded)*
dstore 6 pops two-word `double` to local variables 6–7

Arithmetic instructions *(affect operand stack)*

imul pops two `int`s from stack, pushes multiplication result

Bytecode (cont'd)
Some basic bytecode instructions

Stack $\leftrightarrow$ Local variables

iconst 1	pushes `int` 1 onto operand stack
aload 5	pushes **object** in local variable 5 onto stack
	*(object **type** is deduced when class is loaded)*
dstore 6	pops two-word `double` to local variables 6–7

Arithmetic instructions *(affect operand stack)*

imul pops two `int`s from stack, pushes multiplication result

Control flow *(uses operand stack)*

ifle +13	pops `int`, jumps +13 bytes if value $\leqslant 0$
lreturn	pops two-word `long`, returns to caller's stack

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**
(e.g., can't multiply an `int` by an **Object***)*
Local variables use is **type-consistent**
(e.g., can't read an `int` after storing an **Object***)*
No stack underflow
No reading from uninitialized variables

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

> **Correct bytecode requirements**
>
> Stack use is **type-consistent**
>
> *(e.g., can't multiply an* `int` *by an* **Object***)*
>
> Local variables use is **type-consistent**
>
> *(e.g., can't read an* `int` *after storing an* **Object***)*
>
> No stack underflow
>
> No reading from uninitialized variables

- So, genetic operators are still delicate

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

> **Correct bytecode requirements**
>
> Stack use is **type-consistent**
>   *(e.g., can't multiply an* `int` *by an* **Object***)*
>
> Local variables use is **type-consistent**
>   *(e.g., can't read an* `int` *after storing an* **Object***)*
>
> No stack underflow
>
> No reading from uninitialized variables

- So, genetic operators are still delicate
- Need **good** genetic operators to produce **correct** offspring

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**
(e.g., can't multiply an `int` *by an* **Object***)*
Local variables use is **type-consistent**
(e.g., can't read an `int` *after storing an* **Object***)*
No stack underflow
No reading from uninitialized variables

- So, genetic operators are still delicate
- Need **good** genetic operators to produce **correct** offspring
- Conclusion: Avoid **bad** crossover and mutation

Compatible Crossover
Constraints of unidirectional crossover $A \overleftarrow{\times} B$

Good crossover is achieved by respecting bytecode constraints:
(α *is target section in* ***A***, β *is source section in* ***B***)

Operand stack

e.g., β doesn't pop values with types incompatible to those popped by α

Compatible Crossover
Constraints of unidirectional crossover $\mathsf{A} \overleftarrow{\times} \mathsf{B}$

Good crossover is achieved by respecting bytecode constraints:
(α *is target section in* ***A***, β *is source section in* ***B***)

Operand stack

e.g., β doesn't pop values with types incompatible to those popped by α

Local variables

e.g., variables read by β in **B** must be written before α in **A** with compatible types

Compatible Crossover
Constraints of unidirectional crossover $\mathbf{A} \overleftarrow{\times} \mathbf{B}$

Good crossover is achieved by respecting bytecode constraints:
(α *is target section in* ***A***, β *is source section in* ***B***)

Operand stack

e.g., β doesn't pop values with types incompatible to those popped by α

Local variables

e.g., variables read by β in **B** must be written before α in **A** with compatible types

Control flow

e.g., branch instructions in β have no "outside" destinations

Formal Definition
(Example of operand stack requirement)

α and β have compatible stack frames up to stack depth of β: pops of α have identical or narrower types as pops of β; pushes of β have identical or narrower types as pushes of α

Good crossover

	α	β
pre-stack	`**AB`	`**AA`
post-stack	`**B`	`**C`
depth	3	2

Stack pops "`AB`" *(2 stop tack frames)* are narrower than "`AA`", whereas stack push "`C`" is narrower than "`B`"

Types hierarchy: `C` $\rightarrow$ `B` $\rightarrow$ `A`

(see [Orlov and Sipper, 2009, 2011] for full formal definitions)

Formal Definition
(Example of operand stack requirement)

α and β have compatible stack frames up to stack depth of β:
pops of α have identical or narrower types as pops of β;
pushes of β have identical or narrower types as pushes of α

Bad crossover

	α	β
pre-stack	`**AB`	`**Af`
post-stack	`**B`	`**A`
depth	3	2

Stack pops “`AB`” are not narrower than “`Af`”
(B and f are incompatible);
stack push “`A`” is not narrower than “`B`”

Types hierarchy: `B` $\rightarrow$ `A`; `f` is a `float`

(see [Orlov and Sipper, 2009, 2011] for full formal definitions)

Symbolic Regression
As an evolutionary example...

Parameters

- Objective: symbolic regression, $x^4 + x^3 + x^2 + x$
- Fitness: sum of errors on 20 random data points in $[-1, 1]$
- Input: **Number** `num` *(a Java type)*

Symbolic Regression
As an evolutionary example...

Parameters

- Objective: symbolic regression, $x^4 + x^3 + x^2 + x$
- Fitness: sum of errors on 20 random data points in $[-1, 1]$
- Input: **Number** `num` *(a Java type)*

Seeding

- Population initialized using seeding

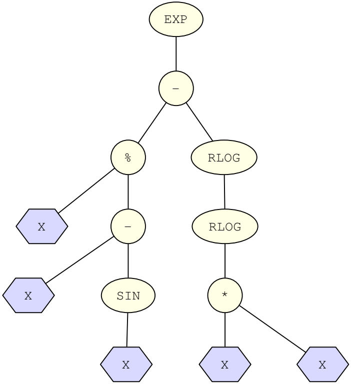
  [Langdon and Nordin, 2000]
- Seed population with clones of Koza's original worst-of-generation-0

  [Koza, 1992]

Symbolic Regression
Seeding with Koza's worst-of-generation-0

Original **Lisp** individual and its **tree** representation:

```
(EXP (- (% X (- X (SIN X))) (RLOG (RLOG (* X
X)))))
```



Symbolic Regression
Seeding with Koza's worst-of-generation-0

Translation to **unrestricted Java**

```java
class SimpleSymbolicRegression {
  Number simpleRegression(Number num) {
    double x     = num.doubleValue();
    double llsq  = Math.log(Math.log(x*x));
    double dv    = x / (x - Math.sin(x));
    double worst = Math.exp(dv - llsq);
    return Double.valueOf(worst + Math.cos(1));
  }

  /* Rest of class omitted */
}
```

We added a couple of building blocks in the last line

Symbolic Regression
Setup and Statistics

Setup *(similar to Koza's)*

- Population: 500 individuals
- Generations: 51 (or less)
- Probabilities: $p_{\text{cross}} = 0.9$
  (α and β segments are uniform over segment sizes)
- Selection: binary tournament

Symbolic Regression
Setup and Statistics

Setup *(similar to Koza's)*

- Population: 500 individuals
- Generations: 51 (or less)
- Probabilities: $p_{\text{cross}} = 0.9$
  (α and β segments are uniform over segment sizes)
- Selection: binary tournament

Statistics

- Yield: 99% of runs successful (out of 100)
- Runtime: 30–60 s on dual-core 2.6 GHz Opteron
- Memory limits: insignificant w.r.t. runtime

Symbolic Regression
Evolved perfect individuals

A perfect solution easily evolves:

(beware of decompiler quirks!)

```
class SimpleSymbolicRegression_0_7199 {
  Number simpleRegression(Number num) {
    double d = num.doubleValue();
    d = num.doubleValue();
    double d1 = d; d = Double.valueOf(d + d * d *
          num.doubleValue()).doubleValue();
    return Double.valueOf(d +
          (d = num.doubleValue()) * num.doubleValue());
  }

  /* Rest of class unchanged */
}
```

Computes $(x + x \cdot x \cdot x) + (x + x \cdot x \cdot x) \cdot x \;=\; x(1 + x)(1 + x^2)$

Symbolic Regression
Evolved perfect individuals

Another solution:

```
class SimpleSymbolicRegression_0_2720 {
  Number simpleRegression(Number num) {
    double d = num.doubleValue();
    d = d; d = d;
    double d1 = Math.exp(d - d);
    return Double.valueOf(num.doubleValue() *
      (num.doubleValue() * (d * d + d) + d) + d);
  }

  /* Rest of class unchanged */
}
```

Computes $x \cdot (x \cdot (x \cdot x + x) + x) + x \;=\; x(1 + x(1 + x(1 + x)))$

Complex Regression

Parameters

- Objective: symbolic regression: $x^9 + x^8 + \cdots + x^2 + x$
- Fitness: incremental evaluation, $\sum_{i=1}^{n} x^i$, up to $n = 9$
- Crossover: Gaussian distribution over segment sizes
- Parsimony pressure, growth limit

Initialization

- Worst of generation-0 from simple regression

Java Wilderness
Complex Regression

A perfect solution:

```
Number simpleRegression(Number num) {
  double d = num.doubleValue();
  return Double.valueOf(d + (d * (d * (d +
      ((d = num.doubleValue()) +
        (((num.doubleValue() * (d = d) + d) *
          d + d) * d + d) * d)
      * d) + d) + d) * d);
}
```

Computes

$$x+(x\cdot(x\cdot(x+(x+(((x\cdot x+x)\cdot x+x)\cdot x+x)\cdot x)\cdot x)+x)+x)\cdot x$$

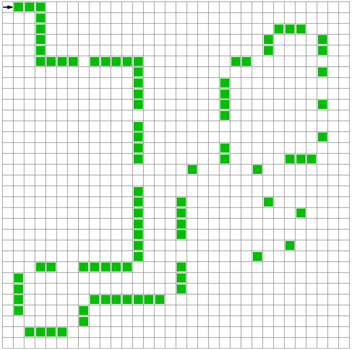
Parameters

- Objective: consume all food pellets on Santa Fe trail
- Fitness: number of food pellets consumed
- Crossover: Gaussian distribution over segment sizes
- Parsimony pressure, growth limit

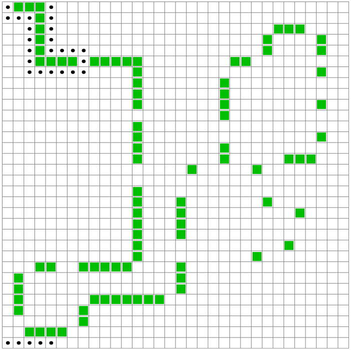
Initialization

- "Avoider" (zero-fitness)

Santa Fe Trail:



(a) Initial setup

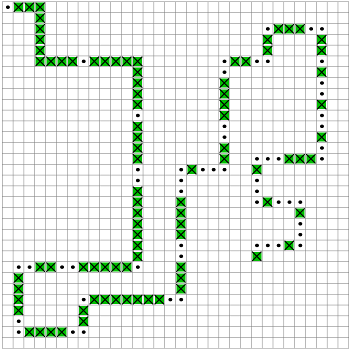


(b) Avoider

Java Wilderness
Artificial Ant

A perfect solution:

```
void step() {
  if (foodAhead()) {
    move(); right();
  }
  else {
    right(); right();
    if (foodAhead())
      left();
    else {
      right(); move();
      left();
    }
    left(); left();
  }
}
```



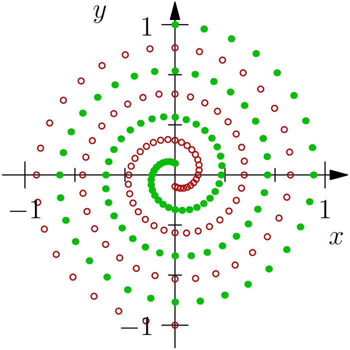
Parameters

- Objective: two-class classification of intertwined spirals
- Fitness: number of correctly classified points

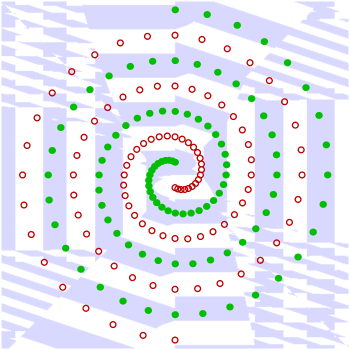
Initialization

- Arbitrarily organized repository of building blocks: floating-point arithmetics, trigonometric functions, and polar-rectangular coordinates conversion

Intertwined spirals:



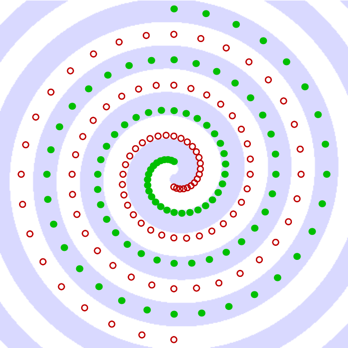
(e) Initial setup



(f) Koza's evolved solution

Java Wilderness
Intertwined Spirals

A perfect solution:



Computes the (approximate) sign of $\sin\left(\frac{9}{4}\pi^2\sqrt{x^2+y^2} - \tan^{-1}\frac{y}{x}\right)$ as the class predictor of (x, y)

Java Wilderness
Intertwined Spirals

Koza's best-of-run:

```
(sin (iflte (iflte (+ Y Y) (+ X Y) (- X Y) (+ Y Y)) (* X X)
(sin (iflte (% Y Y) (% (sin (sin (% Y 0.30400002))) X) (% Y
0.30400002) (iflte (iflte (% (sin (% (% Y (+ X Y))
0.30400002)) (+ X Y)) (% X -0.10399997) (- X Y) (* (+
-0.12499994 -0.15999997) (- X Y))) 0.30400002 (sin (sin
(iflte (% (sin (% (% Y 0.30400002) 0.30400002)) (+ X Y))
(% (sin Y) Y) (sin (sin (sin (% (sin X) (+ -0.12499994
-0.15999997))))) (% (+ (+ X Y) (+ Y Y)) 0.30400002))))
(+ (+ X Y) (+ Y Y))))) (sin (iflte (iflte Y (+ X Y) (- X Y)
(+ Y Y)) (* X X) (sin (iflte (% Y Y) (% (sin (sin (% Y
0.30400002))) X) (% Y 0.30400002) (sin (sin (iflte (iflte
(sin (% (sin X) (+ -0.12499994 -0.15999997))) (% X
-0.10399997) (- X Y) (+ X Y)) (sin (% (sin X) (+
-0.12499994 -0.15999997))) (sin (sin (% (sin X) (+
-0.12499994 -0.15999997)))) (+ (+ X Y) (+ Y Y))))))) (%
Y 0.30400002)))))
```

Java Wilderness
Intertwined Spirals

Our best-of-run:

```
boolean isFirst(double x, double y) {
  double a, b, c, e;
  a = Math.hypot(x, y);  e = y;
  c = Math.atan2(y, b = x) +
    -(b = Math.atan2(a, -a))
    * (c = a + a) * (b + (c = b));
  e = -b * Math.sin(c);
  if (e < -0.0056126487018762772) {
    b = Math.atan2(a, -a);
    b = Math.atan2(a * c + b, x);  b = x;
    return false;
  }
  else
    return true;
}
```

Java Wilderness
Array Sum

Parameters

- Objective: summation of numbers in an input array
- Fitness: differences from actual sums on test inputs
- **Time limit**: 5000 backward branches

Code instrumentation

- Bytecode is instrumented with calls to time-limit check
- Before each backward branch and method invocation
- Robust and portable technique

Initialization

- "Weird" program that does **not** compute the sum

Array sum: array loop solution

```java
int sumlist(int list[]) {
  int sum  = 0;
  int size = list.length;
  for (int tmp = 0; tmp < list.length; tmp++) {
    size = tmp;
    sum = sum - (0 - list[tmp]);
  }
  return sum;
}
```

Array sum: List loop solution

```java
int sumlist(List list) {
  int sum = 0;
  int size = list.size();
  for (Iterator iterator = list.iterator();
                         iterator.hasNext(); ) {
    int tmp = ((Integer) iterator.next())
                        .intValue();
    tmp = tmp + sum;
    if (tmp == list.size() + sum)
      sum = tmp;
    sum = tmp;
  }
  return sum;
}
```

Java Wilderness
Array Sum

Array sum: List-recursive solution

```
int sumlistrec(List list) {
  int sum = 0;
  if (list.isEmpty())
    sum = sum;
  else
    sum += ((Integer)list.get(0)).intValue() +
        sumlistrec(list.subList(1, list.size()));
  return sum;
}
```

Parameters

- Objective: learn to play Tic-Tac-Toe
- Fitness: rounds won in single-elimination tournament

Initialization

- Negamax algorithm with α-β pruning and one of four (plausibly) insidious imperfections

Performance

- All imperfections are easily swept away (with interesting quirks!)

The Tic-Tac-Toe seed:

```java
int negamaxAB(TicTacToeBoard board,
      int alpha, int beta, boolean save) {
  Position[] free = getFreeCells(board);
  // utility is derived from the number of free cells left
  if (board.getWinner() != null)
    alpha = utility(board, free);
  else if (free.length == 0)
    alpha = 0   save = false ;
  else for (Position move: free) {
    TicTacToeBoard copy = board.clone();
    copy.play(move.row(), move.col(),
                copy.getTurn());
    int utility = -  (removed) negamaxAB(copy,
                -beta, -alpha, false  save );
    if (utility > alpha) {
      alpha = utility;
      if (save)
        // save the move into a class instance field
        chosenMove = move;
      if ( alpha >= beta  beta >= alpha )
        break;
    }
  }
  return alpha;
}
```

Conclusions

Completely **unrestricted** Java programs can be **evolved**
(via bytecode)

Loops and recursion are not a problem!

Extant (bad) Java programs can be **improved**

Future Work

- Actively searching for consistent bytecode segments during compatibility checks
- Class-level evolution: cross-method crossover, introduction of new methods
- Development of mutation operators
- Applying FINCH to additional hard problems
- Designing an IDE plugin to leverage FINCH for **software engineers**
- Applying FINCH to meta-evolution
- Automatic improvement of existing applications: the realm of **extant software**

Evolving Game Heuristics

- A row of cells: k black pieces, empty cell, k white pieces.
- Pieces move towards the opposite direction, striving to reverse the initial board situation.
- Pieces can move one step towards the opposite direction, or jump over one complementary-color piece.
- FINCH successfully evolved a `getMove` method, solving the problem consistently and effortlessly.
- Additionally, we had significant progress evolving heuristic evaluation functions for the game of *Connect Four*.

Evolving Game Heuristics

Evolved program solving the *k*-empty-*k* game:

```
Move getMove(Board board) {
    int i = board.findEmpty();
    Piece left1  = board.getPlace(i-1);
    Piece left2  = board.getPlace(i-2);
    Piece right1 = board.getPlace(i+1);
    Piece right2 = board.getPlace(i+2);
    if (left1  == Piece.BLACK && left2 == Piece.WHITE)
        return Move.RIGHT;
    if (right2 == Piece.BLACK)
        return Move.LEFT;
    if (left1  == right2)
        return Move.RIGHT;
    if (right1 == Piece.BLACK)
        return Move.LEFT;
    return Move.RIGHT;
}
```

References

M. Orlov and M. Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In G. Raidl et al., editors, *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, July 8–12, 2009, Montréal Québec, Canada*, pages 1043–1050, New York, NY, USA, July 2009. ACM Press. ISBN 978-1-60558-325-9. doi:10.1145/1569901.1570042.

M. Orlov and M. Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, Apr. 2011. doi:10.1109/TEVC.2010.2052622.

M. Orlov, C. Bregman, and M. Sipper. Automatic evolution of Java-written game heuristics. In M. B. Cohen and M. O. Cinnéide, editors, *Search Based Software Engineering: Proceedings of the Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10–12, 2011*, volume 6956 of *Lecture Notes in Computer Science*, page 277, Berlin Heidelberg, sep 2011. Springer-Verlag. ISBN 978-3-642-23715-7. doi:10.1007/978-3-642-23716-4_30.