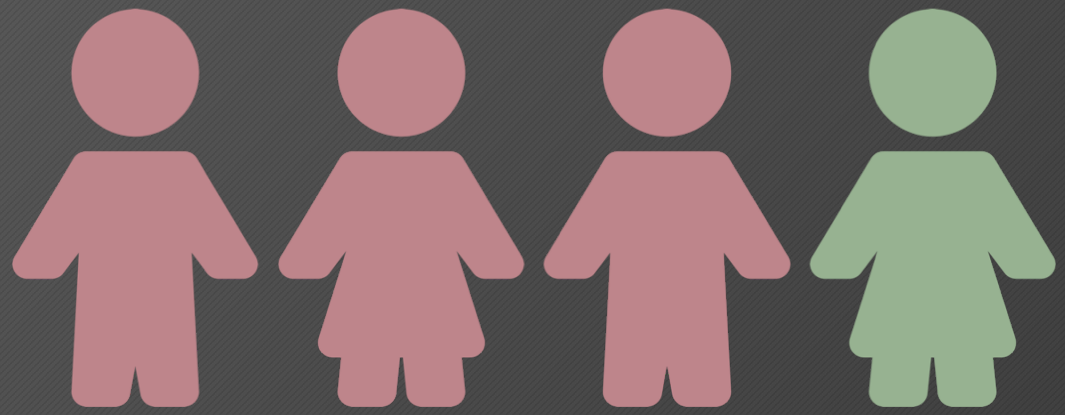
Foundational Analysis Techniques for High-Level Transformation Programs

PhD Candidate: Ahmad Salim Al-Sibahi

Supervisors: Andrzej Wąsowski &

Aleksandar S. Dimovski

Worrisome Fact about Transformations

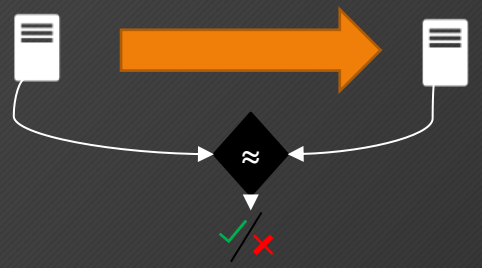


Refactoring comes with a risk of introducing **subtle bugs** and **functionality regression**.

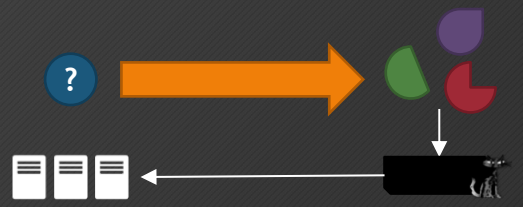
77%, from Kim, M., Zimmermann, T., & Nagappan, N. (2012). A field study of refactoring challenges and benefits. FSE ‘12

Key Contributions

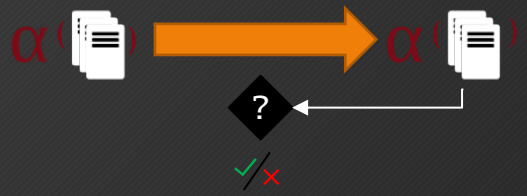
Translation Validation



Symbolic Execution



Static Analysis



High-level Transformations

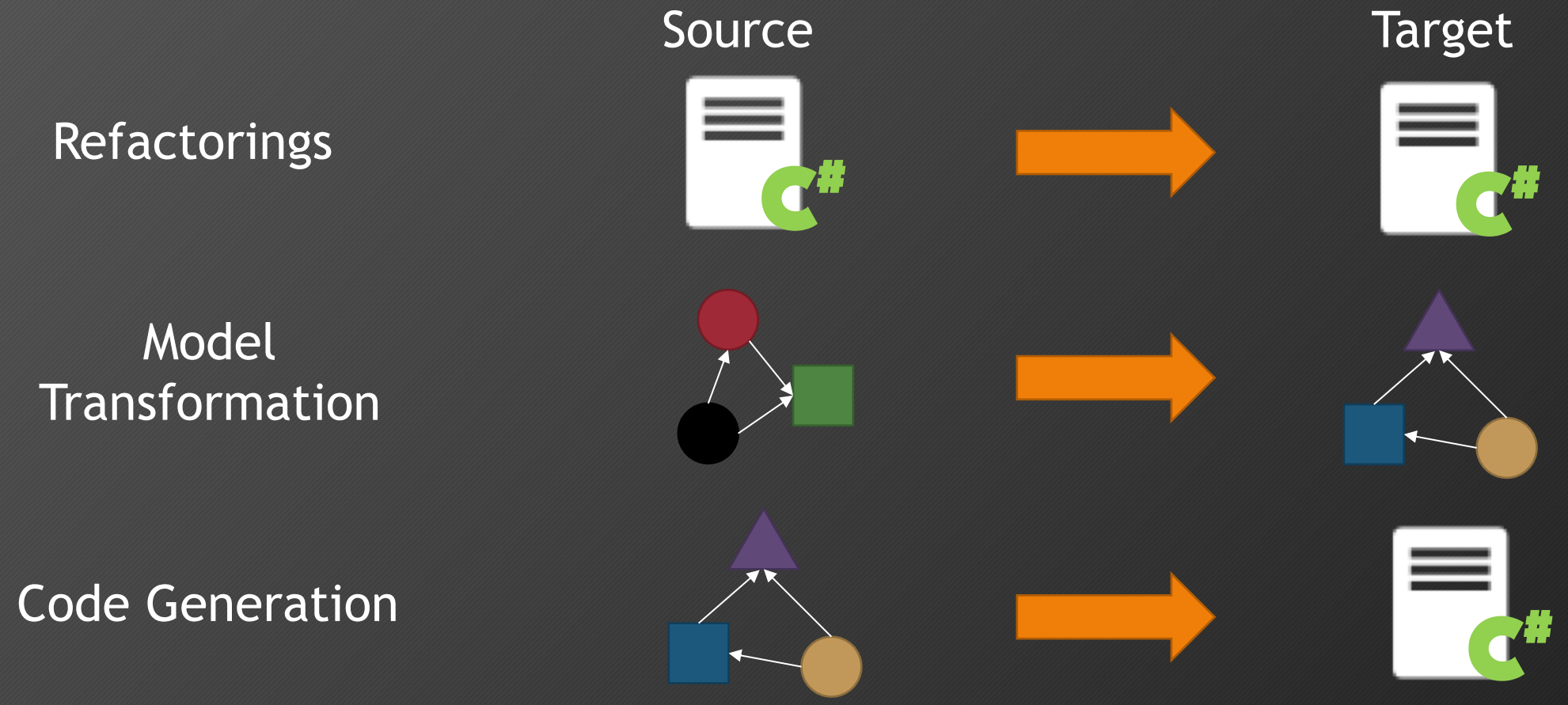
What do I mean by a high-level transformation?


```
class Account {
  Id id;
  Money credit;

  membershipLevel() {
    return max(100,
               this.credit / 100);
  }
  eq(Account o) {
   return
     this.credit == o.credit &&
     this.id == o.id;
  }
}
```

rename credit **to** balance **in** Account

```
class Account {
  Id id;
  Money balance;

  membershipLevel() {
    return max(100,
               this.balance / 100);
  }
  eq(Account o) {
   return
     this.balance == o.balance &&
     this.id == o.id;
  }
}
```

What do I mean by a high-level transformation?



What is a high-level transformation language?

Model Transformation Languages

Graph Rewriting

Rule-based Model Transformation

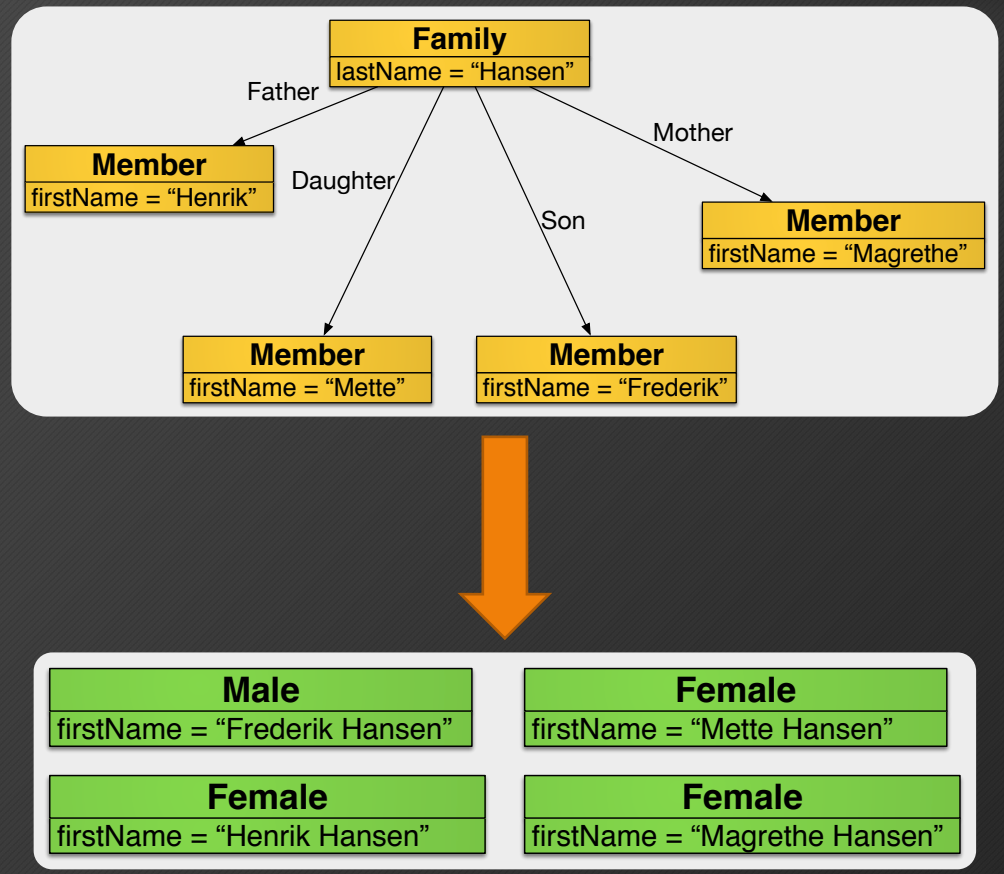
Program Transformation Languages

What is a high-level transformation language?

- Constructs for *traversing* and manipulating structures
- Expressive *pattern matching* and *querying* operations
- First-class *collections* and collection operations

Declarative transformation languages


```
rule Member2Female
transform  member : Families!Member
to person : Persons!Female
{
   guard: member.isFemale()
   person.fullName =
     member.firstName +" "+
          member.familyName();
 }
```



epsilon

Program transformation languages


```
data Nat = zero() | suc(Nat pred);
data Expr = var(str nm) | cst(Nat vl) |
            mult(Expr el, Expr er);

Expr simplify(Expr expr) =
  bottom-up visit (expr) {
      case mult(cst(zero()), y) => cst(zero())
      case mult(x, cst(zero())) => cst(zero())
      case mult(cst(suc(zero())), y) => y
      case mult(x, cst(suc(zero()))) => x
  };
```

1 * (x * 10 * 1) * y

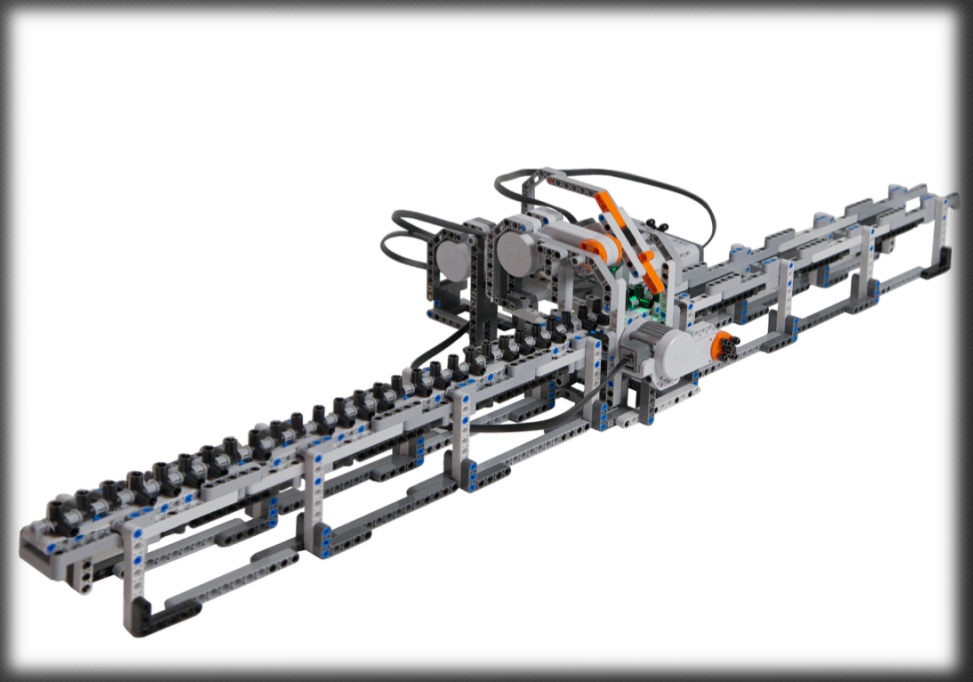
x * 10 * y

1 * ((3 * 0) * z)

0

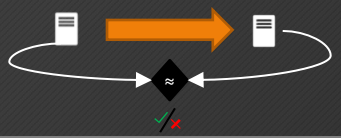
Establishing of Model Transformation Languages

- Popular graph and rule-based model transformation languages are Turing-complete
- Traditional programming language verification techniques needed!



LEGO Turing-machine © CWI Amsterdam

Translation Validation

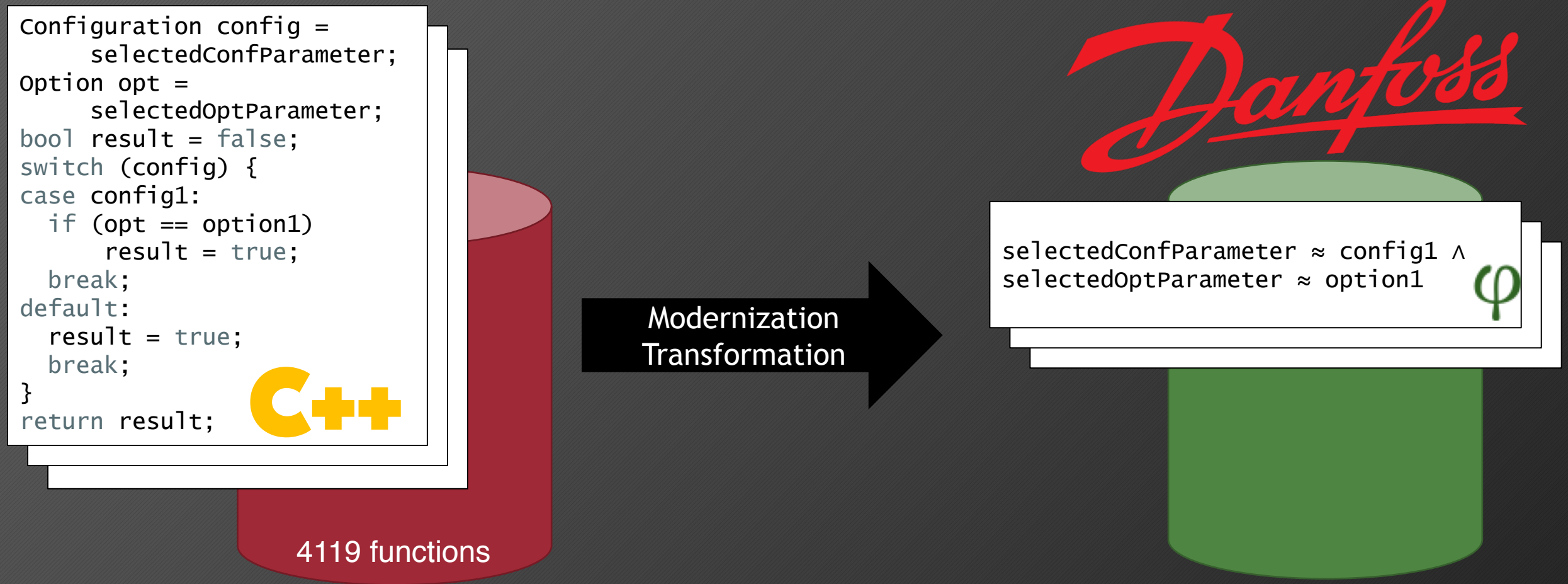


Symbolic Execution

Static Analysis

Validating an Industrial Software Modernization Transformation

Based on Iosif-Lazăr, A. F., Al-Sibahi, A. S., Dimovski, A. S., Savolainen, J. E., Sierszecki, K. & Wąsowski, A. (2015). Experiences from designing and validating a software modernization transformation. ASE ‘15.



Modernizing an Industrial Configuration Tool

Transforming Danfoss' imperative code base for configuring frequency converters to pure logical formulae compatible with off-the-shelf constraint solvers

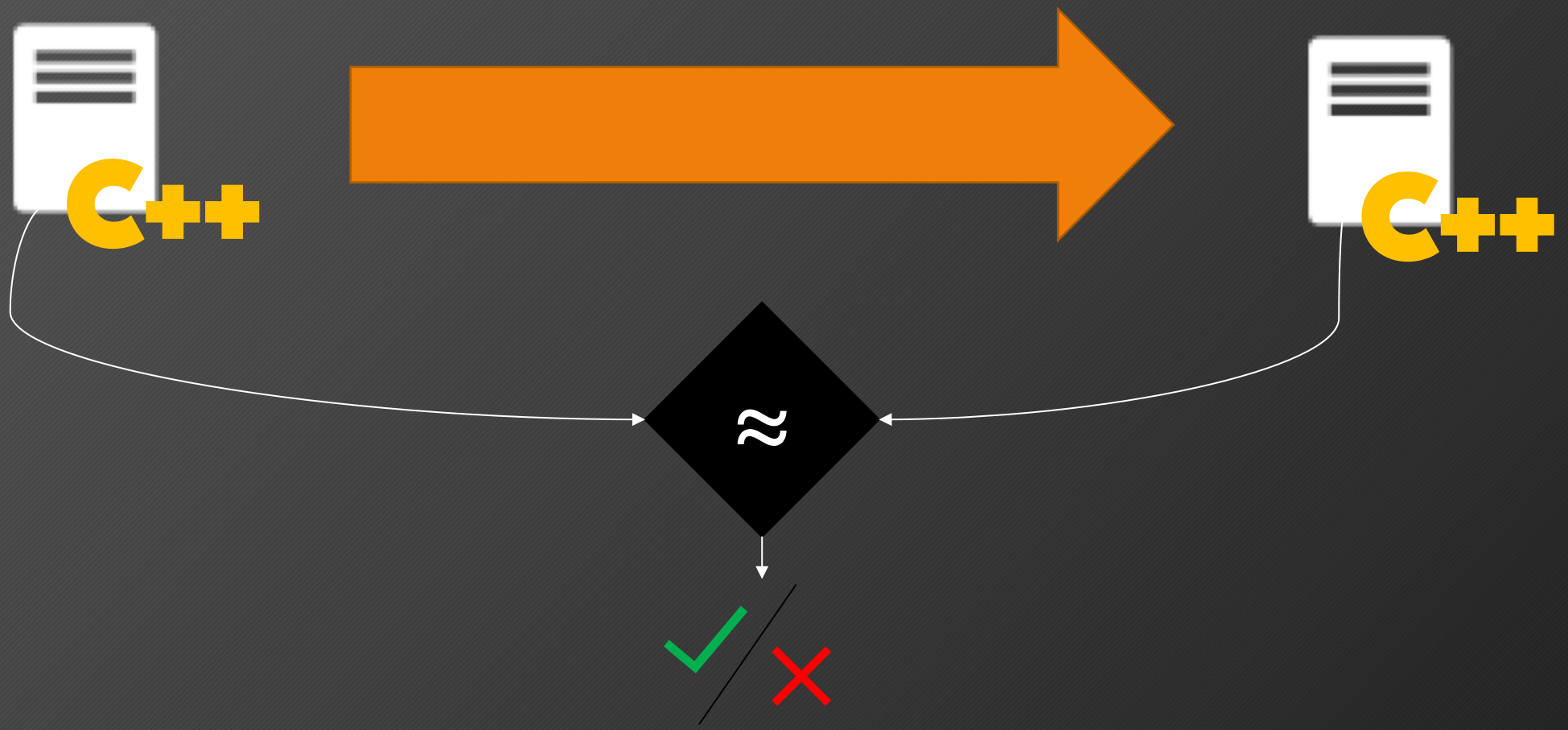
Syntactic Transformation using TXL

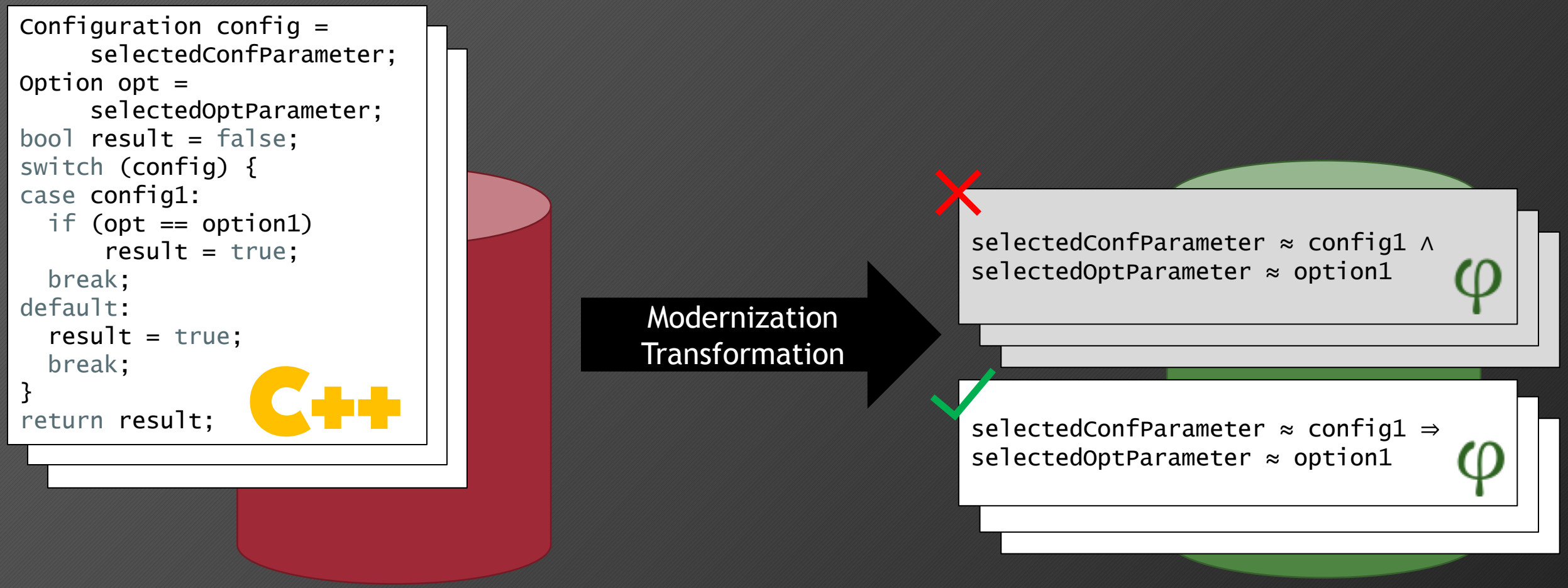
468 rule definitions handling:

- Preprocessor directives
- Inlinining variables
- Converting and simplifying **switch**’s and **if**’s to ternary expressions

```
rule convert_simple_sel_stmt
replace [selection_statement]
  'if '( EXP [expression] ')
        STMT [statement]
where not STMT [contains_selection_stmt]
where not STMT [is_compound_stmt]
construct TRUE_STMT [true_case_statement]
  'TRUE ';
by
 '( EXP ') '? '( STMT ') ': '(TRUE_STMT ')
end rule
```

Checking Correctness of Modernization using Translation Validation





Validating the Industrial Configuration Tool

Discovering a bug in the modernization transformation

“

Correct programs are all alike; every buggy program is buggy in its own way.

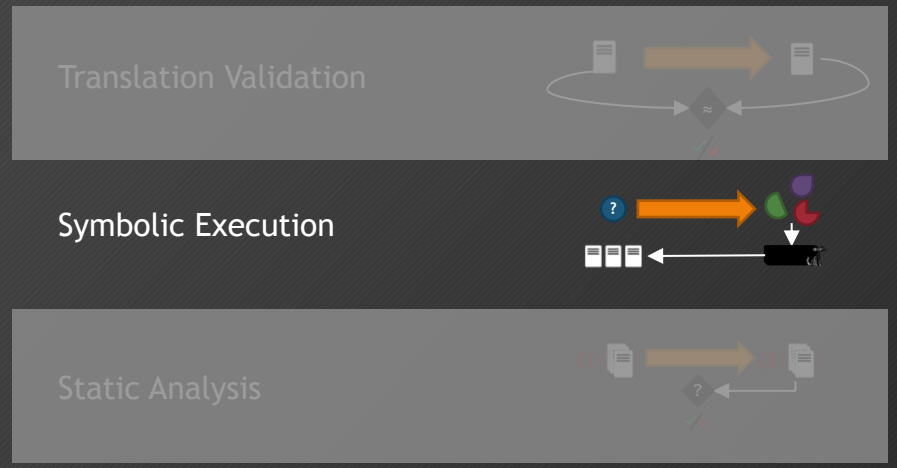
”

Anna Karenina principle as applied to program correctness

Qualitative Understanding of Transformation Bugs

50 bug cases out of 4491 functions

Simple Syntactic	Relational Syntactic	Semantic
• All negations dropped • Structure replaced by a constant integer • Unexpected exceptions in output expression	• Some function calls dropped • Some conditional branches dropped • Conditionals with error code assignments dropped	• Use of undeclared variables • Variable declarations without assignment not handled



Effective Test Generation for High-Level Transformation Programs

Based on Al-Sibahi, A. S., Dimovski, A. S., & Wąsowski, A. (2016). Symbolic Execution of High-Level Transformations. SLE '16.

Test case generation for transformations

- Goal: Generate test cases given transformation program
- Rely on definition of transformation program to efficiently cover interesting paths of program

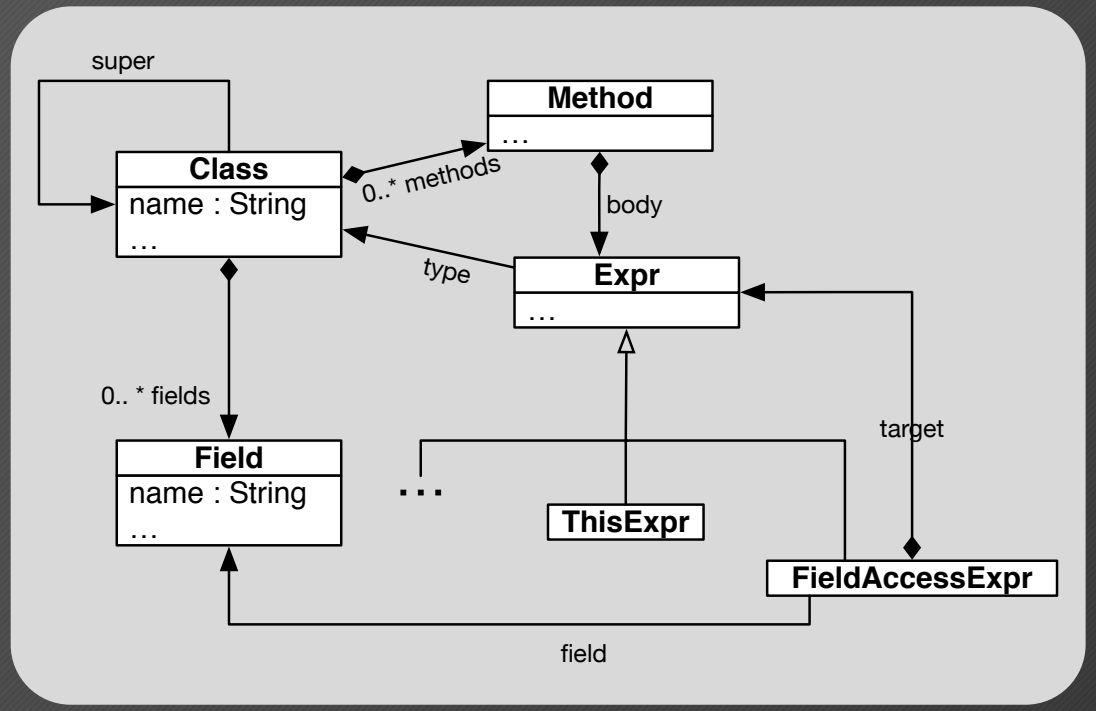
Rename Field Refactoring


```
class Account {
  Id id;
  Money credit;

  membershipLevel() {
    return max(100,
               this.credit / 100);
  }
  eq(Account o) {
   return
     this.credit == o.credit &&
     this.id == o.id;
  }
}
```

rename credit **to** balance **in** Account

```
class Account {
  Id id;
  Money balance;

  membershipLevel() {
    return max(100,
               this.balance / 100);
  }
  eq(Account o) {
   return
     this.balance == o.balance &&
     this.id == o.id;
  }
}
```

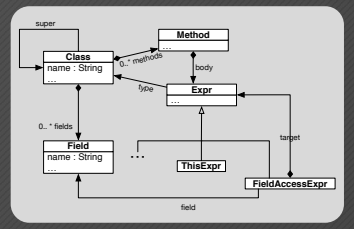
Rename Field Refactoring



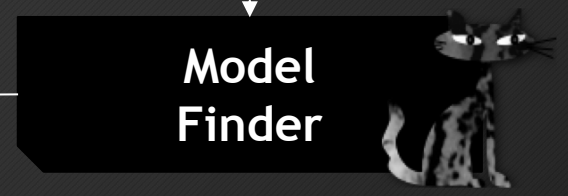

```
target_class.fields :=
  (target_class.fields \ old_field) ∪ new_field
foreach faexpr ∈
  target_class match⋆ FieldAccessExpr do
  if faexpr.field = old_field ∧
       faexpr.target.type = target_class then
    faexpr.field := new_field
  else skip
```

TRON

Test Generation using Symbolic Execution




```
target_class.fields :=
  (target_class.fields \ old_field) ∪ new_field
foreach faexpr ∈
  target_class match* FieldAccessExpr do
  if faexpr.field = old_field ∧
      faexpr.target.type = target_class then
    faexpr.field := new_field
  else skip
```

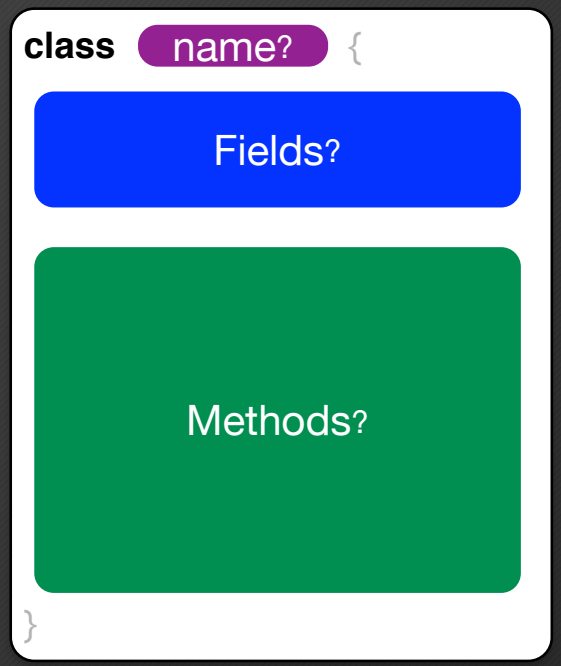


Symbolic Execution

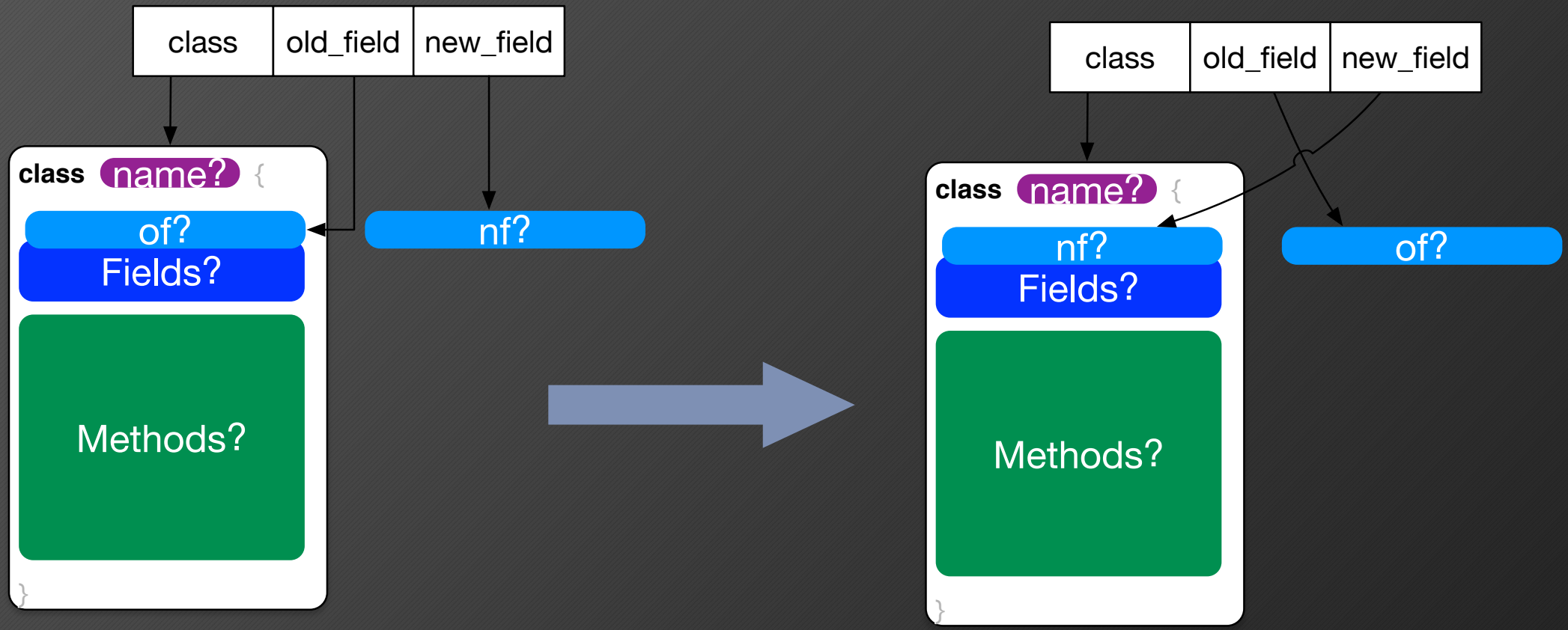
Concrete

```
class Account {
  Id id;
  Money credit;

  membershipLevel() {
    return max(100,
               this.credit / 100);
  }
  eq(Account o) {
   return
     this.credit == o.credit &&
     this.id == o.id;
  }
}
```

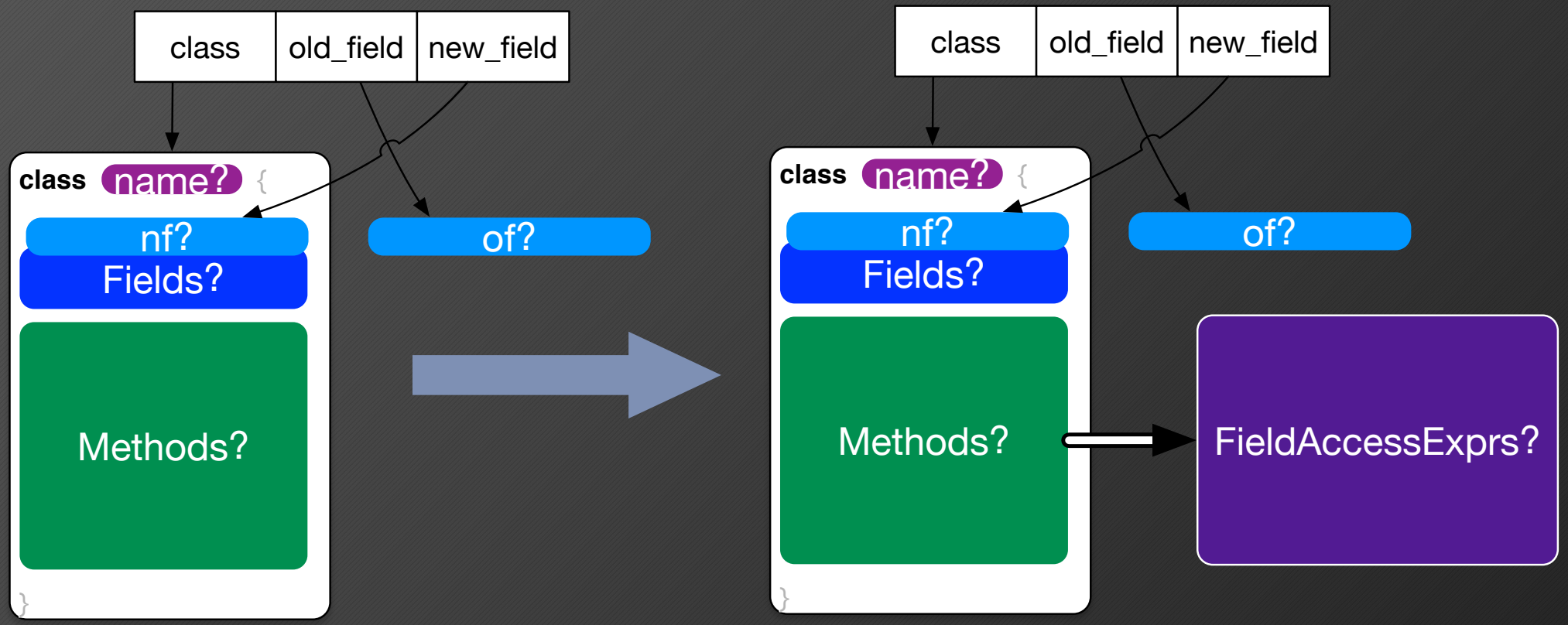
Symbolic



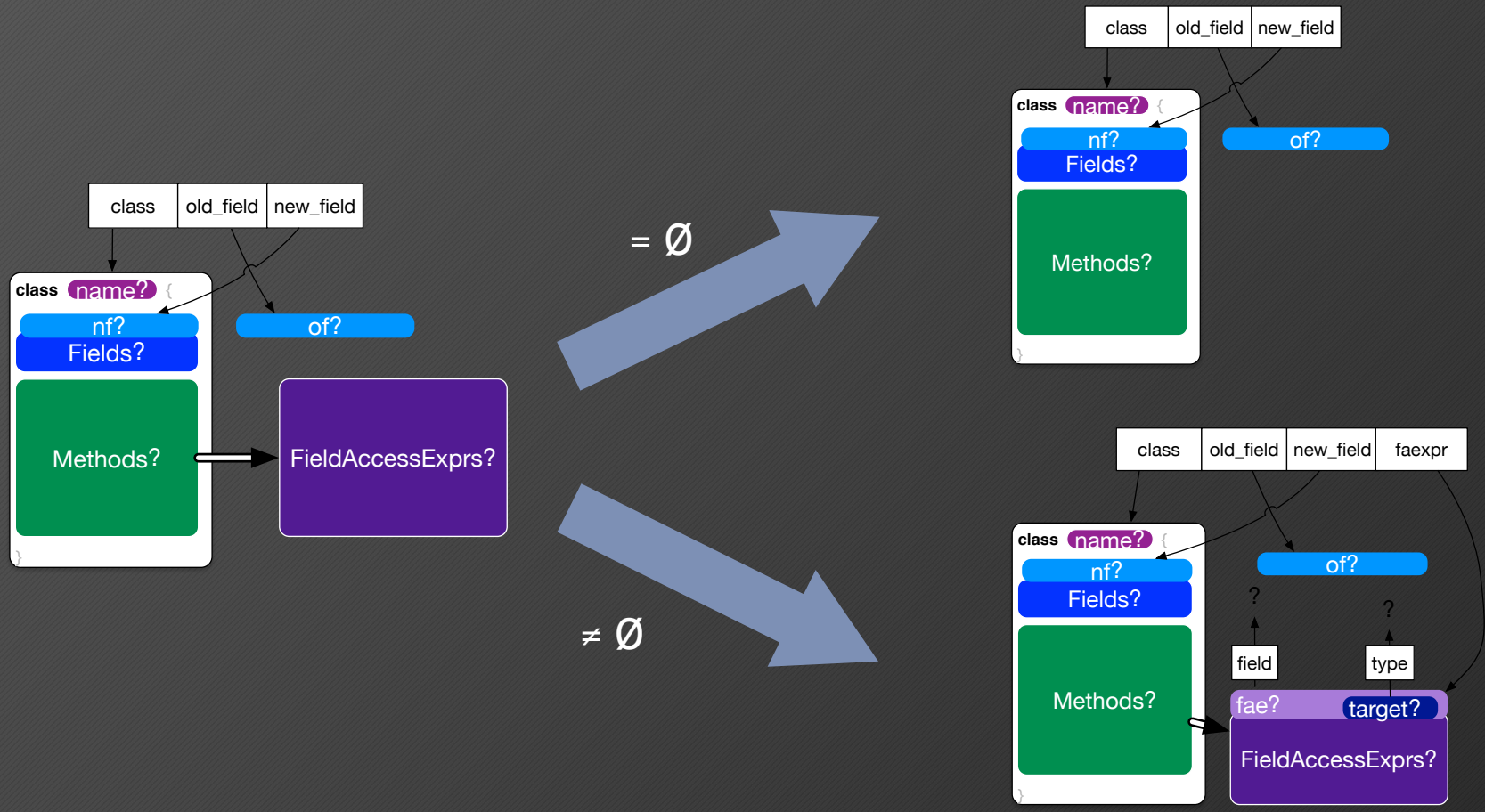
```
class.fields :=
 (class.fields \ old_field) ∪ new_field
foreach faexpr ∈ class match⋆ FieldAccessExpr do
 if (faexpr.field = old_field
        ∧ faexpr.target.type = class) then
    faexpr.field := new_field
 else skip
```



```
class.fields :=
 (class.fields \ old_field) ∪ new_field
foreach faexpr ∈ class match* FieldAccessExpr do
 if (faexpr.field = old_field
        ∧ faexpr.target.type = class) then
    faexpr.field := new_field
 else skip
```



```
class.fields :=
 (class.fields \ old_field) ∪ new_field
foreach faexpr ∈ class match* FieldAccessExpr do
 if (faexpr.field = old_field
        ∧ faexpr.target.type = class) then
    faexpr.field := new_field
 else skip
```

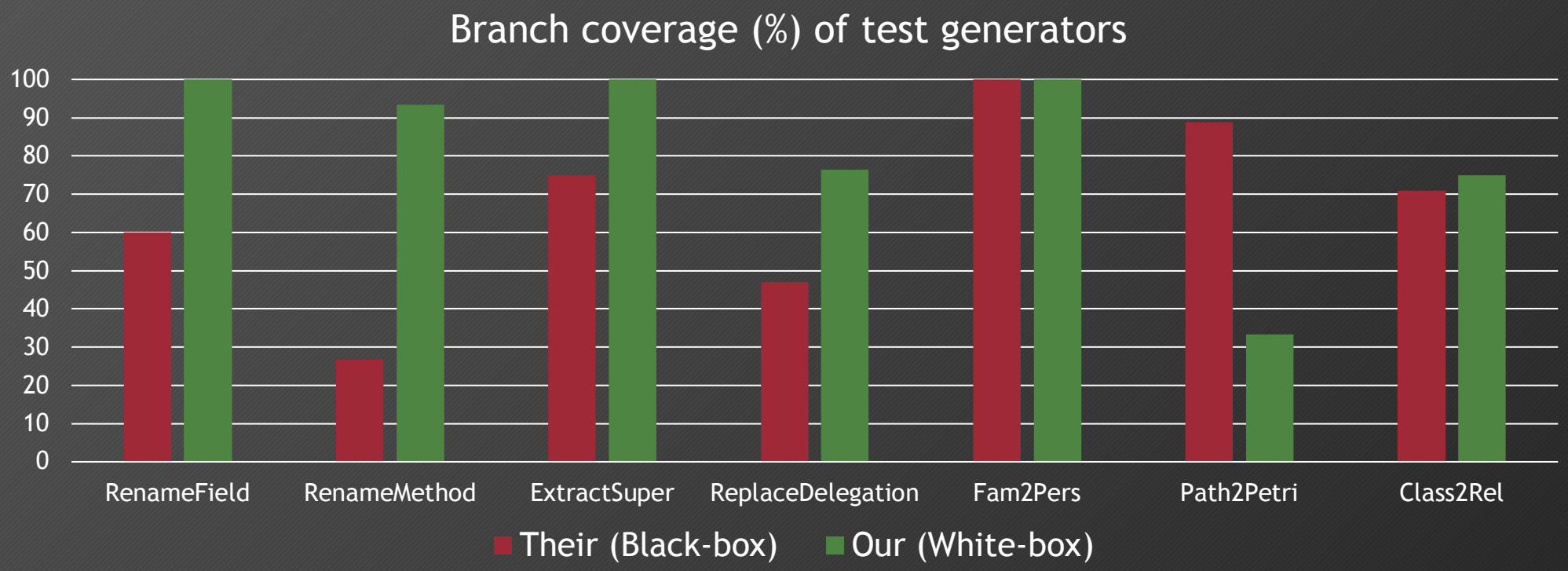


Symbolic Execution Continues…

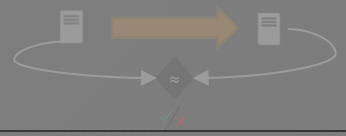
SymexTRON

- Scala-based implementation of symbolic execution for TRON
  - 36 Scala files, ~3,485 SLOC total
- Relies on KodKod model solver and Plingeling SAT solver
- Artifact Evaluated and Proudly Open Source
  - http://itu-square.github.io/SymexTRON/

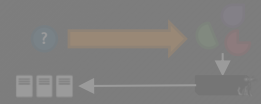
Test Generation Results



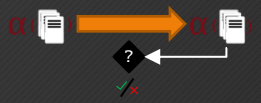
Translation Validation



Symbolic Execution



Static Analysis



Verifying type and shape properties for Rascal

What is Rascal?

- High-level Language for Analysing and Transforming Programs
- Popular in the Software Language Engineering Community
- Developed at CWI Amsterdam by the SWAT team

What is Rascal?

- Full programming language with algebraic datatypes, functions with case analysis, imperative variables, various loops (**for**, **while**, **solve**) with control flow (**break**, **continue**), and exceptions
- Generic traversals using a wide range of strategies (**bottom-up**, **top-down**, **innermost**, **outermost**, **bottom-up-break**, **top-down-break**)
- Expressive pattern matching constructs, including collection patterns, non-linear patterns, negated patterns, and deep matching patterns

What is Rascal?


```
data Config =
    flat(str option, str val)
  | nested(str group, list[Config] subconfigs);

Config deduplicate(Config config) =
 innermost visit(config) {
    case [*xs, x, *zs, x, *ys] =>[*xs, x, *zs, *ys]
 };
```

Rascal Light, a Formal Subset of Rascal

Based on Al-Sibahi, A. S. (2017). The Formal Semantics of Rascal Light. arXiv CoRR, abs/1703.92312.

Rascal Light

- Fully-formalized subset of Rascal
- Captures key features like Traversals and Pattern Matching
- Ideal for developing formal verification techniques

Rascal Light

Includes ✚

- Large subset of expression language
  - Case analysis, Variables, Exceptions, and Loops with control flow operators
- Traversals including all strategies
- Expressive pattern matching operations including backtracking

Excludes ➗

- Concrete syntax support, string interpolation, and regular expressions
- Standard Library, Input/Output, and FFI
- Module system and extensibility
- Advanced type system features like polymorphism and inheritance

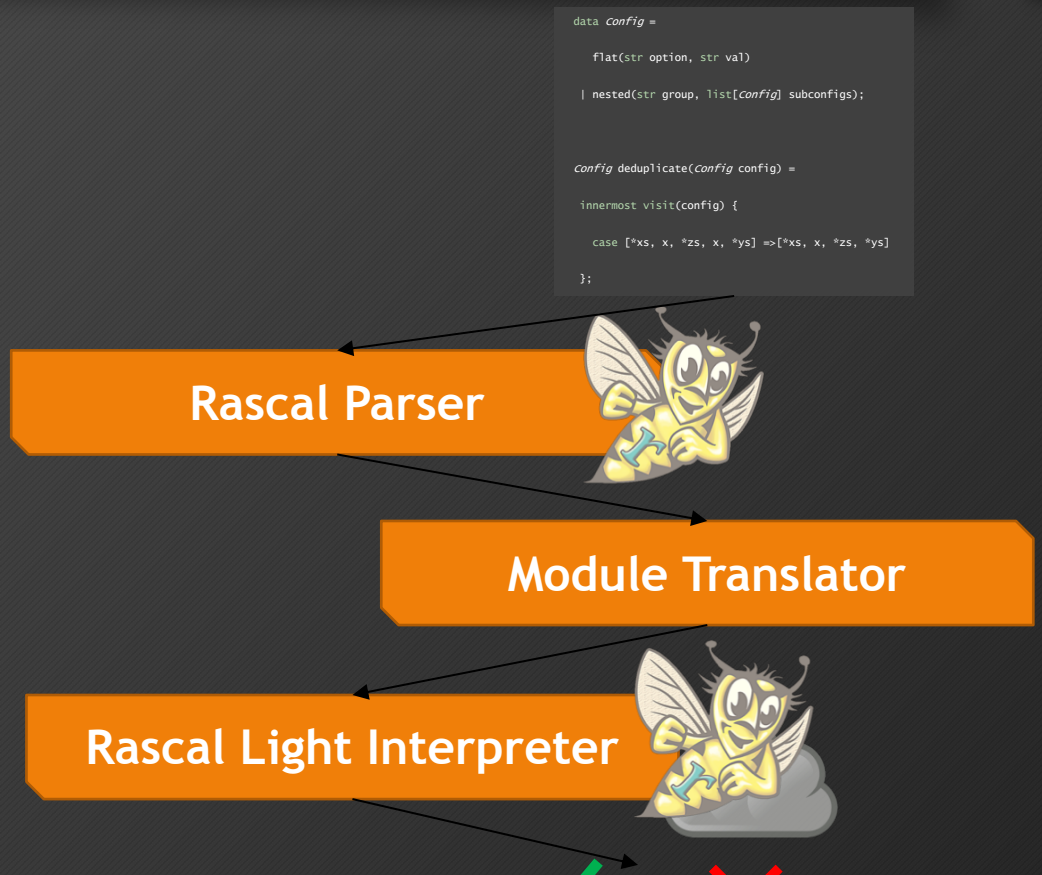
Method of Formalising Rascal Light

- Develop an Operational Semantics for Rascal Light based on:
  - Rascal documentation
  - Implementation of micro-Rascal
  - Correspondence with Rascal developers (esp. Paul Klint)
- Checked using prototype implementation and proofs of target theorems

$$\textsc{E-Visit-Sucs}\ \frac{e;\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v;\sigma'' \qquad \underline{cs};v;\sigma'' \underset{\text{visit}}{\overset{st}{\Longrightarrow}} \textbf{success}\ v';\sigma'}{st\ \textbf{visit}\ e\ \underline{cs};\sigma \underset{\text{expr}}{\Longrightarrow} \textbf{success}\ v';\sigma'}$$

Rascal Light

- Prototype Interpreter implemented in Scala
- Closely corresponds to the operational semantics
- Tested against a series of real and synthetic Rascal transformations



“

The robustness of the semantics depends upon theorems

”

Milner, R., Tofte, M., Harper, R. (1990). The Definition of Standard ML.

Correctness of Rascal Light semantics

Proven theories:

- Purity of backtracking
- Strong typing
- Partial progress
- Terminating subset

Static Analysis Tool for Rascal Light

Based on Al-Sibahi A. S., Jensen, T. P., Dimovski, A. S. & Wąsowski, A. (2017). Verification of High-Level Transformations with Inductive Refinement Types. *Unpublished Draft*.

Type and Shape Properties

Inductive Shape

```
refine Config_simple =
    flat("playerId", 1..10)
  | nested("players", [Config_simple]^2..3)
```

Example represented programs

flat("playerId", 10)

nested("players",
[flat("playerId", 1),
flat("playerId", 3)]

- Types
- Inductive shapes

Rascal Static Analysis Challenges

Challenges

1. **Complex inductive structures** with **collections**
2. **Non-modular** control flow
3. **Substantial** number of **expressive** language constructs

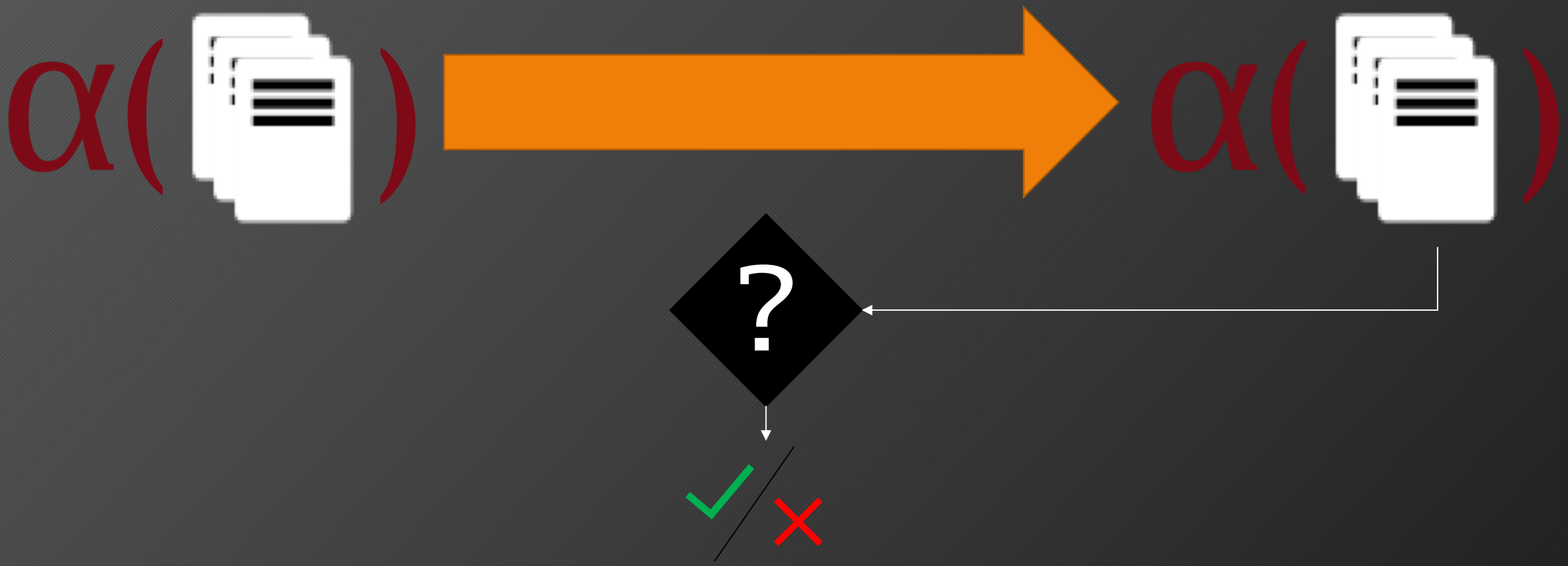
Solutions

1. Modular construction of abstract domains
2. Schmidt-style abstract interpretation directly on operational semantics
3. Systematic mapping of concrete semantics to abstract semantics

“It is convenient to think of an [abstract intepretation] as a “symbolic execution” where the symbols have semantic content.”

Schmidt, D. A. (1998). Trace-Based Abstract Interpretation of Operational Semantics. Journal of LISP and Symbolic Computation 10, pp. 237-271.

Static Analysis using Schmidt-style Abstract Interpretation



Implementation

- Rascal light Abstract Interpretation Tool (RABIT)
- Development effort:
  - ~3 months of part-time programming
  - 5,673 SLOC Scala (incl. concrete interpreter)
- Structured technique helped reduced bugs
  - Intrinsically complex meta-meta-program, makes it hard to debug and log calls

Evaluation Subjects

Transformation	Description
Negation Normal Form (NNF)	Normalize a propositional formula so that all negations (¬) are only in front of atoms
Rename Struct Field (RSF)	Refactor the name of the field of a structure, ensuring all references are updated correctly
Desugar Oberon-0 (DSO0)	Translate **for**-loops and **switch**-statements to **while**-loops and **if**-statements resp. for the Oberon-0
Glagol-to-PHP Expressions (G2PE)	Code generation to PHP from expressions in the Glagol DSL

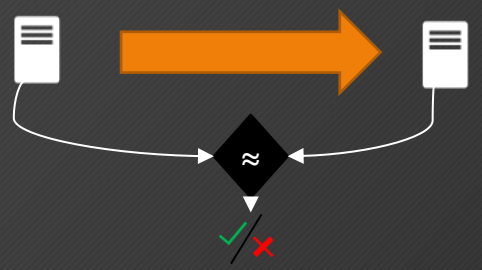
Verified Properties

Transformation	#	Target Property	Verified
NNF	P1	Implication is not used as a connective in the result	✓
	P2	All negations in the result are in front of atoms	✓
RSF	P3	Structures should not define fields with the old name	✗
	P4	There should not be any field access expression to the old field name	✓
DSO0	P5	For-loops correctly desugared to while-loops	✓
	P6	Switch-statements correctly desugared to if-statements	✓
	P7	No auxiliary data in output	✗
G2PE	P8	Only produce simple PHP expressions given simple Glagol expressions	✓
	P9	Not produce unary PHP expressions if there were no +/- markers in input Glagol	✓

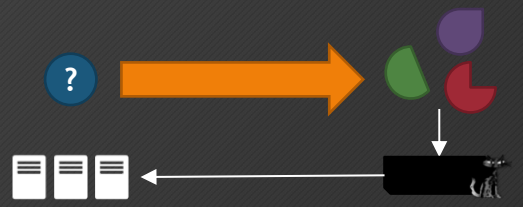
Wrapping up

Key Contributions

Translation Validation



Symbolic Execution



Static Analysis