Trustworthy Refactoring via Decomposition and Schemes

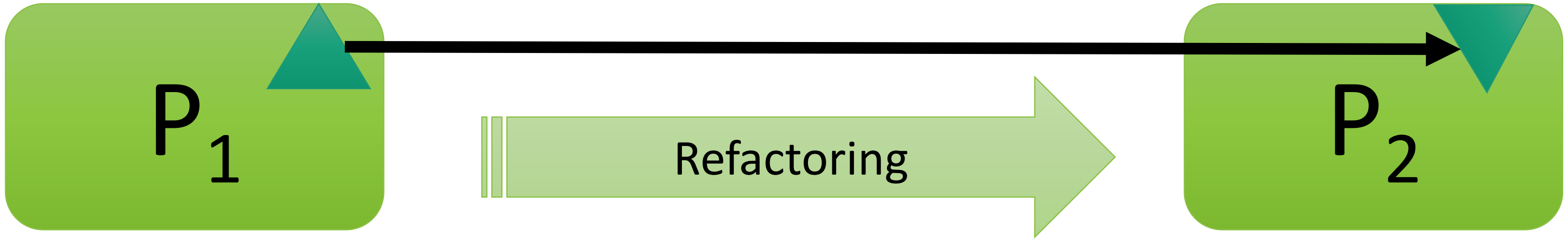
Dániel HORPÁCSI, Judit KŐSZEGI, Zoltán HORVÁTH

Eötvös Loránd University, Budapest

Supported through the New National Excellence Program
of the Ministry of Human Capacities of Hungary

Refactoring correctness

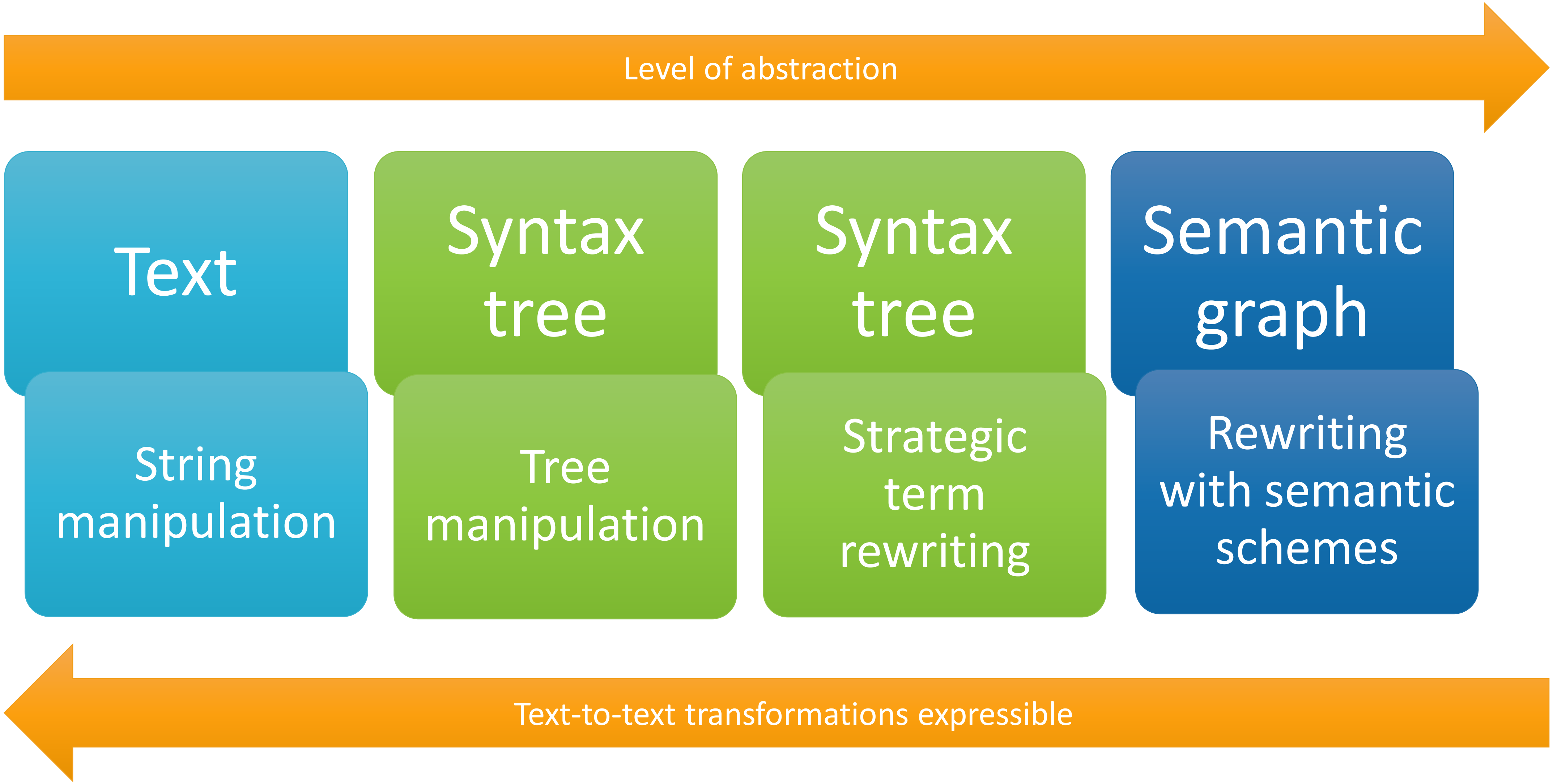
- For any input program, the refactoring transformation results in a program **semantically equivalent** to the input, w.r.t. a definition of language semantics and equivalence.



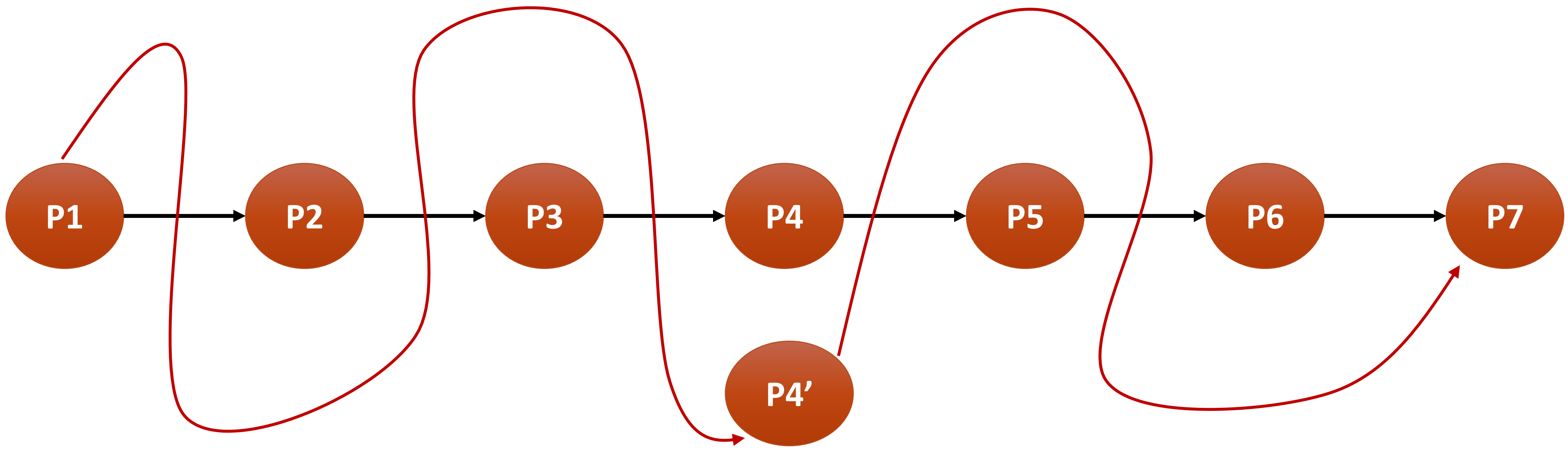
$P_1 \equiv P_2$

- In practice, only part of the program is modified
- **Extensive changes: completeness + consistency**

Representation - transformation



(De)composition

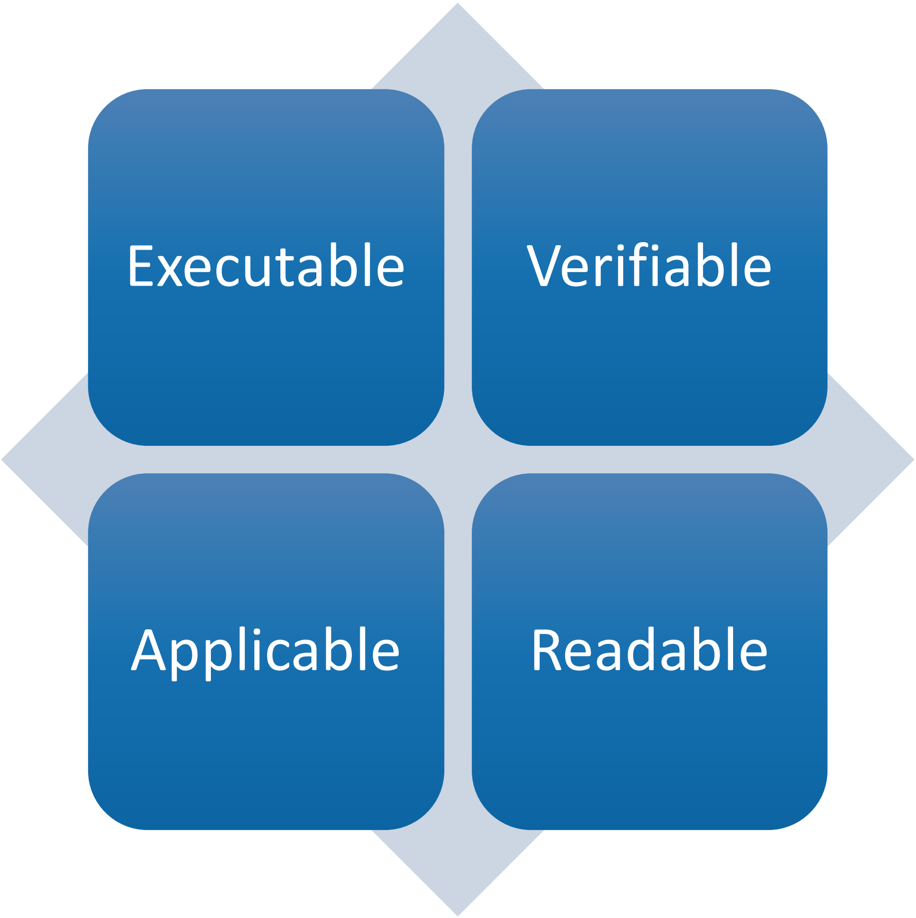


- PRO: likely results in simple, reusable micro-refactorings
- CON: it may be hard to find the proper decomposition

Semantic schemes

- Completeness and consistency
  - Ensures compensation for each change, at every affected location
- Sequential program of term rewrite rules
- With modifiers based on data and control dependency
  - Data-flow
    - Forward
    - Backward
  - Introduce or modify binding
    - Variable
    - Function

A refactoring specification formalism



Term rewriting

+

Semantic predicates

+

Semantic schemes

+

Selector functions

+

Refactoring functions

Case study: „generalise function definition"

```
f(X) -> begin X * 2 end.
g(X) -> f(X+1).
```

⇒

```
f(X, Y) -> begin X * Y() end.
f(X) -> f(X, fun () -> 2 end).
g(X) -> f(X+1).
```

At first glance: only 2 steps

- Create the generalised function
- Make the less general function invoke the generalised one

Compositional approach: 6 steps

- Wrap the selected element into an unnamed function application
- Create a function abstraction from the body of the function
- Create a variable abstraction from the selected element
- Lift the variable to the function scope
- Lift the variable to function parameter
- Rename the generalised function to the original name

```
f(X) -> begin X * 2 end.
g(X) -> f(X + 1).
```

```
f(X) -> begin X * fun () -> 2 end () end.
g(X) -> f(X+1).
```

STEP 1: wrap expression into unnamed function

```
LOCAL REFACTORING wrap ()
                   E
   ______________________________
   fun (Vars..) -> E end (Vars..)
WHEN
   Vars.. = free_vars(E) AND non_bind(E)
```

```
f(X) -> begin X * fun () -> 2 end () end.
g(X) -> f(X+1).
```

```
t(X) -> begin X * fun () -> 2 end () end.
f(X) -> t(X).
g(X) -> f(X+1).
```

STEP 2: extract expression into function (t - temporary)

INTRODUCE FUNCTION extract_to_function(NewName)

DEFINITION

```
NewName(Vars..) -> E .
```

REFERENCE

$$\frac{\texttt{E}}{\texttt{NewName(Vars..)}}$$

WHEN Vars.. = free_vars(E)

```
t(X) -> begin X * fun () -> 2 end () end.
f(X) -> t(X).
g(X) -> f(X+1).
```

```
t(X) -> begin Y = fun () -> 2 end, X* Y() end.
f(X) -> t(X).
g(X) -> f(X+1).
```

STEP 3: extract expression into variable (Y)

INTRODUCE VARIABLE extract_to_variable(NewName)

DEFINITION IN SCOPE

NewName = E

REFERENCE

$$\frac{E}{NewName}$$

```
t(X) -> begin Y = fun () -> 2 end, X* Y() end.
f(X) -> t(X).
g(X) -> f(X+1).
```

```
t(X) -> Y = fun () -> 2 end, begin X* Y() end.
f(X) -> t(X).
g(X) -> f(X+1).
```

STEP 4: lift variable to outer scope

INTRODUCE VARIABLE outer_variable()

DEFINITION IN OUTER SCOPE

Name = E

REFERENCE

$$\frac{\text{Name = E}}{\text{Name}}$$

```
t(X) -> Y = fun () -> 2 end, begin X* Y() end.
f(X) -> t(X).
g(X) -> f(X+1).
```

```
t(X, Y) -> begin X * Y() end.
f(X) -> t(X, fun () -> 2 end).
g(X) -> f(X+1).
```

STEP 5: variable to function parameter

FUNCTION REFACTORING variable_to_parameter()

DEFINITION

$$\frac{\texttt{(Args..) -> X = E, Body..}}{\texttt{(Args.., X) -> Body..}}$$

REFERENCE

$$\frac{\texttt{(Args2..)}}{\texttt{(Args2.., E)}}$$

```
t(X, Y) -> begin X * Y() end.
f(X) -> t(X, fun () -> 2 end).
g(X) -> f(X+1).
```

```
f(X, Y) -> begin X * Y() end.
f(X) -> f(X, fun () -> 2 end).
g(X) -> f(X+1).
```

STEP 6: rename function (to f)

FUNCTION SIGNATURE REFACTORING `rename_function(NewName)`

$$\frac{\texttt{Name(Args..)}}{\texttt{NewName(Args..)}}$$

```
t(X, Y) -> begin X * Y() end.
f(X) -> t(X, fun () -> 2 end).
g(X) -> f(X+1).
```

```
f(X, Y) -> begin X * Y() end.
```

```
REFACTORING generalise_function(ParamName)
DO
  THIS.wrap()
  THIS = THIS.function_part()
  Old = function(THIS)
  Name = name(Old)
  New = Old.body().extract_to_function(t)
  Var = THIS.extract_to_variable(ParamName)
  Var.to_function_parameter()
  New.rename_function(Name)
```

FUNCTION S ... ame)

Name(A

NewName(

Verification technique

Local refactoring

- Simple rewriting: conditional pattern equivalence

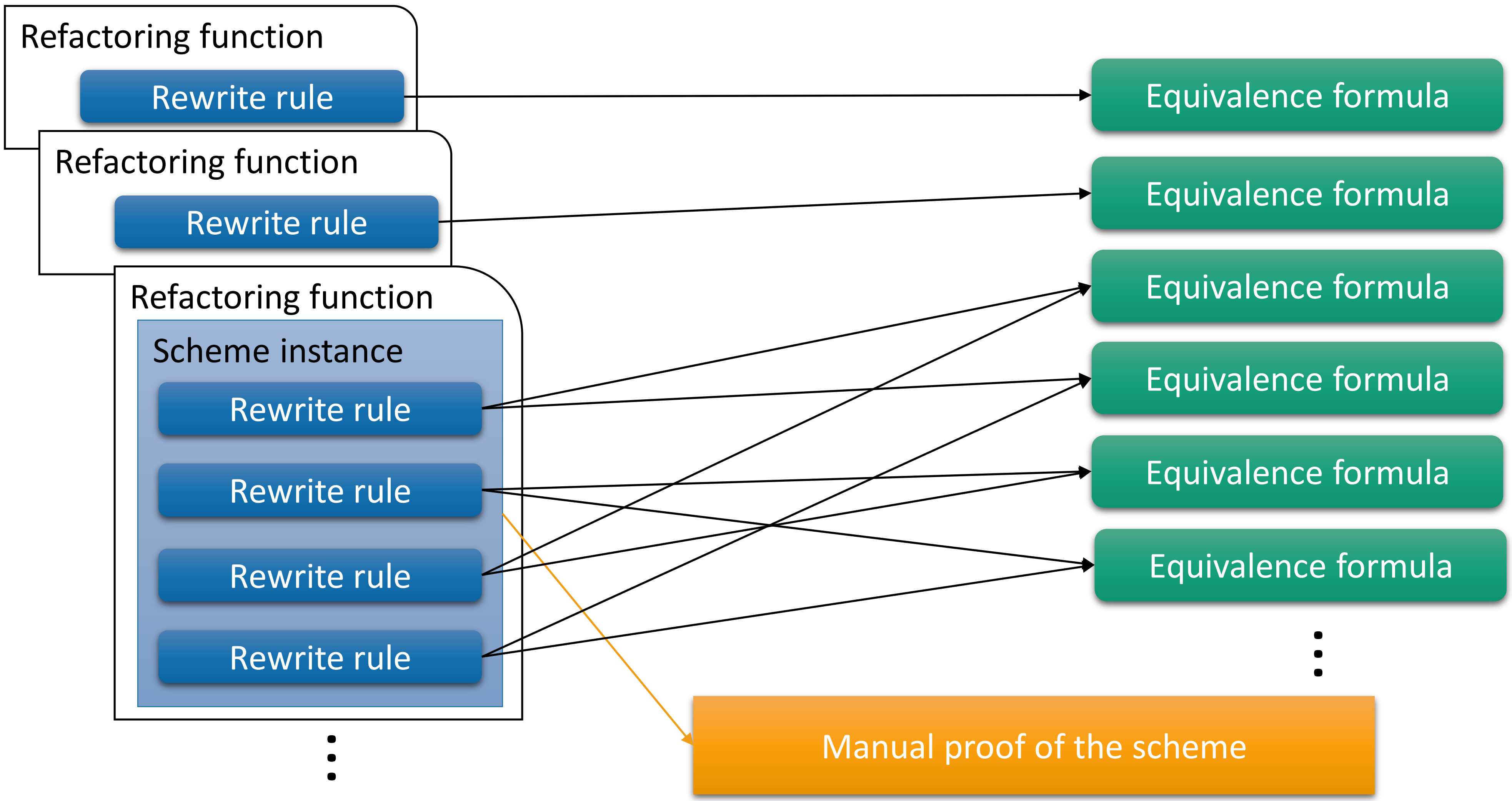
Extensive refactoring

- Scheme verification: inductive proof on the semantic relation
- Scheme instance verification: conditional pattern equivalences

Composite refactoring

- Correct if the components are correct

Verification technique



Equivalence checking (in Matching Logic)

📖 Stefan Ciobaca et al. (2016): A Language-Independent Proof System for Full Program Equivalence.

a formula with a pair of configuration patterns

set of pairs of configuration patterns that are known to be equivalent

$$\varphi \Downarrow^{\infty} Eq$$

All concrete pairs of program configurations that match to $\boldsymbol{\varphi}$ *both diverge or reach an instance of* Eq.

Proof System

AXIOM $\dfrac{\varphi \in Eq}{\vdash \varphi \Downarrow^{\infty} Eq}$

CONSEQ $\dfrac{\vDash \varphi \to \exists \tilde{x}.\varphi' \quad \varphi' \Downarrow^{\infty} Eq}{\vdash \varphi \Downarrow^{\infty} Eq}$

STEP $\dfrac{\vDash \varphi_1 \Rightarrow^{*} \varphi_1' \quad \vDash \varphi_2 \Rightarrow^{*} \varphi_2' \quad \vdash \langle \varphi_1', \varphi_2' \rangle \Downarrow^{\infty} Eq}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^{\infty} Eq}$

CASE ANALYSIS

CIRCULARITY

Verification of „wrap”

```
LOCAL REFACTORING wrap ()
                         E
   ─────────────────────────────────
   fun (Vars..) -> E end (Vars..)
WHEN
  Vars.. = free_vars(E) AND non_bind(E)
```

Verification of „wrap”

$$\left\langle \begin{array}{c} \langle\langle E\rangle_{\mathrm{code}}\langle\varepsilon_0\rangle_{\mathrm{env}}\cdots\rangle_{\mathrm{cfg}} \\ \langle\langle \texttt{fun (}\mathit{Vars}\texttt{) -> } E \texttt{ end (}\mathit{Vars}\texttt{)}\rangle_{\mathrm{code}}\langle\varepsilon_0\rangle_{\mathrm{env}}\cdots\rangle_{\mathrm{cfg}} \end{array} \right\rangle \Downarrow^{\infty} \left\langle \begin{array}{c} \langle\langle \mathit{Code}\rangle_{\mathrm{code}}\langle\varepsilon\rangle_{\mathrm{env}}\rangle_{\mathrm{cfg}} \\ \langle\langle \mathit{Code}\rangle_{\mathrm{code}}\langle\varepsilon\rangle_{\mathrm{env}}\rangle_{\mathrm{cfg}} \end{array} \right\rangle, \ldots$$

$$\land\ \mathit{Vars} = \texttt{free\_vars}(E) \land \texttt{non\_bind}(E)$$

$$1.\quad \vdash \left\langle \begin{array}{ll} \langle\langle \mathit{Code}\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}}\cdots\rangle_{\mathsf{cfg}} \\ \langle\langle \mathit{Code}\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}}\cdots\rangle_{\mathsf{cfg}} \end{array} \right\rangle_{\mathsf{eq}} \quad \Downarrow^{\infty}\ \mathrm{Eq} \quad \textsc{Axiom}$$

$$2.\quad \vdash \left\langle \begin{array}{ll} \langle\langle E\rangle_{\mathsf{code}} & \langle\varepsilon_0\rangle_{\mathsf{env}}\cdots\rangle_{\mathsf{cfg}} \\ \langle\langle E\rangle_{\mathsf{code}} & \langle\varepsilon_0\rangle_{\mathsf{env}}\cdots\rangle_{\mathsf{cfg}} \end{array} \right\rangle_{\mathsf{eq}} \land \Psi \quad \Downarrow^{\infty}\ \mathrm{Eq} \quad \textsc{Conseq}(1)$$

$$3.\quad \vdash \left\langle \begin{array}{ll} \langle\langle E\rangle_{\mathsf{code}} & \langle\varepsilon_0\rangle_{\mathsf{env}}\cdots\rangle_{\mathsf{cfg}} \\ \langle\langle \texttt{fun}(\mathit{Vars})\texttt{->}\,E\ \texttt{end}(\mathit{Vars})\rangle_{\mathsf{code}} & \langle\varepsilon_0\rangle_{\mathsf{env}}\cdots\rangle_{\mathsf{cfg}} \end{array} \right\rangle_{\mathsf{eq}} \land \Psi \quad \Downarrow^{\infty}\ \mathrm{Eq} \quad \textsc{Step}(2)$$

$$\Psi \equiv \mathit{Vars} = \texttt{free\_vars}(E) \land \texttt{non\_bind}(E)$$

Verification of „extract_to_variable” (introduce variable scheme)

Template

```
DEFINITION IN SCOPE
  <name> = <pattern1>
REFERENCE
  <pattern2>
  ----------
  <pattern3>
```

Definition

```
ON scope(THIS)
            E..
  ------------------------
  <name> = <pattern1>, E..
WHEN fresh(<name>)
 AND pure(<pattern1>)
 AND closed(<pattern1>)
THEN ON THIS
  <pattern2>
  ----------
  <pattern3>
```

Contract

```
begin <name> = <pattern1>, <pattern3> end ≡ <pattern2>
```

Verification of „extract_to_variable” (introduce variable scheme)

Template

```
DEFINITION IN SCOPE
  <name> = <pattern1>
REFERENCE
  <
  -
  <
```

Definition

```
ON scope(THIS)
            E..
  -------------------------
```

$$\vdash \left\langle \begin{array}{ll} \langle\langle E\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \\ \langle\langle \mathtt{begin}\ Name = E\, ,\, Name\ \mathtt{end}\rangle_{\mathsf{code}} & \langle\varepsilon\rangle_{\mathsf{env}} \cdots\rangle_{\mathsf{cfg}} \end{array}\right\rangle_{\mathsf{eq}} \wedge\ \mathrm{fresh}(Name) \Downarrow^{\infty} \quad E$$

```
THEN ON THIS
  <pattern2>
  ----------
  <pattern3>
```

Contract

```
begin <name> = <pattern1>, <pattern3> end ≡ <pattern2>
```

Summary

- Refactoring can be seen as a special case of strategic term rewriting
- By decomposition, we can split up big steps into small micro-steps
- Finding a decomposition may be far from trivial, but it pays off
- To check correctness, we translate the refactoring specification into a set of equivalence formulas in matching logic
- Refactoring specifications are executable and verifiable
- With appropriate restrictions (and appropriate program representation) we arrive at refactoring oriented programming