# Program Equivalence in Sequential Core Erlang

Proving Refactoring Correctness

***Dániel Horpácsi***
***Péter Bereczky***
*Eötvös Loránd University*
*Budapest, Hungary*

SZÉCHENYI 2020
HUNGARIAN GOVERNMENT
European Union
European Social Fund
INVESTING IN YOUR FUTURE

# Correctness: behaviour-preservation

$e \xrightarrow{\textit{transformation}} e'$

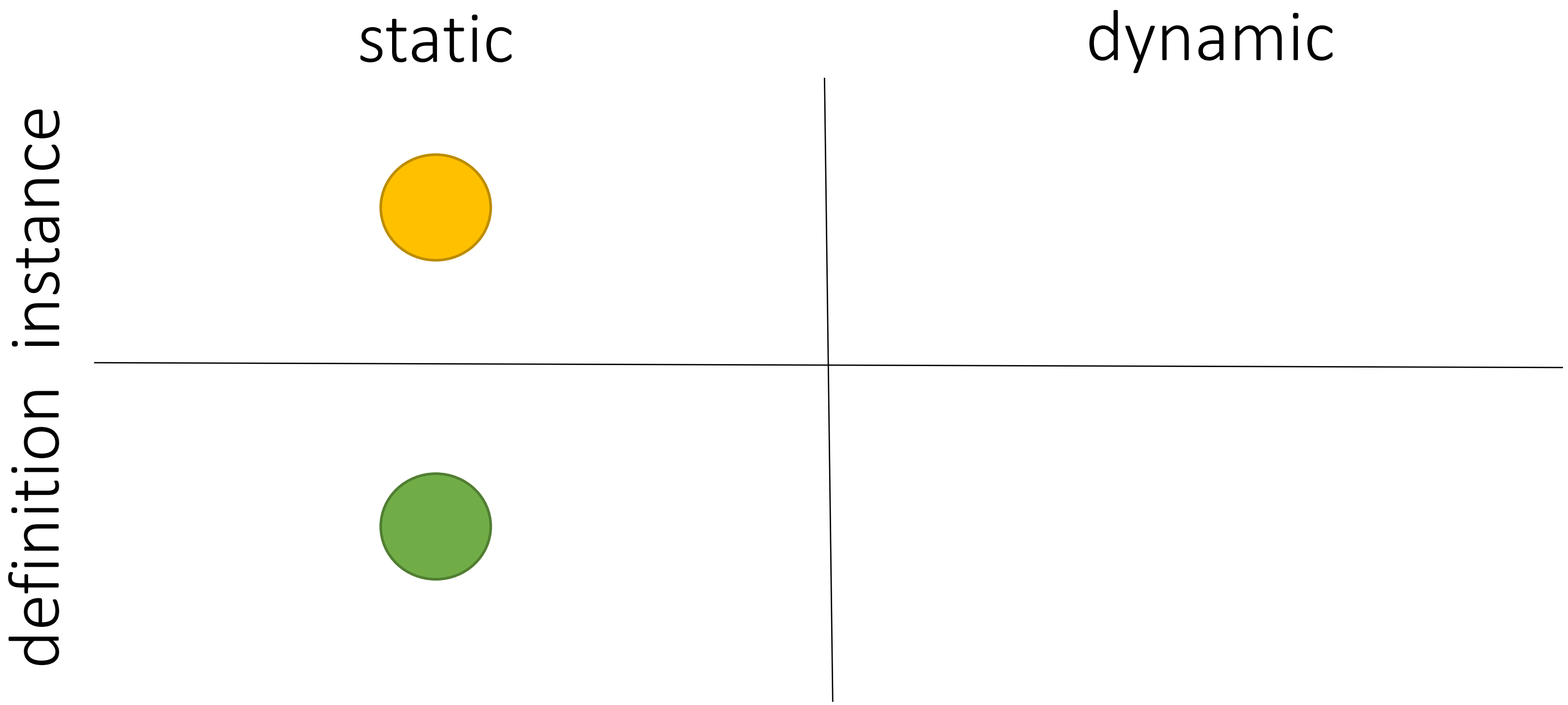
<u>*is refactoring if*</u>

$e \approx e'$

ground terms

behavioural equivalence

# Refactoring verification



# Local refactoring: eta-abstraction

$E \Rightarrow$ `(fun() ->` $E$`)()`

*is refactoring if*

`C[` $E$ `]` $\approx$ `C[ (fun() ->` $E$`)() ]`

# Local refactoring: eta-abstraction

$E \quad \Rightarrow \quad$ `(fun() -> ` $E$ `)()`

*is refactoring if*

$E \quad \approx_{ctx} \quad$ `(fun() -> ` $E$ `)()`

# Local refactoring

$$l \Rightarrow r \quad \textit{when} \quad c$$

first-order terms

*is refactoring if*

$$c \rightarrow l \approx_{\text{ctx}} r$$

contextual equivalence

# Extensive refactoring

Many refactorings are not local…

- *changing function interfaces, altering dataflow paths etc.*

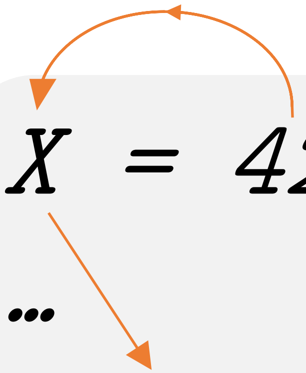
Multiple term rewrite rules applied simultaneously…

- *can we reason about rewrite strategies?*

Solution: semantic refactoring scheme

- *"semantic strategy"*

# Dataflow refactoring: eta-abstraction



```
X = 42,
…
f(X)
```

⇒

```
X = (fun() -> 42)(),
…
f(X)
```

# Dataflow refactoring: eta-abstraction

```
X = 42,
…
f(X)
```

⇒

$E \Rightarrow$ `(fun() -> E)`

```
X = fun() -> 42,
…
f(X())
```

$E \Rightarrow$ `E()`

# Dataflow refactoring: eta-abstraction

$$\mathcal{S} \Rightarrow (\texttt{fun() -> } \mathcal{S}) \quad \circ_{\texttt{df}} \quad \mathcal{R} \Rightarrow \mathcal{R}\texttt{()}$$

*is refactoring if*

$$\texttt{C}_{\texttt{df}}[\mathcal{S},\ \mathcal{R}] \approx \texttt{C}_{\texttt{df}}\ [(\texttt{fun() -> } \mathcal{S}),\ \mathcal{R}\texttt{()}]$$

# Dataflow refactoring scheme

$$S \Rightarrow \mathtt{C}_1[S] \quad \circ_{\mathtt{df}} \quad R \Rightarrow \mathtt{C}_2[R]$$

- Alter a dataflow chain of expressions
  - With a primary and a secondary rewrite rule
  - The secondary rule needs to compensate/void the primary rule
- Correctness: induction on the data flow context
  - The base cases follow from the local equivalence

$$S \approx_{\text{ctx}} \mathtt{C}_2[\mathtt{C}_1[S]]$$

# Schemes in general

$$\mathtt{C}_1[S] \Rightarrow \mathtt{C}_2[S] \quad \circ \quad \mathtt{C}_3[R] \Rightarrow \mathtt{C}_4[R]$$

- Various semantic dependencies induce refactoring schemes

- Plenty of open questions being investigated:
  - Formal definition of semantic contexts for the various schemes
  - Compound matching patterns for context-aware rewriting
  - Multiple alternative compensation rules
  - Multiple roots and loops in the dependency subgraph

By using schemes, refactoring verification boils down to $\approx$ and $\approx_{ctx}$ !

By using schemes, refactoring verification boils down to $\approx$ and $\approx_{ctx}$ !

# Motivational examples

```
mod(_, 0) -> error;
mod(N, D) -> N rem D == 0.

f() ->
  lists:foldr(fun(E, Acc) -> mod(10,E) andalso Acc end,
              true, [0,1,2,3,4,5,6]).

g() ->
  lists:foldr(fun(E, Acc) -> Acc andalso mod(10,E) end,
              true, [0,1,2,3,4,5,6]).
```

# Motivational examples

```
mod(_, 0) -> error;
mod(N, D) -> N rem D == 0.

f() ->
  lists:foldr(fun(E, Acc) -> mod(10,E) andalso Acc end,
              true, [0,1,2,3,4,5,6]).

g() ->
  lists:foldr(fun(E, Acc) -> Acc andalso mod(10,E) end,
              true, [0,1,2,3,4,5,6]).
```

# Motivational examples

```
mod(_, 0) -> error;
mod(N, D) -> N rem D == 0.

f() ->
  lists:foldr(fun(E, Acc) -> mod(10,E) andalso Acc end,
              true, [0,1,2,3,4,5,6]).

g() ->
  lists:foldr(fun(E, Acc) -> Acc andalso mod(10,E) end,
              true, [0,1,2,3,4,5,6]).
```

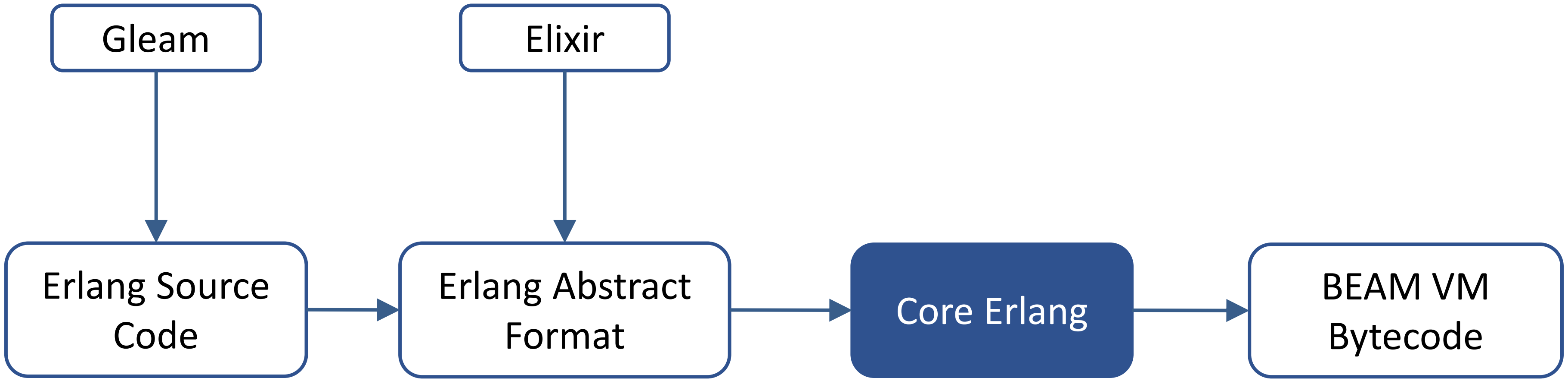
Exception

false

Is $e_1 + e_2 \approx e_2 + e_1$ for every $e_1, e_2$?
$e_1 \coloneqq$ `io:fwrite(foo)`
$e_2 \coloneqq$ `3 rem 0`

Exceptions in both cases, but side effects changed

# Motivation: Why Core Erlang?



# What is behaviour-preservation?

- Suppose, we have the semantics: $\langle \Gamma, e \rangle \Downarrow v$
- Program equivalence relations
- Congruence

Reflexive
Symmetric
Transitive

$$e_1 \approx a_1 \land e_2 \approx a_2$$

# What is behaviour-preservation?

- Suppose, we have the semantics: $\langle \Gamma, e \rangle \Downarrow v$
- Program equivalence relations

  Reflexive
  Symmetric
  Transitive
- Congruence

$$e_1 \approx a_1 \wedge e_2 \approx a_2 \Rightarrow \textbf{do}\ e_1\ e_2 \approx \textbf{do}\ a_1\ a_2$$

- When are programs equivalent?
  - Strong equivalence

$$e \approx 2 + 2,\ e$$

  - Weak equivalence

```
io:fwrite(a), io:fwrite(b), e ≈ io:fwrite(b), io:fwrite(a), e
```

# What does it mean formally?

1st definition (based on [1]):

$$e_1 \approx e_2 \ ::= \forall(\Gamma\!: Environment)(v\!: Value)\!:$$
$$\langle \Gamma, e_1 \rangle \Downarrow v \ \Leftrightarrow \langle \Gamma, e_2 \rangle \Downarrow v$$

- ✓ Reflexive
- ✓ Symmetric
- ✓ Transitive
- ✓ Congruent

[1] Pierce, Benjamin C., et al. "Software foundations." *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html* (2010).

# What does it mean formally?

1st definition (based on [1]):

$$e_1 \approx e_2 \;::= \forall(\Gamma{:}\,Environment)(v{:}\,Value){:}$$
$$\langle\Gamma, e_1\rangle \Downarrow v \;\Leftrightarrow \langle\Gamma, e_2\rangle \Downarrow v$$

- ✓ Reflexive
- ✓ Symmetric
- ✓ Transitive
- ✓ Congruent

Major disadvantage:

$e_1 ::=$ `fun(X) -> X + 2`

$e_2 ::=$ `fun(X) -> X + 1 + 1`

What is the problem?

$$\langle\Gamma, e_1\rangle \Downarrow clos(\Gamma, [X], X + 2)$$
$$\langle\Gamma, e_2\rangle \Downarrow clos(\Gamma, [X], X + 1 + 1)$$

# Contexts

Expression context: expressions can contain ”holes”

```
do (let X = □ in X + 3) □
```

It is an inductive definition:

```
Inductive Context : Set :=
| CVar (v : Var)
| CHole
| CLet (v : Var) (c1 c2 : Context)
...
```

Substitution: $C[e]$

(`let X = □ in do (X + 3) □`)$[3+2] =$
`let X = 3 + 2 in do (X + 3) (3 + 2)`

# Contextual equivalence with equality

$$e_1 \approx_{ctx} e_2 \ ::= \forall(C{:}\,Context)(\Gamma{:}\,Environment)(v{:}\,Value){:}$$
$$\langle \Gamma, C[e_1] \rangle \Downarrow v \Leftrightarrow \langle \Gamma, C[e_2] \rangle \Downarrow v$$

- ✓ Reflexive
- ✓ Symmetric
- ✓ Transitive
- ✓ Congruent

# Contextual equivalence with equality

$$e_1 \approx_{ctx} e_2 ::= \forall(C\colon Context)(\Gamma\colon Environment)(v\colon Value)\colon$$
$$\langle \Gamma, C[e_1] \rangle \Downarrow v \Leftrightarrow \langle \Gamma, C[e_2] \rangle \Downarrow v$$

- ✓ Reflexive
- ✓ Symmetric
- ✓ Transitive
- ✓ Congruent

The previous problem still exists ☹

$e_1$ ::= `fun(X) -> X + 2`

$e_2$ ::= `fun(X) -> X + 1 + 1`

# Equivalent results

$$v_1 \approx_{val} v_2 ::= v_1 = v_2$$

# Equivalent results

$$e_1 \approx_{ctx} e_2 ::= \forall(C{:}\,Context){:}\ \mathrm{C}[e_1] \approx_{exp} C[e_2]$$

$$e_1 \approx_{exp} e_2 ::= \forall(\Gamma{:}\,Environment)(v_1, v_2{:}\,Value){:}$$
$$\langle\Gamma, e_1\rangle \Downarrow v_1 \wedge \langle\Gamma, e_2\rangle \Downarrow v_2 \Rightarrow v_1 \approx_{val} v_2$$

$$v_1 \approx_{val} v_2 ::= v_1 = v_2$$
$$clos(\Gamma, [x_1, \dots, x_n], e_1) \approx_{val} clos(\Gamma', [x_1, \dots, x_n], e_2) ::=$$
$$\forall v_1, \dots, v_n{:}\,(fun(x_1, \dots, x_n) \rightarrow e_1)(v_1, \dots, v_n) \approx_{exp}$$
$$(fun(x_1, \dots, x_n) \rightarrow e_2)(v_1, \dots, v_n)$$

- ✓ Reflexive
- ✓ Symmetric
- ✓ Transitive
- x Congruent

[2] Owens, Scott, et al. "Functional big-step semantics." *European Symposium on Programming*. Springer, Berlin, Heidelberg, 2016.
[3] Pitts, Andrew M. "Operationally-based theories of program equivalence." *Semantics and Logics of Computation* 14 (1997): 241.

# Congruence: application?

- Goal: $C_0(C_1, \ldots, C_n)[e_1] \approx_{exp}$
  $C_0(C_1, \ldots, C_n)[e_2]$

# Congruence: application?

- Goal: $C_0(C_1, \dots, C_n)[e_1] = C_0[e_1](C_1[e_1], \dots, C_n[e_2]) \approx_{exp}$
  $C_0(C_1, \dots, C_n)[e_2] = C_0[e_2](C_1[e_2], \dots, C_n[e_2])$

- Induction hypothesis: $\forall (e_1, e_2 : Exp) : e_1 \approx_{exp} e_2 \rightarrow C_i[e_1] \approx_{exp} C_i[e_2]$

- If $C_0[e_1] \Downarrow clos(\Gamma, [x_1, \dots, x_n], b_1)$ and $C_0[e_2] \Downarrow clos(\Gamma', [x_1, \dots, x_n], b_2)$ then:

$$\forall v_1, \dots, v_n : (fun(x_1, \dots, x_n) \rightarrow b_1)(v_1, \dots, v_n) \approx_{exp}$$
$$(fun(x_1, \dots, x_n) \rightarrow b_2)(v_1, \dots, v_n)$$

# Congruence: application?

- Goal: $C_0(C_1, \dots, C_n)[e_1] = C_0[e_1](C_1[e_1], \dots, C_n[e_2]) \approx_{exp}$
  $C_0(C_1, \dots, C_n)[e_2] = C_0[e_2](C_1[e_2], \dots, C_n[e_2])$

- Induction hypothesis: $\forall(e_1, e_2 : Exp) : e_1 \approx_{exp} e_2 \rightarrow C_i[e_1] \approx_{exp} C_i[e_2]$

- If $C_0[e_1] \Downarrow clos(\Gamma, [x_1, \dots, x_n], b_1)$ and $C_0[e_2] \Downarrow clos(\Gamma', [x_1, \dots, x_n], b_2)$ then:

$$\forall v_1, \dots, v_n : (fun(x_1, \dots, x_n) \rightarrow b_1)(v_1, \dots, v_n) \approx_{exp} (fun(x_1, \dots, x_n) \rightarrow b_2)(v_1, \dots, v_n)$$

- However, we only have equivalent parameters

# Back to the definition

$$e_1 \approx_{ctx} e_2 \ ::= \forall(C{:}\,Context){:}\ \mathrm{C}[e_1] \approx_{exp} C[e_2]$$

$$e_1 \approx_{exp} e_2 \ ::= \forall(\Gamma{:}\,Environment)(v_1, v_2{:}\,Value){:}$$
$$\langle \Gamma, e_1 \rangle \Downarrow v_1 \wedge \langle \Gamma, e_2 \rangle \Downarrow v_2 \Rightarrow v_1 \approx_{val} v_2$$

$$v_1 \approx_{val} v_2 ::= v_1 = v_2$$
$$clos(\Gamma, [x_1, \dots, x_n], e_1) \approx_{val} clos(\Gamma', [x_1, \dots, x_n], e_2) ::=$$
$$\forall v_1, \dots, v_n{:}\ (fun(x_1, \dots, x_n) \rightarrow e_1)(v_1, \dots, v_n) \approx_{exp}$$
$$(fun(x_1, \dots, x_n) \rightarrow e_2)(v_1, \dots, v_n)$$

Same environment?
Same parameters?

- ✓ Reflexive
- ✓ Symmetric
- ✓ Transitive
- x Congruent

# Let us fix the congruence premises

$$v_1 \approx_{val} v_2 ::= v_1 = v_2$$
$$clos(\Gamma, [x_1, \dots, x_n], e_1) \approx_{val} clos(\Gamma', [x_1, \dots, x_n], e_2) ::=$$
$$\forall v_1, \dots, v_n, v_1', \dots, v_n' \colon v_1 \approx_{val} v_1' \land \cdots \land v_n \approx_{val} v_n' \Rightarrow$$
$$\Gamma[x_1 \leftarrow v_1, \dots, x_n \leftarrow v_n], e_1 \approx_{exp} \Gamma'[x_1 \leftarrow v_1', \dots, x_n \leftarrow v_n'], e_2$$

# Let us fix the congruence premises

$$v_1 \approx_{val} v_2 ::= v_1 = v_2$$
$$clos(\Gamma, [x_1, \ldots, x_n], e_1) \approx_{val} clos(\Gamma', [x_1, \ldots, x_n], e_2) ::=$$
$$\forall v_1, \ldots, v_n, v_1', \ldots, v_n' \colon v_1 \approx_{val} v_1' \land \cdots \land v_n \approx_{val} v_n' \Rightarrow$$
$$\Gamma[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n], e_1 \approx_{exp} \Gamma'[x_1 \leftarrow v_1', \ldots, x_n \leftarrow v_n'], e_2$$

# Let us fix the congruence premises

$$v_1 \approx_{val} v_2 ::= v_1 = v_2$$
$$clos(\Gamma, [x_1, \dots, x_n], e_1) \approx_{val} clos(\Gamma', [x_1, \dots, x_n], e_2) ::=$$
$$\forall v_1, \dots, v_n, v'_1, \dots, v'_n\colon v_1 \approx_{val} v'_1 \land \cdots \land v_n \approx_{val} v'_n \Rightarrow$$
$$\Gamma[x_1 \leftarrow v_1, \dots, x_n \leftarrow v_n], e_1 \approx_{exp} \Gamma'[x_1 \leftarrow v'_1, \dots, x_n \leftarrow v'_n], e_2$$

# Let us fix the congruence premises

$$\Gamma, e_1 \approx_{exp} \Gamma', e_2 \ ::= \forall (v_1, v_2 : Value):$$
$$\langle \Gamma, e_1 \rangle \Downarrow v_1 \wedge \langle \Gamma', e_2 \rangle \Downarrow v_2 \Rightarrow v_1 \approx_{val} v_2$$

$$v_1 \approx_{val} v_2 ::= v_1 = v_2$$
$$clos(\Gamma, [x_1, \ldots, x_n], e_1) \approx_{val} clos(\Gamma', [x_1, \ldots, x_n], e_2) ::=$$
$$\forall v_1, \ldots, v_n, v_1', \ldots, v_n' : v_1 \approx_{val} v_1' \wedge \cdots \wedge v_n \approx_{val} v_n' \Rightarrow$$
$$\Gamma[x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n], e_1 \approx_{exp} \Gamma'[x_1 \leftarrow v_1', \ldots, x_n \leftarrow v_n'], e_2$$

Using this: $\Gamma, 0 \approx_{exp} \Gamma,$ `letrec 'f'/0 = fun() -> apply 'f'/0 in apply 'f'/0()`

# Adding termination criteria

$$\Gamma, e_1 \approx_{exp} \Gamma', e_2 \; ::= \forall (v_1, v_2 : Value):$$
$$\langle \Gamma, e_1 \rangle \Downarrow \Leftrightarrow \langle \Gamma', e_2 \rangle \Downarrow \land$$
$$(\langle \Gamma, e_1 \rangle \Downarrow v_1 \land \langle \Gamma', e_2 \rangle \Downarrow v_2 \Rightarrow v_1 \approx_{val} v_2)$$

$$v_1 \approx_{val} v_2 ::= v_1 = v_2$$
$$clos(\Gamma, [x_1, \dots, x_n], e_1) \approx_{val} clos(\Gamma', [x_1, \dots, x_n], e_2) ::=$$
$$\forall v_1, \dots, v_n, v_1', \dots, v_n' : v_1 \approx_{val} v_1' \land \cdots \land v_n \approx_{val} v_n' \Rightarrow$$
$$\Gamma[x_1 \leftarrow v_1, \dots, x_n \leftarrow v_n], e_1 \approx_{exp} \Gamma'[x_1 \leftarrow v_1' \dots x_n \leftarrow v_n'], e_2$$

Work in progress
- ✓ Reflexive
- Symmetric
- Transitive
- Congruent

# Examples

$$e_1 \approx e_2 \ ::= \forall(\Gamma: Environment): \Gamma, e_1 \approx_{exp} \ \Gamma, e_2$$

```
1 ≈ 0 + 1
```

```
sum(2) ≈ 3
```

```
fun() -> 1 ≈ fun() -> 0 + 1
```

```
fun(X) -> e ≈ fun(X) -> (fun(X) -> e)(X)
```

```
0 ≉ letrec 'f'/0 = fun() -> apply 'f'/0 in apply 'f'/0()
```

# Coq embedding

- We have: Core Erlang semantics
- The relations above are not accepted by the positivity checker
- Workaround [4]
  - Define the parts of the relation piece-by-piece
  - Assemble the relation with a Fixpoint
  - However, it recurses over the type : there is no typing in Erlang/Core Erlang
  - Solution: use the size of the terms instead

[4] Culpepper R., Cobb A. (2017) Contextual Equivalence for Probabilistic Programs with Continuous Random Variables and Scoring. In: Yang H. (eds) Programming Languages and Systems. ESOP 2017. Lecture Notes in Computer Science, vol 10201. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-54434-1_14

# Contact

*daniel-h@elte.hu*

*berpeti@inf.elte.hu*

# GitHub

*https://github.com/harp-project*

``Application Domain Specific Highly Reliable IT Solutions'' project has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

This work was supported by the project "Integrált kutatói utánpótlás-képzési program az informatika és számítástudomány diszciplináris területein (Integrated program for training new generation of researchers in the disciplinary fields of computer science) ", No. EFOP-3.6.3-VEKOP-16-2017-00002. The project has been supported by the European Union and co-funded by the European Social Fund.

SZÉCHENYI 2020

HUNGARIAN GOVERNMENT

European Union
European Social Fund

INVESTING IN YOUR FUTURE