A PROOF ASSISTANT BASED FORMALISATION OF CORE ERLANG

Péter Bereczky et al.
Eötvös Loránd University
Budapest, Hungary

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

INTRODUCTION

INTRODUCTION

Motivation: refactorings

INTRODUCTION

Motivation: refactorings

- DSL: high level refactorings
- Arguing about correctness → formal semantics of target language

INTRODUCTION

Motivation: refactorings

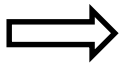
- DSL: high level refactorings
- Arguing about correctness → formal semantics of target language
- Why is a formal argument needed? It seems so simple:

INTRODUCTION

Motivation: refactorings

- DSL: high level refactorings
- Arguing about correctness → formal semantics of target language
- Why is a formal argument needed? It seems so simple:

```
let X = 5 in
  let Y = 4 in
    X + Y
```



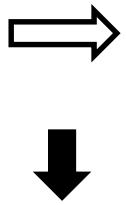
```
let X = 4 in
  let Y = 5 in
    X + Y
```

INTRODUCTION

Motivation: refactorings

- DSL: high level refactorings
- Arguing about correctness → formal semantics of target language
- Why is a formal argument needed? It seems so simple:

```
let X = 5 in
  let Y = 4 in
    X + Y
```



```
let X = 4 in
  let Y = 5 in
    X + Y
```

INTRODUCTION

Motivation: refactorings

- DSL: high level refactorings
- Arguing about correctness → formal semantics of target language
- Why is a formal argument needed? It seems so simple:

```
let X = 5 in
  let Y = 4 in
    X + Y
```

⇨

```
let X = 4 in
  let Y = 5 in
    X + Y
```

⇩

```
let X = e1 in
  let Y = e2 in
    X + Y
```

⇨

```
let X = e2 in
  let Y = e1 in
    X + Y
```

INTRODUCTION

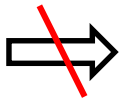
Motivation: refactorings

- DSL: high level refactorings
- Arguing about correctness → formal semantics of target language
- Why is a formal argument needed? It seems so simple:

```
let X = 5 in
  let Y = 4 in
    X + Y
```

```
let X = 4 in
  let Y = 5 in
    X + Y
```

```
let X = e1 in
  let Y = e2 in
    X + Y
```

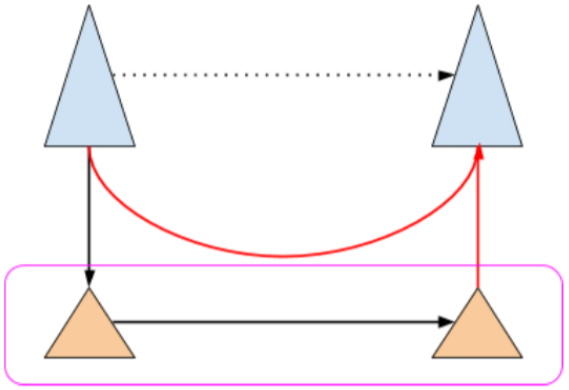


```
let X = e2 in
  let Y = e1 in
    X + Y
```

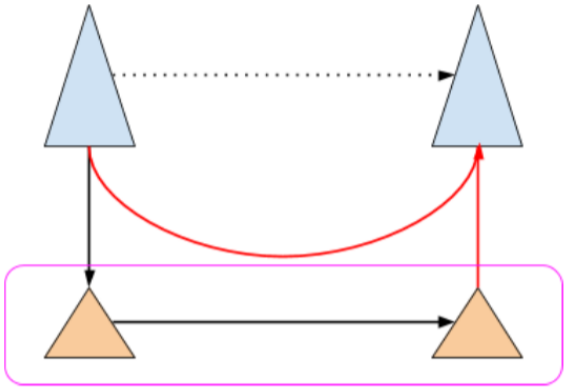
What if X is already bound?

INTRODUCTION – HARP PROJECT

INTRODUCTION – HARP PROJECT

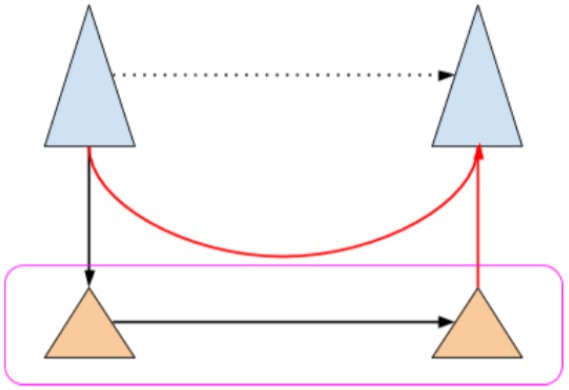


INTRODUCTION – HARP PROJECT



Verified core of a refactoring system

INTRODUCTION – HARP PROJECT



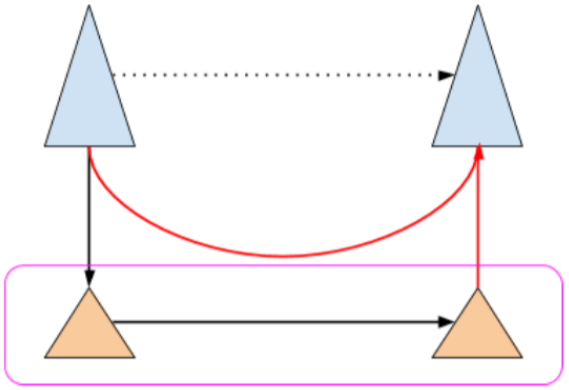
Core Erlang formalisation

AML formalisation

DSL formalisation

Verified core of a refactoring system

INTRODUCTION – HARP PROJECT



Core Erlang formalisation

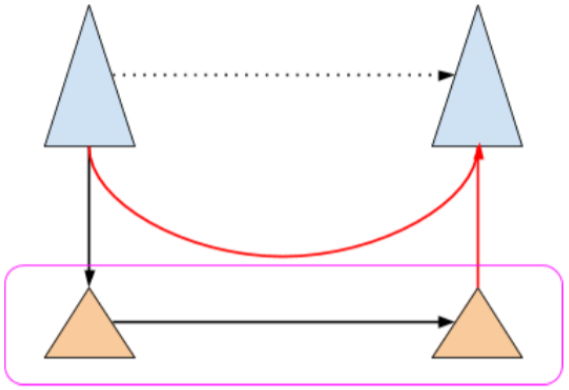
Erlang formalisation

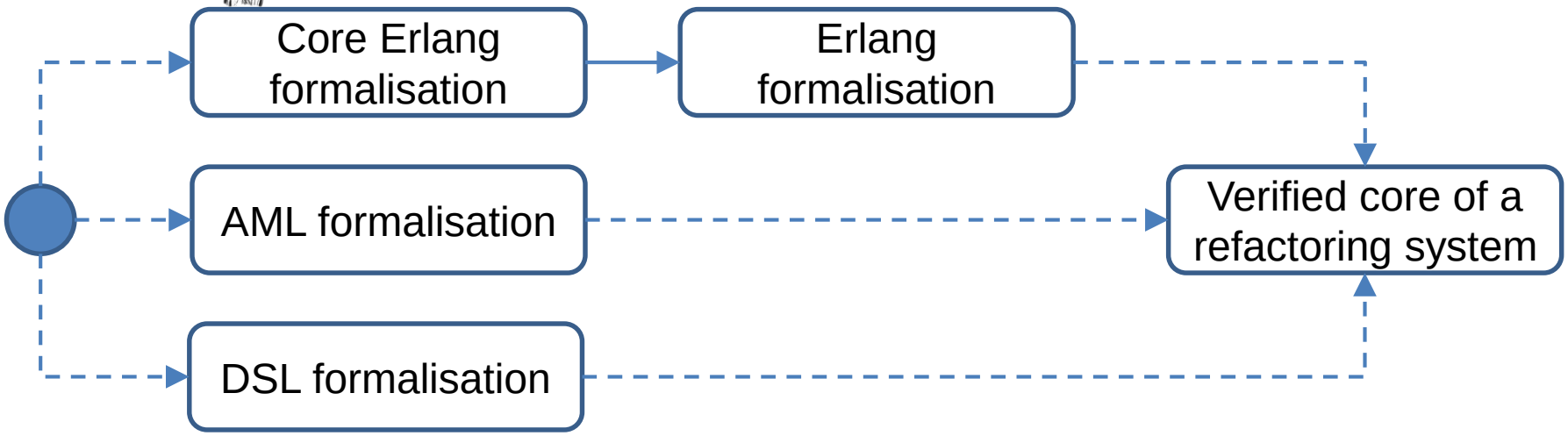
AML formalisation

DSL formalisation

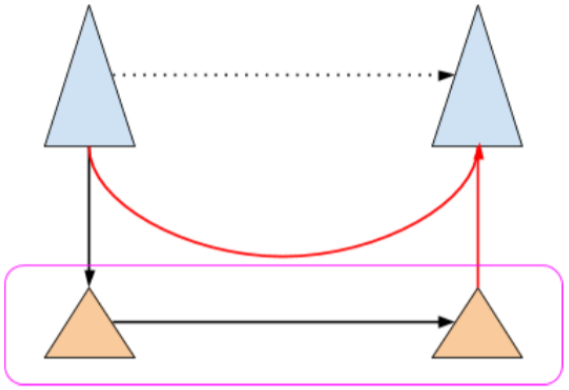
Verified core of a refactoring system

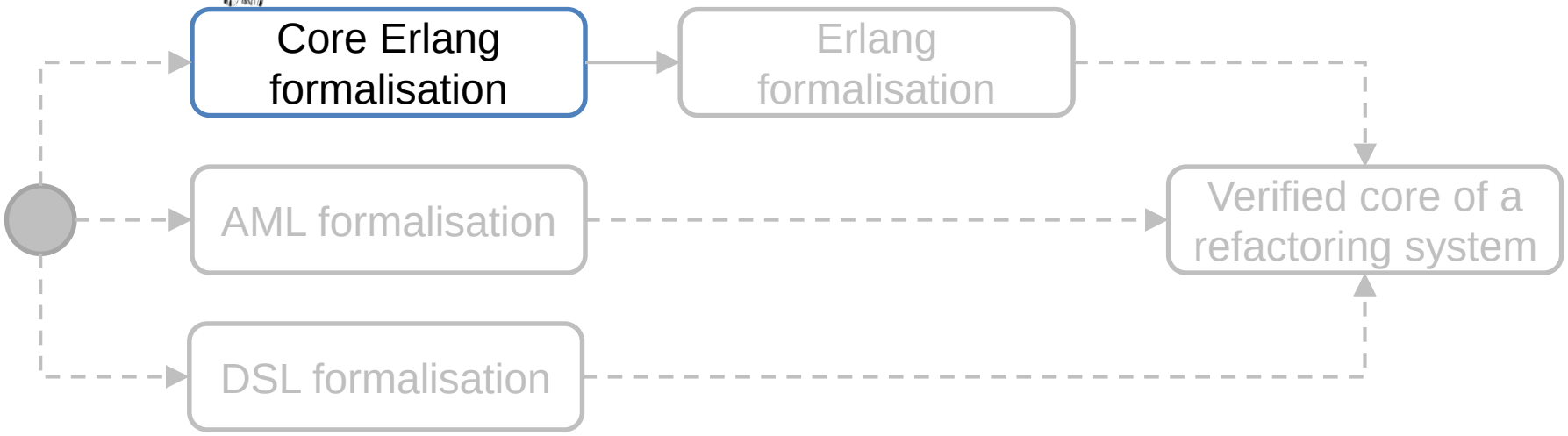
INTRODUCTION – HARP PROJECT





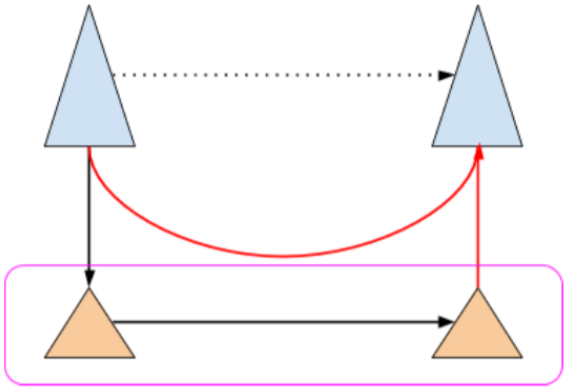
INTRODUCTION – HARP PROJECT – CORE ERLANG





INTRODUCTION – HARP PROJECT – CORE ERLANG

Formal semantics and abstractions



Core Erlang formalisation

Erlang formalisation

AML formalisation

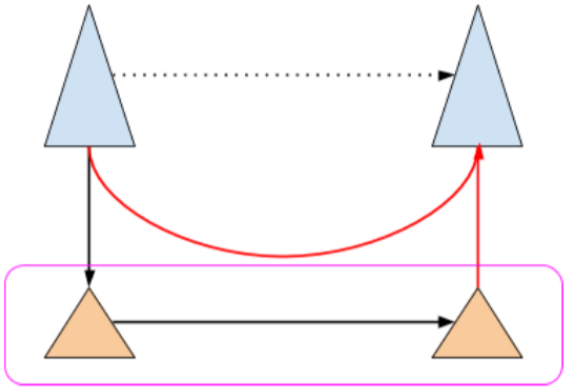
DSL formalisation

Verified core of a refactoring system

INTRODUCTION – HARP PROJECT – CORE ERLANG

Formal semantics and abstractions

Examples and tests



Core Erlang formalisation

Erlang formalisation

AML formalisation

DSL formalisation

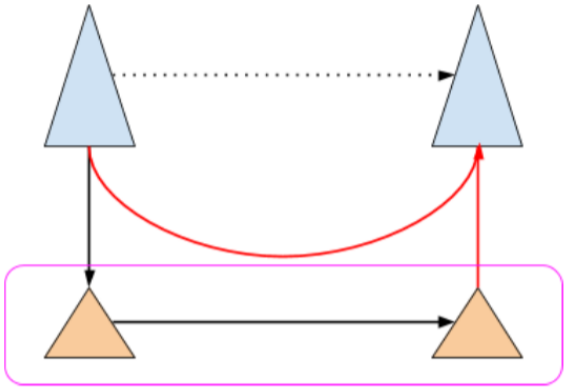
Verified core of a refactoring system

INTRODUCTION – HARP PROJECT – CORE ERLANG

Formal semantics and abstractions

Examples and tests

Equivalence proofs



Core Erlang formalisation

Erlang formalisation

AML formalisation

DSL formalisation

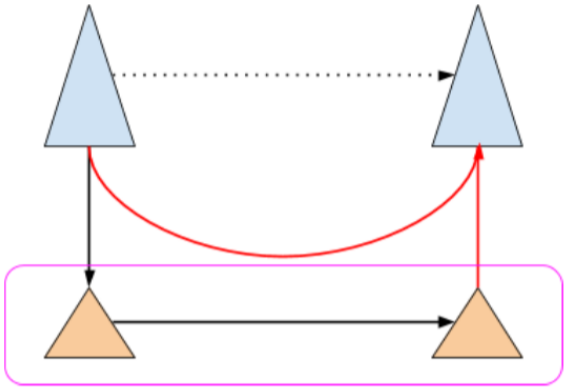
Verified core of a refactoring system

INTRODUCTION – HARP PROJECT – CORE ERLANG

Formal semantics and abstractions

Examples and tests

Equivalence proofs



Core Erlang formalisation

Erlang formalisation

AML formalisation

DSL formalisation

Verified core of a refactoring system

Machine-checked formalisation and proofs by using Coq

CORE ERLANG - ERLANG

CORE ERLANG - ERLANG

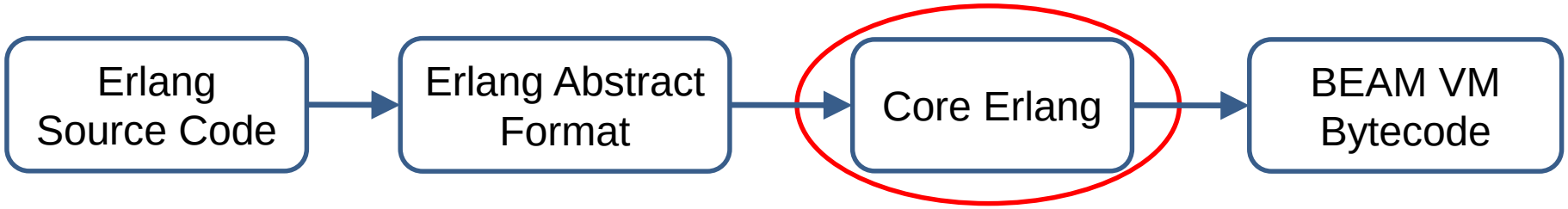
Core Erlang is suitable:

CORE ERLANG - ERLANG

Core Erlang is suitable:

- It is a subset of Erlang

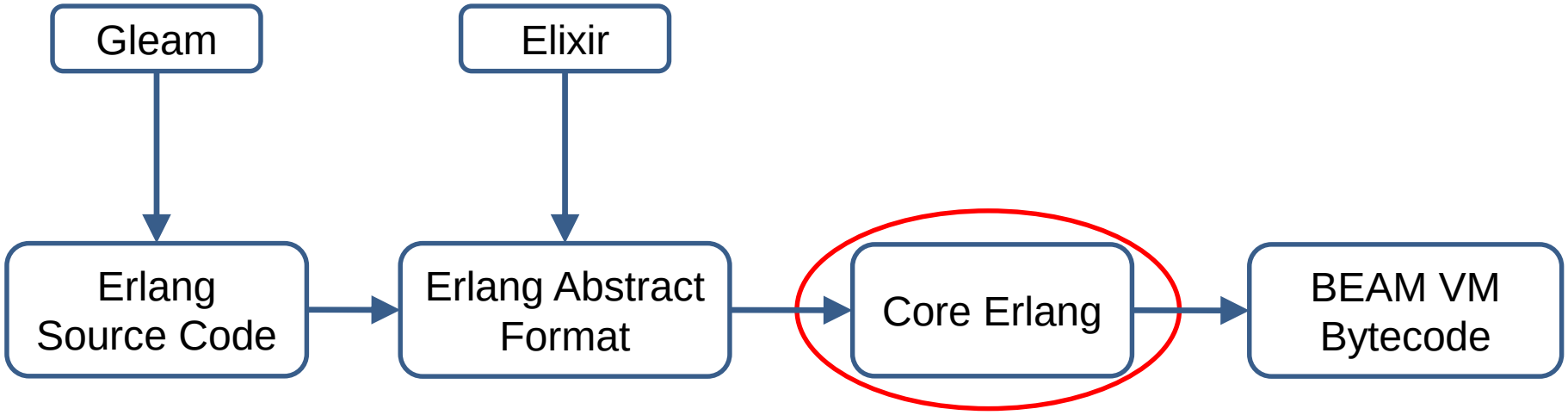
CORE ERLANG - ERLANG



Core Erlang is suitable:

- It is a subset of Erlang
- Erlang translates to Core Erlang

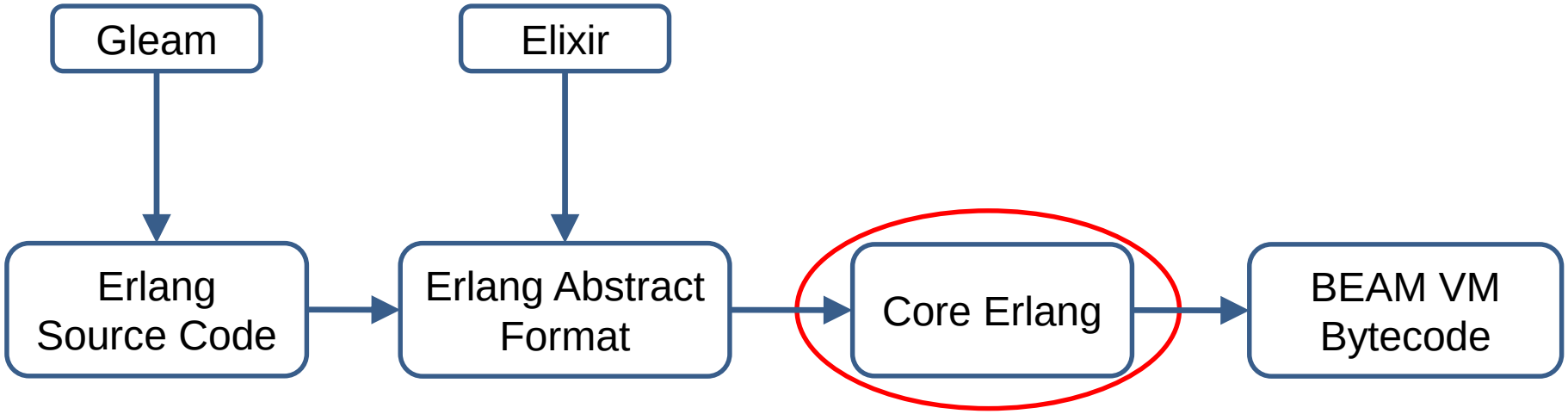
CORE ERLANG - ERLANG



Core Erlang is suitable:

- It is a subset of Erlang
- Erlang translates to Core Erlang
- Modern functional languages (e.g. Elixir, Gleam, LFE) can be translated to Core Erlang

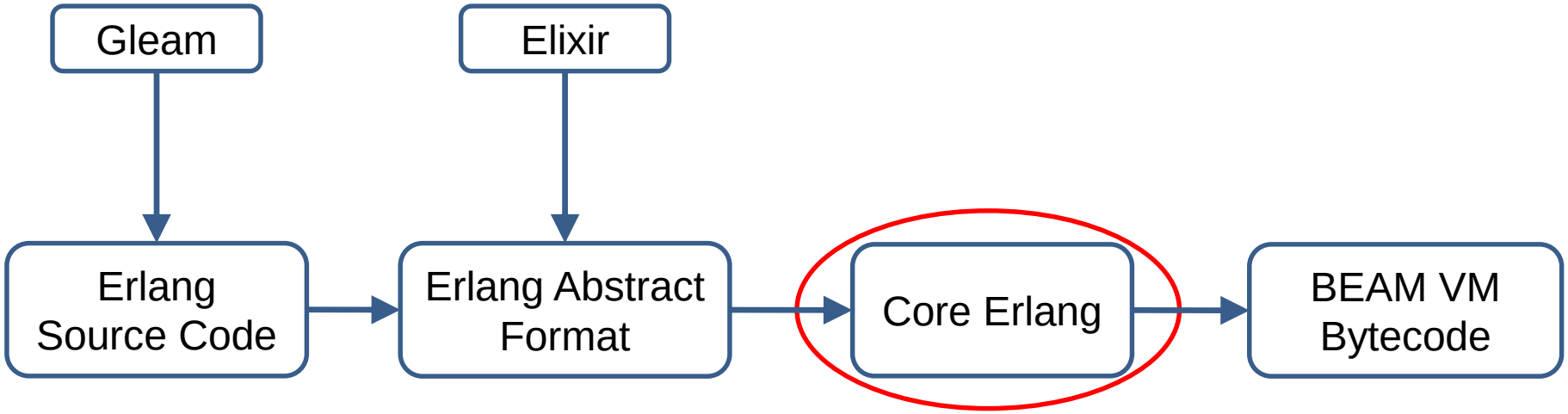
CORE ERLANG - ERLANG



Core Erlang is suitable:

- It is a subset of Erlang
- Erlang translates to Core Erlang
- Modern functional languages (e.g. Elixir, Gleam, LFE) can be translated to Core Erlang
- Existing sources, formalisations

CORE ERLANG - ERLANG



Core Erlang is suitable:

- It is a subset of Erlang
- Erlang translates to Core Erlang
- Modern functional languages (e.g. Elixir, Gleam, LFE) can be translated to Core Erlang
- Existing sources, formalisations

We chose a sequential subset of Core Erlang to formalise, based on the existing sources (papers, reference implementation, language documentation).

SYNTAX

SYNTAX

```
Module 'm' ['f1'/0, 'f2'/0,
'f3'/1]

'f1'/0 = fun() -> ~{}~


'f2'/0 = fun() ->
  let X = fun() -> {1,2,3}
    in apply X()


'f3'/1 = fun(X) ->
  case X of
    <5> when 'true' -> X + 1
  end
```

SYNTAX

```
Module 'm' ['f1'/0, 'f2'/0,
'f3'/1]

'f1'/0 = fun() -> ~{}~


'f2'/0 = fun() ->
  let X = fun() -> {1,2,3}
    in apply X()


'f3'/1 = fun(X) ->
  case X of
    <5> when 'true' -> X + 1
  end
```

```
Inductive Literal : Type :=
| Atom (s : string)
| Integer (z : Z)
| EmptyList
| EmptyTuple
| EmptyMap.
```

SYNTAX

```
Module 'm' ['f1'/0, 'f2'/0,
'f3'/1]

'f1'/0 = fun() -> ~{}~


'f2'/0 = fun() ->
  let X = fun() -> {1,2,3}
    in apply X()


'f3'/1 = fun(X) ->
  case X of
    <5> when 'true' -> X + 1
  end
```

```
Inductive Literal : Type :=
| Atom (s : string)
| Integer (z : Z)
| EmptyList
| EmptyTuple
| EmptyMap.

Inductive Pattern : Type :=
| PVar (v : Var)
| PLiteral (l : Literal)
| PList (hd tl : Pattern)
| PTuple (t : Tuple)
with Tuple : Type :=
| TNil
| TCons (hd : Pattern) (tl : Tuple).
```

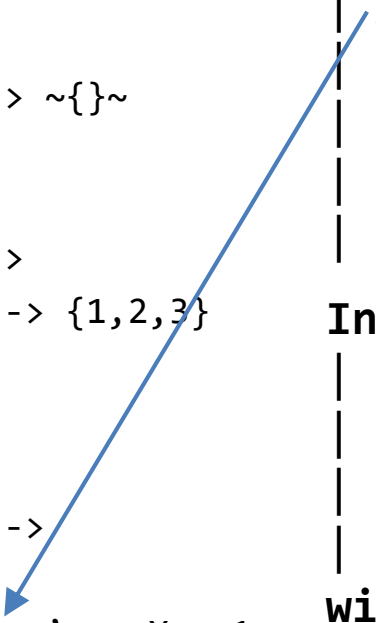
SYNTAX

```
Module 'm' ['f1'/0, 'f2'/0,
'f3'/1]

'f1'/0 = fun() -> ~{}~


'f2'/0 = fun() ->
  let X = fun() -> {1,2,3}
    in apply X()


'f3'/1 = fun(X) ->
  case X of
    <5> when 'true' -> X + 1
  end
```



```
Inductive Literal : Type :=
| Atom (s : string)
| Integer (z : Z)
| EmptyList
| EmptyTuple
| EmptyMap.

Inductive Pattern : Type :=
| PVar (v : Var)
| PLiteral (l : Literal)
| PList (hd tl : Pattern)
| PTuple (t : Tuple)
with Tuple : Type :=
| TNil
| TCons (hd : Pattern) (tl : Tuple).
```

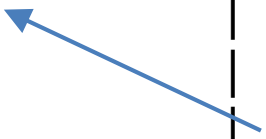
SYNTAX

```
Module 'm' ['f1'/0, 'f2'/0,
'f3'/1]

'f1'/0 = fun() -> ~{}~


'f2'/0 = fun() ->
  let X = fun() -> {1,2,3}
    in apply X()


'f3'/1 = fun(X) ->
  case X of
    <5> when 'true' -> X + 1
  end
```



```
Inductive Literal : Type :=
| Atom (s : string)
| Integer (z : Z)
| EmptyList
| EmptyTuple
| EmptyMap.

Inductive Pattern : Type :=
| PVar (v : Var)
| PLiteral (l : Literal)
| PList (hd tl : Pattern)
| PTuple (t : Tuple)
with Tuple : Type :=
| TNil
| TCons (hd : Pattern) (tl : Tuple).
```

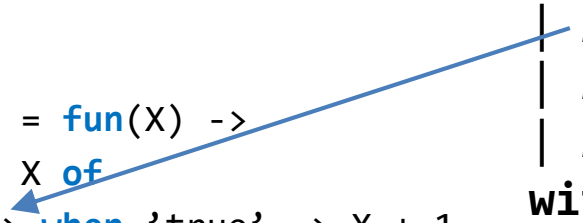
SYNTAX

```
Module 'm' ['f1'/0, 'f2'/0,
'f3'/1]

'f1'/0 = fun() -> ~{}~


'f2'/0 = fun() ->
  let X = fun() -> {1,2,3}
    in apply X()


'f3'/1 = fun(X) ->
  case X of
    <5> when 'true' -> X + 1
  end
```



```
Inductive Literal : Type :=
| Atom (s : string)
| Integer (z : Z)
| EmptyList
| EmptyTuple
| EmptyMap.

Inductive Pattern : Type :=
| PVar (v : Var)
| PLiteral (l : Literal)
| PList (hd tl : Pattern)
| PTuple (t : Tuple)
with Tuple : Type :=
| TNil
| TCons (hd : Pattern) (tl : Tuple).
```

SYNTAX

```
Module 'm' ['f1'/0, 'f2'/0,
'f3'/1]

'f1'/0 = fun() -> ~{}~


'f2'/0 = fun() ->
  let X = fun() -> {1,2,3}
    in apply X()


'f3'/1 = fun(X) ->
  case X of
    <5> when 'true' -> X + 1
  end
```

```
Inductive Expression : Type :=
| ELiteral (l : Literal)
…
| ETuple (l : list Expression)
| ECall (f : string) (l : list Expression)
| EApply (name : Expression) (l : list
Expression)
| ECase (e : Expression) (l : list Clause)
| ELet (s : list Var) (el : list Expression)
(e : Expression)
…
with Clause : Type :=
| CCons (p : Pattern) (guard e : Expression).
```

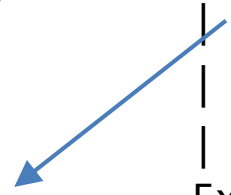
SYNTAX

```
Module 'm' ['f1'/0, 'f2'/0,
'f3'/1]

'f1'/0 = fun() -> ~{}~


'f2'/0 = fun() ->
  let X = fun() -> {1,2,3}
    in apply X()


'f3'/1 = fun(X) ->
  case X of
    <5> when 'true' -> X + 1
  end
```



```
Inductive Expression : Type :=
| ELiteral (l : Literal)
…
| ETuple (l : list Expression)
| ECall (f : string) (l : list Expression)
| EApply (name : Expression) (l : list
Expression)
| ECase (e : Expression) (l : list Clause)
| ELet (s : list Var) (el : list Expression)
(e : Expression)
…
with Clause : Type :=
| CCons (p : Pattern) (guard e : Expression).
```

SYNTAX

```
Module 'm' ['f1'/0, 'f2'/0,
'f3'/1]

'f1'/0 = fun() -> ~{}~


'f2'/0 = fun() ->
  let X = fun() -> {1,2,3}
    in apply X()


'f3'/1 = fun(X) ->
  case X of
    <5> when 'true' -> X + 1
  end
```

```
Inductive Expression : Type :=
| ELiteral (l : Literal)
…
| ETuple (l : list Expression)
| ECall (f : string) (l : list Expression)
| EApply (name : Expression) (l : list
Expression)
| ECase (e : Expression) (l : list Clause)
| ELet (s : list Var) (el : list Expression)
(e : Expression)
…
with Clause : Type :=
| CCons (p : Pattern) (guard e : Expression).
```

SYNTAX – VALUES - NORMALFORM

SYNTAX – VALUES - NORMALFORM

```
{}
{1, 2, 3}
[14|[43|[45]]]
X
~{1 => 5}~
{1, 2, 3, 5 + 6}
{1, 2, 3, X}
```

SYNTAX – VALUES - NORMALFORM

✓
```
{}
{1, 2, 3}
[14|[43|[45]]]
X
~{1 => 5}~
{1, 2, 3, 5 + 6}
{1, 2, 3, X}
```

SYNTAX – VALUES - NORMALFORM

✓ `{}`

✓ `{1, 2, 3}`

`[14|[43|[45]]]`

`X`

`~{1 => 5}~`

`{1, 2, 3, 5 + 6}`

`{1, 2, 3, X}`

SYNTAX – VALUES - NORMALFORM

✓ `{}`

✓ `{1, 2, 3}`

✓ `[14|[43|[45]]]`

`X`

`~{1 => 5}~`

`{1, 2, 3, 5 + 6}`

`{1, 2, 3, X}`

SYNTAX – VALUES - NORMALFORM

✓ `{}`
✓ `{1, 2, 3}`
✓ `[14|[43|[45]]]`
✗ `X`
`~{1 => 5}~`
`{1, 2, 3, 5 + 6}`
`{1, 2, 3, X}`

SYNTAX – VALUES - NORMALFORM

✓ `{}`
✓ `{1, 2, 3}`
✓ `[14|[43|[45]]]`
✗ `X`
✓ `~{1 => 5}~`
`{1, 2, 3, 5 + 6}`
`{1, 2, 3, X}`

SYNTAX – VALUES - NORMALFORM

✓ `{}`
✓ `{1, 2, 3}`
✓ `[14|[43|[45]]]`
✗ `X`
✓ `~{1 => 5}~`
✗ `{1, 2, 3, 5 + 6}`
  `{1, 2, 3, X}`

SYNTAX – VALUES - NORMALFORM

✓ `{}`
✓ `{1, 2, 3}`
✓ `[14|[43|[45]]]`
✗ `X`
✓ `~{1 => 5}~`
✗ `{1, 2, 3, 5 + 6}`
✗ `{1, 2, 3, X}`

SYNTAX – VALUES - NORMALFORM

- ✓ `{}`
- ✓ `{1, 2, 3}`
- ✓ `[14|[43|[45]]]`
- ✗ `X`
- ✓ `~{1 => 5}~`
- ✗ `{1, 2, 3, 5 + 6}`
- ✗ `{1, 2, 3, X}`

```
Inductive Value : Type :=
| VLiteral (l : Literal)
| VClosure (ref : Environment + FunctionSignature)
  (params: list Var) (body : Expression)
| VList (vhd vtl : Value)
| VTuple (vl : list Value)
| VMap (kl vl : list Value).
```

SYNTAX – VALUES - NORMALFORM

✓ `{}`
✓ `{1, 2, 3}`
✓ `[14|[43|[45]]]`
✗ `X`
✓ `~{1 => 5}~`
✗ `{1, 2, 3, 5 + 6}`
✗ `{1, 2, 3, X}`

```
Inductive Value : Type :=
| VLiteral (l : Literal)
| VClosure (ref : Environment + FunctionSignature)
  (params: list Var) (body : Expression)
| VList (vhd vtl : Value)
| VTuple (vl : list Value)
| VMap (kl vl : list Value).
```

Values are a subset of patterns		
+ Less code duplication + Big-step semantics with domain changing		
- Expressions evaluate to values, not patterns - Closure representation		

SYNTAX – VALUES - NORMALFORM

✓ `{}`
✓ `{1, 2, 3}`
✓ `[14|[43|[45]]]`
✗ `X`
✓ `~{1 => 5}~`
✗ `{1, 2, 3, 5 + 6}`
✗ `{1, 2, 3, X}`

```
Inductive Value : Type :=
| VLiteral (l : Literal)
| VClosure (ref : Environment + FunctionSignature)
  (params: list Var) (body : Expression)
| VList (vhd vtl : Value)
| VTuple (vl : list Value)
| VMap (kl vl : list Value).
```

Values are a subset of patterns	Values are an implicit subset of expressions	
+ Less code duplication + Big-step semantics with domain changing	+ Reusable code + Big-step semantics with domain changing	
- Expressions evaluate to values, not patterns - Closure representation	- Closure representation	

SYNTAX – VALUES - NORMALFORM

✓ `{}`
✓ `{1, 2, 3}`
✓ `[14|[43|[45]]]`
✗ `X`
✓ `~{1 => 5}~`
✗ `{1, 2, 3, 5 + 6}`
✗ `{1, 2, 3, X}`

```
Inductive Value : Type :=
| VLiteral (l : Literal)
| VClosure (ref : Environment + FunctionSignature)
  (params: list Var) (body : Expression)
| VList (vhd vtl : Value)
| VTuple (vl : list Value)
| VMap (kl vl : list Value).
```

Values are a subset of patterns	Values are an implicit subset of expressions	Values are an explicit subset of expressions
+ Less code duplication + Big-step semantics with domain changing	+ Reusable code + Big-step semantics with domain changing	+ Less code duplication
- Expressions evaluate to values, not patterns - Closure representation	- Closure representation	- Semantics is a relation from Expressions to Expressions - Closure representation - Hard to prove statements this way in Coq

DYNAMIC SEMANTICS - ENVIRONMENT

DYNAMIC SEMANTICS - ENVIRONMENT

```
a) let X = 5 in X
b) let Y = fun() -> 5 in apply Y()
c) letrec 'fun1'/1 = fun(X) -> X in
       apply 'fun1/1'(Y)
```

DYNAMIC SEMANTICS - ENVIRONMENT

```
a) let X = 5 in X
b) let Y = fun() -> 5 in apply Y()
c) letrec 'fun1'/1 = fun(X) -> X in
       apply 'fun1/1'(Y)
```

To evaluate variables and function calls, an execution environment is needed.

DYNAMIC SEMANTICS - ENVIRONMENT

```
a) let X = 5 in X
b) let Y = fun() -> 5 in apply Y()
c) letrec 'fun1'/1 = fun(X) -> X in
       apply 'fun1/1'(Y)
```

This is a mapping from variables or signatures to values.

To evaluate variables and function calls, an execution environment is needed.

DYNAMIC SEMANTICS - ENVIRONMENT

```
a) let X = 5 in X
b) let Y = fun() -> 5 in apply Y()
c) letrec 'fun1'/1 = fun(X) -> X in
       apply 'fun1/1'(Y)
```

This is a mapping from variables or signatures to values.

To evaluate variables and function calls, an execution environment is needed.

```
a)  {X : 5}
b)  {Y : some closure}
c)  {'fun1'/1 : some closure, Y -> ?}
```

DYNAMIC SEMANTICS - ENVIRONMENT

```
a) let X = 5 in X
b) let Y = fun() -> 5 in apply Y()
c) letrec 'fun1'/1 = fun(X) -> X in
       apply 'fun1/1'(Y)
```

To evaluate variables and function calls, an execution environment is needed.

This is a mapping from variables or signatures to values.

```
a)  {X : 5}
b)  {Y : some closure}
c)  {'fun1'/1 : some closure, Y -> ?}
```

`get_value`

`append_{funs|vars}_to_env (or '+' in the presentation)`

To reach and modify the environment getter and update methods are needed.

DYNAMIC SEMANTICS - CLOSURES

DYNAMIC SEMANTICS - CLOSURES

```
let X = 5 in
  let Z = fun(Y) -> Y + X in
    let X = 10 in
      apply Z(0)
```

DYNAMIC SEMANTICS - CLOSURES

```
let X = 5 in
  let Z = fun(Y) -> Y + X in
    let X = 10 in
      apply Z(0)
```

Application environment:

{X : 5, Y : 0} or
{X : 10, Y : 0}?

DYNAMIC SEMANTICS - CLOSURES

```
let X = 5 in
  let Z = fun(Y) -> Y + X in
    let X = 10 in
      apply Z(0)
```

Application environment:

{X : 5, Y : 0} or
{X : 10, Y : 0}?

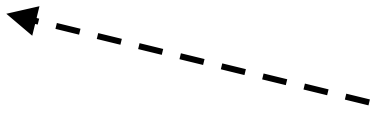
Static binding should be applied here

DYNAMIC SEMANTICS - CLOSURES

```
let X = 5 in
  let Z = fun(Y) -> Y + X in
    let X = 10 in
      apply Z(0)
```

Application environment:

{X : 5, Y : 0} or
{X : 10, Y : 0}?



Static binding should be applied here

- Closures can be applied

DYNAMIC SEMANTICS - CLOSURES

```
let X = 5 in
  let Z = fun(Y) -> Y + X in
    let X = 10 in
      apply Z(0)
```

Application environment:

{X : 5, Y : 0} or
{X : 10, Y : 0}?

Static binding should be applied here

- Closures can be applied
- Closure captures:
  - Body expression
  - Parameter list
  - Evaluation environment

DYNAMIC SEMANTICS - CLOSURES

```
let X = 5 in
  let Z = fun(Y) -> Y + X in
    let X = 10 in
      apply Z(0)
```

Application environment:

`{X : 5, Y : 0}` or
`{X : 10, Y : 0}`?

Static binding should be applied here

- Closures can be applied
- Closure captures:
  - Body expression
  - Parameter list
  - Evaluation environment
- In this case: `{Z : VClosure {X : 5} [Y] (Y + X), …}`

DYNAMIC SEMANTICS - CLOSURES

```
let X = 5 in
  let Z = fun(Y) -> Y + X in
    let X = 10 in
      apply Z(0)
```

Application environment:

{X : 5, Y : 0} or
{X : 10, Y : 0}?

Static binding should be applied here

- Closures can be applied
- Closure captures:
  - Body expression
  - Parameter list
  - Evaluation environment
- In this case: {Z : *VClosure* {X : 5} [Y] (Y + X), …}

DYNAMIC SEMANTICS - CLOSURES

```
let X = 5 in
  let Z = fun(Y) -> Y + X in
    let X = 10 in
      apply Z(0)
```

Application environment:

{X : 5, Y : 0} or
{X : 10, Y : 0}?

Static binding should be applied here

- Closures can be applied
- Closure captures:
  - Body expression
  - Parameter list
  - Evaluation environment
- In this case: {Z : *VClosure* {X : 5} [Y] (Y + X), …}
- The evaluation of **apply** Z(0) must happen in the environment stored in the closure (extended with Y)

DYNAMIC SEMANTICS – RECURSIVE CLOSURES

DYNAMIC SEMANTICS – RECURSIVE CLOSURES

Is the previous approach always usable?

DYNAMIC SEMANTICS – RECURSIVE CLOSURES

Is the previous approach always usable?

```
letrec 'f1'/0 = fun() -> apply 'f1'/0()
```

DYNAMIC SEMANTICS – RECURSIVE CLOSURES

Is the previous approach always usable?

```
letrec 'f1'/0 = fun() -> apply 'f1'/0()
```

```
VClosure {'f1'/0 : VClosure {'f1'/0 : ?} …} [] (apply 'f1'/0())
```

DYNAMIC SEMANTICS – RECURSIVE CLOSURES

Is the previous approach always usable?

```
letrec 'f1'/0 = fun() -> apply 'f1'/0()
```

Recursive

```
VClosure {'f1'/0 : VClosure {'f1'/0 : ?} …} [] (apply 'f1'/0())
```

DYNAMIC SEMANTICS – RECURSIVE CLOSURES

Is the previous approach always usable?

```
letrec 'f1'/0 = fun() -> apply 'f1'/0()
```

Recursive

```
VClosure {'f1'/0 : VClosure {'f1'/0 : ?} …} [] (apply 'f1'/0())
```

- Solve with references:
  ```
  VClosure 'f1'/0 [] (apply 'f1'/0())
  ```

DYNAMIC SEMANTICS – RECURSIVE CLOSURES

Is the previous approach always usable?

```
letrec 'f1'/0 = fun() -> apply 'f1'/0()
```

Recursive

```
VClosure {'f1'/0 : VClosure {'f1'/0 : ?} …} [] (apply 'f1'/0())
```

- Solve with references:
  `VClosure 'f1'/0 [] (apply 'f1'/0())`
- The evaluation environment is stored in the closure environment
  - A mapping from function signatures to environments

DYNAMIC SEMANTICS – RECURSIVE CLOSURES

Is the previous approach always usable?

```
letrec 'f1'/0 = fun() -> apply 'f1'/0()
```

Recursive

```
VClosure {'f1'/0 : VClosure {'f1'/0 : ?} …} [] (apply 'f1'/0())
```

- Solve with references:
  `VClosure 'f1'/0 [] (apply 'f1'/0())`
- The evaluation environment is stored in the closure environment
  - A mapping from function signatures to environments
- In this case:
  `{'f1'/0 : {'f1'/0 : VClosure 'f1'/0 [] (apply 'f1'/0())}}`

DYNAMIC SEMANTICS

DYNAMIC SEMANTICS

- Big-step, operational semantics

DYNAMIC SEMANTICS

- Big-step, operational semantics
- Type in Coq:
  `Environment → Closures → Expression → Value → Prop`

DYNAMIC SEMANTICS

- Big-step, operational semantics
- Type in Coq:
  `Environment → Closures → Expression → Value → Prop`

$$\frac{}{(\Gamma, \Delta, ELiteral\ l) \xrightarrow{e} VLiteral\ l}$$

DYNAMIC SEMANTICS

- Big-step, operational semantics
- Type in Coq:
  `Environment → Closures → Expression → Value → Prop`

$$\frac{}{(\Gamma, \Delta, ELiteral\ l) \xrightarrow{e} VLiteral\ l}$$

$$\frac{(\Gamma, \Delta, hd) \xrightarrow{e} hdv \qquad (\Gamma, \Delta, tl) \xrightarrow{e} tlv}{(\Gamma, \Delta, EList\ hd\ tl) \xrightarrow{e} VList\ hdv\ tlv}$$

DYNAMIC SEMANTICS

- Big-step, operational semantics
- Type in Coq:
  `Environment → Closures → Expression → Value → Prop`

$$\frac{}{(\Gamma, \Delta, ELiteral\ l) \xrightarrow{e} VLiteral\ l}$$

$$\frac{(\Gamma, \Delta, hd) \xrightarrow{e} hdv \qquad (\Gamma, \Delta, tl) \xrightarrow{e} tlv}{(\Gamma, \Delta, EList\ hd\ tl) \xrightarrow{e} VList\ hdv\ tlv}$$

- Theorems: determinism, and its helpers

DYNAMIC SEMANTICS

- Big-step, operational semantics
- Type in Coq:
  `Environment → Closures → Expression → Value → Prop`

$$\frac{}{(\Gamma, \Delta, ELiteral\ l) \xrightarrow{e} VLiteral\ l}$$

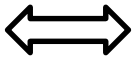
$$\frac{(\Gamma, \Delta, hd) \xrightarrow{e} hdv \qquad (\Gamma, \Delta, tl) \xrightarrow{e} tlv}{(\Gamma, \Delta, EList\ hd\ tl) \xrightarrow{e} VList\ hdv\ tlv}$$

- Theorems: determinism, and its helpers
- Lemmas about helper functions (e.g. reflexivity and symmetry of the equivalence of objects of union, product types)

EQUIVALENCE EXAMPLES

EQUIVALENCE EXAMPLES

```
let <X, Y> = <e1, e2> in
  X + Y
```



```
let <X, Y> = <e2, e1> in
  X + Y
```

EQUIVALENCE EXAMPLES

```
let <X, Y> = <e1, e2> in
  X + Y
```

⟺

```
let <X, Y> = <e2, e1> in
  X + Y
```

Formalise the problem:

```
(Γ,Δ,ELet ["X";"Y"] [e1;e2] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
→
(Γ,Δ,ELet ["X";"Y"] [e2;e1] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
```

EQUIVALENCE EXAMPLES

```
let <X, Y> = <e1, e2> in
  X + Y
```

⟺

```
let <X, Y> = <e2, e1> in
  X + Y
```

Formalise the problem:

```
(Γ,Δ,ELet ["X";"Y"] [e1;e2] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
→
(Γ,Δ,ELet ["X";"Y"] [e2;e1] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
```

Idea:

1. Deconstruct the hypothesis for *ELet*

EQUIVALENCE EXAMPLES

```
let <X, Y> = <e1, e2> in
  X + Y
```

⟺

```
let <X, Y> = <e2, e1> in
  X + Y
```

Formalise the problem:

```
(Γ,Δ,ELet ["X";"Y"] [e1;e2] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
→
(Γ,Δ,ELet ["X";"Y"] [e2;e1] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
```

Idea:

1. Deconstruct the hypothesis for *ELet*
2. Deconstruct the hypothesis for *ECall*

EQUIVALENCE EXAMPLES

```
let <X, Y> = <e1, e2> in
  X + Y
```

⟺

```
let <X, Y> = <e2, e1> in
  X + Y
```

Formalise the problem:

```
(Γ,Δ,ELet ["X";"Y"] [e1;e2] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
→
(Γ,Δ,ELet ["X";"Y"] [e2;e1] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
```

Idea:

1. Deconstruct the hypothesis for *ELet*
2. Deconstruct the hypothesis for *ECall*
3. Prove evaluation using a derivation tree

EQUIVALENCE EXAMPLES

```
let <X, Y> = <e1, e2> in
  X + Y
```

⟺

```
let <X, Y> = <e2, e1> in
  X + Y
```

Formalise the problem:

```
(Γ,Δ,ELet ["X";"Y"] [e1;e2] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
→
(Γ,Δ,ELet ["X";"Y"] [e2;e1] (ECall "plus" [EVar "X";EVar "Y"])) -e> t
```

Idea:

1. Deconstruct the hypothesis for *ELet*
2. Deconstruct the hypothesis for *ECall*
3. Prove evaluation using a derivation tree

This idea can be used in most cases

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

Hypotheses:

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

Hypotheses:

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

Hypotheses:

- H1: (Γ,Δ,e_1) –e> v_1

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

Hypotheses:

- H1: (Γ,Δ,e_1) -e> v_1
- H2: (Γ,Δ,e_2) -e> v_2

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

Hypotheses:

- H1: (Γ,Δ,e_1) -e> v_1
- H2: (Γ,Δ,e_2) -e> v_2
- H3: (Γ + {X : v_1, Y : v_2},Δ,*ECall* …) -e> t

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ params\ vals \qquad eval\ fname\ vals = v}{(\Gamma, \Delta, ECall\ fname\ params) \xrightarrow{e} v}$$

Hypotheses:

- H1: (Γ,Δ,e_1) -e> v_1
- H2: (Γ,Δ,e_2) -e> v_2
- H3: (Γ + {X : v_1, Y : v_2},Δ,*ECall* …) -e> t

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ params\ vals \qquad eval\ fname\ vals = v}{(\Gamma, \Delta, ECall\ fname\ params) \xrightarrow{e} v}$$

Hypotheses:

- H1: (Γ,Δ,e_1) –e> v_1
- H2: (Γ,Δ,e_2) –e> v_2
- H3: (Γ + {X : v_1, Y : v_2},Δ,*ECall* …) –e> t
  - H4: (Γ + {X : v_1, Y : v_2},Δ,*EVar* ”X”) –e> v_1

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ params\ vals \qquad eval\ fname\ vals = v}{(\Gamma, \Delta, ECall\ fname\ params) \xrightarrow{e} v}$$

Hypotheses:

- H1: (Γ,Δ,e_1) -e> v_1
- H2: (Γ,Δ,e_2) -e> v_2
- H3: (Γ + {X : v_1, Y : v_2},Δ,*ECall* …) -e> t
  - H4: (Γ + {X : v_1, Y : v_2},Δ,*EVar* ”X”) -e> v_1
  - H5: (Γ + {X : v_1, Y : v_2},Δ,*EVar* ”Y”) -e> v_2

DECONSTRUCT HYPOTHESIS

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ params\ vals \qquad eval\ fname\ vals = v}{(\Gamma, \Delta, ECall\ fname\ params) \xrightarrow{e} v}$$

Hypotheses:

- H1: (Γ,Δ,e_1) -e> v_1
- H2: (Γ,Δ,e_2) -e> v_2
- H3: (Γ + {X : v_1, Y : v_2},Δ,*ECall* …) -e> t
  - H4: (Γ + {X : v_1, Y : v_2},Δ,*EVar* ”X”) -e> v_1
  - H5: (Γ + {X : v_1, Y : v_2},Δ,*EVar* ”Y”) -e> v_2
  - H6: eval ”plus” [v_1;v_2] = t

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

---

$$(\Gamma, \Delta, let <X, Y>=<e_2, e_1> in\ X + Y) \xrightarrow{e} t$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t
→
(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

$$\frac{}{(\Gamma, \Delta, let\ <X, Y>=<e_2, e_1>\ in\ X + Y) \xrightarrow{e} t}$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t
→
(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

$$\frac{(\Gamma, \Delta, e_2) \xrightarrow{e} v_2 \qquad (\Gamma, \Delta, e_1) \xrightarrow{e} v_1 \qquad (\Gamma + \{X : v_2, Y : v_1\}, \Delta, X + Y) \xrightarrow{e} t}{(\Gamma, \Delta, let <X, Y>=<e_2, e_1> in\ X + Y) \xrightarrow{e} t}$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t
→
(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

H2✓

$$\frac{}{(\Gamma, \Delta, e_2) \xrightarrow{e} v_2} \qquad \frac{}{(\Gamma, \Delta, e_1) \xrightarrow{e} v_1} \qquad \frac{}{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X + Y) \xrightarrow{e} t}$$

$$(\Gamma, \Delta, let <X, Y> = <e_2, e_1> in\ X + Y) \xrightarrow{e} t$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

$$\frac{\text{H2}\checkmark \qquad \text{H1}\checkmark \qquad}{(\Gamma, \Delta, e_2) \xrightarrow{e} v_2 \qquad (\Gamma, \Delta, e_1) \xrightarrow{e} v_1 \qquad (\Gamma + \{X : v_2, Y : v_1\}, \Delta, X + Y) \xrightarrow{e} t}{(\Gamma, \Delta, let <X, Y> = <e_2, e_1> in\ X + Y) \xrightarrow{e} t}$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

$$(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X + Y) \xrightarrow{e} t$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t
→
(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ params\ vals \qquad eval\ fname\ vals = v}{(\Gamma, \Delta, ECall\ fname\ params) \xrightarrow{e} v}$$

$$(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X + Y) \xrightarrow{e} t$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t
→
(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{eval\_all\ \Gamma\ \Delta\ params\ vals \qquad eval\ fname\ vals = v}{(\Gamma, \Delta, ECall\ fname\ params) \xrightarrow{e} v}$$

$$\frac{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X) \xrightarrow{e} v_2 \qquad (\Gamma + \{X : v_2, Y : v_1\}, \Delta, Y) \xrightarrow{e} v_1 \qquad eval\ "plus"\ [v_2, v_1] = t}{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X + Y) \xrightarrow{e} t}$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t
→
(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{}{(\Gamma, \Delta, EVar\ s) \xrightarrow{e} get\_value\ \Gamma\ (inl\ s)}$$

$$\frac{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X) \xrightarrow{e} v_2 \qquad (\Gamma + \{X : v_2, Y : v_1\}, \Delta, Y) \xrightarrow{e} v_1 \qquad eval\ "plus"\ [v_2, v_1] = t}{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X + Y) \xrightarrow{e} t}$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t
→
(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{}{(\Gamma, \Delta, EVar\ s) \xrightarrow{e} get\_value\ \Gamma\ (inl\ s)}$$

$$\frac{\dfrac{get\_value\ \Gamma + \{X : v_2, Y : v_1\}\ X = v_2}{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X) \xrightarrow{e} v_2} \quad \dfrac{get\_value\ \Gamma + \{X : v_2, Y : v_1\}\ Y = v_1}{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, Y) \xrightarrow{e} v_1} \quad \dfrac{}{eval\ "plus"\ [v_2, v_1] = t}}{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X + Y) \xrightarrow{e} t}$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t
→
(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) -e> t

$$\frac{}{(\Gamma, \Delta, EVar\ s) \xrightarrow{e} get\_value\ \Gamma\ (inl\ s)}$$

The append operation overwrites existing variable-value bindings

$$\frac{\dfrac{\dfrac{}{get\_value\ \Gamma + \{X : v_2, Y : v_1\}\ X = v_2}}{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X) \xrightarrow{e} v_2} \quad \dfrac{\dfrac{}{get\_value\ \Gamma + \{X : v_2, Y : v_1\}\ Y = v_1}}{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, Y) \xrightarrow{e} v_1} \quad \dfrac{}{eval\ "plus"\ [v_2, v_1] = t}}{(\Gamma + \{X : v_2, Y : v_1\}, \Delta, X + Y) \xrightarrow{e} t}$$

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

- Remaining: eval ”plus” [v_2;v_1] = t

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

- Remaining: eval ”plus” [v_2;v_1] = t
- According to H6: eval ”plus” [v_1;v_2] = t

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

- Remaining: eval ”plus” [v_2;v_1] = t
- According to H6: eval ”plus” [v_1;v_2] = t
- eval ”plus” [v_1;v_2] = eval ”plus” [v_2;v_1]

BUILD DERIVATION TREE

(Γ,Δ,*ELet* [”X”;”Y”] [e_1;e_2] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

→

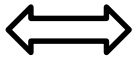
(Γ,Δ,*ELet* [”X”;”Y”] [e_2;e_1] (*ECall* ”plus” [*EVar* ”X”;*EVar* ”Y”])) –e> t

- Remaining: eval ”plus” [v_2;v_1] = t
- According to H6: eval ”plus” [v_1;v_2] = t
- eval ”plus” [v_1;v_2] = eval ”plus” [v_2;v_1]
  - The eval auxiliary function is commutative regarding the ”plus” operation

OTHER EXAMPLES

OTHER EXAMPLES

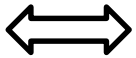
```
let X = e1 in
  let Y = e2 in
    X + Y
```



```
let X = e2 in
  let Y = e1 in
    X + Y
```

OTHER EXAMPLES

```
let X = e1 in
  let Y = e2 in
    X + Y
```

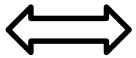


```
let X = e2 in
  let Y = e1 in
    X + Y
```

- This equivalence cannot be proved in this form

OTHER EXAMPLES

```
let X = e1 in
  let Y = e2 in
    X + Y
```

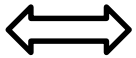


```
let X = e2 in
  let Y = e1 in
    X + Y
```

- This equivalence cannot be proved in this form
- Additional hypotheses are needed:

OTHER EXAMPLES

```
let X = e1 in
  let Y = e2 in
    X + Y
```



```
let X = e2 in
  let Y = e1 in
    X + Y
```

- This equivalence cannot be proved in this form
- Additional hypotheses are needed:
  - `(Γ,Δ,e1) -e> v1` and `(Γ+{X : x},Δ,e1) -e> v1`
  - This is needed for e_2 too

OTHER EXAMPLES

```
let X = e1 in
  let Y = e2 in
    X + Y
```

$\Longleftrightarrow$

```
let X = e2 in
  let Y = e1 in
    X + Y
```

- This equivalence cannot be proved in this form
- Additional hypotheses are needed:
  - `(Γ,Δ,e1) -e> v1` and `(Γ+{X : x},Δ,e1) -e> v1`
  - This is needed for e_2 too

`e` $\Longleftrightarrow$

```
let X = fun() -> e in
  apply X()
```

OTHER EXAMPLES

```
let X = e1 in
  let Y = e2 in
    X + Y
```

$\Longleftrightarrow$

```
let X = e2 in
  let Y = e1 in
    X + Y
```

- This equivalence cannot be proved in this form
- Additional hypotheses are needed:
  - `(Γ,Δ,e1) -e> v1` and `(Γ+{X : x},Δ,e1) -e> v1`
  - This is needed for e_2 too

`e` $\Longleftrightarrow$

```
let X = fun() -> e in
  apply X()
```

The concrete proofs are available on Github

CONCLUSION

CONCLUSION

Contributions

CONCLUSION

Contributions

A formal description of a part of sequential Core Erlang

CONCLUSION

Contributions

A formal description of a part of sequential Core Erlang

Essential proofs about the properties of the presented formalisation (e.g. determinism)

CONCLUSION

Contributions

A formal description of a part of sequential Core Erlang

Essential proofs about the properties of the presented formalisation (e.g. determinism)

Example program formal descriptions and proofs about the meaning-preservation of refactorings

CONCLUSION

Contributions

A formal description of a part of sequential Core Erlang

Essential proofs about the properties of the presented formalisation (e.g. determinism)

Example program formal descriptions and proofs about the meaning-preservation of refactorings

Some Possible Future Work

CONCLUSION

Contributions

A formal description of a part of sequential Core Erlang

Essential proofs about the properties of the presented formalisation (e.g. determinism)

Example program formal descriptions and proofs about the meaning-preservation of refactorings

Some Possible Future Work

Extension of the formalisation with additional statements (e.g. try, do)

CONCLUSION

Contributions

A formal description of a part of sequential Core Erlang

Essential proofs about the properties of the presented formalisation (e.g. determinism)

Example program formal descriptions and proofs about the meaning-preservation of refactorings

Some Possible Future Work

Extension of the formalisation with additional statements (e.g. try, do)

Error handling (and the introduction of exceptions) and logging side effects

CONCLUSION

Contributions

A formal description of a part of sequential Core Erlang

Essential proofs about the properties of the presented formalisation (e.g. determinism)

Example program formal descriptions and proofs about the meaning-preservation of refactorings

Some Possible Future Work

Extension of the formalisation with additional statements (e.g. try, do)

Error handling (and the introduction of exceptions) and logging side effects

Formalisation of the sequential part of Erlang

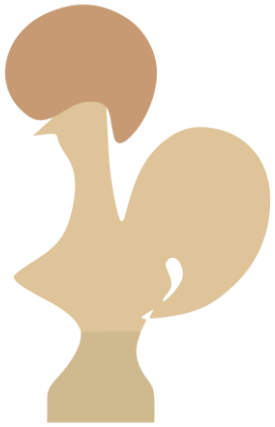
Github repository:
https://github.com/harp-project/Core-Erlang-Formalization

Contact:
berpeti@inf.elte.hu

This work was supported by the „Thematic fundamental research collaborations grounding innovation in informatics and infocommunications (3in)” EFOP-3.6.2-16-2017-00013 project, supported by the European Union, co-financed by the European Social Fund.

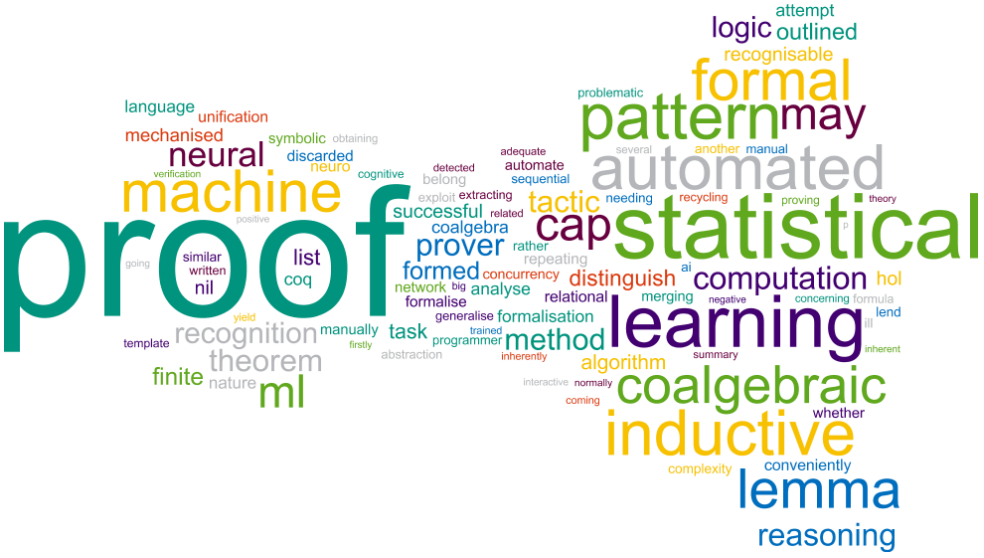
BACKUP

COQ IMPLEMENTATION



- Inductive types for expressions, patterns, value judgment and operational semantics
- Induction principles for expressions and operational semantics

- Definitions for environment, closures and their functions (update, getter)
- Essential theorems and their proofs



PROPERTIES – INDUCTION PRINCIPLES

Coq automatically generates induction principles

Type	Coq induction	What we want

PROPERTIES – INDUCTION PRINCIPLES

Coq automatically generates induction principles

Type	Coq induction	What we want

`Inductive Nat : Type :=`
`| O`
`| S (n : Nat).`
	1. Prove P for O
	2. Prove P for (S n) with (P n)
		The Coq principle is correct.

PROPERTIES – INDUCTION PRINCIPLES

Coq automatically generates induction principles.

Type	Coq induction	What we want

`Inductive Nat : Type :=`
`| O`
`| S (n : Nat).`
	1. Prove P for O
	2. Prove P for (S n) with (P n)
		The Coq principle is correct.

`(* RoseTree *)`
`Inductive RT : Type :=`
`| Nil`
`| Node (chs : list RT).`
	1. Prove P for Nil
	2. Prove P for (Node chs)
		1. Prove P for Nil
		2. Prove P for (Node chs) with ∀c ∈ chs: (P c)

When using nested types, Coq can not guess what principles do we need.
Solution: writing principles by hand

This was needed for Expressions and the operational semantics, because both have nested stdlib lists.

INDUCTION PRINCIPLES

```
Inductive T : list A → Prop :=
| Cons1 : T nil
| Cons2 (l : list A) : ∀i∈[0..length l – 1]: B v T nil →T l.
```

1. Prove P for Cons1
2. Prove P for (Cons2 l) with ∀i∈[0..length l – 1]: B v T nil.

This cannot be rewritten to ∀i∈[0..length l – 1]: B and ∀i∈[0..length l – 1]: T

DYNAMIC SEMANTICS

- Strict (everything evaluates to values first)
  - Core Erlang is strict
  - First, every part of an expression must be evaluated, then it can be evaluated
- Left-to-right

$$
\begin{aligned}
&\mathit{eval\_all}\ \Gamma\ \Delta\ \mathit{exps}\ \mathit{vals} := \\
&\quad \mathit{length}\ \mathit{exps} = \mathit{length}\ \mathit{vals} \rightarrow \\
&\quad (\forall\ \mathit{exp} : \mathit{Expression}, \forall\ \mathit{val} : \mathit{Value}, \\
&\qquad \mathit{In}\ (\mathit{exp}, \mathit{val})\ (\mathit{combine}\ \mathit{exps}\ \mathit{vals}) \rightarrow \\
&\qquad\quad (\Gamma, \Delta, \mathit{exp}) \xrightarrow{e} \mathit{val})
\end{aligned}
$$

FORMAL EVALUATION EXAMPLE

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

$$(\varnothing, \varnothing, \text{let } X = 42 \text{ in let } Y = fun() \rightarrow X \text{ } in \text{ } let \text{ } X = 5 \text{ } in \text{ } apply \text{ } Y()) \xrightarrow{e} 42$$

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

$$(\varnothing, \varnothing, \text{let } X = 42 \text{ in let } Y = fun() \rightarrow X \text{ in let } X = 5 \text{ in apply } Y()) \xrightarrow{e} 42$$

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

$$\frac{}{(\{X : 42\}, \varnothing, \text{let } Y = fun() \rightarrow X \text{ in let } X = 5 \text{ in apply } Y()) \xrightarrow{e} 42}$$

$$\frac{(\{X : 42\}, \varnothing, \text{let } Y = fun() \rightarrow X \text{ in let } X = 5 \text{ in apply } Y()) \xrightarrow{e} 42}{(\varnothing, \varnothing, \text{let } X = 42 \text{ in let } Y = fun() \rightarrow X \text{ in let } X = 5 \text{ in apply } Y()) \xrightarrow{e} 42}$$

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

$$\frac{\dfrac{\dfrac{}{(\{X : 42, Y : VClosure\ \{X : 42\}\ []\ X\}, \varnothing, let\ X\ = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\{X : 42\}, \varnothing, \text{let } Y = fun() \rightarrow X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\varnothing, \varnothing, \text{let } X = 42 \text{ in let } Y = fun() \rightarrow X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}$$

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

$$\frac{eval\_all\ \Gamma\ \Delta\ exps\ vals \qquad (append\_vars\_to\_env\ vars\ vals\ \Gamma, \Delta, e) \xrightarrow{e} v}{(\Gamma, \Delta, ELet\ vars\ exps\ e) \xrightarrow{e} v}$$

$$\frac{\dfrac{\dfrac{\dfrac{}{(\{X:5, Y:\ VClosure\ \{X:42\}\ []\ X\}, \varnothing, apply\ Y()) \xrightarrow{e} 42}}{(\{X:42, Y:\ VClosure\ \{X:42\}\ []\ X\}, \varnothing, let\ X\ = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\{X:42\}, \varnothing, \text{let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\varnothing, \varnothing, \text{let } X = 42 \text{ in let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}$$

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

$$\frac{eval\_all\ \Gamma\ \Delta\ params\ vals \qquad (\Gamma, \Delta, exp) \xrightarrow{e} VClosure\ ref\ var\_list\ body \qquad (append\_vars\_to\_env\ var\_list\ vals\ (get\_env\ ref\ \Delta\ \Gamma), \Delta, body) \xrightarrow{e} v}{(\Gamma, \Delta, EApply\ exp\ params) \xrightarrow{e} v}$$

$$\frac{\dfrac{\dfrac{\dfrac{}{(\{X:5, Y:VClosure\ \{X:42\}\ []\ X\}, \varnothing, apply\ Y()) \xrightarrow{e} 42}}{(\{X:42, Y:VClosure\ \{X:42\}\ []\ X\}, \varnothing, let\ X\ = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\{X:42\}, \varnothing, \text{let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\varnothing, \varnothing, \text{let } X = 42 \text{ in let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}$$

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

$$\frac{eval\_all\ \Gamma\ \Delta\ params\ vals \qquad (\Gamma, \Delta, exp) \xrightarrow{e} VClosure\ ref\ var\_list\ body \qquad (append\_vars\_to\_env\ var\_list\ vals\ (get\_env\ ref\ \Delta\ \Gamma), \Delta, body) \xrightarrow{e} v}{(\Gamma, \Delta, EApply\ exp\ params) \xrightarrow{e} v}$$

$$\frac{\dfrac{\dfrac{\dfrac{}{(\{X : 42\}, \varnothing, X) \xrightarrow{e} 42}}{(\{X : 5, Y : VClosure\ \{X : 42\}\ []\ X\}, \varnothing, apply\ Y()) \xrightarrow{e} 42}}{(\{X : 42, Y : VClosure\ \{X : 42\}\ []\ X\}, \varnothing, let\ X\ = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\{X : 42\}, \varnothing, \text{let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\varnothing, \varnothing, \text{let } X = 42 \text{ in let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}$$

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

$$\frac{}{(\Gamma, \Delta, EVar\ s) \xrightarrow{e} get\_value\ \Gamma\ (inl\ s)}$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{(\{X : 42\}, \varnothing, X) \xrightarrow{e} 42}}{(\{X : 5, Y : VClosure\ \{X : 42\}\ []\ X\}, \varnothing, apply\ Y()) \xrightarrow{e} 42}}{(\{X : 42, Y : VClosure\ \{X : 42\}\ []\ X\}, \varnothing, let\ X\ = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\{X : 42\}, \varnothing, \text{let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\varnothing, \varnothing, \text{let } X = 42 \text{ in let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}$$

FORMAL EVALUATION EXAMPLE

```
let X = 42 in
  let Y = fun() -> X in
    let X = 5 in
      apply Y()
→ 42
```

$$\frac{}{(\Gamma, \Delta, EVar\ s) \xrightarrow{e} get\_value\ \Gamma\ (inl\ s)}$$

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{get\_value\ \{X : 42\}\ X = 42}}{(\{X : 42\}, \varnothing, X) \xrightarrow{e} 42}}{(\{X : 5, Y : VClosure\ \{X : 42\}\ []\ X\}, \varnothing, apply\ Y()) \xrightarrow{e} 42}}{(\{X : 42, Y : VClosure\ \{X : 42\}\ []\ X\}, \varnothing, let\ X\ = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\{X : 42\}, \varnothing, \text{let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}}{(\varnothing, \varnothing, \text{let } X = 42 \text{ in let } Y = fun() \to X\ in\ let\ X = 5\ in\ apply\ Y()) \xrightarrow{e} 42}$$