Formalizing Real World Programming Languages with Skeletal Semantics

Alan Schmitt

September 18, 2023

Wat

A lightning talk by Gary Bernhardt from CodeMash 2012

```
failbowl:~(master!?) $ jsc
> [] + []

> [] + {}
[object Object]
> {} + []
0
> {} + {}
NaN
>
```

Programming Language Techniques for JavaScript Isolation

Sergio Maffeis
EPSRC Research Fellow, Imperial College London.
Currently visiting INRIA Rocquencourt.

In collaboration with:
John Mitchell, Ankur Taly (Stanford University),
Philippa Gardner, Gareth Smith (Imperial College London).

Rennes, November 4, 2011.

JSCert

Two JavaScript semantics in Coq

descriptive: given a program and a result, say if they are related

executable: given a program, compute the result

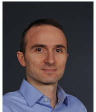
Correctness

If program P executes to v, then P and v are related

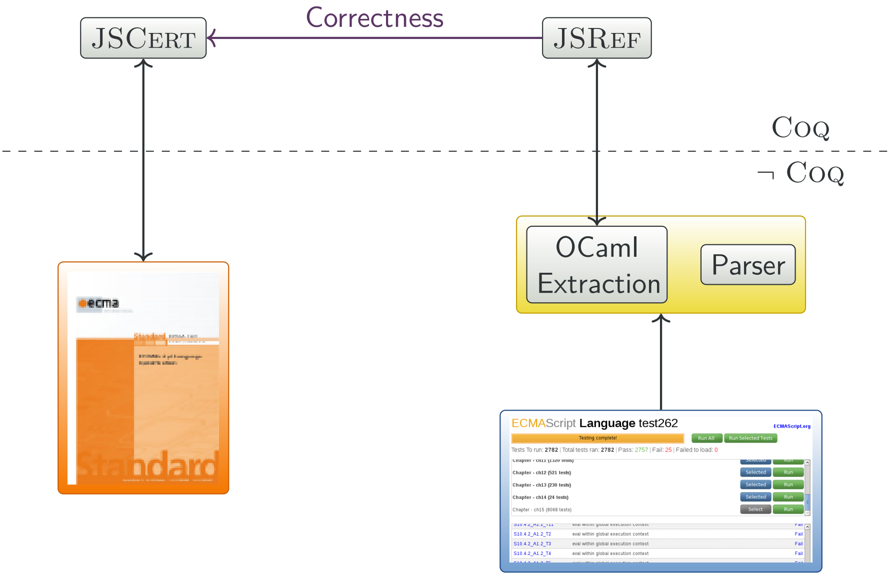
- 2 years, 8 people
- 18 klocs of Coq







JSCert



Correctness
JSCert
JSRef
Coq
¬ Coq
OCaml Extraction
Parser
ecma
Standard
ECMAScript Language test262
ECMAScript.org
Testing complete!
Run All
Run Selected Tests
Tests To run: 2782 | Total tests ran: 2782 | Pass: 2757 | Fail: 25 | Failed to load: 0
Chapter - ch12 (521 tests)
Chapter - ch13 (230 tests)
Chapter - ch14 (24 tests)
Chapter - ch15 (8068 tests)
S10.4.2_A1.2_T2 eval within global execution context Fail
S10.4.2_A1.2_T3 eval within global execution context Fail
S10.4.2_A1.2_T4 eval within global execution context Fail

12.6.2 The `while` Statement

The production *IterationStatement* **: while (** *Expression* **)** *Statement* is evaluated as follows:

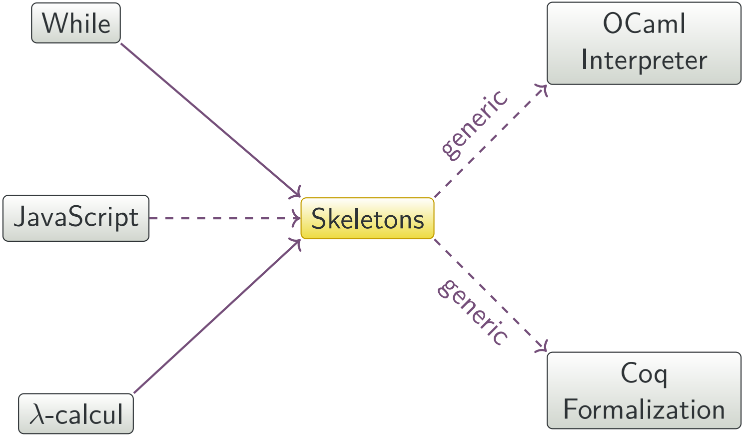
1. Let *V* = empty.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. If ToBoolean(GetValue(*exprRef*)) is **false**, return (normal, *V*, empty).
   c. Let *stmt* be the result of evaluating *Statement*.
   d. If *stmt*.value is not empty, let *V* = *stmt*.value.
   e. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
      i. If *stmt*.type is break and *stmt*.target is in the current label set, then
         1. Return (normal, *V*, empty).
      ii. If *stmt* is an abrupt completion, return *stmt*.

```
| red_stat_while : forall S C labs e1 t2 o,
    red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
    red_stat S C (stat_while labs e1 t2) o

| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o

| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)

| red_stat_while_2_true : forall S0 S C labs e1 t2 rv o1 o,
    red_stat S C t2 o1 ->
    red_stat S C (stat_while_3 labs e1 t2 rv o1) o ->
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S true)) o

| red_stat_while_3 : forall rv S0 S C labs e1 t2 rv' R o,
    rv' = (If res_value R <> resvalue_empty then res_value R else rv) ->
    red_stat S C (stat_while_4 labs e1 t2 rv' R) o ->
    red_stat S0 C (stat_while_3 labs e1 t2 rv (out_ter S R)) o

| red_stat_while_4_continue : forall S C labs e1 t2 rv R o,
    res_type R = restype_continue /\ res_label_in R labs ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_4_not_continue : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_continue /\ res_label_in R labs) ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_5_break : forall S C labs e1 t2 rv R,
    res_type R = restype_break /\ res_label_in R labs ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) (out_ter S rv)

| red_stat_while_5_not_break : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_break /\ res_label_in R labs) ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o

| red_stat_while_6_abort : forall S C labs e1 t2 rv R,
    res_type R <> restype_normal ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) (out_ter S R)

| red_stat_while_6_normal : forall S C labs e1 t2 rv R o,
    res_type R = restype_normal ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o
```

JSCert: The Problem

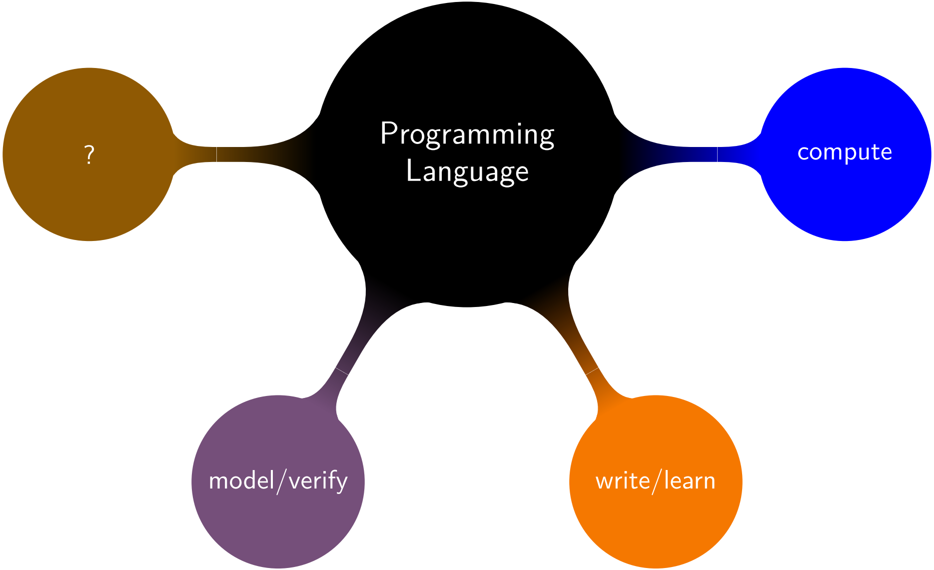
```
(** If statement (12.5) *)

| red_stat_if : forall S C e1 t2 t3opt y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_if_1 y1 t2 t3opt) o ->
    red_stat S C (stat_if e1 t2 t3opt) o

| red_stat_if_1_true : forall S0 S C t2 t3opt o,
    red_stat S C t2 o ->
    red_stat S0 C (stat_if_1 (vret S true) t2 t3opt) o

| red_stat_if_1_false : forall S0 S C t2 t3 o,
    red_stat S C t3 o ->
    red_stat S0 C (stat_if_1 (vret S false) t2 (Some t3)) o

| red_stat_if_1_false_implicit : forall S0 S C t2,
    red_stat S0 C (stat_if_1 (vret S false) t2 None) (out_ter S resvalue_empty)
```

- 900 mutually inductive rules
- inversion during an induction runs out of memory


While
JavaScript
λ-calcul
Skeletons
generic
generic
OCaml Interpreter
Coq Formalization

(POPL 2019)

The Meaning of Programs



Goals

What

- Computable
- Readable
- Maintainable
- Usable

How

- Syntactic description of semantics
- No silo, always have an escape hatch
- Simple, to be extended & reused
- Supports non-determinism and partiality
- Incremental or incomplete specification

Why Not?

- OCaml, Haskell, Scheme
- Coq, Isabelle
- Lem, Ott
- $\mathbb{K}$

Coupling too tight between definition and use

Skeletal Semantics

Specifying the Syntax of a Language

$n \in \mathit{Lit}$

$x \in \mathit{Ident}$

$$
\begin{aligned}
e ::= {} & n \mid x \mid e + e \\
\mid {} & e = e \mid !e
\end{aligned}
$$

$$
\begin{aligned}
s ::= {} & \texttt{skip} \mid x := e \mid s; s \\
\mid {} & \texttt{if } e \texttt{ then } s \texttt{ else } s \\
\mid {} & \texttt{while } e \texttt{ do } s
\end{aligned}
$$

Specifying the Syntax of a Language

$n \in \mathit{Lit}$

$x \in \mathit{Ident}$

$e ::= n \mid x \mid e + e$
$\quad \mid e = e \mid !e$

$s ::= \texttt{skip} \mid x := e \mid s; s$
$\quad \mid \texttt{if } e \texttt{ then } s \texttt{ else } s$
$\quad \mid \texttt{while } e \texttt{ do } s$

```
(* unspecified type *)
type lit
type ident
```

Specifying the Syntax of a Language

$n \in \mathit{Lit}$

$x \in \mathit{Ident}$

$e ::= n \mid x \mid e + e \mid e = e \mid\ !e$

$s ::= \texttt{skip} \mid x := e \mid s; s \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ e\ \texttt{do}\ s$

```
(* unspecified type *)
type lit
type ident
```

```
(* specified type *)
type expr =
| Const lit
| Var ident
| Plus (expr, expr)
| Equal (expr, expr)
| Not expr
```

```
(* specified type *)
type stmt =
| Skip
| Assign (ident, expr)
| Seq (stmt, stmt)
| If (expr, stmt, stmt)
| While (expr, stmt)
```

Specifying the Semantics of a Language

$$b ::= \texttt{tt} \mid \texttt{ff}$$

$$v ::= n \mid b$$

$$\sigma \in State$$

$$\sigma, e \Downarrow_e v$$

$$\sigma, s \Downarrow_s \sigma$$

Specifying the Semantics of a Language

$$b ::= \texttt{tt} \mid \texttt{ff}$$
$$v ::= n \mid b$$
$$\sigma \in State$$

```
type boolean = | True | False
type int
type value = | Int int | Bool boolean
type state
```

$$\sigma, e \Downarrow_e v$$
$$\sigma, s \Downarrow_s \sigma$$

Specifying the Semantics of a Language

$$b ::= \texttt{tt} \mid \texttt{ff}$$
$$v ::= n \mid b$$
$$\sigma \in State$$

```
type boolean = | True | False
type int
type value = | Int int | Bool boolean
type state
```

$$\sigma, e \Downarrow_e v$$
$$\sigma, s \Downarrow_s \sigma$$

```
(* specified term *)
val eval_expr ((st:state), (e:expr)) : value = ...

val eval_stmt ((st:state), (s:stmt)) : state = ...
```

```
(* specified term *)
val eval_expr ((st:state), (e:expr)) : value =
```

Specifying the Semantics of a Language

$$\frac{}{\sigma, n \Downarrow_e n}$$

```
(* specified term *)
val eval_expr ((st:state), (e:expr)) : value =

  let Const n = e in
  let i = int_of_lit n in
  Int i
```

```
(* unspecified term *)
val int_of_lit: lit → int
```

Specifying the Semantics of a Language

$$\frac{}{\sigma, n \Downarrow_e n}$$

$$\frac{\sigma, e_1 \Downarrow_e n_1 \qquad \sigma, e_2 \Downarrow_e n_2}{\sigma, e_1 + e_2 \Downarrow_e n_1 + n_2}$$

```
(* unspecified term *)
val int_of_lit: lit → int
val add: (int, int) → int
```

```
(* specified term *)
val eval_expr ((st:state), (e:expr)) : value =

  let Const n = e in
  let i = int_of_lit n in
  Int i

 let Plus(e1,e2) = e in
  let Int n1 = eval_expr (st, e1) in
  let Int n2 = eval_expr (st, e2) in
  let n = add (n1, n2) in
  Int n
```

Specifying the Semantics of a Language

$$\frac{}{\sigma, n \Downarrow_e n}$$

$$\frac{\sigma, e_1 \Downarrow_e n_1 \qquad \sigma, e_2 \Downarrow_e n_2}{\sigma, e_1 + e_2 \Downarrow_e n_1 + n_2}$$

```
(* unspecified term *)
val int_of_lit: lit → int
val add: (int, int) → int
```

```
(* specified term *)
val eval_expr ((st:state), (e:expr)) : value =
branch
  let Const n = e in
  let i = int_of_lit n in
  Int i
or
 let Plus(e1,e2) = e in
  let Int n1 = eval_expr (st, e1) in
  let Int n2 = eval_expr (st, e2) in
  let n = add (n1, n2) in
  Int n
end
```

Specifying the Semantics of a Language

$$\frac{\sigma, e \Downarrow_e \texttt{ff}}{\sigma, \texttt{while}\ \ e\ \ \texttt{do}\ \ s \Downarrow_s \sigma}$$

$$\frac{\sigma, e \Downarrow_e \texttt{tt} \qquad \sigma, s \Downarrow_s \sigma' \qquad \sigma', \texttt{while}\ \ e\ \ \texttt{do}\ \ s \Downarrow_s \sigma''}{\sigma, \texttt{while}\ \ e\ \ \texttt{do}\ \ s \Downarrow_s \sigma''}$$

```
val eval_stmt ((st:state), (s:stmt)) : state =
branch
  let While(e, s') = s in
  let Bool False = eval_expr (st, e) in
  st
or
  let While(e, s') = s in
  let Bool True = eval_expr (st, e) in
  let st' = eval_stmt (st, s') in
  eval_stmt (st', s)
or ...
end
```

Specifying the Semantics of a Language

$$\frac{\sigma, e \Downarrow_e \texttt{ff}}{\sigma, \texttt{while}\ \ e\ \ \texttt{do}\ \ s \Downarrow_s \sigma}$$

$$\frac{\sigma, e \Downarrow_e \texttt{tt} \qquad \sigma, s \Downarrow_s \sigma' \qquad \sigma', \texttt{while}\ \ e\ \ \texttt{do}\ \ s \Downarrow_s \sigma''}{\sigma, \texttt{while}\ \ e\ \ \texttt{do}\ \ s \Downarrow_s \sigma''}$$

```
val eval_stmt ((st:state), (s:stmt)) : state =
branch
  let While(e, s') = s in
  let Bool b = eval_expr (st, e) in
  branch
    let False = b in st
  or
    let True = b in
    let st' = eval_stmt (st, s') in
    eval_stmt (st', s)
  end
or ...
end
```

Higher Order

```
val eval_stmt ((st:state), (s:stmt)) : state = ...

(* is syntactic sugar for *)
val eval_stmt : (state, stmt) → state =
  λ (st, s) : (state, stmt) → ...



val app: nat → (nat → nat) → nat =
  λ x: nat →
  λ f: (nat → nat) →
  f x
```

Polymorphism

```
type list<a> =
| Nil | Cons (a, list<a>)

val map<a, b> ((f: (a → b)), (l: list<a>)) : list<b> =
  branch
    let Nil = l in Nil<b>
  or
    let Cons (x, xs) = l in
    let y = f x in
    let ys = map<a, b> (f, xs) in
    Cons<b> (y, ys)
  end
```

Monads

Language Monads

Polymorphism + first class functions is sufficient for monads

```
type st<a> = state → (a, state)

val ret<a> (v: a) : st<a> =
  λ s:state → (v, s)

val bind<a, b> ((w: st<a>), (f: a → st<b>)) : st<b> =
  λ s:state →
  let (v, s') = w s in
  let w' = f v in
  w' s'
```

GetValue(V)

1. ReturnIfAbrupt(V).
2. If V is not a Reference Record, return V.
3. If IsUnresolvableReference(V) is `true`, throw a `ReferenceError` exception.
4. If IsPropertyReference(V) is `true`, then
   1. Let baseObj be ! ToObject(V.`[[Base]]`).
   2. Return ? baseObj.`[[Get]]`(V.`[[ReferencedName]]`, GetThisValue(V)).
5. Else,
   1. Let base be V.`[[Base]]`
   2. Assert: base is an Environment Record.
   3. Return ? base.GetBindingValue(V.`[[ReferencedName]]`, V.`[[Strict]]`).

State Monad

```
type st<a> := state → (a, state)

val ret<a> (v: a) : st<a> =
  λ s:state → (v, s)

val bind<a, b> ((w: st<a>), (f: a → st<b>)) : st<b> =
  λ s:state →
  let (v, s') = w s in
  let w' = f v in
  w' s'

val eval_expr (e:expr) : st<value> = ...

val eval_stmt (s:stmt) : st<()> = ...
```

While in State Monad

```
val eval_stmt (s:stmt) : st<()> =
branch
  let While(e, s') = s in
  let w = eval_expr e in
  bind<value, ()> (w, λ Bool b : value →
  branch
    let False = b in ret<()> ()
  or
    let True = b in
    let w' = eval_stmt s' in
    bind<(),()> (w', λ () → eval_stmt s)
  end)
or ...
end
```

While in State Monad

```
val eval_stmt (s:stmt) : st<()> =
branch
  let While(e, s') = s in
  let Bool b =%bind eval_expr e in
  branch
    let False = b in ret<()> ()
  or
    let True = b in
    let () =%bind eval_stmt s' in
    eval_stmt s
  end
or ...
end
```

While in State Monad

```
binder @ := bind

val eval_stmt (s: stmt): st<()> =
branch
  let While (e, s') = s in
  let Bool b =@ eval_expr e in
  branch
    let False = b in ret<()> ()
  or
    let True = b in
    eval_stmt t';@
    eval_stmt t
  end
or ...
end
```

The Monad Zoo

- reader monad (environment)
- writer monad (log)
- option monad (exceptions, simpler control flow)
- state monad (heap)
- delimited continuation monad (generators, effects)

GetValue(V)

1. ReturnIfAbrupt(V).
2. If V is not a Reference Record, return V.
3. If IsUnresolvableReference(V) is `true`, throw a `ReferenceError` exception.
4. If IsPropertyReference(V) is `true`, then
   1. Let baseObj be ! ToObject(V.`[[Base]]`).
   2. Return ? baseObj.`[[Get]]`(V.`[[ReferencedName]]`, GetThisValue(V)).
5. Else,
   1. Let base be V.`[[Base]]`
   2. Assert: base is an Environment Record.
   3. Return ? base.GetBindingValue(V.`[[ReferencedName]]`, V.`[[Strict]]`).

GetValue(V)

```
val getValue: (v: out<valref>) -> st<out<value>> =
  let result =@r
    let v =%returnIfAbrupt v in
    branch valref_Type(v, T_Reference);@false let Value v = v in ret v end;@
    let Reference v = v in
    branch isUnresolvableReference v;@true throw referenceError _getValue_ end;@
    branch let T = isPropertyReference v in
           let R_Value v_base = v._Base_ in let baseObj =! toObject(v_base) in
           let baseObj =/o baseObj in
           let thisVal =? getThisValue(v) in
           let r =? baseObj._O_Get__(v._ReferencedName_, thisVal) in ret r
    or     let F = isPropertyReference v in
           let base = v._Base_ in
           assT ref_Type (base, T_R_EnvironmentRecord) _getValue_;@
           let R_EnvironmentRecord base = base in let base =/er base in
           let r =? base.__GetBindingValue__(v._ReferencedName_, v._Strict_) in ret r
    end
  in result
```

Delimited Continuations

In Languages

- generators (JavaScript, Python)
- effects (OCaml 5)
- and also...

JavaScript Function Calls

13.3.6.1 Runtime Semantics: Evaluation

- `Return ? EvaluateCall(func, ref, arguments, tailCall)`

13.3.6.2 `EvaluateCall ( func, ref, arguments, tailPosition )`

- `Let result be Call(func, thisValue, argList)`

...

10.2.10 `FunctionDeclarationInstantiation ( func, argumentsList )`

- `Let iteratorRecord be CreateListIteratorRecord(argumentsList)`

Iterators as Generators

7.4.9 `CreateListIteratorRecord ( list )`

1. Let closure be a new Abstract Closure with no parameters that captures list and performs the following steps when called:
    1. For each element E of list, do
        1. Perform `? Yield(E)`.
    2. Return undefined.
2. Let `iterator` be `! CreateIteratorFromClosure(closure, empty, %IteratorPrototype%)`.
3. Return `Record { [[Iterator]]: iterator, [[NextMethod]]: %GeneratorFunction.prototype.prototype.next%, [[Done]]: false }`.

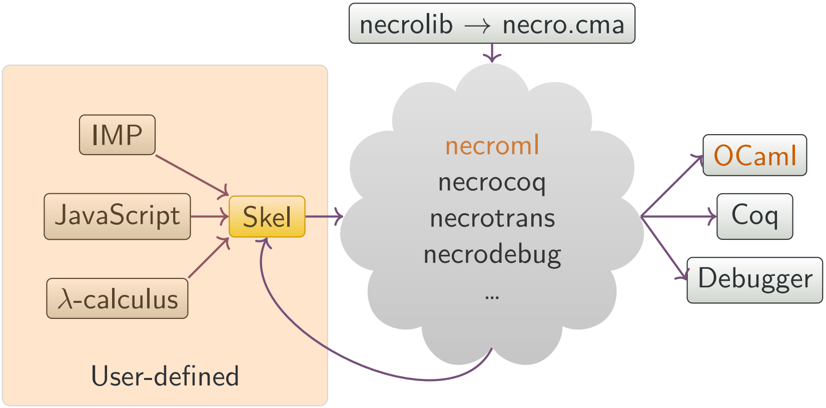
Whole Monad

```
type exc<a> = | Exc | Ok a
type rd<a> = env → a
type st<a> = heap → (a, heap)
type cont<a> = a -> cstack<a> -> a
type cstack<a> =
| Nil
| Cons(cont<a>,cstack<a>)
type contM<a> = cont<a> -> cstack<a> -> a
type ste<a> = st<exc<a>>
type k<a> = cont<ste<a>>
type cs<a> = cstack<ste<a>>
type m<a> = rd<contM<ste<a>>>
```

```
val return<a> (v:a) : m<a> =
    λ_:env → λk:k<a> → λks:cs<a> → λh:heap →
    k (λh':heap → (Ok<a> v, h')) ks h

val bind<a> (w:m<a>) (f:a → m<a>) : m<a> =
    λs:env → λk:k<a> → λks:cs<a> → λh:heap →
    w s (λste:ste<a> → λks1:cs<a> → λh1:heap →
        let (vo, h2) = ste h1 in
        match vo with
        | Exc → k (λh3:heap → (Exc<a>,h3))
                  ks1 h2
        | Ok v → f v s k ks1 h2
        end) ks h
```

Necro ML

Necro Ecosystem



Necro ML

1. Write the skeletal semantics
2. Write a module implementing unspecified types and terms
3. Choose how to interpret branches
4. Run `necroml` and apply the `MakeInterpreter` functor
5. Profit!

Necro ML Example

```
(* arith.sk *)


type lit
type value




val litToVal: lit → value
val add: (value, value) → value
val sub: (value, value) → value
val mul: (value, value) → value
val div: (value, value) → value

(* next are specified types
   and terms *)
```

```
open Arith (* file generated with necroml *)

module Types = struct
  type lit = int
  type value = int
end

module Input = struct
  include Unspec(Monads.ID)(Types)
  let litToVal l = l
  let add (l1, l2) = l1 + l2
  let sub (l1, l2) = l1 - l2
  let mul (l1, l2) = l1 * l2
  let div (l1, l2) = l1 / l2
end

module ArithInterp = MakeInterpreter(Input)
```

Interpretation Monads

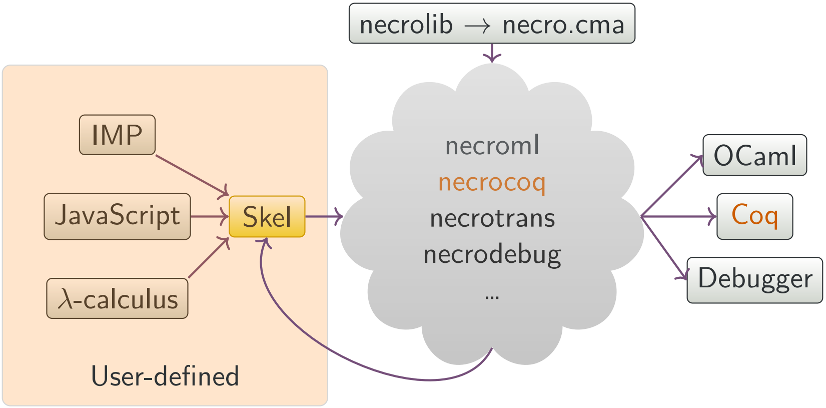
Terms are pure, skeletons are computations

```
module type MONAD = sig
  type 'a t
  val ret: 'a -> 'a t
  val bind: 'a t -> ('a -> 'b t) -> 'b t
  val branch: (unit -> 'a t) list -> 'a t
  val fail: string -> 'a t
  val apply: ('a -> 'b t) -> 'a -> 'b t
  val extract: 'a t -> 'a
end
```

- Main challenge: how to interpret branching?
- Instantiations: sequential, lists, non-deterministic, continuations

Necro Coq

Necro Ecosystem



A Deep Embedding in Coq (Skeleton.v)

```
Inductive term: Type :=
| term_constructor : string -> list type -> term -> term
| term_var: typed_var -> term
| term_tuple: list term -> term
| term_func: pattern -> skeleton -> term
with skeleton: Type :=
| skel_branch : type -> list skeleton -> skeleton
| skel_match : term -> type -> list (pattern * skeleton) -> skeleton
| skel_return : term -> skeleton
| skel_apply : term -> list term -> skeleton
| skel_letin : pattern -> skeleton -> skeleton -> skeleton.
```

Dynamic Semantics

- `Concrete.v`, natural (big-step) semantics using Coq induction

```
| i_letin: forall e e' p s1 s2 v w,
    interp_skel e s1 v ->
    add_asn e p v = Some e' ->
    interp_skel e' s2 w ->
    interp_skel e (skel_letin p s1 s2) w
```

- `Concrete_ss.v`, small-step semantics
- `ConcreteRec.v`, iterative semantics
- `Concrete_ndam.v`, non-deterministic abstract machine
- `Concrete_am.v`, backtracking abstract machine, can compute
- many equivalence proofs

Conclusion

Current Status

- Programming Languages
  - WebAssembly (Thomas Rubiano)
  - JavaScript (Adam Khayam)
  - Python (Martin Andrieux)
- Many applications
  - Generation of OCaml interpreter and Coq formalization (Victoire Noizet)
  - In-browser debugger (Victoire Noizet)
  - Certified interpreter (extracted from Coq) (Guillaume Ambal)
  - Skel to Skel transformation
    - big-step to small-step (Guillaume Ambal)
    - big-step to abstract machines (Martin Andrieux)
  - Generation of abstract analyzers (Vincent Rébiscoul)
  - Hoare Logic (Laura-Andrea Schimbător)
  - Abstract machines for process calculi (Sergueï Lenglet)

Future Work

- More languages
  - Rust
  - Esterel
- Skel improvements
  - `include` support
  - type inference
- New backends
  - generic compilation
  - symbolic execution

Questions?

Many thanks to Guillaume Ambal, Martin Andrieux, Martin Bodin, Nathanaëlle Courant, Enzo Crance, Philippa Gardner, Olivier Idir, Thomas Jensen, Adam Khayam, Sergueï Lenglet, Victoire Noizet, Vincent Rébiscoul, Thomas Rubiano

https://skeletons.inria.fr

Extra Slides

Pattern Matching

```
val eval_expr ((st:state), (e:expr)) : value =
match e with
| Const n -> let i = int_of_lit n in Int n
| Plus(e1,e2) ->
    let Int n1 = eval_expr (st, e1) in
    let Int n2 = eval_expr (st, e2) in
    let n = add (n1, n2) in
    Int i
end
```

Choice

```
val insert<a> ((e:a), (l: list<a>)): list <a> =
  branch
    Cons<a>(e, l)
  or
    let Cons(e', l') = l in
    let l'' = insert<a>(e, l') in
    Cons<a>(e', l'')
  end

val permut<a> (l: list<a>): list <a> =
  match l with
  | Nil → Nil<a>
  | Cons(e,es) → let es' = permut<a> es in insert<a>(e, es')
  end
```

Skel, Formally

$$
\begin{array}{rccl}
\textsc{Term} & t & ::= & x \mid C\ t \mid (t, \ldots, t) \mid \lambda p : \tau \cdot S \\
\textsc{Pattern} & p & ::= & x \mid \_ \mid C\ p \mid (p, \ldots, p) \\
\textsc{Skeleton} & S & ::= & t\ t \mid \text{let}\ p = S\ \text{in}\ S \mid \text{let}\ p : \tau\ \text{in}\ S \\
 & & & \mid \oplus (S..S) \mid \mathcal{M}\,(t)\,(p \to S..p \to S) \mid t \\
\textsc{Type spec} & r & ::= & \text{type}\ b \mid \text{type}\ b := \tau \mid \text{type}\ b = "\mid" C\ \tau \ldots "\mid" C\ \tau \\
\textsc{Term spec} & r' & ::= & \text{val}\ x : \tau \mid \text{val}\ x : \tau = t
\end{array}
$$

Note: this is almost in administrative normal form

Existentials

$$\frac{\Gamma, x : \tau \vdash m : \nu}{\Gamma \vdash \lambda x \cdot m : \tau \rightarrow \nu}$$

Existentials

$$\frac{\Gamma, x : \tau \vdash m : \nu}{\Gamma \vdash \lambda x \cdot m : \tau \to \nu}$$

Predicate

input: whole judgement, output: unit

```
val ctype ((gamma: env), (t: term), (tp: ltype)) : () =
  branch
    let Lam (x, m) = t in
    let Arrow (tau,nu) = tp in
    let gamma' = ext_env (gamma, x, tau) in
    ctype (gamma', m, nu)
```

Existentials

$$\frac{\Gamma, x : \tau \vdash m : \nu}{\Gamma \vdash \lambda x \cdot m : \tau \to \nu}$$

Algorithmic

input: typing env and term, output: type

```
val ctype ((gamma: env), (t: term)) : ltype =
  branch
    let Lam (x, m) = t in
    let tau : ltype in
    let gamma' = ext_env (gamma, x, tau) in
    let nu = ctype (gamma', m) in
    Arrow (tau, nu)
```