Finding Meaning: Semantics and Verification in Programming Languages

Remy Seassau

8 December 2024

Cambium, Inria Paris

Who am I?

Supervisor: François Pottier

- PhD Student in my 2nd year
- Co-organiser of the Junior Seminar
- Part of the Cambium team

Thesis title:

”Osiris : a Separation Logic Verification Framework for OCaml”

Cambium

Programming Languages

- Design: Programming Constructs, Type Systems
- Implementation: Compilers, Parsing, Memory Models
- Formalisation: Proof assistants, Logic, Specification

Cambium

Programming Languages

- Design: Programming Constructs, Type Systems
- Implementation: Compilers, Parsing, Memory Models
- Formalisation: Proof assistants, Logic, Specification

Keywords: Functional, Strongly Typed, Concurrent, Garbage Collected

Who are we?

Cambium

Programming Languages

- Design: Programming Constructs, Type Systems
- Implementation: Compilers, Parsing, Memory Models
- Formalisation: Proof assistants, Logic, Specification

Keywords: Functional, Strongly Typed, Concurrent, Garbage Collected

Bloomberg

Who are we?

Cambium

Programming Languages

- Design: Programming Constructs, Type Systems
- Implementation: Compilers, Parsing, Memory Models
- Formalisation: Proof assistants, Logic, Specification

Keywords: Functional, Strongly Typed, Concurrent, Garbage Collected

Bloomberg

In this talk

1. Mathematically modelling program behaviour
2. Reasoning on our Model

A programming language is a set of instructions that “do things”

A programming language is a set of instructions that “do things”

Syntax

A programming language is a set of instructions that “do things”

An example of a programming language

An example of a programming language

Python is defined in natural language in the **Python Language Reference**:

An example of a programming language

Python is defined in natural language in the **Python Language Reference**:

"*The if statement is used for conditional execution:*
It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true; then that suite is executed."

An example of a programming language

Python is defined in natural language in the **Python Language Reference**:

"*The if statement is used for conditional execution:*
It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true; then that suite is executed."

It also has a **Reference Implementation** called CPython (probably installed on your computer)

Natural language isn't enough

```
class D(dict):
    def __getitem__(self, key):
        return "overridden"

class A(object):
    pass

a = A()
a.__dict__ = D()
a.foo = "not overridden"
print(a.foo)
```

Natural language isn't enough

```
class D(dict):
    def __getitem__(self, key):
        return "overridden"

class A(object):
    pass

a = A()
a.__dict__ = D()
a.foo = "not overridden"
print(a.foo)
```

CPython: `"not overridden"`

Natural language isn't enough

```
class D(dict):
    def __getitem__(self, key):
        return "overridden"

class A(object):
    pass

a = A()
a.__dict__ = D()
a.foo = "not overridden"
print(a.foo)
```

CPython: `"not overridden"`

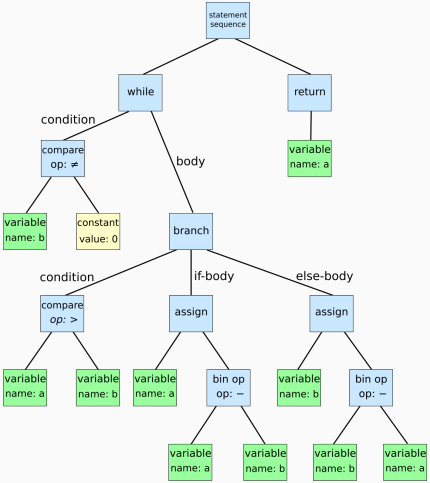
PyPy: `"overridden"`

A programming language is a set of instructions that “do things”

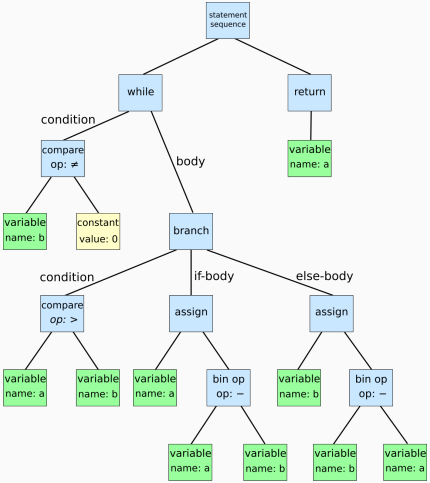
Defining Meaning - Semantics of the Syntax

```
while b != 0:
    if a > b:
        a = a - b
    else:
        b = b - a
return a
```

Defining Meaning - Semantics of the Syntax



Defining Meaning - Semantics of the Syntax



We will refer to the nodes of the AST as **expressions**, and say that they evaluate to **values**

Operational Semantics

$$EAdd\ e_1\ e_2 \longrightarrow ??$$

Operational Semantics

$$EAdd\ e_1\ e_2 \longrightarrow n_1 + n_2$$

Operational Semantics

If $e_1 \longrightarrow n_1$

$EAdd\ e_1\ e_2 \longrightarrow n_1 + n_2$

Operational Semantics

If $e_1 \longrightarrow n_1$ and $e_2 \longrightarrow n_2$ then $EAdd\ e_1\ e_2 \longrightarrow n_1 + n_2$

Operational Semantics

$$\mathit{Elf}\ e_b\ e_1\ e_2 \longrightarrow\ ??$$

Operational Semantics

$$If\ e_b\ e_1\ e_2 \longrightarrow n_1$$

Operational Semantics

$\text{If } e_b \longrightarrow \texttt{True}$

$\mathit{Elf}\ e_b\ e_1\ e_2 \longrightarrow n_1$

Operational Semantics

If $e_b \longrightarrow \texttt{True}$ and $e_1 \longrightarrow n_1$ then $\mathit{Elf}\ e_b\ e_1\ e_2 \longrightarrow n_1$

Operational Semantics

If $e_b \longrightarrow \texttt{True}$ and $e_1 \longrightarrow n_1$ then $\mathit{EIf}\ e_b\ e_1\ e_2 \longrightarrow n_1$

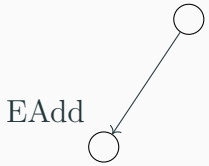
If $e_b \longrightarrow \texttt{False}$ and $e_2 \longrightarrow n_2$ then $\mathit{EIf}\ e_b\ e_1\ e_2 \longrightarrow n_2$

Modelling Computation

```
if b:
    x = x + 1
    return x
else:
    return x
```

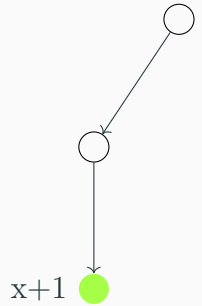
Modelling Computation

```
if b:
    x = x + 1
    return x
else:
    return x
```



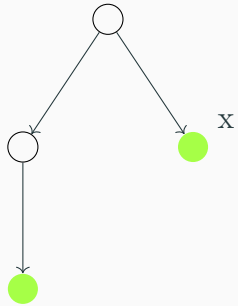
Modelling Computation

```
if b:
    x = x + 1
    return x
else:
    return x
```



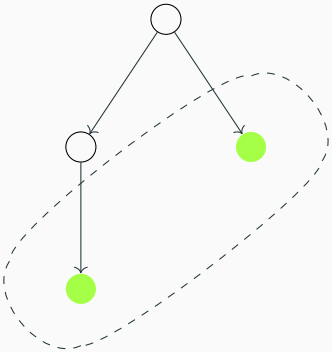
Modelling Computation

```
if b:
    x = x + 1
    return x
else:
    return x
```



Modelling Computation

```
if b:
    x = x + 1
    return x
else:
    return x
```



Progam Verification

Once we have a model, we can **reason** on it

Progam Verification

Once we have a model, we can **reason** on it

Properties we can verify:

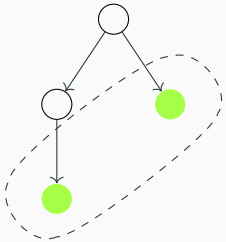
- Runtime safety
- Functional correctness
- Resource usage
- Effectful behaviour

Progam Verification

Once we have a model, we can **reason** on it

Properties we can verify:

- Runtime safety
- Functional correctness
- Resource usage
- Effectful behaviour

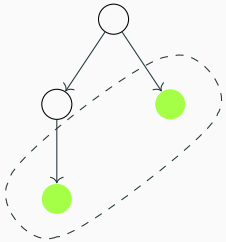


Progam Verification

Once we have a model, we can **reason** on it

Properties we can verify:

- Runtime safety
- Functional correctness
- Resource usage
- Effectful behaviour



Program Logics

$$c \textit{ respects } P \coloneqq \begin{cases} c \text{ is a value } v \text{ s.t. } P\ v \\ c \text{ can step, and } \forall c' \text{ s.t. } c \longrightarrow c',\ c' \textit{ respects } P \end{cases}$$

Program Logics

$$c \textit{ respects } P \coloneqq \begin{cases} c \text{ is a value } v \text{ s.t. } P\ v \\ c \text{ can step, and } \forall c' \text{ s.t. } c \longrightarrow c',\ c' \textit{ respects } P \end{cases}$$

$$\mathit{EIf}\ e_b\ e_1\ e_2 \textit{ respects } P$$

Program Logics

$$c \textit{ respects } P \coloneqq \begin{cases} c \text{ is a value } v \text{ s.t. } P\ v \\ c \text{ can step, and } \forall c' \text{ s.t. } c \longrightarrow c',\ c' \textit{ respects } P \end{cases}$$

e_b *respects* $(\lambda b,$ $) \implies$

EIf $e_b\ e_1\ e_2$ *respects* P

Program Logics

$$c \textit{ respects } P \coloneqq \begin{cases} c \text{ is a value } v \text{ s.t. } P\ v \\ c \text{ can step, and } \forall c' \text{ s.t. } c \longrightarrow c',\ c' \textit{ respects } P \end{cases}$$

e_b *respects* $(\lambda b,\ \textit{if}\ b\ \textit{then}\qquad\qquad \textit{else}\qquad\qquad) \implies$

$\textit{EIf}\ e_b\ e_1\ e_2$ *respects* P

Program Logics

$$c \textit{ respects } P \coloneqq \begin{cases} c \text{ is a value } v \text{ s.t. } P\ v \\ c \text{ can step, and } \forall c' \text{ s.t. } c \longrightarrow c',\ c' \textit{ respects } P \end{cases}$$

$$e_b \textit{ respects } (\lambda b,\ \textit{if } b \textit{ then } e_1 \textit{ respects } P \textit{ else } e_2 \textit{ respects } P) \implies$$
$$\textit{EIf } e_b\ e_1\ e_2 \textit{ respects } P$$

Program Logics

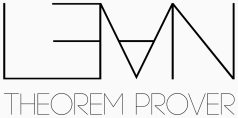
$$c \textit{ respects } P \coloneqq \begin{cases} c \text{ is a value } v \text{ s.t. } P\ v \\ c \text{ can step, and } \forall c' \text{ s.t. } c \longrightarrow c',\ c' \textit{ respects } P \end{cases}$$

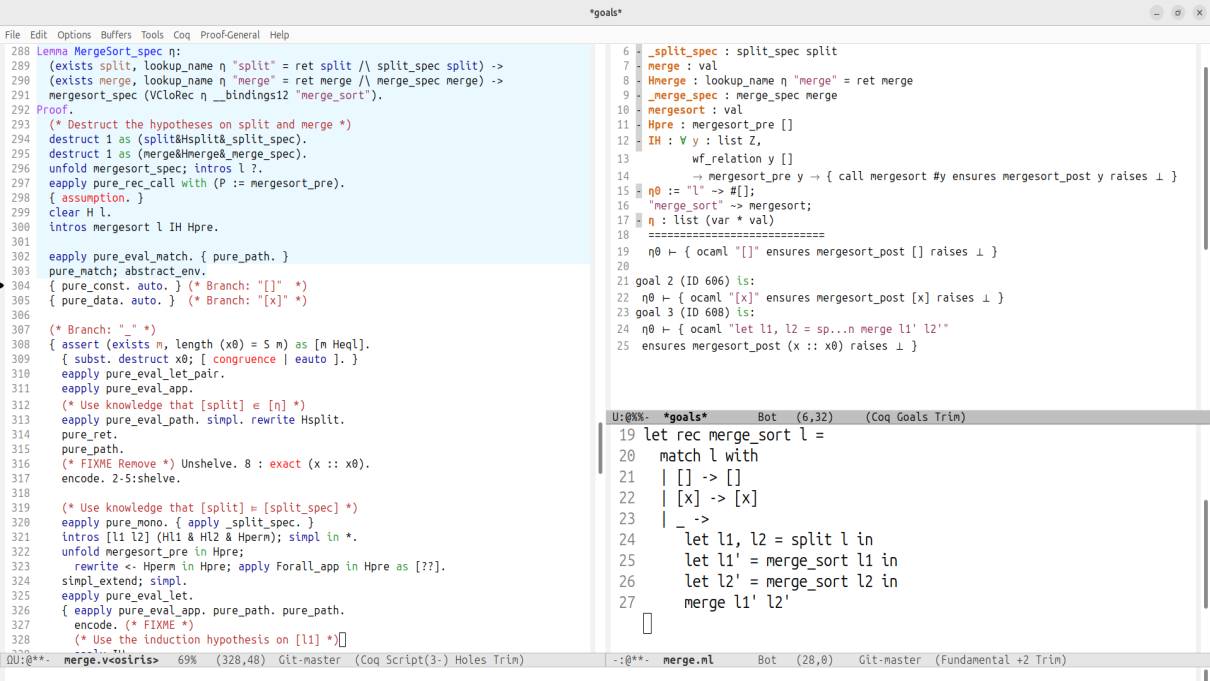
$$e_b \textit{ respects } (\lambda b,\ \textit{if } b \textit{ then } e_1 \textit{ respects } P \textit{ else } e_2 \textit{ respects } P) \implies$$
$$\mathit{EIf}\ e_b\ e_1\ e_2 \textit{ respects } P$$

P describes functional correctness, we get runtime safety implicitly!

Proof Assistants

Don’t write out formal proofs by hand, use **proof assistants** instead




goals
File Edit Options Buffers Tools Coq Proof-General Help
Lemma MergeSort_spec η:
(exists split, lookup_name η "split" = ret split /\ split_spec split) ->
(exists merge, lookup_name η "merge" = ret merge /\ merge_spec merge) ->
mergesort_spec (VCloRec η __bindings12 "merge_sort").
Proof.
(* Destruct the hypotheses on split and merge *)
destruct 1 as (split&Hsplit&_split_spec).
destruct 1 as (merge&Hmerge&_merge_spec).
unfold mergesort_spec; intros l ?.
eapply pure_rec_call with (P := mergesort_pre).
{ assumption. }
clear H l.
intros mergesort l IH Hpre.
eapply pure_eval_match. { pure_path. }
pure_match; abstract_env.
{ pure_const. auto. } (* Branch: "[]" *)
{ pure_data. auto. } (* Branch: "[x]" *)
(* Branch: "_" *)
{ assert (exists m, length (x0) = S m) as [m Heql].
{ subst. destruct x0; [congruence | eauto]. }
eapply pure_eval_let_pair.
eapply pure_eval_app.
(* Use knowledge that [split] ∈ [η] *)
eapply pure_eval_path. simpl. rewrite Hsplit.
pure_ret.
pure_path.
(* FIXME Remove *) Unshelve. 8 : exact (x :: x0).
encode. 2-5:shelve.
(* Use knowledge that [split] ⊨ [split_spec] *)
eapply pure_mono. { apply _split_spec. }
intros [l1 l2] (Hl1 & Hl2 & Hperm); simpl in *.
unfold mergesort_pre in Hpre;
rewrite <- Hperm in Hpre; apply Forall_app in Hpre as [??].
simpl_extend; simpl.
eapply pure_eval_let.
{ eapply pure_eval_app. pure_path. pure_path.
encode. (* FIXME *)
(* Use the induction hypothesis on [l1] *)
U:@**- merge.v<osiris> 69% (328,48) Git-master (Coq Script(3-) Holes Trim)
_split_spec : split_spec split
merge : val
Hmerge : lookup_name η "merge" = ret merge
_merge_spec : merge_spec merge
mergesort : val
Hpre : mergesort_pre []
IH : ∀ y : list Z,
wf_relation y []
→ mergesort_pre y → { call mergesort #y ensures mergesort_post y raises ⊥ }
η0 := "l" ~> #[];
"merge_sort" ~> mergesort;
η : list (var * val)
============================
η0 ⊢ { ocaml "[]" ensures mergesort_post [] raises ⊥ }
goal 2 (ID 606) is:
η0 ⊢ { ocaml "[x]" ensures mergesort_post [x] raises ⊥ }
goal 3 (ID 608) is:
η0 ⊢ { ocaml "let l1, l2 = sp...n merge l1' l2'"
ensures mergesort_post (x :: x0) raises ⊥ }
U:@%%- *goals* Bot (6,32) (Coq Goals Trim)
let rec merge_sort l =
match l with
| [] -> []
| [x] -> [x]
| _ ->
let l1, l2 = split l in
let l1' = merge_sort l1 in
let l2' = merge_sort l2 in
merge l1' l2'
-:@**- merge.ml Bot (28,0) Git-master (Fundamental +2 Trim)

Large scale verification projects:

- CompCert: An optimizing C compiler
- Catala: A formalisation of the French tax code
- HACL*: A cryptographic algorithm library