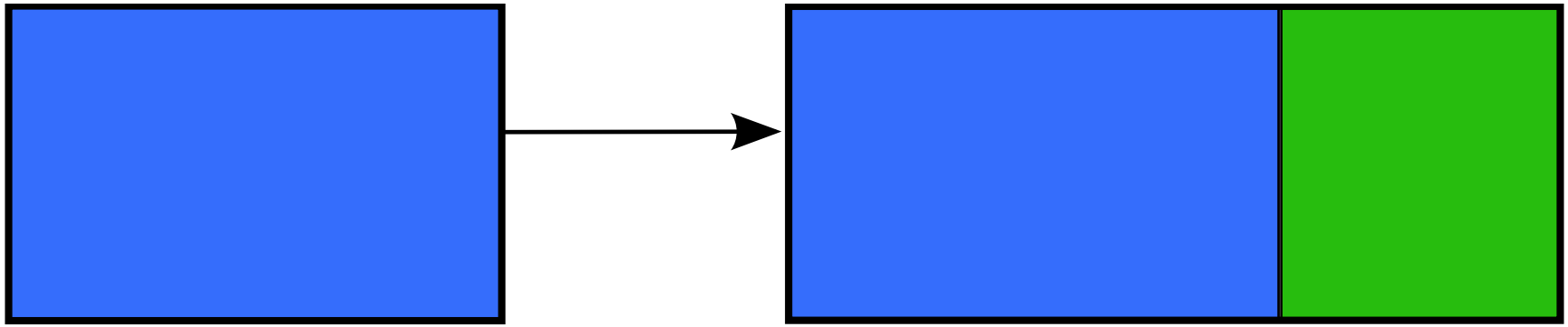
Extending Operational Semantics of the Java Bytecode

Patryk Czarnik Aleksy Schubert

Institute of Informatics
Warsaw University

Trustworthy Global Computing 2007

Extending a Semantics



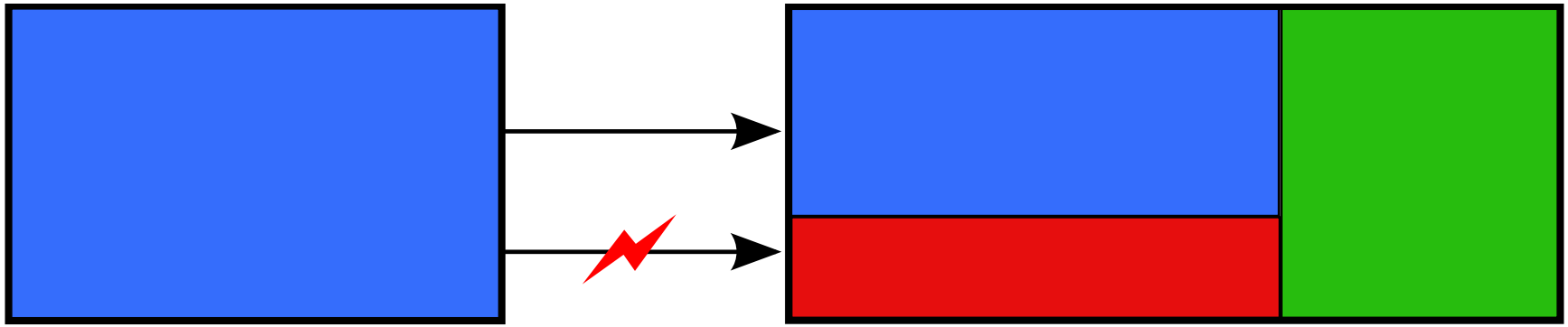
Existing formalisation

- only basic properties

New formalisation

- basic properties formalisation reused
- additional details specified

Extending a Semantics



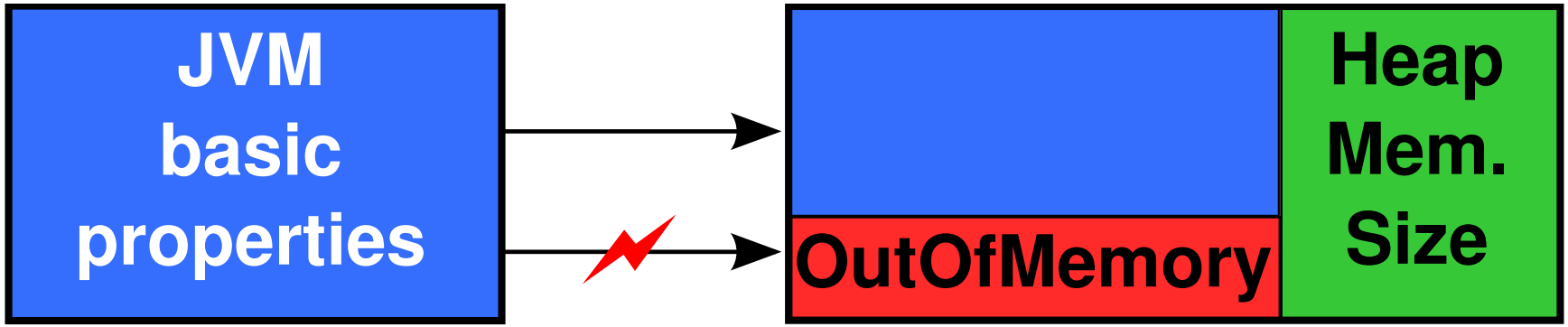
Existing formalisation

- only basic properties

New formalisation

- basic properties formalisation reused
- additional details specified
- change of behaviour

Example: JVM, Heap Memory Size



- New information to track: heap memory size
- `new` or `newarray` JVM instruction: size increased
- Memory size exceeded: `OutOfMemoryError`

Background

- **Mobius** – Mobility, Ubiquity and Security:
  - EU 6th Framework Program
  - enabling proof-carrying code for Java on mobile devices
- **Coq** proof assistant:
  - strong logic base (Calculus of Inductive Constructions)
  - module system
- **Bicolano** [*Pichardie 2006*]:
  - basic JVM formalisation for Mobius
  - large set of instructions (CLDC)
  - modularly implemented in Coq (11.500 lines of code)

Limitations of Original Bicolano

- Idealised in various ways:
  - unlimited resources
  - single-threaded
  - no non-deterministic JVM errors
- No support for extending state with additional data

Why not just modify Bicolano?

The choice of additional features depends on application.

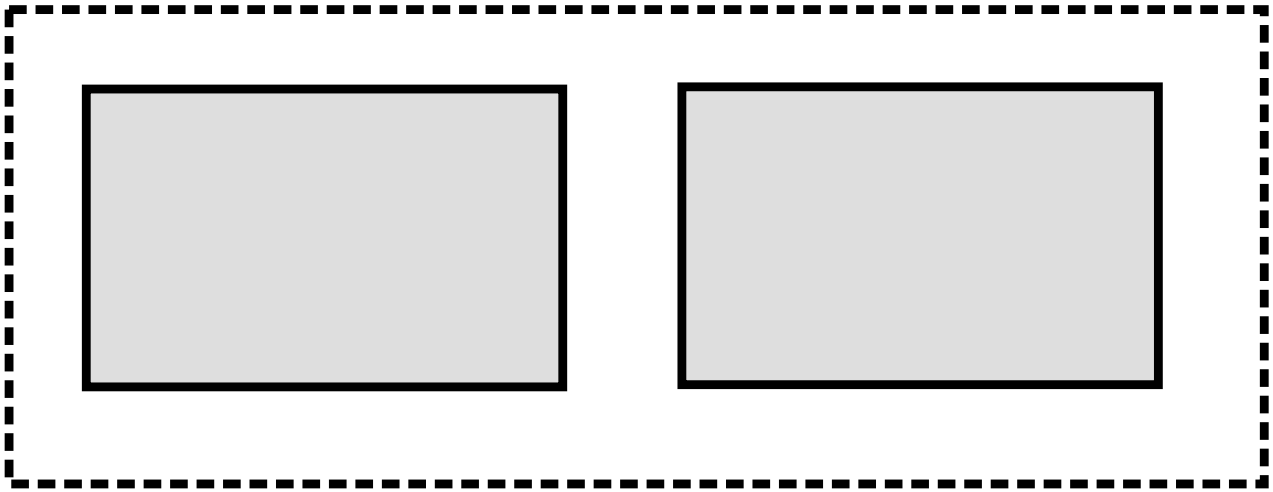
Motivation for Extensions

- JVM and system resources:
  - heap memory size
  - call stack size
  - network transfer
  - paid services (SMS etc) . . .
- Logic:
  - program traces
  - ghost variables
- Non-determinism:
  - multi-threading
  - non-deterministic JVM errors

Our Work

- General frameworks for extending semantics:
  - **horizontal** (HF)
  - **vertical** (VF)
- Module-based implementation in Coq:
  - flexible
  - no changes in original Bicolano code
- Extension examples:
  - empty extension :)
  - executed instructions counter
  - heap memory size
  - call stack size
  - embedding of HF within VF

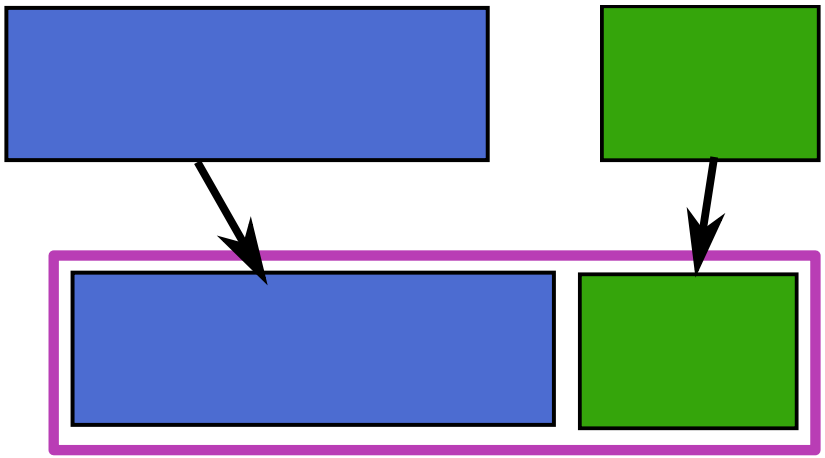
Horizontal framework



Horizontal Framework

Idea

- Observation of program runs
- Additional information collected
- No change in behaviour



Writing Extension in Horizontal Framework

Implementation of extension module:

- `ACT: Type`
  - *action trace*
  - extended part of state
- `bookkeep: Program -> State.t -> State.t -> ACT -> ACT`
  - how ACT is changed in a step
- `bookkeep` input:
  - Bicolano state before step
  - Bicolano state after step
  - ACT before step
- `bookkeep` output:
  - ACT after step

Horizontal Framework Realisation

Bicolano

```
State.t = heap * frame_stack

step : Program
  -> State.t -> State.t -> Prop
```

Extension

```
ACT: Type

bookkeep:
    Program -> State.t
    -> State.t -> ACT -> ACT
```

Extended semantics

```
ExtState.t = State.t * ACT

Inductive ext_step : Program -> ExtState.t -> ExtState.t -> Prop :=
  ESS_step: forall ...,
    step p s t ->
    T = bookkeep p s t S ->
    ext_step p (s,S) (t,T)
```

Horizontal Framework Realisation

Bicolano

```
State.t = heap * frame_stack

step : Program
  -> State.t -> State.t -> Prop
```

Extension

```
ACT: Type

bookkeep:
    Program -> State.t
    -> State.t -> ACT -> ACT
```

Extended semantics

```
ExtState.t = State.t * ACT

Inductive ext_step : Program -> ExtState.t -> ExtState.t -> Prop :=
  ESS_step: forall ...,
    step p s t ->
    T = bookkeep p s t S ->
    ext_step p (s,S) (t,T)
```

Horizontal Framework Realisation

Bicolano

```
State.t = heap * frame_stack

step : Program
  -> State.t -> State.t -> Prop
```

Extension

```
ACT: Type

bookkeep:
    Program -> State.t
    -> State.t -> ACT -> ACT
```

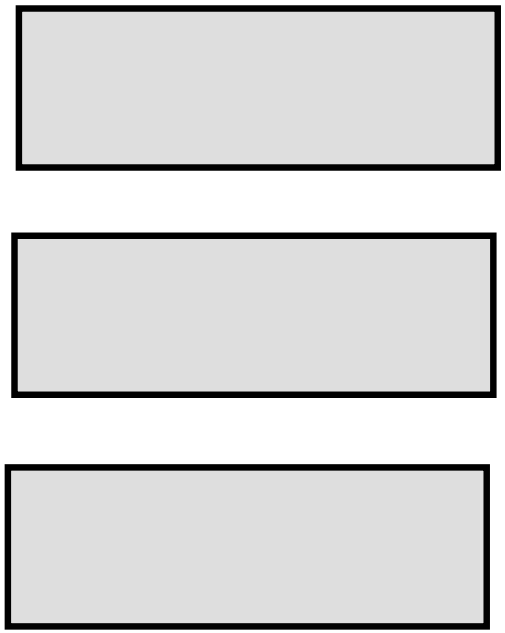
Extended semantics

```
ExtState.t = State.t * ACT

Inductive ext_step : Program -> ExtState.t -> ExtState.t -> Prop :=
  ESS_step: forall ...,
    step p s t ->
    T = bookkeep p s t S ->
    ext_step p (s,S) (t,T)
```

Memory Example: HF Implementation

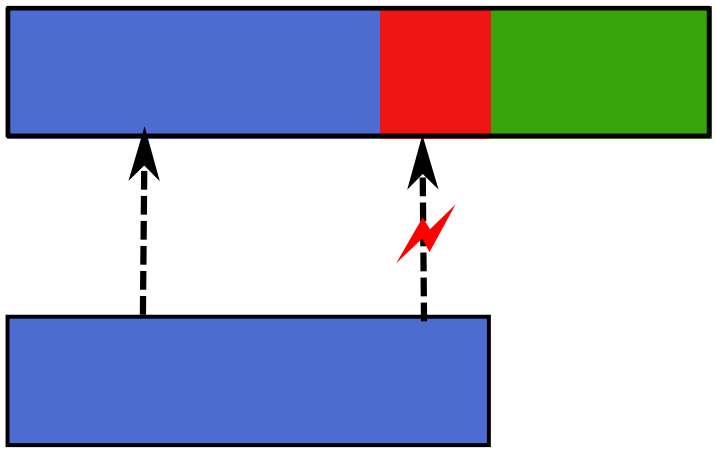
```
Module MemoryExt <: EXTENSION.

  Definition ACT := nat.

  Definition bookkeep (p: Program)
    (s1: State.t) (s2: State.t) (old_size: nat): nat :=
    (* ... *)
      match instruction with
      | New c =>
        old_size + (size_of_object c)

      | Newarray t =>
        old_size + (size_of_array t)

      | _ => old_size
      end
   (* ... *)
End MemoryExt.
```

Vertical framework



Vertical Framework Idea

- New, arbitrary definitions of state and step
- Original definitions available for reuse
- Typical use:
  - extending original state type
  - changing behaviour in special circumstances
  - leaving original behaviour in most cases



Vertical Framework Components

- `SEM` – signature for semantics instances:
  - `state_t: Type`
    type of state
  - `get_bottom_state: state_t -> bt_state_t -> Prop`
    projection to Bicolano state
  - `step: Program -> state_t -> state_t -> Prop`
    semantic step
  - `additional_step: Program -> state_t -> bt_state_t -> state_t -> Prop`
    used by further extensions to construct Bicolano states in case of change of behaviour
- `BottomSemantics` – Bicolano semantics wrapped in `SEM` signature
- Extension: functor `SEM` $\longrightarrow$ `SEM`

Vertical Framework Components

- `SEM` – signature for semantics instances:
  - `state_t: Type`
    type of state
  - `get_bottom_state: state_t -> bt_state_t -> Prop`
    projection to Bicolano state
  - `step: Program -> state_t -> state_t -> Prop`
    semantic step
  - `additional_step: Program -> state_t -> bt_state_t -> state_t -> Prop`
    used by further extensions to construct Bicolano states in case of change of behaviour
- `BottomSemantics` – Bicolano semantics wrapped in `SEM` signature
- Extension: functor `SEM` $\longrightarrow$ `SEM`

Vertical Framework Components

- `SEM` – signature for semantics instances:
  - `state_t: Type`
    type of state
  - `get_bottom_state: state_t -> bt_state_t -> Prop`
    projection to Bicolano state
  - `step: Program -> state_t -> state_t -> Prop`
    semantic step
  - `additional_step: Program -> state_t -> bt_state_t -> state_t -> Prop`
    used by further extensions to construct Bicolano states in case of change of behaviour
- `BottomSemantics` – Bicolano semantics wrapped in `SEM` signature
- Extension: functor `SEM` $\longrightarrow$ `SEM`

Vertical Framework Components

- `SEM` – signature for semantics instances:
  - `state_t: Type`
    type of state
  - `get_bottom_state: state_t -> bt_state_t -> Prop`
    projection to Bicolano state
  - `step: Program -> state_t -> state_t -> Prop`
    semantic step
  - `additional_step: Program -> state_t -> bt_state_t -> state_t -> Prop`
    used by further extensions to construct Bicolano states in case of change of behaviour
- `BottomSemantics` – Bicolano semantics wrapped in `SEM` signature
- Extension: functor `SEM` $\longrightarrow$ `SEM`

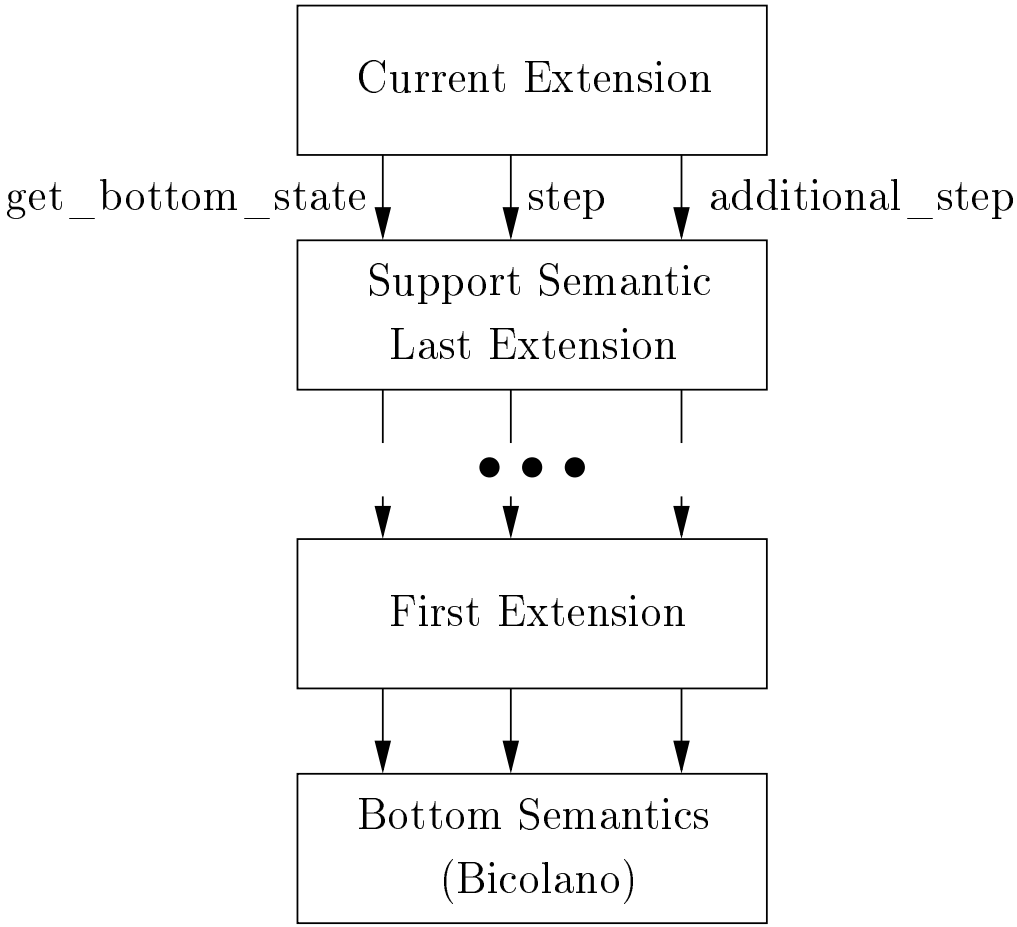
Vertical Framework Components

- `SEM` – signature for semantics instances:
  - `state_t: Type`
    type of state
  - `get_bottom_state: state_t -> bt_state_t -> Prop`
    projection to Bicolano state
  - `step: Program -> state_t -> state_t -> Prop`
    semantic step
  - `additional_step: Program -> state_t -> bt_state_t -> state_t -> Prop`
    used by further extensions to construct Bicolano states in case of change of behaviour
- `BottomSemantics` – Bicolano semantics wrapped in `SEM` signature
- Extension: functor `SEM` $\longrightarrow$ `SEM`

Vertical Framework Components

- `SEM` – signature for semantics instances:
  - `state_t: Type`
    type of state
  - `get_bottom_state: state_t -> bt_state_t -> Prop`
    projection to Bicolano state
  - `step: Program -> state_t -> state_t -> Prop`
    semantic step
  - `additional_step: Program -> state_t -> bt_state_t -> state_t -> Prop`
    used by further extensions to construct Bicolano states in case of change of behaviour
- `BottomSemantics` – Bicolano semantics wrapped in `SEM` signature
- Extension: functor `SEM` $\longrightarrow$ `SEM`

Semantics Hierarchy



Memory Example: VF Implementation(1)

```
Module MemorySem (Support: SEM) <: SEM.
  Parameter max_heap_size: nat.

  Definition state_t := Support.state_t * nat.

  Inductive step :=
  | step_inherit_nn: forall ...,
(* ... *)
    ~(current_ins sust1 (New c)) ->
    ~(current_ins sust1 (Newarray t)) ->
    Support.step p sust1 sust2 ->
    step p (sust1, chs) (sust2, chs)

  | step_new_ok: forall ...,
(* ... *)
    current_ins sust1 (New c) ->
    chs + (size_of_object c) <= max_heap_size ->
    Support.step p sust1 sust2 ->
    step p (sust1, chs) (sust2, (chs + (size_of_object c)))
(* ... *)
```

Memory Example: VF Implementation(1)

```
Module MemorySem (Support: SEM) <: SEM.
  Parameter max_heap_size: nat.

  Definition state_t := Support.state_t * nat.

  Inductive step :=
  | step_inherit_nn: forall ...,
(* ... *)
    ~(current_ins sust1 (New c)) ->
    ~(current_ins sust1 (Newarray t)) ->
    Support.step p sust1 sust2 ->
    step p (sust1, chs) (sust2, chs)

  | step_new_ok: forall ...,
(* ... *)
    current_ins sust1 (New c) ->
    chs + (size_of_object c) <= max_heap_size ->
    Support.step p sust1 sust2 ->
    step p (sust1, chs) (sust2, (chs + (size_of_object c)))
(* ... *)
```

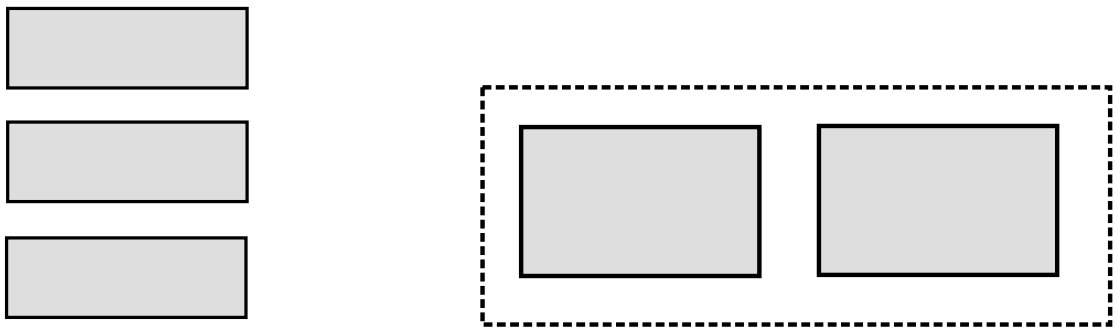
Memory Example: VF Implementation(1)

```
Module MemorySem (Support: SEM) <: SEM.
  Parameter max_heap_size: nat.

  Definition state_t := Support.state_t * nat.

  Inductive step :=
  | step_inherit_nn: forall ...,
(* ... *)
    ~(current_ins sust1 (New c)) ->
    ~(current_ins sust1 (Newarray t)) ->
    Support.step p sust1 sust2 ->
    step p (sust1, chs) (sust2, chs)

  | step_new_ok: forall ...,
(* ... *)
    current_ins sust1 (New c) ->
    chs + (size_of_object c) <= max_heap_size ->
    Support.step p sust1 sust2 ->
    step p (sust1, chs) (sust2, (chs + (size_of_object c)))
(* ... *)
```

Memory Example: VF Implementation (2)

```
(* ... *)
  | step_new_OutOfMemory: forall ...,
    current_ins sust1 (New c) ->
    size_of_object p c = size ->
    chs + (size_of_object c) > max_heap_size ->
(* ...
   esize = OutOfMemoryError size,
   bost2 = bottom state after OutOfMemory thrown
 *)
    Support.additional_step p sust1 bost2 sust2 ->
    step p (sust1, chs) (sust2, (chs + esize))
(* ... *)
```

The frameworks compared



Change of Behaviour Only in VF

Horizontal framework

```
Inductive ext_step :=
ESS_step: forall ...,
  step p s t ->
  T = bookkeep p s t S ->
  ext_step p (s,S) (t,T)
```

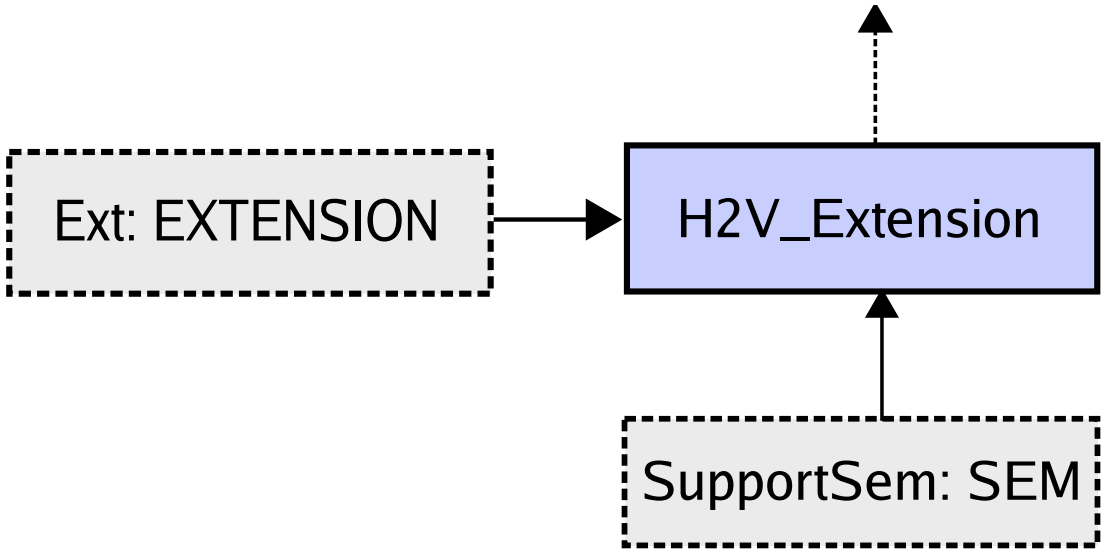
- inductive definition
- `s` and `t` unchanged
- behaviour guaranteed not to change

Vertical framework

```
Parameter step : Program
  -> state_t -> state_t -> Prop
```

- arbitrary `step` definition

Embedding of HF within VF



Properties

- any horizontal extension allowed
- equivalent to ordinary horizontal framework (when applied to `BottomSem`)

Cost of Frameworks

Source code size (lines)

Extension	HF	VF
Heap mem. size	50	204
Instr. counter	26	173

Compiled code size (KB)

Extension	HF	VF
Heap mem. size	148	181
Instr. counter	136	158

HF conceptually simpler

- only the extended part of state in mind
- `bookkeep` function easy to write
- `step` and `additional_step` relations hard to write

Choosing Appropriate Framework

Horizontal Framework

- No change in behaviour
- Tracking additional information:
  - counting use of resources
  - logic strengthening (ghost variables, program traces)

Vertical framework

- Additional behaviour needed:
  - new errors and exceptions signalled
  - step not allowed in some circumstances
- Adding non-determinism:
  - multi-threading
  - non-deterministic errors

Summary

- Two frameworks for extending semantics:
  - applied to Bicolano (Mobius basic JVM formalisation)
  - developed in Coq
  - modular
- Extension examples and frameworks comparison
- Vertical framework more general:
  - arbitrary state and semantics definitions
  - change in behaviour allowed
- Horizontal framework still usable:
  - less expensive in use
  - behaviour guaranteed not to be changed
  - actually used in (extended) Mobius base logic

Story

1. Primary motivation – resource control for Bicolano
2. Horizontal framework idea
   - no change of behaviour recognised as a limitation
3. Vertical framework idea and implementation
   - change of behaviour found disadvantageous in some applications
4. Horizontal framework implemented for ”simpler cases”

Related Work

- Peter D. Mosses; *Action Semantics*; 1992
  - similar to HF
- Sheng Liang & Paul Hudak; *Modular Denotational Semantics for Compiler Construction*; 1996
  - modular monadic semantics
- In both of them framework build with extensions in mind
- Our framworks: attention paid on reusing Bicolano formalisation

Future Work

- Horizontal framework:
  - composing several extensions into one extension
- Vertical framework:
  - extending instructions set
  - extending basic datatypes set
  - make ”extensions subclasses” modularly

Bicolano Flavours

- Bicolano semantics given in 2 flavours:
  - **small step**
  - **big step**
- Vertical framework – small step flavour
- Horizontal framework – both flavours
  - small step – no changes in existing Bicolano code
  - big step – small changes in Bicolano code (cyclic module dependencies)
  - big step actually used in (extended) Mobius program logic

`additional_step` in Details

- `additional_step : Program -> state_t -> bt_state_t -> state_t`
- Provided for further extensions applied to current one:
  - assume SEM_2 extends SEM_1
- Used when SEM_2 changes behaviour, example:
  1. S_1 – SEM_1's part of $SEM2$'s state before step
  2. B_1 – projection of S_1 to bottom state
  3. SEM_2 changes bottom part of state from B_1 to B_2
  4. SEM_2 needs SEM_1's state constructed over B_2
  5. SEM_2 calls SEM_1's `additional_step` p S_1 B_2
  6. S_2 – result, SEM_1's state after "additional step"
  7. SEM_2 builds its state over S_2
- In fact `additional_step : Program -> state_t -> bt_state_t -> state_t -> Prop`

VF: Axiom in SEM

```
Axiom add_step_step_compat: forall ...,
  step p st1 st2 ->
  get_bottom_state st2 bost2 ->
  additional_step p st1 bost2 st2.
```

”Subclasses” in VF

Deterministic projections

```
Parameter fget_bottom_state: state_t -> bottom_state_t.
Axiom get_bottom_state_ok: forall sust bost,
  get_bottom_state sust bost <-> fget_bottom_state sust = bost.

Axiom additional_step_proj: forall p st1 bost2 st2,
  additional_step p st1 bost2 st2 -> get_bottom_state st2 bost2.
```

Conservativity

```
Axiom support_step_implies_step: forall p t s',
  Support.step p (get_support_state t) s' ->
  exists t': state_t, step p t t' AND (get_support_state t') = s'.
Axiom step_implies_support_step:
  forall p t t', step p t t' ->
    Support.step p (get_support_state t) (get_support_state t').
```