

Syntax

```
...

SYNTAX #Id ::= object
      | null
SYNTAX Variable ::= #Int
SYNTAX VariableName ::= v #Int
SYNTAX MethodName ::= <init>
      | #Id
SYNTAX FormalParams ::= List(TypeName, " " )
SYNTAX Params ::= List(Variable, " " )
SYNTAX Selector ::= MethodName ( FormalParams ) V
SYNTAX TypeName ::= #Id
      | #Id / TypeName
SYNTAX TypeReference ::= < #Id , TypeName >
SYNTAX FieldReference ::= < #Id , TypeName , #Id , TypeReference >
SYNTAX MethodReference ::= < #Id , TypeName , Selector >
SYNTAX NewInstructionBase ::= Variable =new TypeReference @ #Int
      | NewInstructionBase ( Params )
SYNTAX GetInstruction ::= Variable =getField FieldReference Variable
      | Variable =getstatic FieldReference
SYNTAX PutInstruction ::= putfield Variable = Variable FieldReference
      | putstatic Variable FieldReference
SYNTAX PhiInstruction ::= Variable =phi( Params )
SYNTAX PhiPhiInstruction ::= Variable =phiphi( Params )
SYNTAX InvokeSpecialInstruction ::= invokespecial MethodReference Params @ #Int exception: Variable
SYNTAX Instruction ::= NewInstruction
      | GetInstruction
      | PutInstruction
      | PhiInstruction
      | PhiPhiInstruction
      | return
      | InvokeSpecialInstruction
      | noinstruction
SYNTAX BBEdge ::= #Id -> #Id ;
SYNTAX BlockBody ::= List(Instruction, " " )
SYNTAX Block ::= #Id { BlockBody }
SYNTAX TaskUnit ::= BBEdge
      | Block
SYNTAX Task ::= TaskUnit
      | Task Task
SYNTAX Program ::= MethodDefinition
      | start
      | analysis
      | done
      | Program and here comes another method Program
SYNTAX MethodDefinition ::= MethodReference { Task }
END MODULE
```

MODULE KWALA

IMPORTS KWALA-SYNTAX

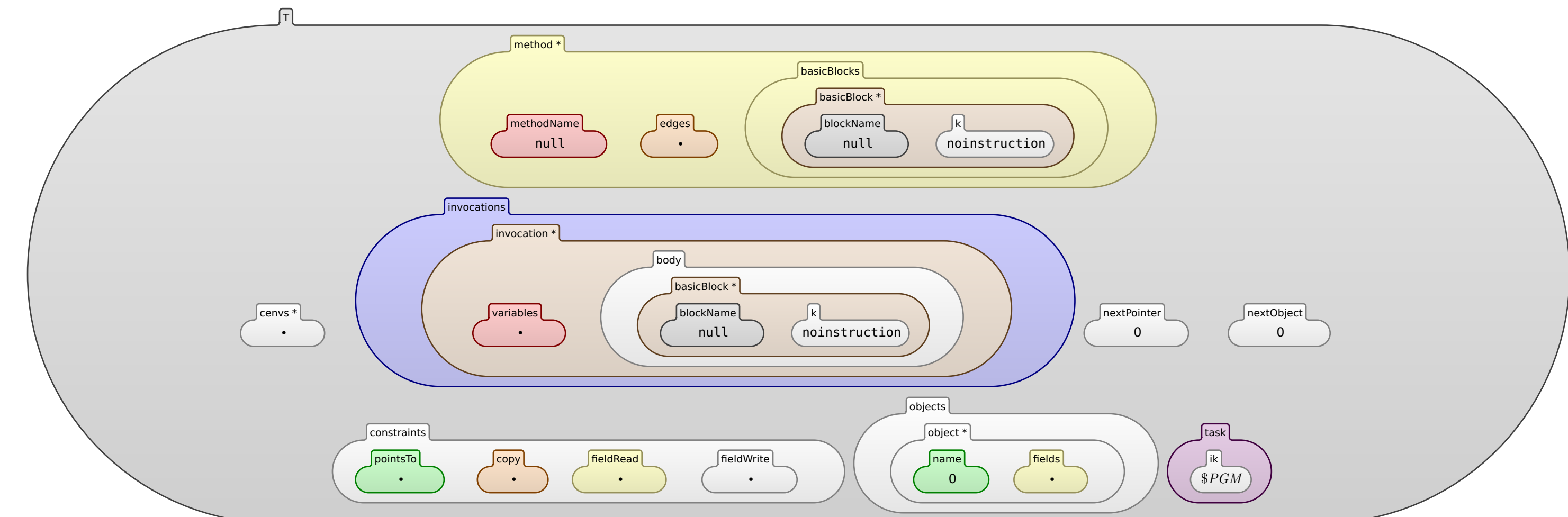
Semantics

...

Configuration

...

CONFIGURATION:



Processing Basic Blocks

...

RULE  $T_1 \ T_2 \Rightarrow T_1 \cap T_2$

RULE  $I_1 \ ; \ BB_2 \Rightarrow I_1 \cap BB_2$

SYNTAX  $ListItem ::= [ \#Id , \#Id ]$

RULE  $BB_1 \rightarrow BB_2 ;$  edges  $[ BB_1 , BB_2 ]$

RULE  $BB : \{ BB \}$  method  $basicBlock$   $blockName$   $BB$   $k$   $BB$   $BB$

RULE  $BB : \{ \}$  method  $basicBlock$   $blockName$   $BB$   $k$   $noinstruction$

RULE  $start$

RULE  $analysis$  basicBlocks  $BBs$  invocation  $variables$   $body$   $BBs$

Gathering Constraints

...

SYNTAX  $K ::= wrappedList( Bag )$

Phi functions

RULE  $V_i = phi( P )$  variables  $V_i \mapsto NP$  nextPointer  $NP$   $NP + Int \ 1$

RULE  $V_i = phiphi( V_2 , P )$  variables  $V_i \mapsto P_i \mid V_i \mapsto P_2$  copy  $P_2 \mapsto wrappedList( censvs RestSet )$  when  $\neg Bool \ P_i$  in  $RestSet$

RULE  $V_i = phiphi( V_2 , P )$  variables  $V_i \mapsto P_i \mid V_i \mapsto P_2$  copy  $P_2 \mapsto wrappedList( censvs P_i )$  when  $\neg Bool \ P_2$  in keys  $Rest$

RULE  $V_i = phi( )$

Get field

SYNTAX  $FieldToPointer ::= ( FieldReference \> \#Int )$

RULE  $V_i = getField \ F \ V_2$  variables  $V_i \mapsto NP \mid V_i \mapsto P_2$  nextPointer  $NP$   $NP + Int \ 1$  fieldRead  $P_2 \mapsto wrappedList( censvs ( F \> NP ) )$  when  $\neg Bool \ P_2$  in keys  $Rest$

RULE  $V_i = getField \ F \ V_2$  variables  $V_i \mapsto NP \mid V_i \mapsto P_2$  nextPointer  $NP$   $NP + Int \ 1$  fieldRead  $P_2 \mapsto wrappedList( censvs RestSet )$   $( F \> P_2 )$

Put field

RULE  $putfield \ V_i = V_2 \ F$  variables  $V_i \mapsto P_i \mid V_i \mapsto P_2$  fieldWrite  $P_i \mapsto wrappedList( censvs ( F \> P_2 ) )$  pointsTo  $P_i \mapsto wrappedList( censvs O )$  object  $name$   $O$   $fields$   $F \mapsto NP$  nextPointer  $NP$   $NP + Int \ 1$  when  $\neg Bool \ P_i$  in keys  $Rest$

RULE  $putfield \ V_i = V_2 \ F$  variables  $V_i \mapsto P_i \mid V_i \mapsto P_2$  fieldWrite  $P_i \mapsto wrappedList( censvs RestSet )$   $( F \> P_2 )$  pointsTo  $P_i \mapsto wrappedList( censvs O )$  object  $name$   $O$   $fields$   $F \mapsto NP$  nextPointer  $NP$   $NP + Int \ 1$

New instruction

RULE  $V_i = new \ TR \ @$  variables  $V_i \mapsto NP$  nextPointer  $NP$   $NP + Int \ 1$  nextObject  $NO$   $NO + Int \ 1$  pointsTo  $NP \mapsto wrappedList( censvs NO )$  objects  $object$   $name$   $NO$   $fields$   $•$

Resolving Constraints

First type of graph: if copy encountered, propagate points-to

RULE  $B \mapsto wrappedList( censvs A )$  pointsTo  $B \mapsto wrappedList( censvs O )$   $A \mapsto wrappedList( censvs OldSet )$  when  $\neg Bool \ O$  in  $OldSet$

RULE  $B \mapsto wrappedList( censvs A )$  pointsTo  $B \mapsto wrappedList( censvs O )$   $RestOfTheWorld$   $A \mapsto wrappedList( censvs O )$  when  $\neg Bool \ A$  in keys  $RestOfTheWorld$

Second type of graph: if field-read and points-to encountered, propagate copy

RULE  $P_i \mapsto wrappedList( censvs OldSet )$  pointsTo  $B \mapsto wrappedList( censvs O )$  fieldRead  $B \mapsto wrappedList( censvs ( F \> A ) )$  object  $name$   $O$   $fields$   $F \mapsto P_i$  when  $\neg Bool \ A$  in  $OldSet$

RULE  $P_i \mapsto wrappedList( censvs OldMap )$  pointsTo  $B \mapsto wrappedList( censvs O )$  fieldRead  $B \mapsto wrappedList( censvs ( F \> A ) )$  object  $name$   $O$   $fields$   $F \mapsto P_i$  when  $\neg Bool \ P_i$  in keys  $OldMap$

Third type of graph: if field-write and points-to encountered, propagate copy

RULE  $B \mapsto wrappedList( censvs OldSet )$  pointsTo  $A \mapsto wrappedList( censvs O )$  fieldWrite  $A \mapsto wrappedList( censvs ( F \> B ) )$  object  $name$   $O$   $fields$   $F \mapsto P_i$  when  $\neg Bool \ P_i$  in  $OldSet$

RULE  $B \mapsto wrappedList( censvs OldMap )$  pointsTo  $A \mapsto wrappedList( censvs O )$  fieldWrite  $A \mapsto wrappedList( censvs ( F \> B ) )$  object  $name$   $O$   $fields$   $F \mapsto P_i$  when  $\neg Bool \ B$  in keys  $OldMap$