MODULE KWALA-SYNTAX

## Syntax

...

SYNTAX *Hnt* ::= `object`
| `null`
SYNTAX *Variable* ::= *Hnt*
SYNTAX *VariableName* ::= *Hnt*
SYNTAX *MethodName* ::= `<init>`
| *Hnt*
| `main`
SYNTAX *FormalParams* ::= *List{TypeName, ","}*
SYNTAX *Params* ::= *List{Variable, ","}*
SYNTAX *Selector* ::= *MethodName* `[` *FormalParams* `]`
SYNTAX *TypeName* ::= *Hnt*
| *Hnt* `[` *TypeName*
SYNTAX *TypeReference* ::= `<` *Hnt* `,` *TypeName* `>`
SYNTAX *FieldReference* ::= `<` *Hnt* `,` *TypeName* `,` *Hnt* `,` *TypeReference* `>`
SYNTAX *MethodReference* ::= `<|` *Hnt* `,` *TypeName* `,` *Selector* `|>`
| `none`
SYNTAX *NewInstructionBasic* ::= *Variable* `=new` *TypeReference* `@` *Hnt*
SYNTAX *NewInstruction* ::= *NewInstructionBase*
| *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* `@` *Hnt* `exception:` *Variable*
SYNTAX *Instruction* ::= *NewInstruction*
| *GetInstruction*
| *PutInstruction*
| *PhiInstruction*
| *PhiPhiInstruction*
| `return`
| *InvokeSpecialInstruction*
| `noinstruction`
SYNTAX *BBEdge* ::= *Hnt* `->` *Hnt* `;`
SYNTAX *BlockBody* ::= *List{Instruction, ";"}*
SYNTAX *Block* ::= *Hnt* `:` `{` *BlockBody* `}`
SYNTAX *TaskUnit* ::= *BBEdge*
| *Block*
SYNTAX *Task* ::= *TaskUnit*
| *Task* *Task*
SYNTAX *Program* ::= *MethodDefinition*
| `start`
| `analysis`
| `done`
| *Program* `+++` *Program*
SYNTAX *MethodDefinition* ::= *MethodReference* `{` *Task* `}`

END MODULE

MODULE KWALA

IMPORTS KWALA-SYNTAX

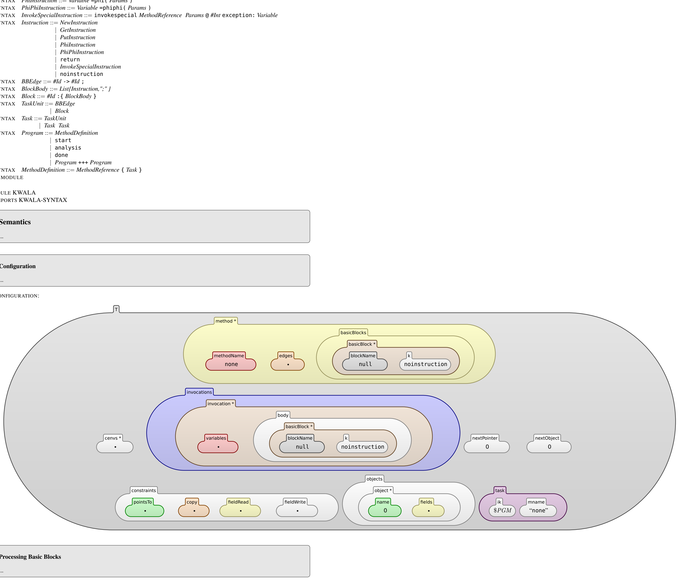
## Semantics

...

## Configuration

...

CONFIGURATION:




## Processing Basic Blocks

...

RULE $T_1 \;\; T_0 \Rightarrow T_1 \curvearrowright T_0$

RULE $I_1 \; ; \; BB_2 \Rightarrow I_1 \curvearrowright BB_2$

RULE $Pg_1 \;\texttt{+++}\; Pg_2 \Rightarrow Pg_1 \curvearrowright Pg_2$

SYNTAX *Listhem* ::= `[` *Hnt* `,` *Hnt* `]`

RULE



RULE



RULE



RULE



RULE



RULE



## Gathering Constraints

...

SYNTAX *K* ::= `wrappedList(` *Hnt* `)`

### Phi functions

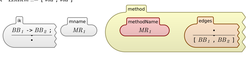
RULE



RULE



when $\neg_{Bool}\, P_1$ in $RestSet$

RULE


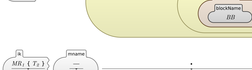

when $\neg_{Bool}\, P_0$ in keys $Rest$

RULE

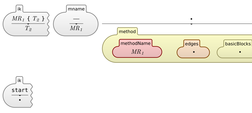


### Get field

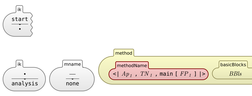
SYNTAX *FieldToPointer* ::= `{` *FieldReference* `|>` *Hnt* `}`

RULE


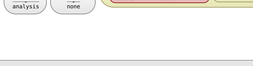

when $\neg_{Bool}\, P_0$ in keys $Rest$

RULE


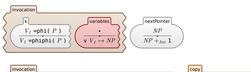

### Put field

RULE



when $\neg_{Bool}\, P_1$ in keys $Rest$

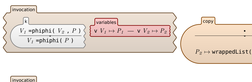
RULE



### New instruction

RULE



## Resolving Constraints

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

RULE


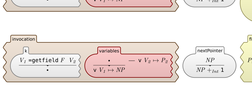

when $\neg_{Bool}\, O$ in $OldSet$

RULE



when $\neg_{Bool}\, A$ in keys $RestOfTheWorld$

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

RULE



when $\neg_{Bool}\, A$ in $OldSet$

RULE



when $\neg_{Bool}\, P_2$ in keys $OldMap$

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

RULE



when $\neg_{Bool}\, P_1$ in $OldSet$

RULE



when $\neg_{Bool}\, B$ in keys $OldMap$

END MODULE