MODULE KWALA-SYNTAX

Syntax

...

SYNTAX *Rhs* ::= `object`
| `null`

SYNTAX *Variable* ::= *Rhs*

SYNTAX *VariableName* ::= `v` *Int*

SYNTAX *MethodName* ::= `<init>`
| *Rhs*

SYNTAX *FormalParams* ::= *List*{*TypeName*,","}

SYNTAX *Params* ::= *List*{*Variable*,","}

SYNTAX *Selector* ::= *MethodName* `(` *FormalParams* `)V`

SYNTAX *TypeName* ::= *Rhs*
| *Rhs* `/` *TypeName*

SYNTAX *TypeReference* ::= `<` *Rhs* `,` *TypeName* `>`

SYNTAX *FieldReference* ::= `<` *Rhs* `,` *TypeName* `,` *Rhs* `,` *TypeReference* `>`

SYNTAX *MethodReference* ::= `<` *Rhs* `,` *TypeName* `,` *Selector* `>`

SYNTAX *NewInstructionBase* ::= *Variable* `=new` *TypeReference* `@` *Rhs*

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* `@` *Rhs* `exception:` *Variable*

SYNTAX *Instruction* ::= *NewInstruction*
| *GetInstruction*
| *PutInstruction*
| *PhiInstruction*
| *PhiPhiInstruction*
| `return`
| *InvokeSpecialInstruction*
| `noinstruction`

SYNTAX *BBEdge* ::= *Rhs* `->` *Rhs* `;`

SYNTAX *BlockBody* ::= *List*{*Instruction*,";"}

SYNTAX *Block* ::= *Rhs* `:{` *BlockBody* `}`

SYNTAX *TaskUnit* ::= *BBEdge*
| *Block*

SYNTAX *Task* ::= *TaskUnit*
| *Task* *Task*

SYNTAX *Program* ::= *MethodDefinition*
| `start`
| `analysis`
| `done`
| *Program* `andherecomesanothermethod` *Program*

SYNTAX *MethodDefinition* ::= *MethodReference* `{` *Task* `}`

END MODULE

MODULE KWALA
IMPORTS KWALA-SYNTAX

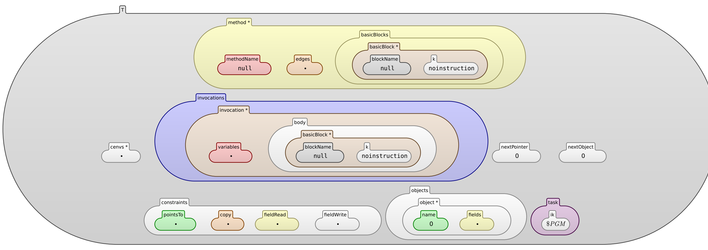
Semantics

...

Configuration

...

CONFIGURATION:



Processing Basic Blocks

...

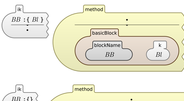
RULE $T_1 \;\; T_2 \Rightarrow T_1 \curlywedge T_2$

RULE $I_1 \; ; \; BB_2 \Rightarrow I_1 \curlywedge BB_2$

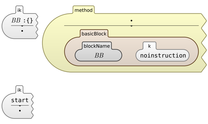
SYNTAX *ListItem* ::= `[` *Rhs* `,` *Rhs* `]`

RULE

RULE

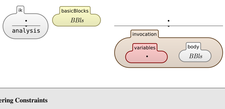


RULE



RULE

RULE



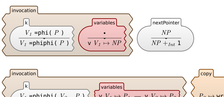
Gathering Constraints

...

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

Phi functions

RULE



RULE

when $\neg_{Bool}\ P_1$ `in` *RestSet*

RULE

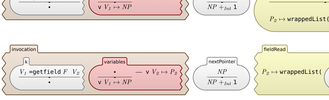
when $\neg_{Bool}\ P_2$ `in keys` *Rest*

RULE

Get field

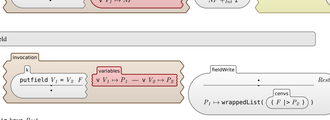
SYNTAX *FieldToPointer* ::= `{` *FieldReference* `|>` *Rhs* `}`

RULE



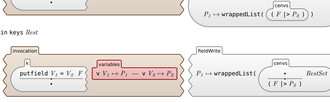
when $\neg_{Bool}\ P_2$ `in keys` *Rest*

RULE



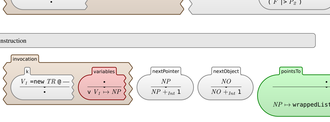
Put field

RULE



when $\neg_{Bool}\ P_1$ `in keys` *Rest*

RULE



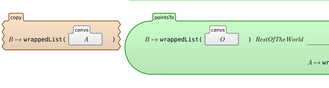
New instruction

RULE

Resolving Constraints

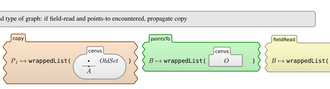
First type of graph: if copy encountered, propagate points-to

RULE



when $\neg_{Bool}\ O$ `in` *OldSet*

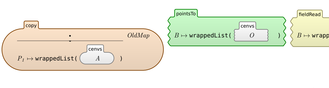
RULE



when $\neg_{Bool}\ A$ `in keys` *RestOfTheWorld*

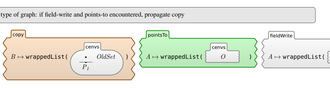
Second type of graph: if field-read and points-to encountered, propagate copy

RULE



when $\neg_{Bool}\ P_1$ `in keys` *OldSet*

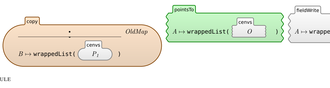
RULE



when $\neg_{Bool}\ P_1$ `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