MODULE KWALA-SYNTAX

Syntax

...

SYNTAX *#Id* ::= `object`
| `null`
SYNTAX *Variable* ::= *#Int*
SYNTAX *VariableName* ::= `v` *#Int*
SYNTAX *Selector* ::= `<init>()V`
SYNTAX *TypeName* ::= *#Id*
| *#Id* `/` *TypeName*
SYNTAX *TypeReference* ::= `<` *#Id* `,` *TypeName* `>`
SYNTAX *FieldReference* ::= `<` *#Id* `,` *TypeName* `,` *#Id* `,` *TypeReference* `>`
SYNTAX *MethodReference* ::= `<` *#Id* `,` *TypeName* `,` *Selector* `>`
SYNTAX *Params* ::= *List{Variable,","}*
SYNTAX *NewInstructionBase* ::= *Variable* `=new` *TypeReference* `@` *#Int*
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* `@` *#Int* `exception:` *Variable*
SYNTAX *Instruction* ::= *NewInstruction*
| *GetInstruction*
| *PutInstruction*
| *PhiInstruction*
| *PhiPhiInstruction*
| `return`
| *InvokeSpecialInstruction*
| `noinstruction`
| `main`
SYNTAX *BBEdge* ::= *#Id* `->` *#Id* `;`
SYNTAX *BlockBody* ::= *Instruction*
| *BlockBody* `;` *BlockBody*
SYNTAX *Block* ::= *#Id* `:{` *BlockBody* `}`
| *#Id* `:{}`
SYNTAX *TaskUnit* ::= *BBEdge*
| *Block*
| `start`
| `analysis`
| `done`
SYNTAX *Task* ::= *TaskUnit*
| *Task* *Task*

END MODULE

MODULE KWALA
IMPORTS KWALA-SYNTAX

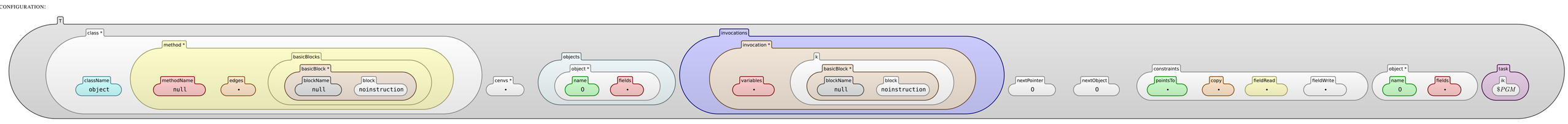
Semantics

...

Configuration

...

CONFIGURATION:



Processing Basic Blocks

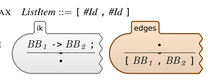
...

RULE $T_1 \quad T_2 \Rightarrow T_1 \curvearrowright T_2$

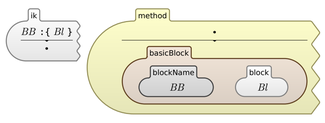
RULE $I_1 \ ; \ BBl_2 \Rightarrow I_1 \curvearrowright BBl_2$

SYNTAX *ListItem* ::= `[` *#Id* `,` *#Id* `]`

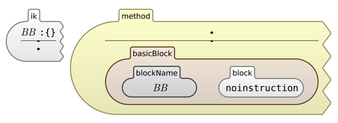
RULE



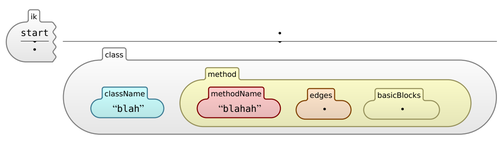
RULE



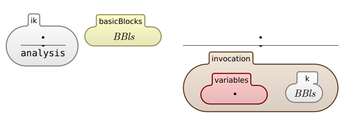
RULE



RULE



RULE



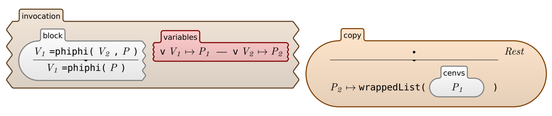
Gathering Constraints

...

Phi functions

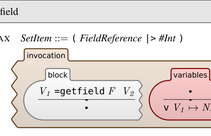
RULE

RULE



RULE

RULE

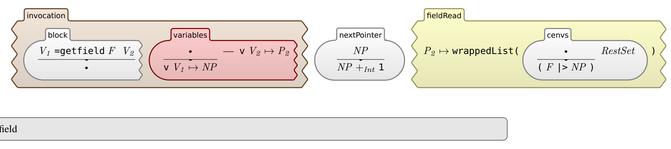


Get field

SYNTAX *SetItem* ::= `(` *FieldReference* `|>` *#Int* `)`

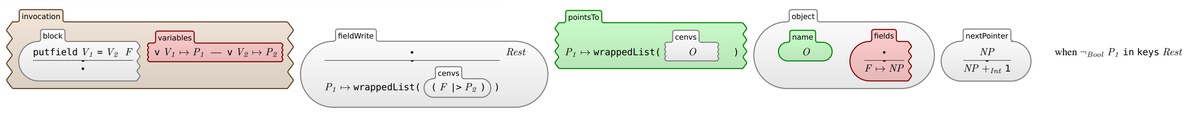
RULE

RULE



Put field

RULE

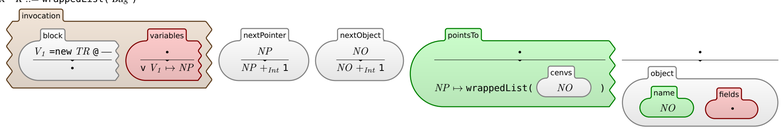


RULE

New instruction

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

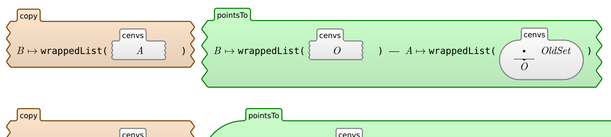
RULE



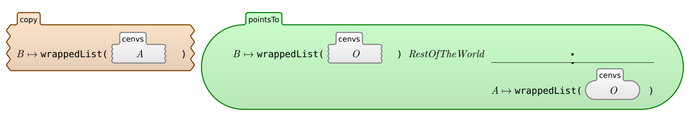
Resolving Constraints

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

RULE

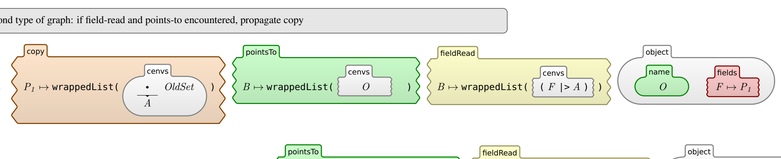


RULE

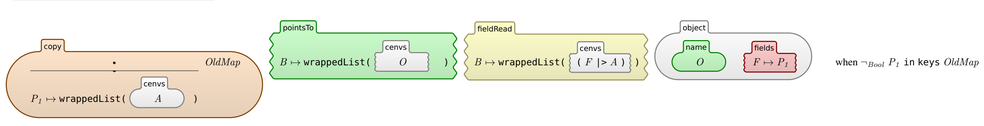


Second type of graph: if field-read and points-to encountered, propagate copy

RULE

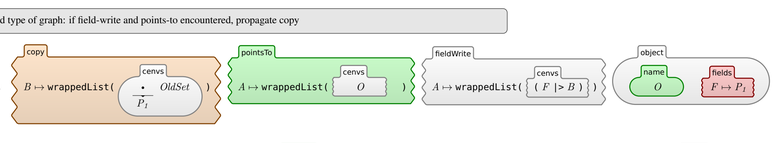


RULE

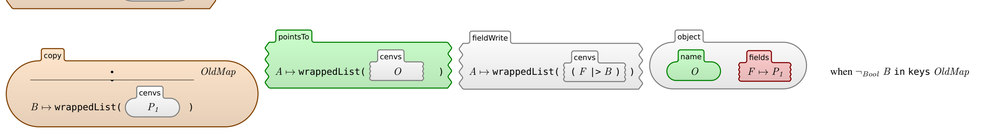


Third type of graph: if field-write and points-to encountered, propagate copy

RULE



RULE



END MODULE