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 ::= List{Instruction,";"}
SYNTAX  Block ::= #Id :{ BlockBody }
SYNTAX  TaskUnit ::= BBEdge
                   | Block
                   | start
                   | analysis
                   | done
SYNTAX  Task ::= TaskUnit
               | Task  Task
```

END MODULE

MODULE KWALA
IMPORTS KWALA-SYNTAX

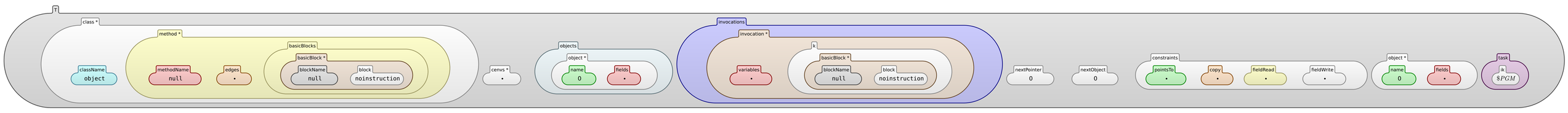
Semantics

...

Configuration

...

CONFIGURATION:



Processing Basic Blocks

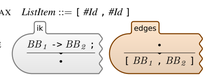
...

RULE $T_1 \;\; T_2 \Rightarrow T_1 \frown T_2$

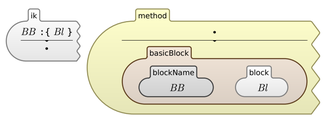
RULE $I_1 \;;\; BBl_2 \Rightarrow I_1 \frown BBl_2$

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

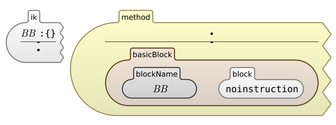
RULE



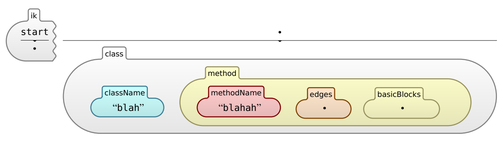
RULE



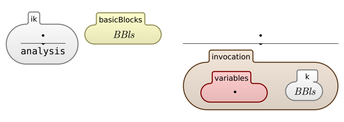
RULE



RULE



RULE

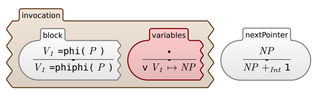


Gathering Constraints

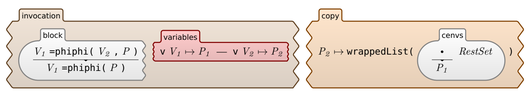
...

Phi functions

RULE

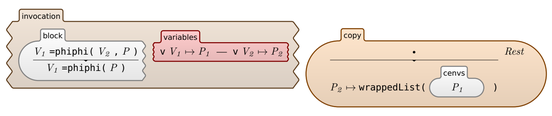


RULE



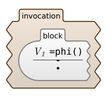
when $\neg_{Bool}$ P_1 in $RestSet$

RULE



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

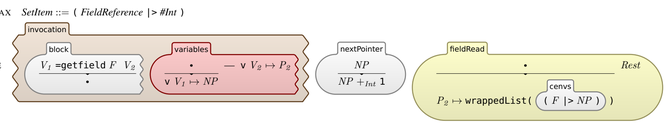
RULE



Get field

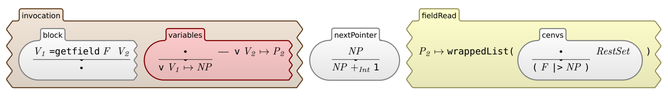
SYNTAX *SetItem* ::= (*FieldReference* |> *#Int*)

RULE



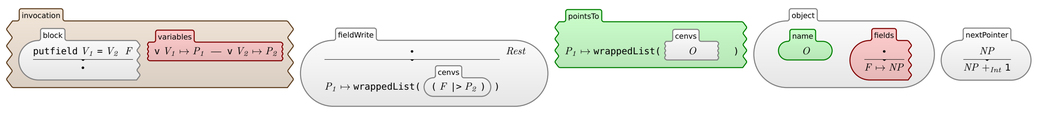
when $\neg_{Bool}$ P_2 in keys $Rest$

RULE



Put field

RULE



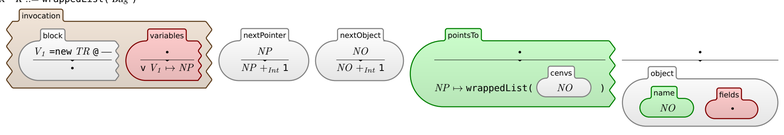
when $\neg_{Bool}$ P_1 in keys $Rest$

RULE

New instruction

SYNTAX K ::= wrappedList(*Bag*)

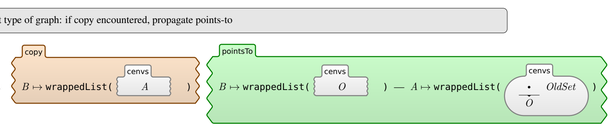
RULE



Resolving Constraints

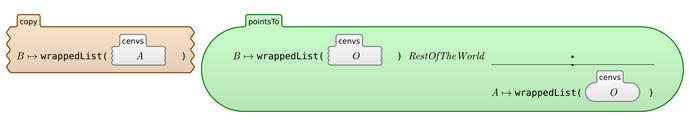
First type of graph: if copy encountered, propagate points-to

RULE



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

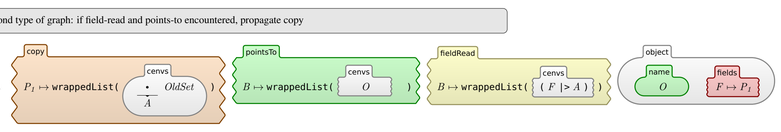
RULE



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

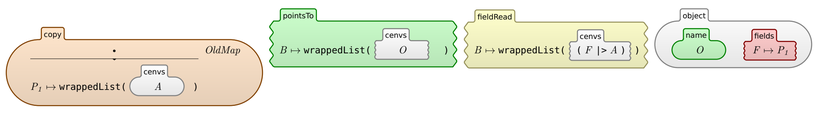
Second type of graph: if field-read and points-to encountered, propagate copy

RULE



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

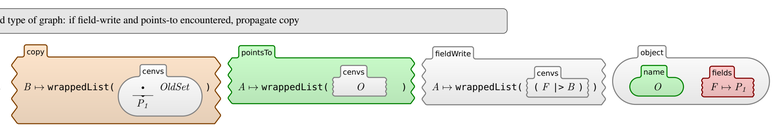
RULE



when $\neg_{Bool}$ P_1 in keys $OldMap$

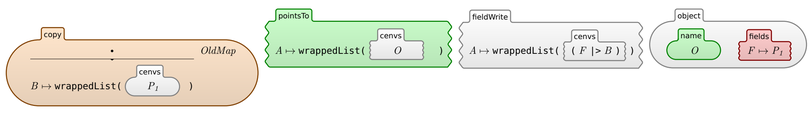
Third type of graph: if field-write and points-to encountered, propagate copy

RULE



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

RULE



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

END MODULE