IDFFL: An Intraprocedural Data Flow Framework for LLVM

Project Report Presentation By:
Sunil Singh (18111076)

Sections:

- A brief Intro to LLVM
- Introduction to Data Flow Analysis
- Foundation Theory for Framework
- Implementation of Framework for LLVM
- Possible future work

A brief Intro to LLVM

Compiler Infrastructure, open source

Tools, analysis passes, transformation passes and some subprojects

Being used in compiler related research a lot now

A brief Intro to LLVM

Compiler Infrastructure, open source

Tools, analysis passes, transformation passes and some subprojects

Being used in compiler related research a lot now

Frontend	---->	Optimizer	---->	Backend
high level language		in LLVM IR		Target dependent
C, C++, Fortran, etc				X86, PowerPC, ARM, etc

A brief Intro to LLVM

Compiler Infrastructure, open source

Tools, analysis passes, transformation passes and some subprojects

Being used in compiler related research a lot now

Frontend	---->	Optimizer	---->	Backend
high level language		in LLVM IR		Target dependent
C, C++, Fortran, etc				X86, PowerPC, ARM, etc

LLVM IR

- Low level RISC like instructions
- Can represent high level language constructs
- Defined with well-defined semantics
- Three forms bitcode, human-readable, and in-memory

A brief Intro to LLVM

Compiler Infrastructure, open source
Tools, analysis passes, transformation passes and some subprojects
Being used in compiler related research a lot now

Frontend	---->	Optimizer	---->	Backend
high level language		in LLVM IR		Target dependent
C, C++, Fortran, etc				X86, PowerPC, ARM, etc

LLVM IR

- Low level RISC like instructions
- Can represent high level language constructs
- Defined with well-defined semantics
- Three forms bitcode, human-readable, and in-memory

```
clang -S -emit-llvm hello.c     <-----     gives a .ll file
clang -c -emit-llvm hello.c     <-----     gives a .bc file
```

A brief Intro to LLVM(cont...)

bitcode(*.bc)
Efficient format for storage
Backward compatibility

llvm-dis →

← llvm-as

human readable(*.ll)

A brief Intro to LLVM(cont...)

bitcode(*.bc)
Efficient format for storage
Backward compatibility

$\xrightarrow{\text{llvm-dis}}$

$\xleftarrow[\text{llvm-as}]{}$

human readable(*.ll)

- Typed IR
- No explicit conversions
- SSA form

A brief Intro to LLVM(cont...)

bitcode(*.bc) $\xrightarrow{\text{llvm-dis}}$ human readable(*.ll)

human readable(*.ll) $\xrightarrow{\text{llvm-as}}$ bitcode(*.bc)

Efficient format for storage
Backward compatibility

- Typed IR
- No explicit conversions
- SSA form

IR Layout

Module : a translation unit or many merged together

- consists Target info, global variables, function declarations, function definitions , and other stuff.

Function :

- consists a list of arguments, an entry Basic block, and more Basic blocks

Basic Block :

- consists a label, some phi instructions, some other instructions, a terminator instruction

A brief Intro to LLVM(cont...)

```
int factorial(int val);

int main(int argc, char** argv){
	return factorial(2) * 7 == 42;
}
```

LLVM IR

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv){
	%1 = call i32 @factorial(i32 2)
	%2 = mul i32 %1, 7
	%3 = icmp eq i32 %2, 42
	%res = zext i1 %3 to i32
	ret i32 %res
}
```

A brief Intro to LLVM(cont...)

```
int factorial(int val);

int main(int argc, char** argv){
	return factorial(2) * 7 == 42;
}
```

declaration of function

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv){
	%1 = call i32 @factorial(i32 2)
	%2 = mul i32 %1, 7
	%3 = icmp eq i32 %2, 42
	%res = zext i1 %3 to i32
	ret i32 %res
}
```

A brief Intro to LLVM(cont...)

```
int factorial(int val);

int main(int argc, char** argv){
        return factorial(2) * 7 == 42;
}
```

declaration of function

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv){
        %1 = call i32 @factorial(i32 2)
        %2 = mul i32 %1, 7
        %3 = icmp eq i32 %2, 42
        %res = zext i1 %3 to i32
        ret i32 %res
}
```

definition of function

A brief Intro to LLVM(cont...)

```
int factorial(int val);

int main(int argc, char** argv){
        return factorial(2) * 7 == 42;
}
```

declaration of function

typed argument

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv){
        %1 = call i32 @factorial(i32 2)
        %2 = mul i32 %1, 7
        %3 = icmp eq i32 %2, 42
        %res = zext i1 %3 to i32
        ret i32 %res
}
```

definition of function

A brief Intro to LLVM(cont...)

```
int factorial(int val);

int main(int argc, char** argv){
        return factorial(2) * 7 == 42;
}
```

declaration of function

typed argument

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv){
        %1 = call i32 @factorial(i32 2)
        %2 = mul i32 %1, 7
        %3 = icmp eq i32 %2, 42
        %res = zext i1 %3 to i32
        ret i32 %res
}
```

definition of function

A brief Intro to LLVM(cont...)

```
int factorial(int val);

int main(int argc, char** argv){
        return factorial(2) * 7 == 42;
}
```

declaration of function

typed argument

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv){
        %1 = call i32 @factorial(i32 2)
        %2 = mul i32 %1, 7
        %3 = icmp eq i32 %2, 42
        %res = zext i1 %3 to i32
        ret i32 %res
}
```

definition of function

terminator instruction

A brief Intro to LLVM(cont...)

```
int factorial(int val);

int main(int argc, char** argv){
        return factorial(2) * 7 == 42;
}
```

@... are global symbols : functions and global variables

%.... -- virtual registers

unnamed -- %<numeric>

named -- %<alphanumeric>

infinite registers

declaration of function

typed argument

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv){
        %1 = call i32 @factorial(i32 2)
        %2 = mul i32 %1, 7
        %3 = icmp eq i32 %2, 42
        %res = zext i1 %3 to i32
        ret i32 %res
}
```

definition of function

terminator instruction

A brief Intro to LLVM(cont...)

```
int factorial(int val);

int main(int argc, char** argv){
        return factorial(2) * 7 == 42;
}
```

declaration of function

typed argument

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv){
        %1 = call i32 @factorial(i32 2)
        %2 = mul i32 %1, 7
        %3 = icmp eq i32 %2, 42
        %res = zext i1 %3 to i32
        ret i32 %res
}
```

definition of function

terminator instruction

@... are global symbols : functions and global variables

%.... -- virtual registers

- unnamed -- %<numeric>
- named -- %<alphanumeric>
- infinite registers

some types are

- void type, function type, integer type, floating point type, pointer type, array type, structure type, etc.

A brief Intro to LLVM(cont...)

```c
int factorial(int val);

int main(int argc, char** argv){
        return factorial(2) * 7 == 42;
}
```

declaration of function

typed argument

```llvm
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv){
        %1 = call i32 @factorial(i32 2)
        %2 = mul i32 %1, 7
        %3 = icmp eq i32 %2, 42
        %res = zext i1 %3 to i32
        ret i32 %res
}
```

definition of function

terminator instruction

@... are global symbols : functions and global variables

%.... -- virtual registers

- unnamed -- %<numeric>
- named -- %<alphanumeric>
- infinite registers

some types are

- void type, function type, integer type, floating point type, pointer type, array type, structure type, etc.

phi inst : phi function in SSA form

- select value based on the bb that executed previously
- -O1 compiled IR usually have phi inst
- -O0 compiled IR has load and store insts, usually do not have phi insts

Introduction to Data Flow Analysis

- A static program analysis technique

Introduction to Data Flow Analysis

- A static program analysis technique
- Being used for Semantic Validity of a Program, Program Transformation, Debugging, etc.

Introduction to Data Flow Analysis

- A static program analysis technique
- Being used for Semantic Validity of a Program, Program Transformation, Debugging, etc.
- Consider these compiler optimizations:

  Common subexpression elimination
  Copy propagation
  Dead code elimination
  Constant folding
  Code motion

Introduction to Data Flow Analysis

- A static program analysis technique
- Being used for Semantic Validity of a Program, Program Transformation, Debugging, etc.
- Consider these compiler optimizations:
  Common subexpression elimination
  Copy propagation
  Dead code elimination
  Constant folding
  Code motion
- Local Analysis - only in a basic block
- Global Analysis - in a Control flow graph of a Function

Introduction to Data Flow Analysis

- A static program analysis technique
- Being used for Semantic Validity of a Program, Program Transformation, Debugging, etc.
- Consider these compiler optimizations:
  Common subexpression elimination
  Copy propagation
  Dead code elimination
  Constant folding
  Code motion
- Local Analysis - only in a basic block
- Global Analysis - in a Control flow graph of a Function
- Interprocedural DFA - across functions (not implemented yet)

Introduction to Data Flow Analysis(Cont...)

In a Control Flow Graph two dummy basic blocks are added

- start : will work as first block of the procedure
  end : the last block of the procedure

Introduction to Data Flow Analysis(Cont...)

In a Control Flow Graph two dummy basic blocks are added

- start : will work as first block of the procedure
  end : the last block of the procedure

Program point : before and after statement and before and after basic block

- compute information on each program point on each possible execution path
- start with imprecise value and go to precise and safe value
- flow values associated with each PP shows the states possible

Introduction to Data Flow Analysis(Cont...)

In a Control Flow Graph two dummy basic blocks are added

- start : will work as first block of the procedure
  end : the last block of the procedure

Program point : before and after statement and before and after basic block

- compute information on each program point on each possible execution path
- start with imprecise value and go to precise and safe value
- flow values associated with each PP shows the states possible

Transfer function : to define relationship between program points before and after stmt

- can be defined for basic blocks

Merge operation : to combine information from predecessor blocks or successors blocks

- for forward analysis : combine predecessor info
- for backward analysis : combine successor info

Local information of basic block : generated by BB and destroyed by BB

- used in transfer functions

Introduction to Data Flow Analysis(Cont...)

Equations for forward analysis:

$$OUT[b] = f_b(IN[b])$$

$$IN[b] = \oplus_{p \in PRED(b)} OUT[p]$$

Where,

IN[b] is incoming info for basic block b

OUT[b] is outgoing info for basic block b

f_b is transfer function for basic block b

$\oplus$ is merge operation

PRED(b) is a list of predecessor of b

For backward analysis,

- exchange IN and OUT and replace predecessor list by successor list

f_b uses kill and gen local information and equation becomes

$$OUT[b] = IN[b] - kill[b] \cup gen[b]$$

Foundation Theory for Framework

A unified framework to defined the core concepts, to give answers for correctness, precision, convergence and speed of convergence for many similar problems and to reuse code.

Framework has four elements (V, $\wedge$, F, D):

V is the domain of values
$\wedge$ is the meet operator, a binary relation, with reflexivity, antisymmetry, and transitivity
F is a set of flow functions
D is the direction of the flow

(V,$\wedge$) define a semi-lattice with a top element $\top$ and an optional bottom element $\perp$.

- $x,y \in V$ are ordered : $x \leq y$ then $x \wedge y = x$
- The lattice is finite. Height = max # $\leq$ relations
- called finite descending chain

F has identity function $I(x) = x$, is closed under composition and are monotone and distributive.

Foundation Theory for Framework(cont...)

The iterative algorithm resemble the fixed point algorithm

```
x = ⊤;
while(x != f(x))
        x = f(x);
return x;
```

Fixed point algorithm
Where,
f is a monotone function defined on a semilattice

```
In{Exit} = BI
for (every other block B)
        In[B] = ⊤
change = 1
while (change)
        change = 0
        for (every basic block b)
                Out_b = ⊓_{s∈succ(n)} In_b
                tmp_b = f_p(Out_b)
                if tmp_b and In_b not same
                In_b = tmp_b
                change = 1
```

Iterative algorithm for data flow analysis

Foundation Theory for Framework(cont...)

The iterative algorithm resemble the fixed point algorithm

```
x = ⊤;
while(x != f(x))
        x = f(x);
return x;
```

Fixed point algorithm
Where,
f is a monotone function
defined on a semilattice

```
In{Exit} = BI
for (every other block B)
        In[B] = ⊤
change = 1
while (change)
        change = 0
        for (every basic block b)
                Out_b = ⊓_{s∈succ(n)} In_b
                tmp_b = f_p(Out_b)
                if tmp_b and In_b not same
                In_b = tmp_b
                change = 1
```

Iterative algorithm for data flow analysis

- Ideal Solution : solution of all executable paths(undecidable)
- Meet over paths: MOP(n) = f_{pi}(T), for all paths p_i reaching n (MOP)
- Result of iterative algorithm is called maximal fixedpoint solution(MFP)

Foundation Theory for Framework(cont...)

The iterative algorithm resemble the fixed point algorithm

```
x = ⊤;
while(x != f(x))
        x = f(x);
return x;
```

Fixed point algorithm
Where,
f is a monotone function
defined on a semilattice

```
In{Exit} = BI
for (every other block B)
        In[B] = ⊤
change = 1
while (change)
        change = 0
        for (every basic block b)
                Out_b = ⊓_{s∈succ(n)} In_b
                tmp_b = f_p(Out_b)
                if tmp_b and In_b not same
                In_b = tmp_b
                change = 1
```

Iterative algorithm for data flow analysis

$$MFP \leq MOP \leq IDEAL$$

- Ideal Solution : solution of all executable paths(undecidable)
- Meet over paths: MOP(n) = $f_{pi}(T)$, for all paths p_i reaching n (MOP)
- Result of iterative algorithm is called maximal fixedpoint solution(MFP)

Implementation of Framework for LLVM

We use C++ templates and LLVM as backbone in our framework

We define mostly all classes as templates

- user can defined own properties

Implementation of Framework for LLVM

We use C++ templates and LLVM as backbone in our framework

We define mostly all classes as templates

- user can defined own properties

Two sections : Storage section and Analysis section

Storage section :

- Two available classes to use BoundedSet<> and UnBoundedSet<>

BoundedSet<>

UnBoundedSet<>

Implementation of Framework for LLVM

We use C++ templates and LLVM as backbone in our framework

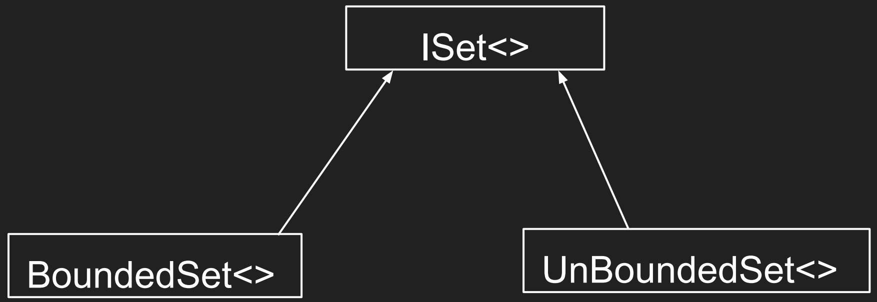
We define mostly all classes as templates

- user can defined own properties

Two sections : Storage section and Analysis section

Storage section :

- Two available classes to use BoundedSet<> and UnBoundedSet<>
- Both provide similar functions(Union, Intersection, etc), inherit same abstract class
- ISet<> works as a interface class



Implementation of Framework for LLVM

We use C++ templates and LLVM as backbone in our framework

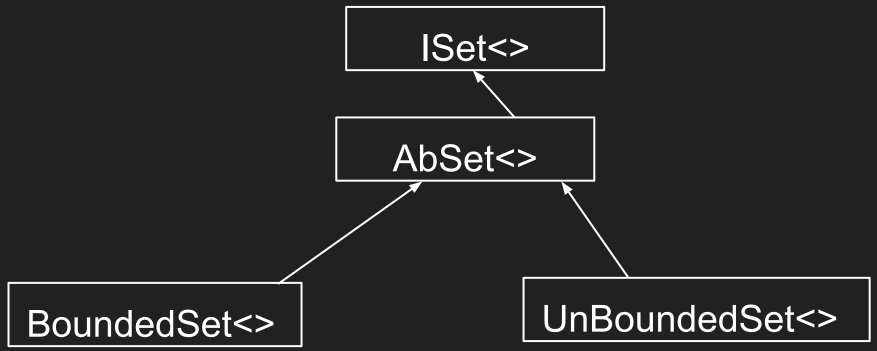
We define mostly all classes as templates

- user can defined own properties

Two sections : Storage section and Analysis section

Storage section :

- Two available classes to use BoundedSet<> and UnBoundedSet<>
- Both provide similar functions(Union, Intersection, etc), inherit same abstract class
- ISet<> works as a interface class
- AbSet<> implement functions to handle two different objects



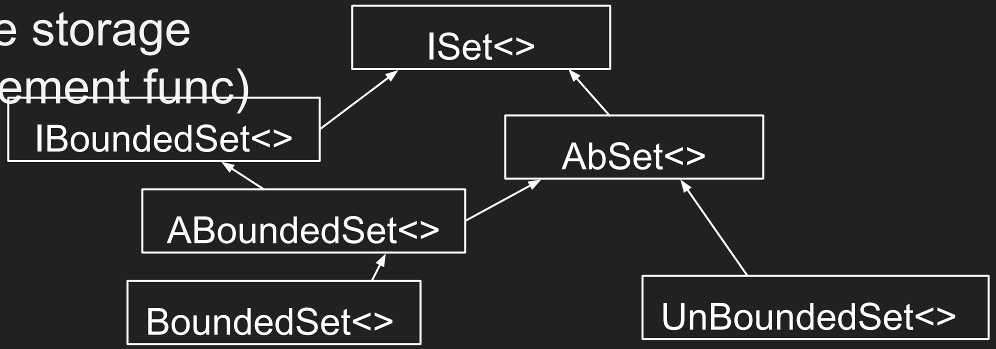
Implementation of Framework for LLVM

We use C++ templates and LLVM as backbone in our framework
We define mostly all classes as templates

- user can defined own properties

Two sections : Storage section and Analysis section
Storage section :

- Two available classes to use BoundedSet<> and UnBoundedSet<>
- Both provide similar functions(Union, Intersection, etc), inherit same abstract class
- ISet<> works as a interface class
- AbSet<> implement functions to handle two different objects
- BoundedSet<> gives a bounded lattice storage
    - One more level(topset and complement func)
    - Uses a bidirectional map and Bitvector



Implementation of Framework for LLVM

We use C++ templates and LLVM as backbone in our framework

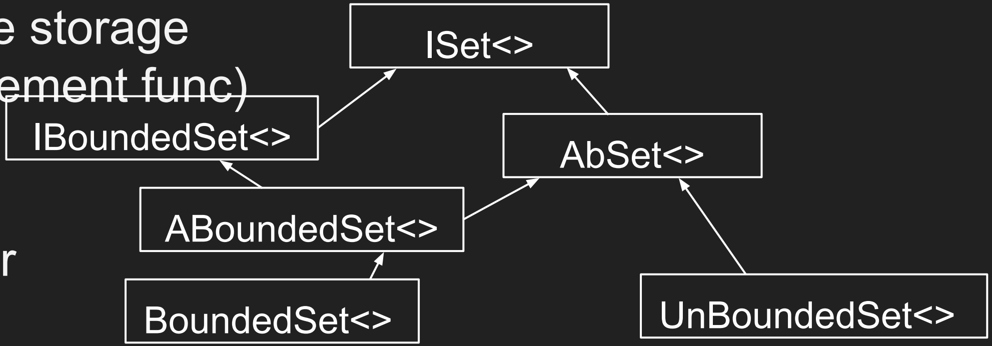
We define mostly all classes as templates

- user can defined own properties

Two sections : Storage section and Analysis section

Storage section :

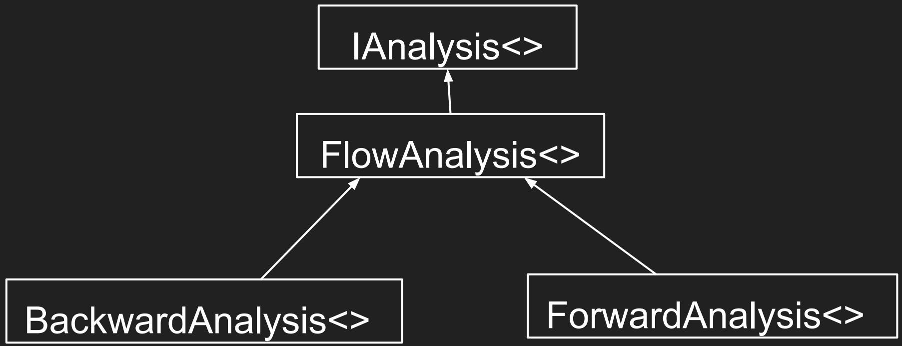
- Two available classes to use BoundedSet<> and UnBoundedSet<>
- Both provide similar functions(Union, Intersection, etc), inherit same abstract class
- ISet<> works as a interface class
- AbSet<> implement functions to handle two different objects
- BoundedSet<> gives a bounded lattice storage
  - One more level(topset and complement func)
  - Uses a bidirectional map and Bitvector
- UnBoundedSet<> uses a set container



Implementation of Framework for LLVM(cont..)

Analysis section :

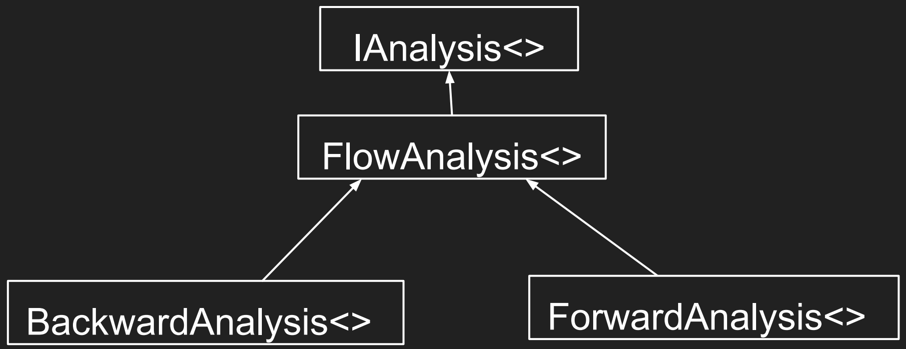
- IAnalysis<> provides functions that user defines
  - NewInitialFlowSet, EntryInitialFlowSet, Merge, Copy, FlowThrough
- User inherits either of BackwardAnalysis<> or ForwardAnalysis<>



Implementation of Framework for LLVM(cont..)

Analysis section :

- IAnalysis<> provides functions that user defines
  - NewInitialFlowSet, EntryInitialFlowSet, Merge, Copy, FlowThrough
- User inherits either of BackwardAnalysis<> or ForwardAnalysis<>
- FlowAnalysis<> does the work
  - Creates a FunctionCFG (LLVM does not provide a CFG object)
  - Creates a NodeData<> object for each CFG node
  - NodeData<> class stores results and pointers to NodeData<> objects of the predecessor nodes and successor nodes
  - Adds the NodeData<> objects in a vector in post order or reverse post order
  - Uses this vector in iterative algorithm



Usage details

We give some macros to use framework easily.

Define analysis in either

FORWARDANALYSIS(...){...} or BACKWARDANALYSIS(...){...}

- Pass three args 1. name for analysis, 2. Storage class name, 3. user-defined property class
- User defines a class for property and a function definition in EXTRACT(){...}
    - Outside the analysis
- Inside analysis
    - function setup with SETUP_FUNCTION()
    - Call DoAnalysis()
    - Use BS_INITIALVALUE(), BS_ENTRYVALUE(), BS_MERGE(), BS_COPY(), and BS_FLOWTH() to give various framework definition
        - For UnBoundedSet use UBS_*
    - In *_FLOWTH() give definition for Gen and Kill also and call AddGenKill(...) on them and use GetGen() and GetKill().

Usage details

```
// define a property class here
EXTRACT(property_class_name){ // definition for extracting properties
  ...
}
BACKWARDANALYSIS(analysis_name,storage_class_name,property_class_name){
  static char ID; // used by LLVM internals
  string funcName; // store function name for analysis
  storage_class_name<property_class_name> *domain; // store properties

  analysis_name(string f): funcName(f),FunctionPass(ID){} // constructor

  RUN(){ // run the analysis
    SETUP_FUNCTION(); // function setup call
    domain = new storage_class_name<property_class_name>(&F); // extracting properties and assign
    DoAnalysis(); // start the analysis
    return false;
  }
  BS_INITIALVALUE(){ return domain->EmptySet();} // setting initial value to emptyset
  BS_ENTRYVALUE(){ return domain->EmptySet();} // setting entry value to emptyset
  BS_MERGE(property_class_name){ in1->Union(in2,out);} // setting meet operator to union
  BS_COPY(property_class_name){ in1->Copy(in2);} // providing copy definition
  BS_FLOWTH(property_class_name){ // definition for flow function
    // define kill
    // define gen

    storage_class_name<property_class_name>* tmp = domain->EmptySet();
    in->Difference(kill,tmp);
    tmp->Union(gen,out);
  }
};
```

Possible future work

- Add graphical interface
- Make even easier to use but still providing a general framework
    - User can define own properties
- Add Interprocedural Data flow analysis

Possible future work

- Add graphical interface
- Make even easier to use but still providing a general framework
  - User can define own properties
- Add Interprocedural Data flow analysis

Thanks for listening...