A brief introduction to our PDG construction on LLVM

Input and Output

- Input: an LLVM module file (e.g. test.bc)
- Output: a .dot file which can be transformed into a .png file for visualization.
- For some large programs, it's difficult to generate a visualizable .png file but the PDG does exist in our in-memory data structures.

  It's good to use a simple example to show our basic workflow first...

Example: original C

Assume we have a C program in file

test.c as follows...

```
#include <stdio.h>

int CalcSum (int s, int i){
  return s + i;
}

int main(){
  int sum = 0 ;
  int i = 0;
  while(i < 10){
    sum = CalcSum(sum , i) ;
    i = i + 1;
  }
  return 0;
}
```

test.c

Example: C → IR

We first compile it into an IR file – test.bc

```
; Function Attrs: nounwind uwtable
define i32 @CalcSum(i32 %s, i32 %i) #0
{
entry:
  %s.addr = alloca i32, align 4
  %i.addr = alloca i32, align 4
  store i32 %s, i32* %s.addr, align 4
  store i32 %i, i32* %i.addr, align 4
  %0 = load i32* %s.addr, align 4
  %1 = load i32* %i.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %sum = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 0, i32* %sum, align 4
  store i32 0, i32* %i, align 4
  br label %while.cond
```

```
while.cond:   ; preds = %while.body, %entry
  %0 = load i32* %i, align 4
  %cmp = icmp slt i32 %0, 10
  br i1 %cmp, label %while.body, label
%while.end

while.body:            ; preds = %while.cond
  %1 = load i32* %sum, align 4
  %2 = load i32* %i, align 4
  %call = call i32 @CalcSum(i32 %1, i32 %2)
  store i32 %call, i32* %sum, align 4
  %3 = load i32* %i, align 4
  %add = add nsw i32 %3, 1
  store i32 %add, i32* %i, align 4
  br label %while.cond

while.end:             ; preds = %while.cond
  ret i32 0
}
```

test.bc

Example: IR processing

- Our program will process the input LLVM module (test.bc) function by function, instruction by instruction, and create a “node” for each instruction, and insert this node into our PDG.
- we also create some dummy nodes for connecting the callers and callees, such as the “Entry” nodes for entering a callee function, and the nodes for parameter passing(parameter trees)
- When the module processing done, the PDG is already constructed in memory.

Example: .dot

We can also dump the in-memory PDG

into a .dot file, which looks like this:

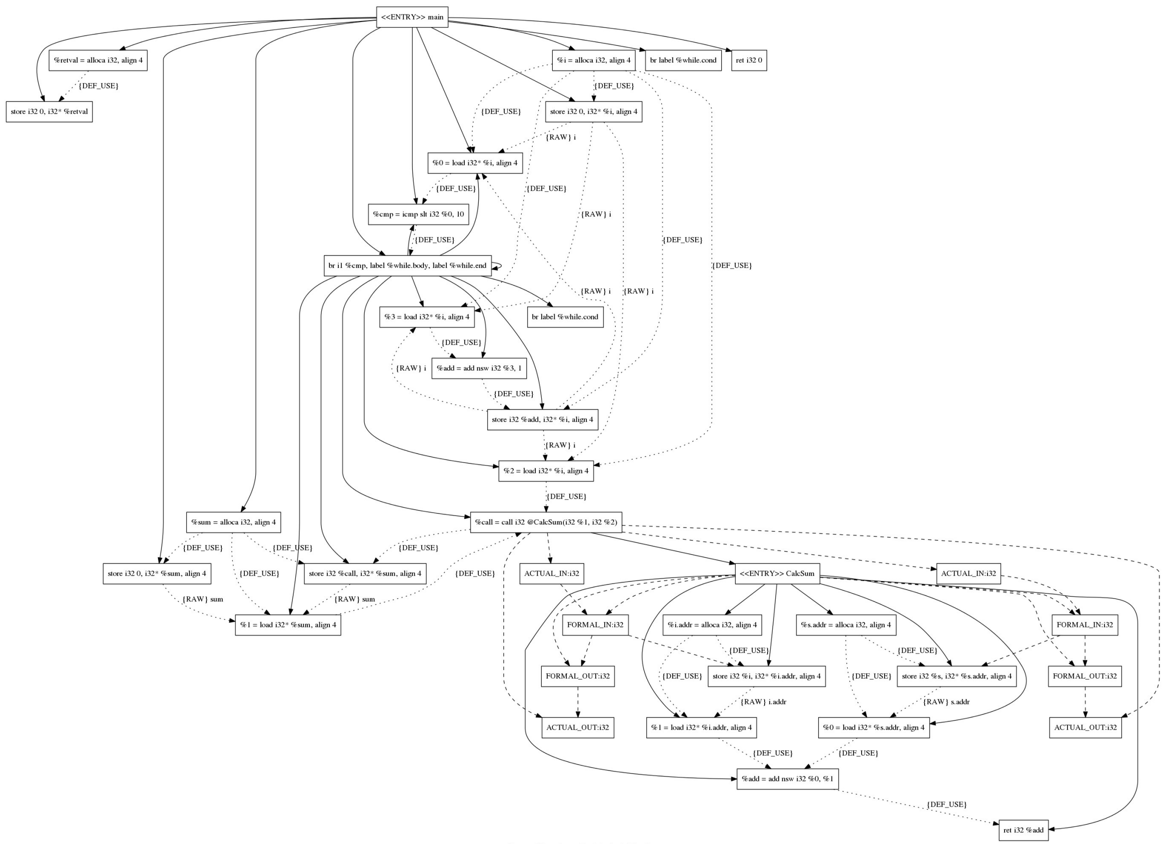
```
digraph "Program Dependency Graph for 'main' function" {
     label="Program Dependency Graph for 'main' function";

     Node0x2e3b590 [shape=record,label="{  %s.addr = alloca i32, align 4}"];
     Node0x2e3b590 -> Node0x2e3b600[style=dotted,label = "{DEF_USE}" ];
     Node0x2e3b590 -> Node0x2e3b670[style=dotted,label = "{DEF_USE}" ];
     Node0x2e3b600 [shape=record,label="{  store i32 %s, i32* %s.addr, align 4}"];
     Node0x2e3b600 -> Node0x2e3b670[style=dotted,label = "{RAW} s.addr"];
     Node0x2e3b670 [shape=record,label="{  %0 = load i32* %s.addr, align 4}"];
     Node0x2e3b670 -> Node0x2e3b890[style=dotted,label = "{DEF_USE}" ];

     …
     …

     Node0x2e40b00 -> Node0x2e3f580;
     Node0x2e40b00 -> Node0x2e40bf0;
     Node0x2e40b70 [shape=record,label="{  br label %while.cond}"];
     Node0x2e40bf0 [shape=record,label="{  ret i32 0}"];
}
```

pdgraph.dot

Example: visualizable PDG

Then use dot command to transform the .dot file into

a .png file as follows for visualization



Program Dependency Graph for 'main' function

pdg.png

Control Dependency: Algorithm

From Ferrante[1] et al. the algorithm for obtaining control dependency information is as follows:

- **Step 1**: Let edge set S consists of all edges(A,B) in the CFG (A->B) such that B does not post-dominate A.
- **Step 2**: Examine each pair (A,B) in S. Let L denote the least common ancestor of A and B in the post-dominator tree. L can be two things (see the paper for proofs)
  - Case 1. L = parent of A. All nodes in the post-dominator tree on the path from L to B, including B but not L, should be made control dependent on A.
  - Case 2. L = A. All nodes in the post-dominator tree on the path from A to B, including A and B, should be made control dependent on A. (This case captures loop dependence.)

[1] https://www.cs.utexas.edu/~pingali/CS395T/2009fa/papers/ferrante87.pdf

Control Dependency: Implementation

- Constructing CDG is easy since LLVM already has a built-in pass to compute the PostDominatorTree
- The Dependences established by using PostDominatorTree pass are among basicblocks, but it's quite easy to convert the CDG from the basicblock-level to instruction-level (see function `addDependency()` in ControlDependencies.cpp).
- In our code, the ultimate control dependence graph for each llvm function is stored in pointer `ControlDepGraph *CDG`

Data Dependency: Intra-procedural

- There are two kinds of intra-procedural data dependences in our code.
  - Definition-use dependence – this represents the data dependency between temporaries(**register** values).
  - Flow dependence – this represents the read-after-write (RAW) **memory** dependency between StoreInst and LoadInst.
- One thing needs to be stated clearly is **we only compute the flow dependence (true data dependence)**. The other two kinds of data dependences(anti-, output-) are not in our consideration.
- It's easy to detect a DD between register values because LLVM uses the SSA form. So, in order to detect a DD, it suffices to find all operands in an instruction and add a dependency between it and the instruction that defines the operand.

Data Dependency: Intra-procedural

- For **flow dependencies**, since they only exist between Store and Load instructions, our algorithm for computing all of intra-procedural flow dependencies is as follows:
  - Step 1: Check each instruction in F, if it is a LoadInst, do Step 2; otherwise, continue to check the next.
  - Step 2: For LoadInst LI, check each StoreInst SI in F, if LI and SI **must or may** access the same memory location, SI->LI is a flow dependence. The AliasAnalysis pass is required here, now we are using DSA analysis to do this, see https://github.com/jtcriswell/llvm-dsa
- Both def-use and flow dependences will be stored in one DDG in the end (DataDepGraph *DDG).

Data Dependency: Inter-procedural

- The inter-procedural data flow fully depends on the parameter passing. So, the hardest part is how to represent function parameters. In our framework, each parameter is represented by a set of **parameter trees,** which includes 4 tree types: actual-in tree, actual-out tree, formal-in tree and formal-out tree.
- The actual-in and actual-out trees represent the flow of the actual parameters (arguments) to call temporaries and from return temporaries respectively. The formal-in and formal-out trees are the callee analogs of actual-in/out trees.
- A parameter tree for atomic data types (e.g. int, float, char...) will shrink to a single parameter node.

Data Dependency: parameter trees

- The inter-procedural data flow fully depends on the parameter passing. So, the hardest part is how to represent function parameters. In our framework, each parameter is represented by a set of **parameter trees,** which includes 4 tree types: actual-in tree, actual-out tree, formal-in tree and formal-out tree.
- The actual-in and actual-out trees represent the flow of the actual parameters (arguments) to call temporaries and from return temporaries respectively. The formal-in and formal-out trees are the callee analogs of actual-in/out trees.
- We greatly benefit from the work by Susan Horwitz et al [1]. and Jürgen Graf [2], you can read the papers below for more details.

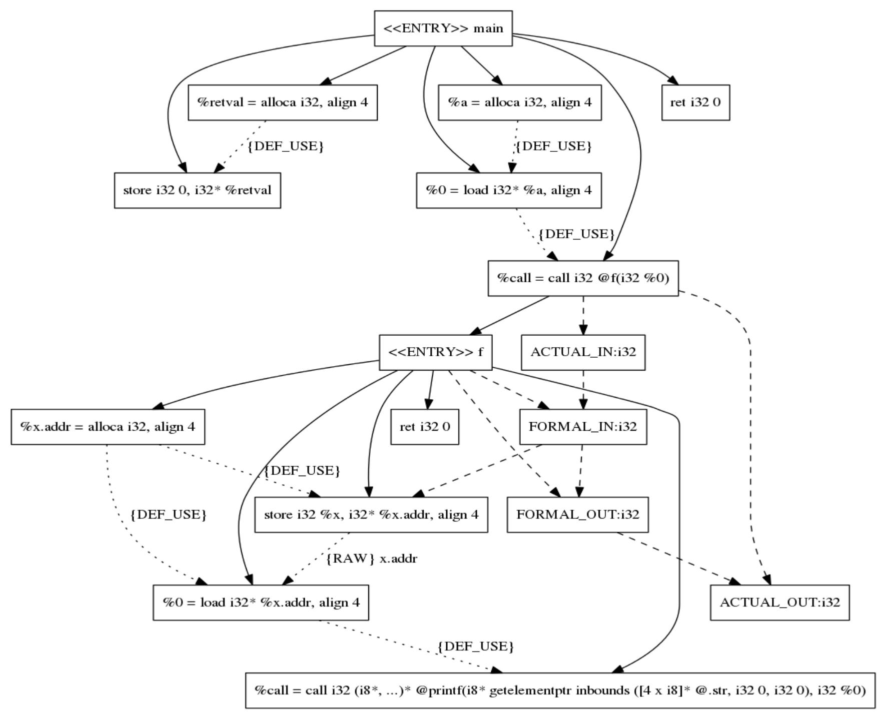
[1]https://courses.cs.washington.edu/courses/cse590md/01sp/w2l1.pdf

[2]http://pp.info.uni-karlsruhe.de/uploads/publikationen/graf10scam.pdf

Parameter Trees: example

Still, let's start with the simplest example to show how parameter trees look like. In the following example, the only parameter is an integer. For all atomic (built-in) C data types (e.g int, float, char...), a parameter tree will shrink to a single node. The dependence from Formal-in node to the StoreInst inside callee f represents the inter-procedural DD.

```
#include <stdio.h>
int f(int x){
  printf("%d\n",x);
  return 0;
}
int main(){
  int a;
  f(a);
  return 0;
}
```

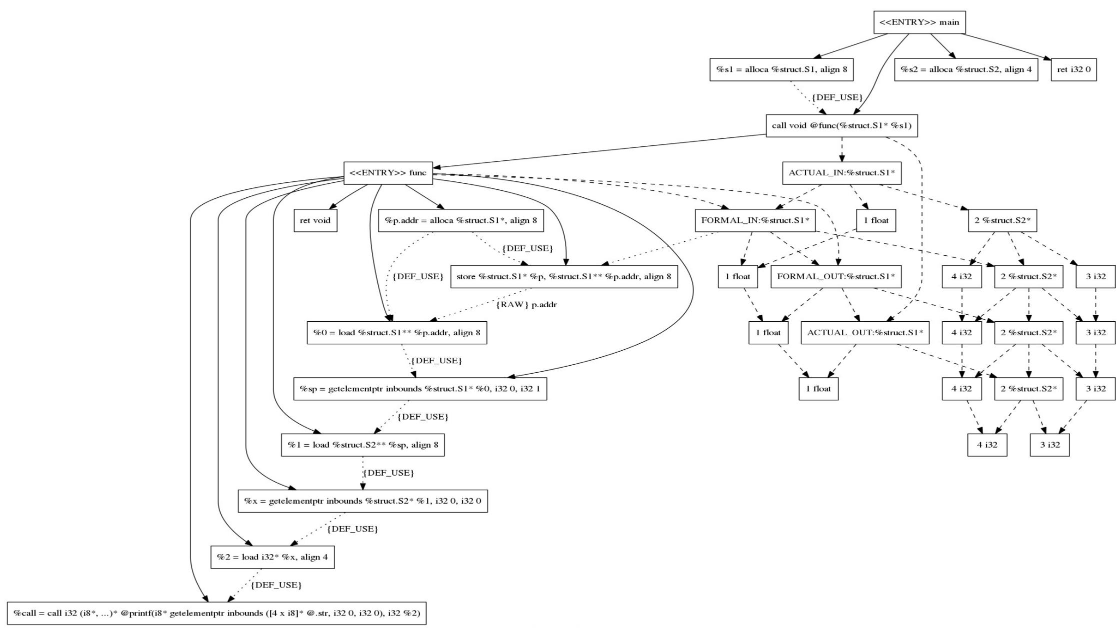


Program Dependency Graph for 'main' function

Parameter Trees: example

Now, look at an example with struct pointer parameter. As you can see, parameter &si is represented as four parameter trees. The root of each tree is [%struct *S1], each root has two leafs: [float] and [%Struct *S2](by static analysis we can just use types instead of value names to label), leaf [%Struct *S2] has two leafs [i32][i32]. The default data flow for each parameter field among parameter trees is [actual-in →formal-in → formal-out → actual-out] (see Susan Horwitz's paper for more detailed explanation).

```
#include<stdio.h>
typedef struct{
 int x;
 int y;
}S2;

typedef struct{
 float f;
 S2 *sp;
}S1;

void func(S1 *p){
 printf("%d\n",p->sp->x);
}
int main(){
 S1 s1;
 S2 s2;
 func(&s1);
}
```



Program Dependency Graph for 'main' function

The main modules in our code

- DependencyNode
  - this template defines all low level common things about dependence graph such as node representations and operations (e.g. add dependence from node A to node B, check whether node A depends on B or not).
- For example, the low level interaction between dependence nodes are implemented by the following data structures:

```
typedef std::pair<DependencyNode<NodeT>*, Type> DependencyLink;
typedef std::vector<DependencyLink> DependencyLinkList;
```

(see Dependences.h/.cpp)

The main modules in our code

- ControlDependenceGraph

  this function pass defines ControlDependenceGraph class and generates control dependence graph. Pass PostDominatorTree will be called inside.

- DataDependenceGraph

  this function pass defines DataDependenceGraph class and generates data dependence graph.

- ProgramDependenceGraph

  this pass calls ControlDependenceGraph and DataDependenceGraph to generate an ultimate program dependence graph.

The main modules in our code

- DepPrinter

  This module generates the .dot files, write the in-memorydata into .dot files based on the `llvm::DOTGraphTraits<Ty> Struct` template.

  (see DepPrinter.cpp)

- FlowDependenceAnalysis

  This pass is called by DataDependenceGraph module to process the Load/Store instructions. The **DSA** analysis will be called inside.

  (see FlowDependenceAnalysis.cpp)

- Now it's another good time to show how these modules work, for example, a data dependence graph(DDG) can be constructed in 3 steps...

An workflow example

- **Step 1**: Initialization, use DataDependenceGraph to analyze the input IR file and put all instructions into a static data structure called instMap.

  ```
  static std::map<const llvm::Instruction*,
  InstructionWrapper*> instMap;
  ```

- **Step 2:** For each instruction, check whether it is a load instruction or not. If yes, call `FlowDependenceAnalysis` to find the related flow dependences, otherwise reiterate the whole function to detect all def-use data dependences.

An workflow example

- **Step 3**: A `DataDependenceGraph*` pointer will be transferred into the DepPrinter module and generate a ddg.dot file(ddg graph description).

- The workflow for CDG generation is quite similar. If we want to generate PDG instead of CDG and DDG separately, just call ProgramDependeceGraph pass to generate PDG at the same time.