- Data flow analyses
  - Liveness Analysis
  - Constant Propagation
- Control flow analyses
  - Dead Code and Data Detection
  - Inteprocedural Dominators
- Some high-level transformations
- Instrumentation Support
- A graphical interface

Data Flow Analyses

- Same as compiler at RTL level, but
  - whole-program scope (on ICFG)
  - on register contents: memory is a black box
- Use calling conventions for greater precision
- 2 analysis/optimization phases
  - Analyses rely on calling conventions, transformations respect them
  - Analyses ignore calling conventions, transformations may break them
  - Transition period between them

Liveness Analysis

- A register holds a *live* value at a program point if this value can be used in some subsequently executed program point.
- Goal:
  - Remove useless instructions
  - Find free registers for use in optimizations

Liveness Analysis: the Algorithm

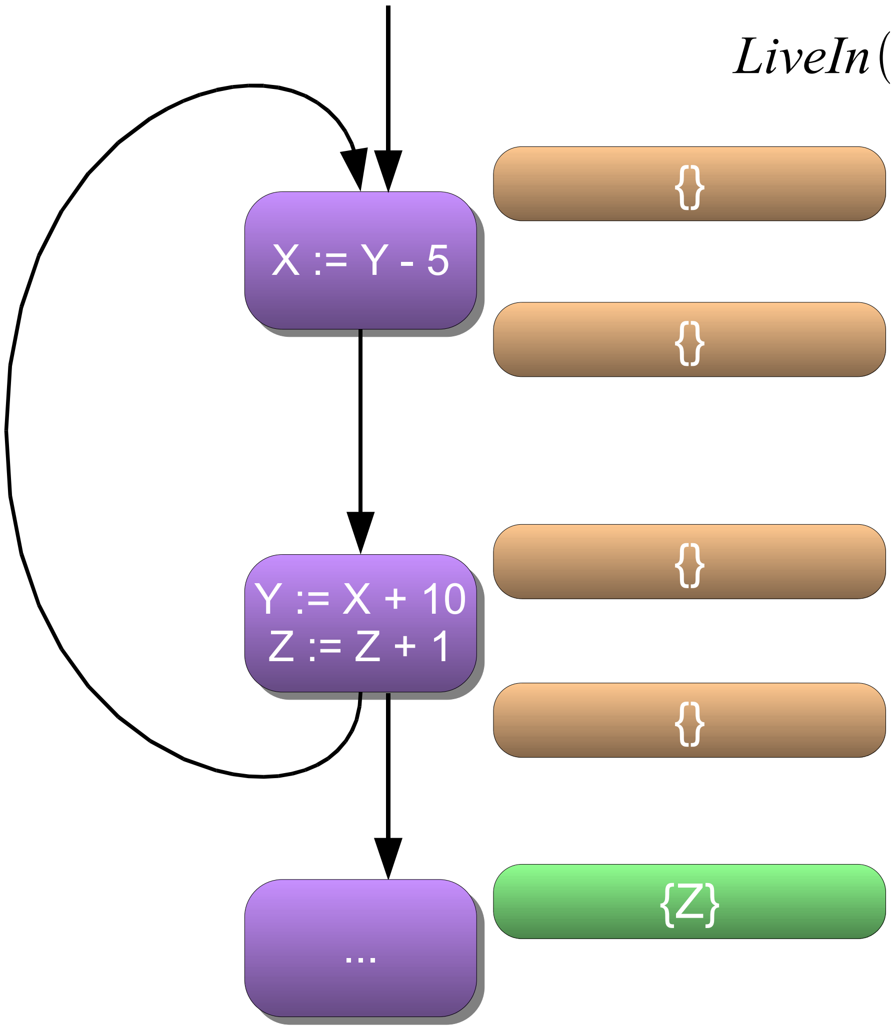
- Classical backward context-sensitive analysis
  - Muth's algorithm [*]
  - can rely on calling conventions to get more accurate results
- Forward analysis finds extra dead registers, but not useful for finding useless code
  - relies on calling conventions

(*) Muth, R., Debray, S. K., Watterson, S., and De Bosschere, K. 2001. *Alto: a link-time optimizer for the Compaq alpha*. Softw. Pract. Exper. 31, 1 (Jan. 2001), 67-101.

Backward Analysis

$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

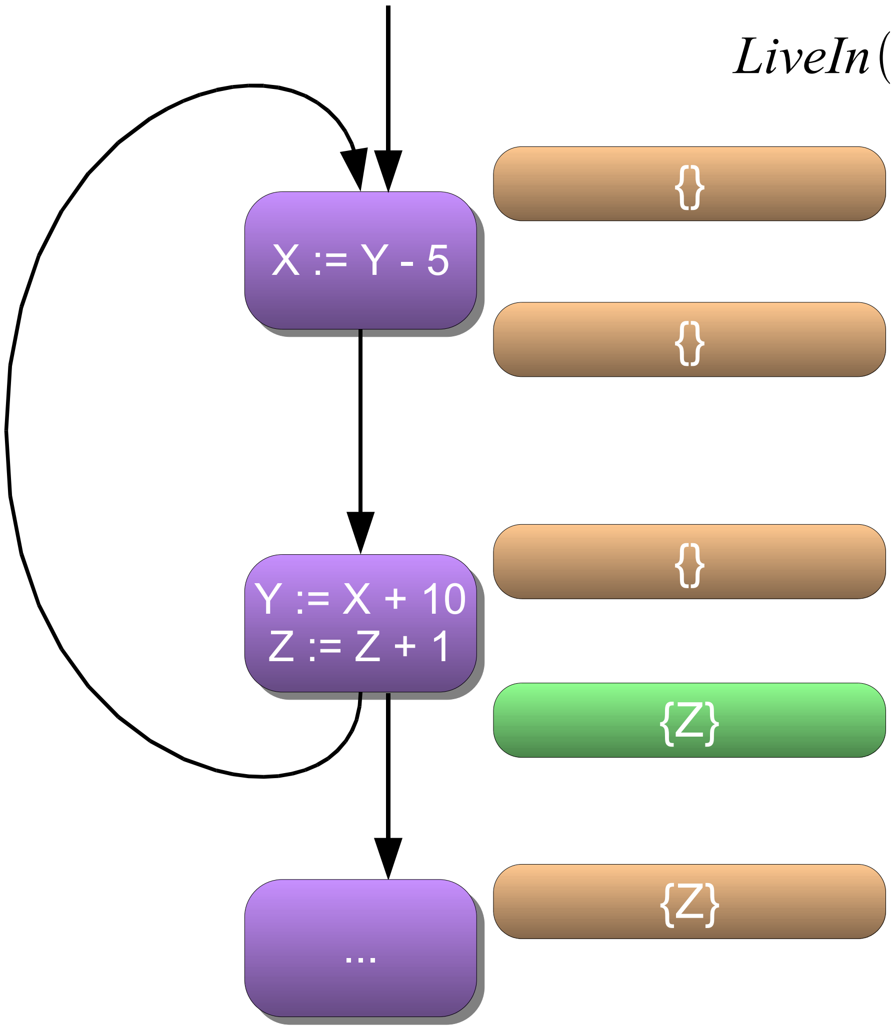
$$LiveIn(b) = (LiveOut(b) \setminus BlockDefines(b)) \cup BlockUses(b)$$



Backward Analysis

$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

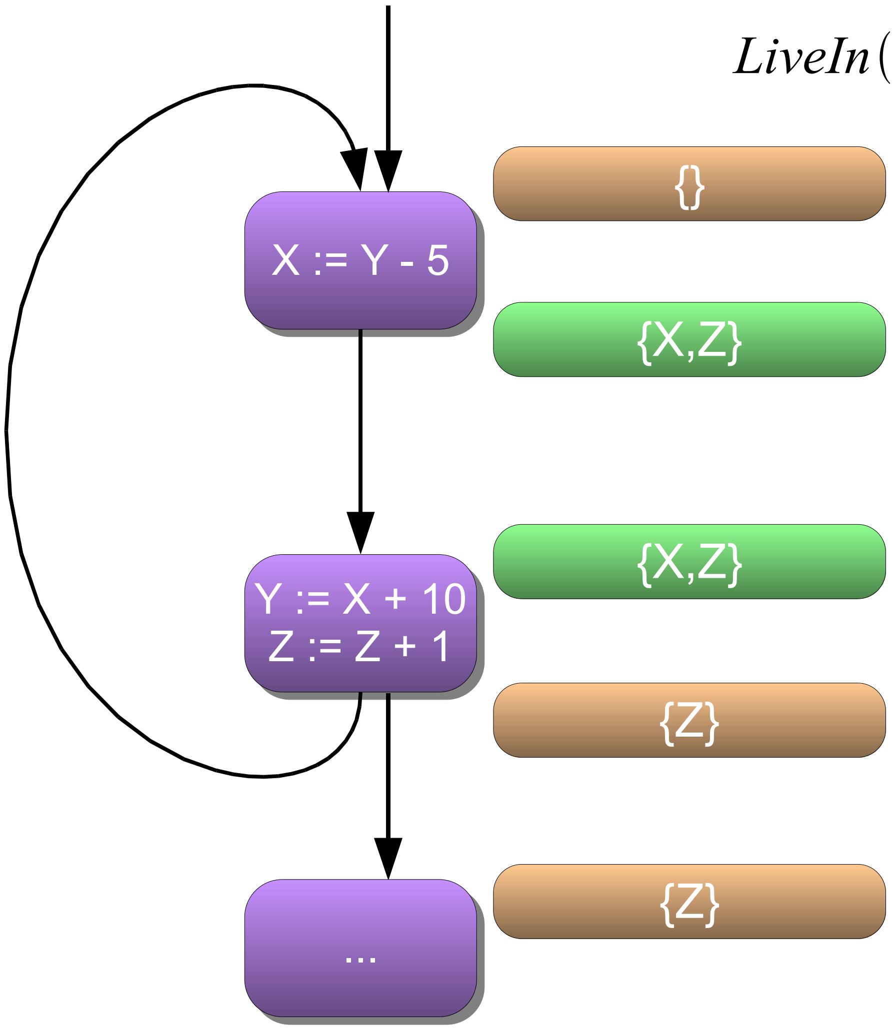
$$LiveIn(b) = (LiveOut(b) \setminus BlockDefines(b)) \cup BlockUses(b)$$



Backward Analysis

$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

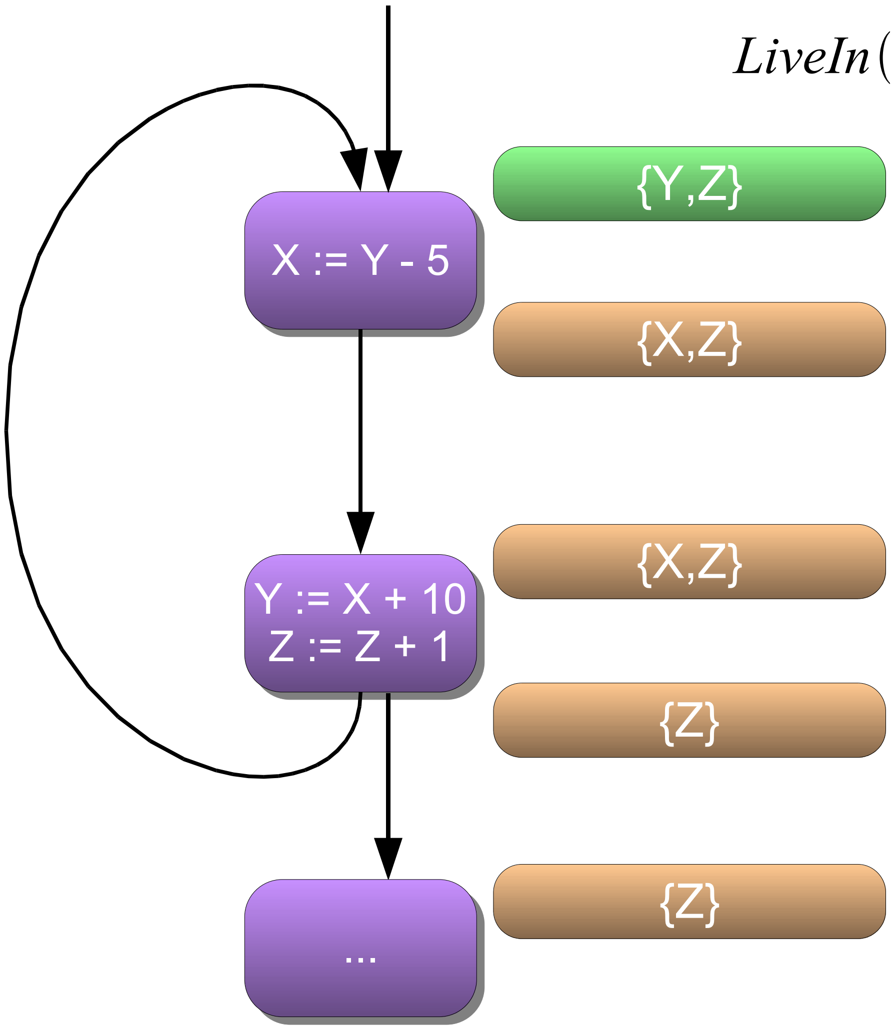
$$LiveIn(b) = (LiveOut(b) \setminus BlockDefines(b)) \cup BlockUses(b)$$



Backward Analysis

$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

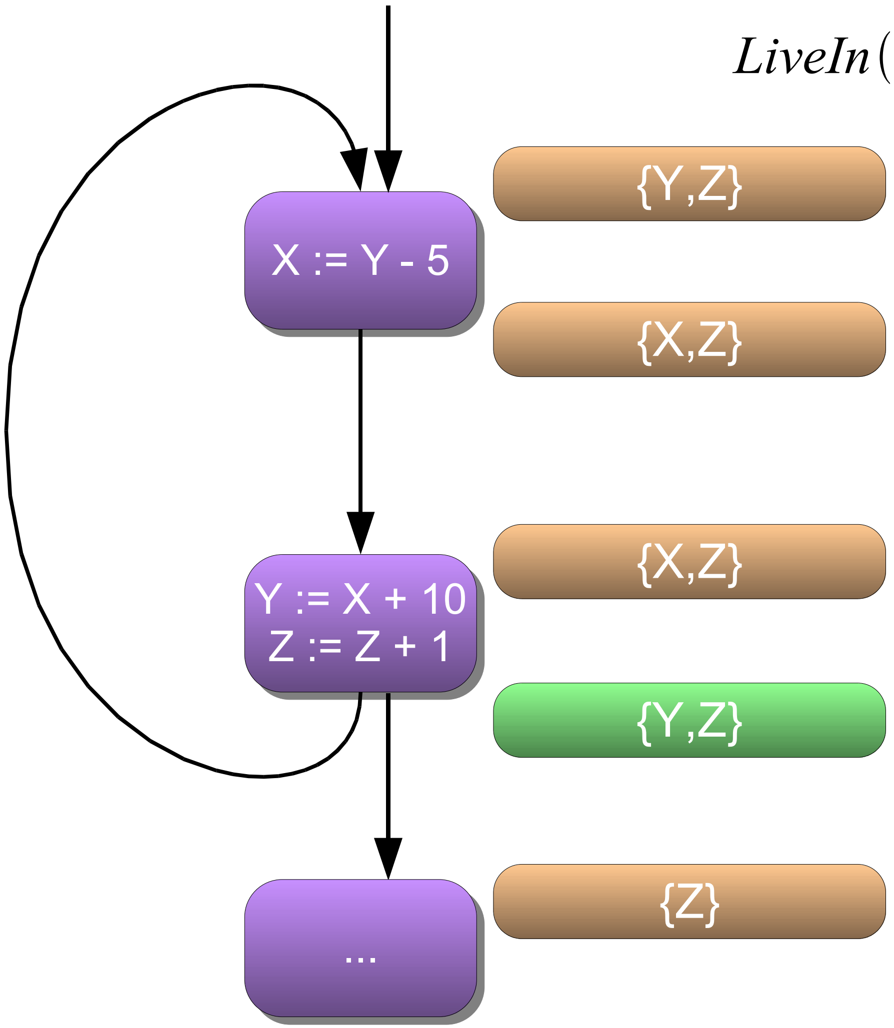
$$LiveIn(b) = (LiveOut(b) \setminus BlockDefines(b)) \cup BlockUses(b)$$



Backward Analysis

$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

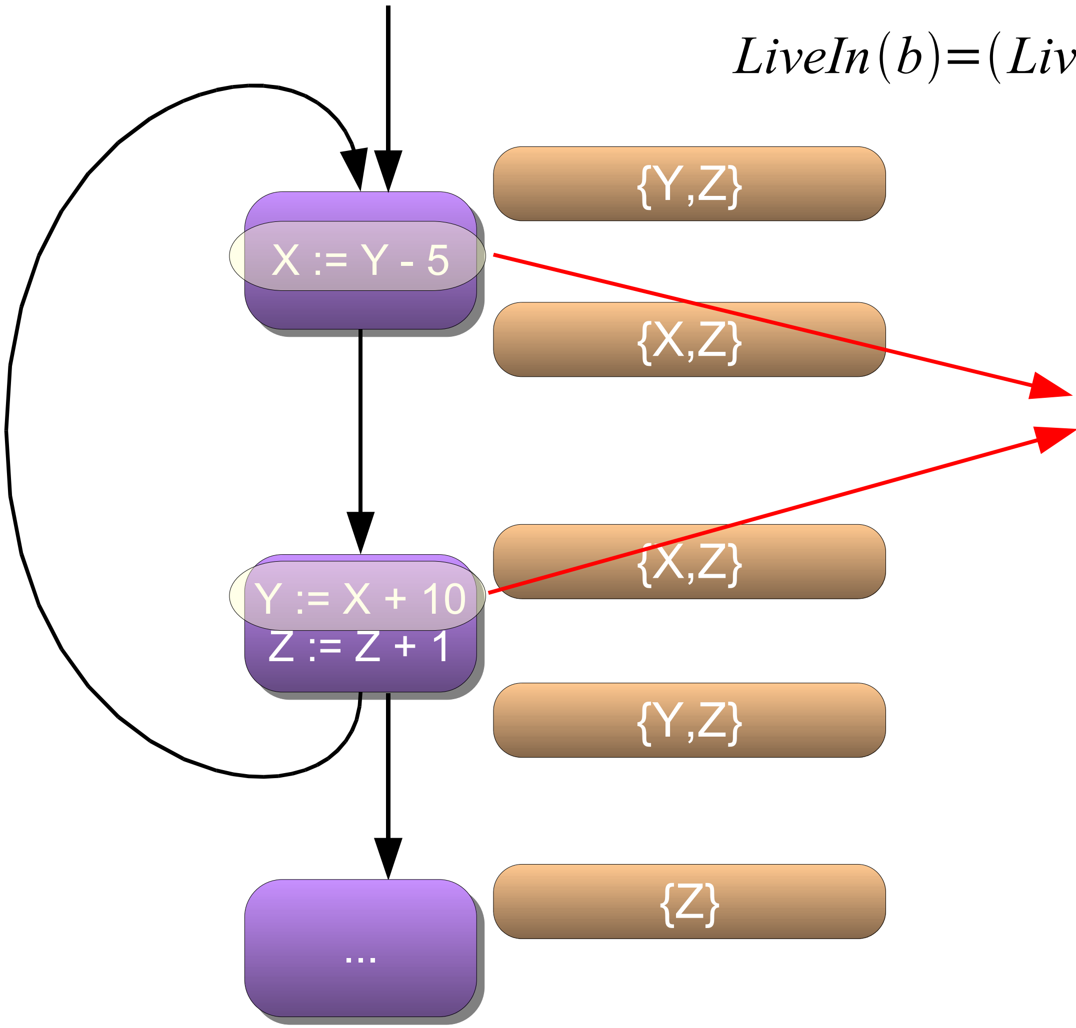
$$LiveIn(b) = (LiveOut(b) \setminus BlockDefines(b)) \cup BlockUses(b)$$



Backward Analysis

$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

$$LiveIn(b) = (LiveOut(b) \setminus BlockDefines(b)) \cup BlockUses(b)$$

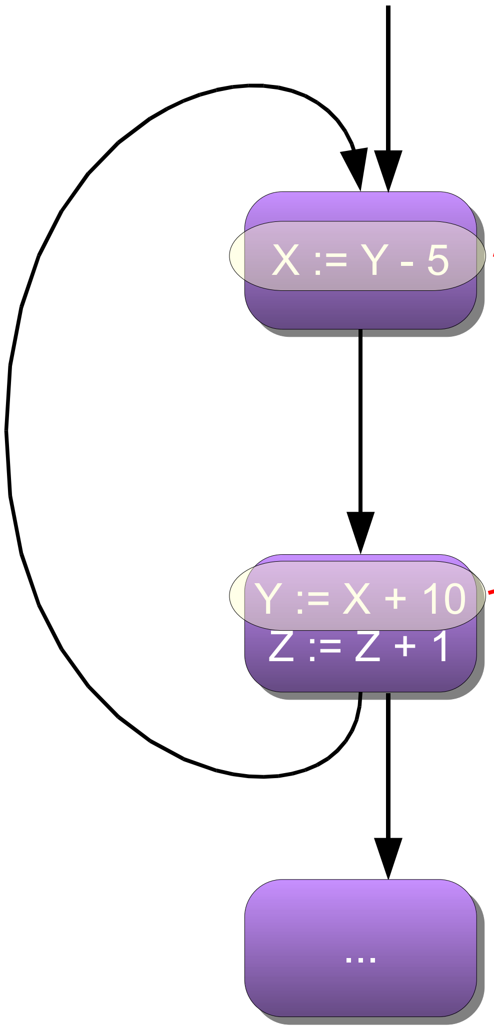


Ultimately, these instructions are useless as X and Y are dead outside of the loop. Liveness analysis cannot detect this!

Backward Analysis

$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

~~$LiveIn(b) = (LiveOut(b) \setminus BlockDefines(b)) \cup BlockUses(b)$~~

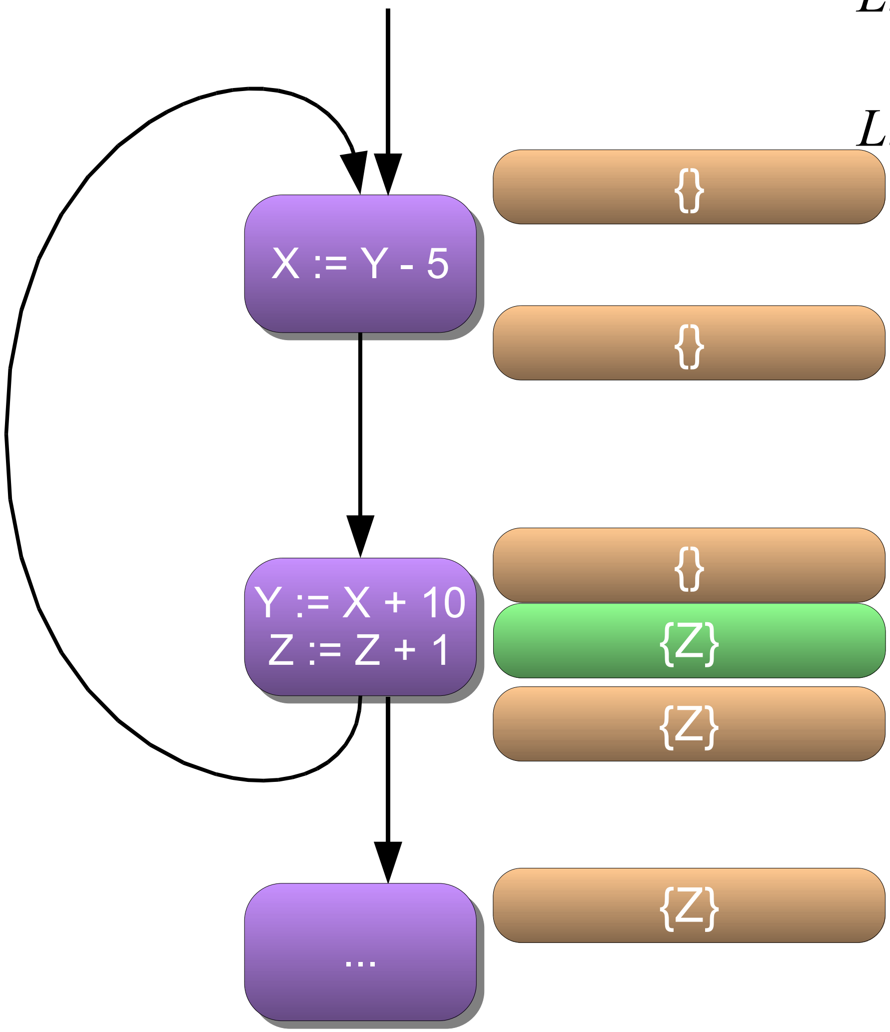


Ultimately, these instructions are useless as X and Y are dead outside of the loop. Liveness analysis cannot detect this!

To remove useless instructions, compute liveness information per instruction instead of per block.

$$LiveOut(i) \cap InsDefines(i) \neq \emptyset \Rightarrow$$
$$LiveIn(i) = (LiveOut(i) \setminus InsDefines(i)) \cup InsUses(i)$$
$$LiveOut(i) \cap InsDefines(i) = \emptyset \Rightarrow$$
$$LiveIn(i) = LiveOut(i)$$

Backward Analysis



$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

$$LiveOut(i) \cap InsDefines(i) \neq \emptyset \Rightarrow$$
$$LiveIn(i) = (LiveOut(i) \setminus InsDefines(i)) \cup InsUses(i)$$

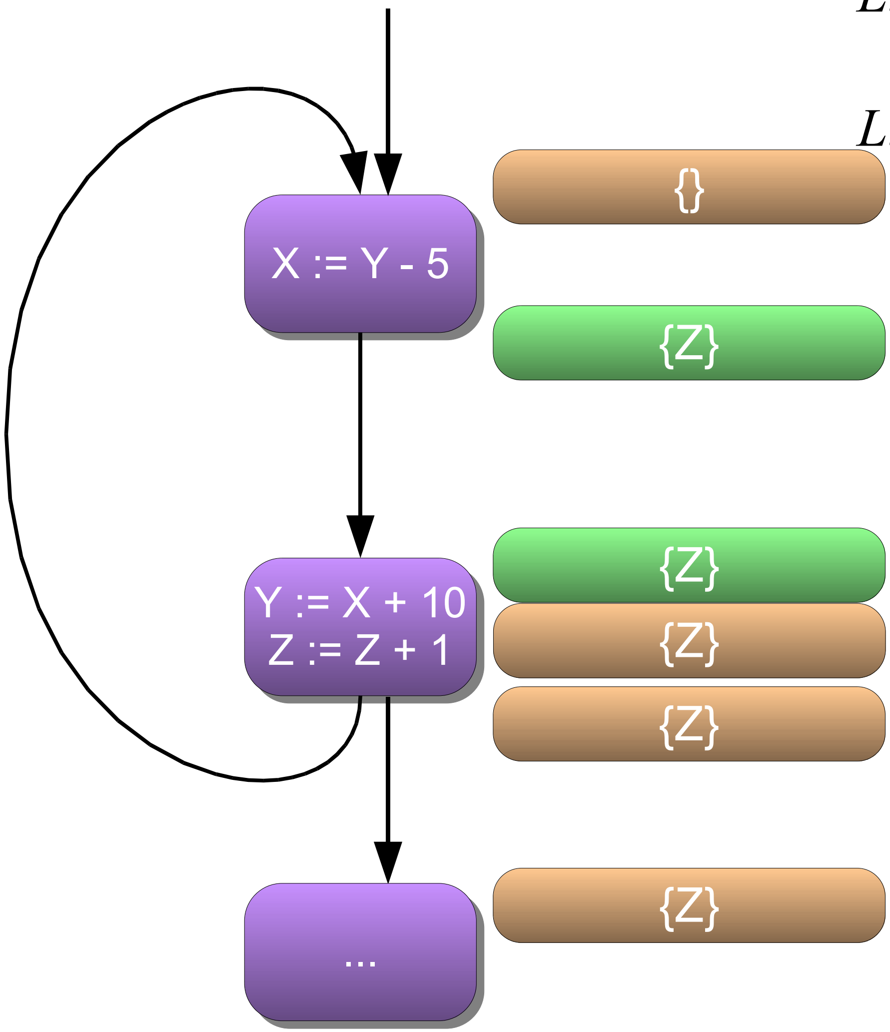
$$LiveOut(i) \cap InsDefines(i) = \emptyset \Rightarrow$$
$$LiveIn(i) = LiveOut(i)$$

Backward Analysis

$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

$$LiveOut(i) \cap InsDefines(i) \neq \emptyset \Rightarrow$$
$$LiveIn(i) = (LiveOut(i) \setminus InsDefines(i)) \cup InsUses(i)$$

$$LiveOut(i) \cap InsDefines(i) = \emptyset \Rightarrow$$
$$LiveIn(i) = LiveOut(i)$$

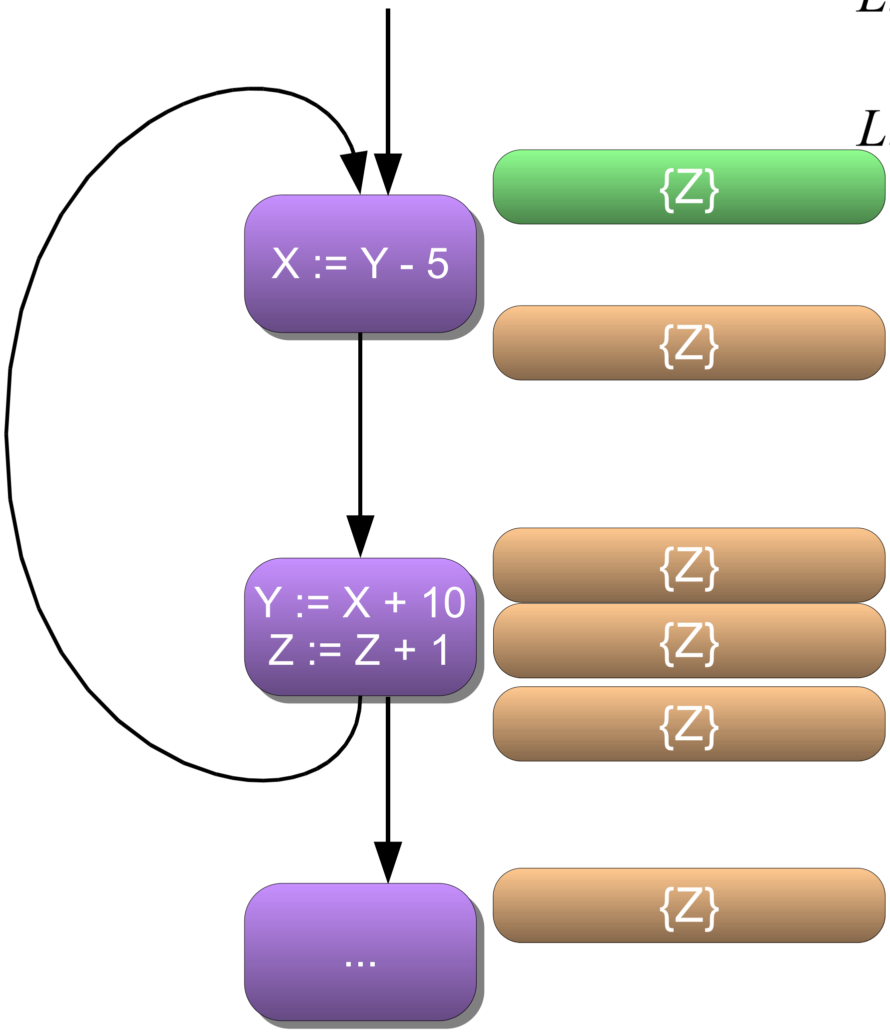


Backward Analysis

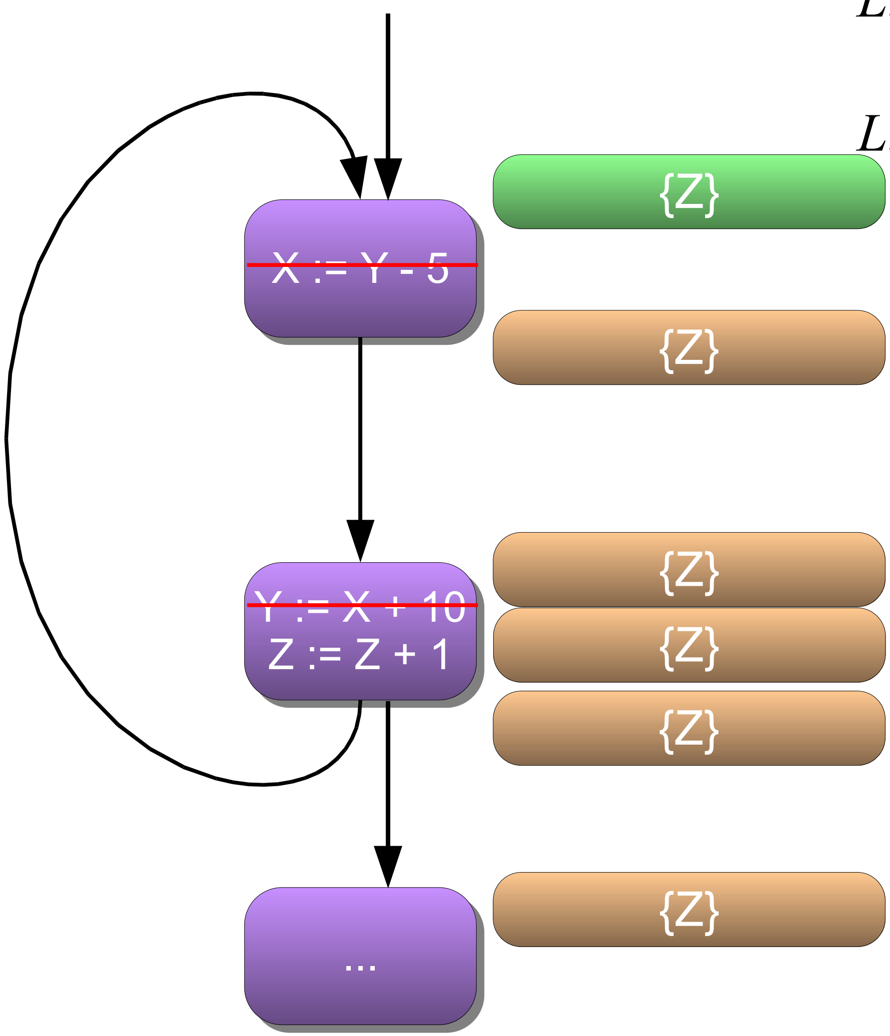
$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

$$LiveOut(i) \cap InsDefines(i) \neq \emptyset \Rightarrow$$
$$LiveIn(i) = (LiveOut(i) \setminus InsDefines(i)) \cup InsUses(i)$$

$$LiveOut(i) \cap InsDefines(i) = \emptyset \Rightarrow$$
$$LiveIn(i) = LiveOut(i)$$



Backward Analysis



$$LiveOut(b) = \bigcup_{b' \in Successors(b)} LiveIn(b')$$

$$LiveOut(i) \cap InsDefines(i) \neq \emptyset \Rightarrow$$
$$LiveIn(i) = (LiveOut(i) \setminus InsDefines(i)) \cup InsUses(i)$$

$$LiveOut(i) \cap InsDefines(i) = \emptyset \Rightarrow$$
$$LiveIn(i) = LiveOut(i)$$

This is only valid if you actually remove the useless instructions!

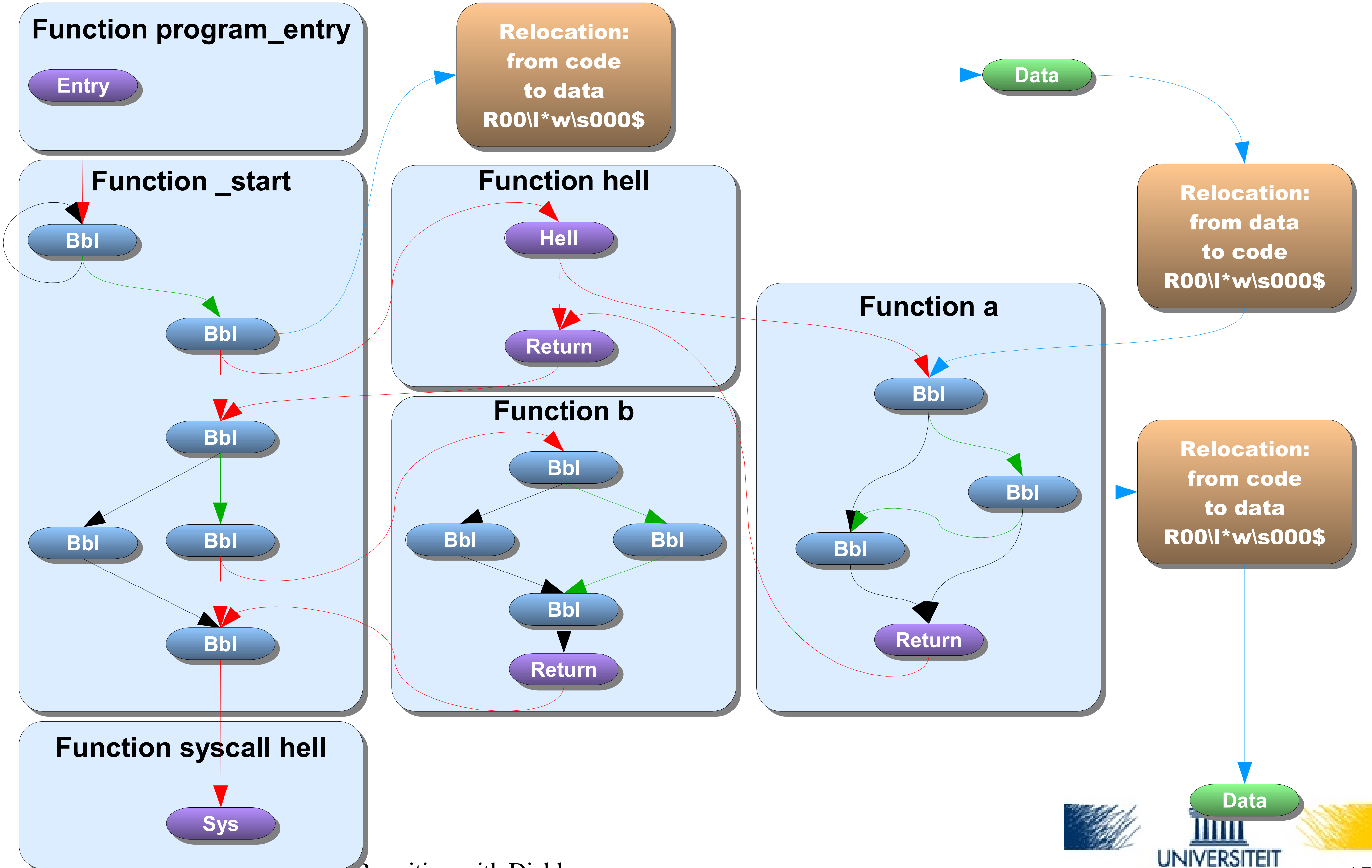
Diablo has both analyses: `ComLiveness()` and `ComputeUselessInstructions()`

Forward Liveness Analysis

- Calling conventions specify *caller-save* registers
- Compiler assumes these registers are not preserved over function calls
- These registers can be considered *dead* from the return site up to their first subsequent definition

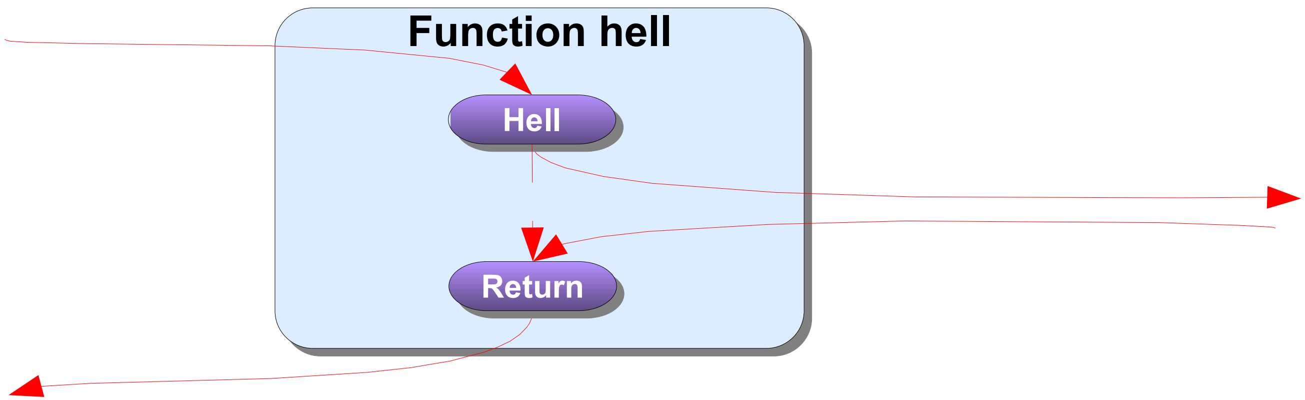
$\Rightarrow$ This analysis finds extra dead registers, but is not useful for finding useless instructions

Hell Node



Function program_entry
Entry
Relocation:
from code
to data
R00\l*w\s000$
Data
Function _start
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Function hell
Hell
Return
Relocation:
from data
to code
R00\l*w\s000$
Function a
Bbl
Bbl
Bbl
Return
Function b
Bbl
Bbl
Bbl
Bbl
Return
Relocation:
from code
to data
R00\l*w\s000$
Function syscall hell
Sys
Data

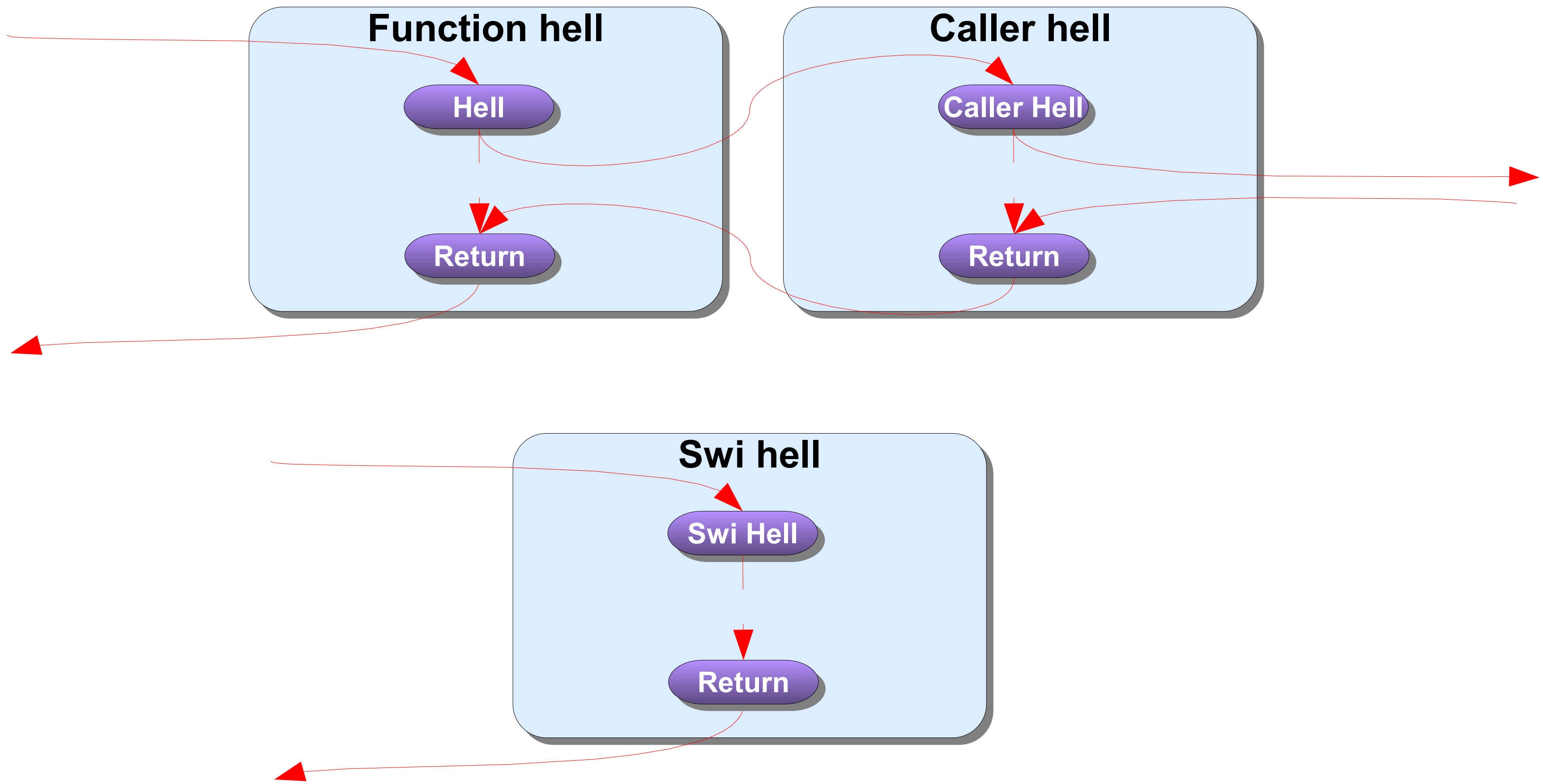
Hell Node



One function cannot be caller and callee.

One function and two nodes cannot model all calling conventions.

Multiple Hell Nodes



Implementation

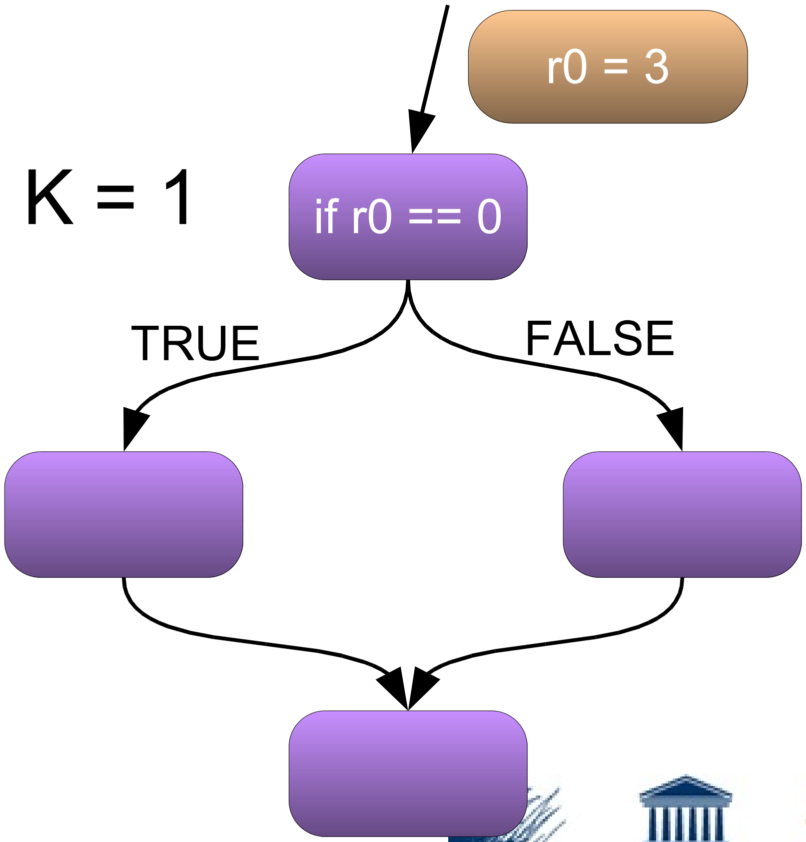
- Largely architecture-independent: operates on bit-vectors
- Some architecture-dependent knowledge provided by the architecture backend
  - ABI info (caller save/callee save registers, argument registers, return registers, ...) for hell nodes.
  - Used and defined registers for each instruction
  - Which instructions have side-effects

Using Liveness Information in Diablo

- `ComLiveness(cfg,CONTEXT_SENSITIVE)` performs both the forward and backward analyses
- `KillUselessInstructions(cfg)` computes liveness, remove useless instructions
- Analyses operate on bit vectors of type `t_regset`
- Liveness information at a specific program point:
  - `BblRegsLiveBefore(bbl)`
  - `BblRegsLiveAfter(bbl)`
  - `InsRegsLiveBefore(ins)`
  - `InsRegsLiveAfter(ins)`

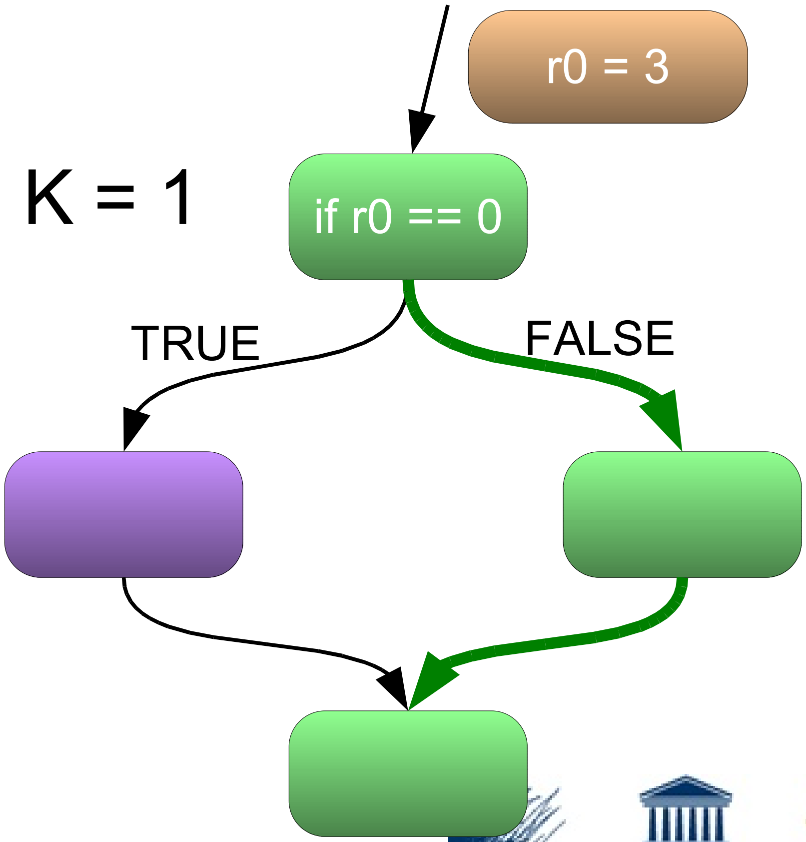
Constant Propagation

- Find registers that hold constant values at certain program points
- Diablo's implementation
  - K-depth context-sensitive, K = 1
  - conditional



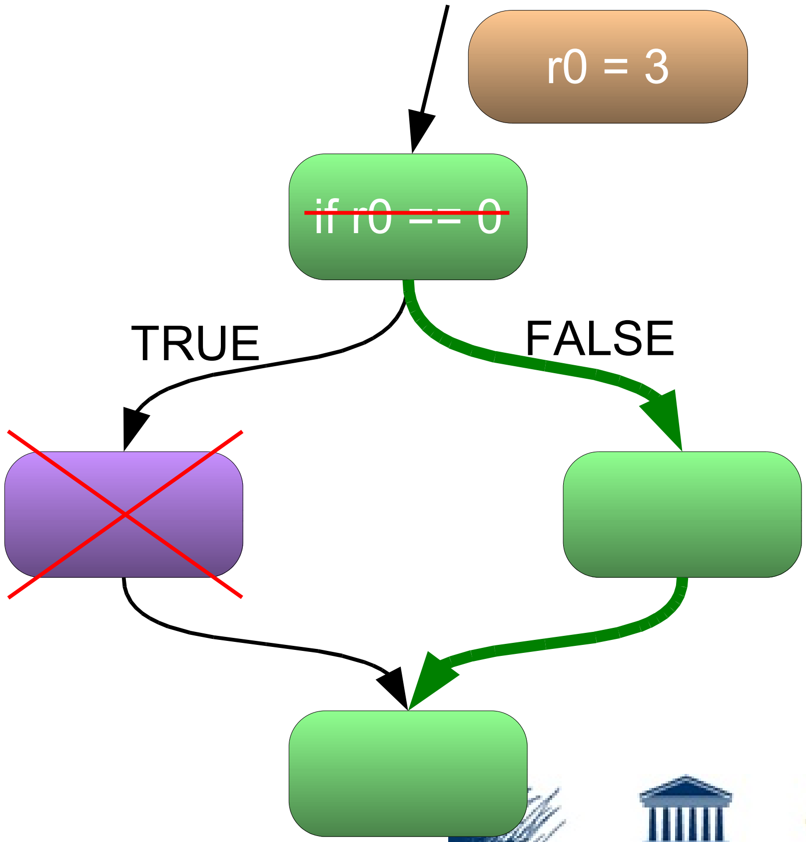
Constant Propagation

- Find registers that hold constant values at certain program points
- Diablo's implementation
  - K-depth context-sensitive, K = 1
  - conditional



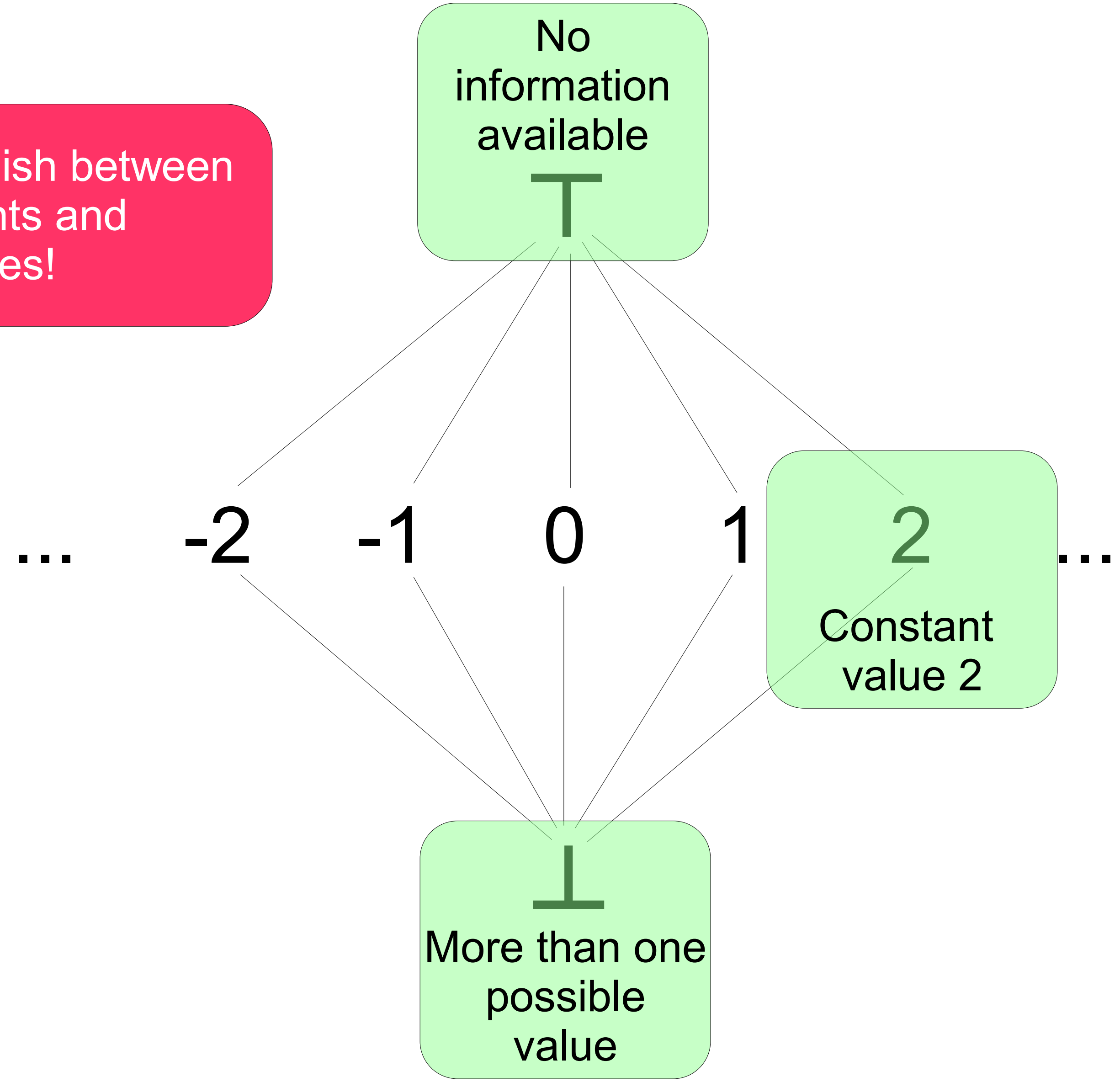
Constant Propagation

- Find registers that hold constant values at certain program points
- Diablo's implementation
  - K-context sensitive, K = 1
  - conditional
  - allows for unreachable code removal



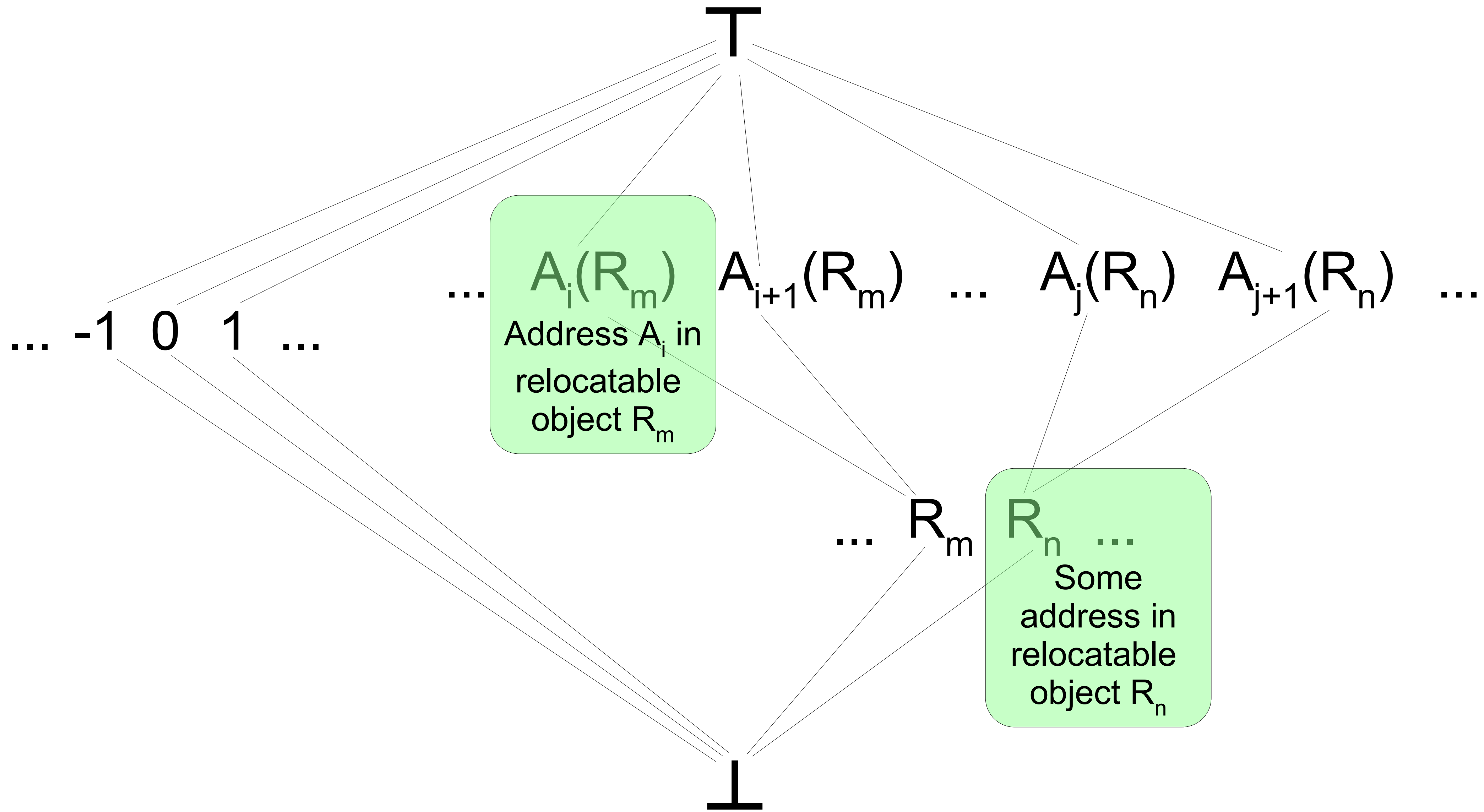
Constant Propagation: the lattice

Does not distinguish between real constants and addresses!



Traditional constant propagation lattice

Constant Propagation: the lattice



Diablo's constant propagation lattice

Implementation

- Actual data flow framework is architecture independent
- Architecture dependent
  - constant optimizations: i.e. encoding constants as immediate operands in instructions
  - instruction emulation
    no need to implement whole ISA

Using Constant Propagation in Diablo

- `ConstantPropagationInit(cfg)`
  Initialize data structures
- `ConstantPropagation(cfg, CONTEXT_SENSITIVE)`
  Interprocedural fixpoint computation, marks unreachable code
- `OptUseConstantInformation (cfg,CONTEXT_SENSITIVE)`
  Remove unreachable code, perform constant optimizations
- `ConstantPropagationFini(cfg)`
  Clean up data structures

Retrieving Constant Information

- After the call to `ConstantPropagation()`, constant information is only available on the interprocedural edges (limits memory consumption)
- Use `FunctionPropagateConstantsAfterIterativeSolution (fun, CONTEXT_SENSITIVE)` to propagate constant information down to basic block level for a given function
- Constant information at a specific program point:
  - `BblProcStateBefore()`
  - `BblProcStateAfter()`
  - `InsProcStateBefore()`
  - `InsProcStateAfter()`

Constant Information Representation

- Constant information is represented in the `t_procstate` type
- `ProcStateGetReg(procstate, register, &value)`
  Retrieve constant for register
- `ProcStateGetTag(procstate, register, &reloc)`
  Retrieve address for register
  `CP_TOP`: value was real constant, otherwise potentially an address
- `ProcStateGetCond(procstate, flag, &value)`
  Retrieve processor status flag state:
- These calls return either `CP_BOT`, `CP_TOP` or `CP_VALUE`, the actual value (if appropriate) is returned in the last parameter

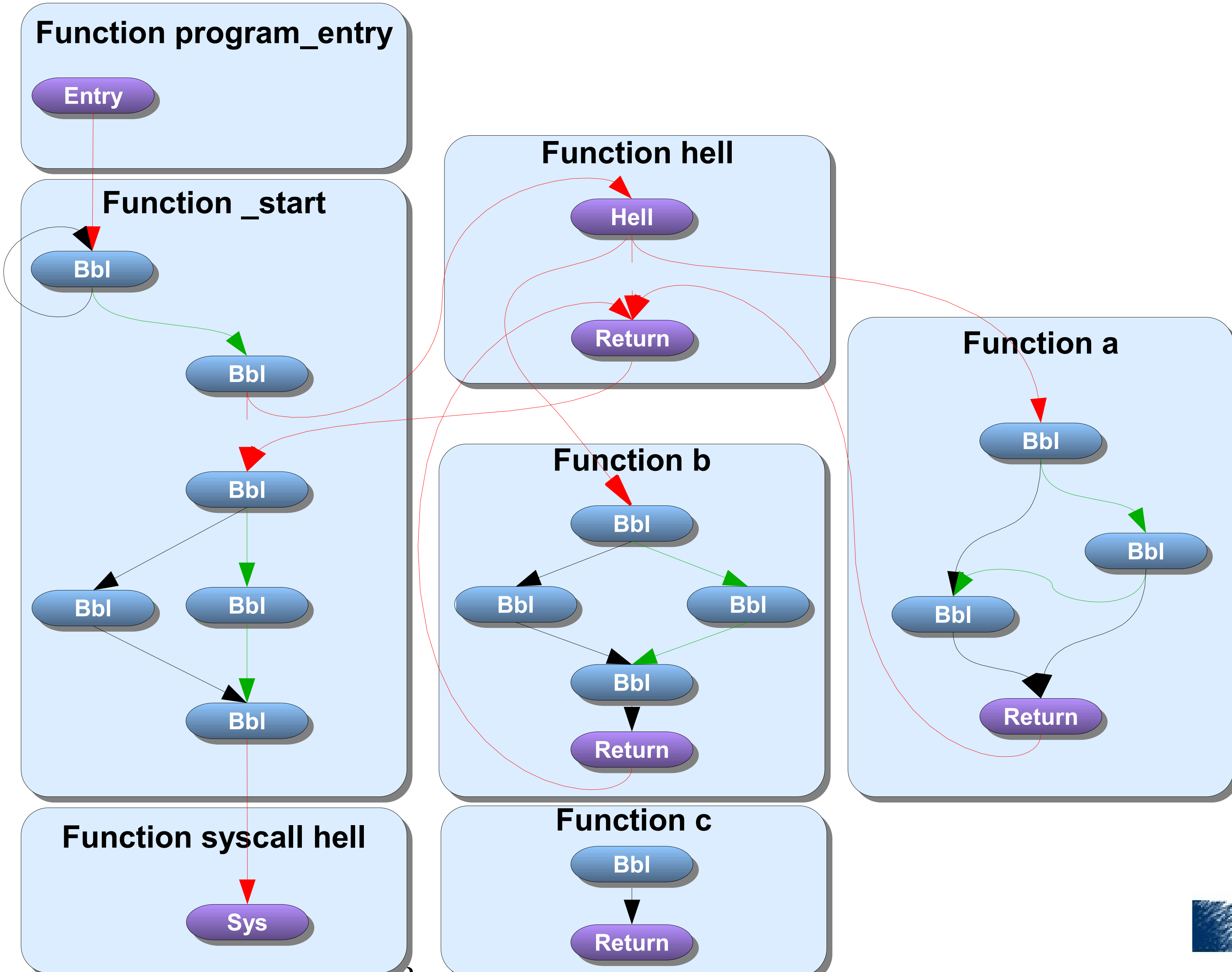
Other Data Flow Analyses

- Copy propagation
- Load-store forwarding
- Spill code removal

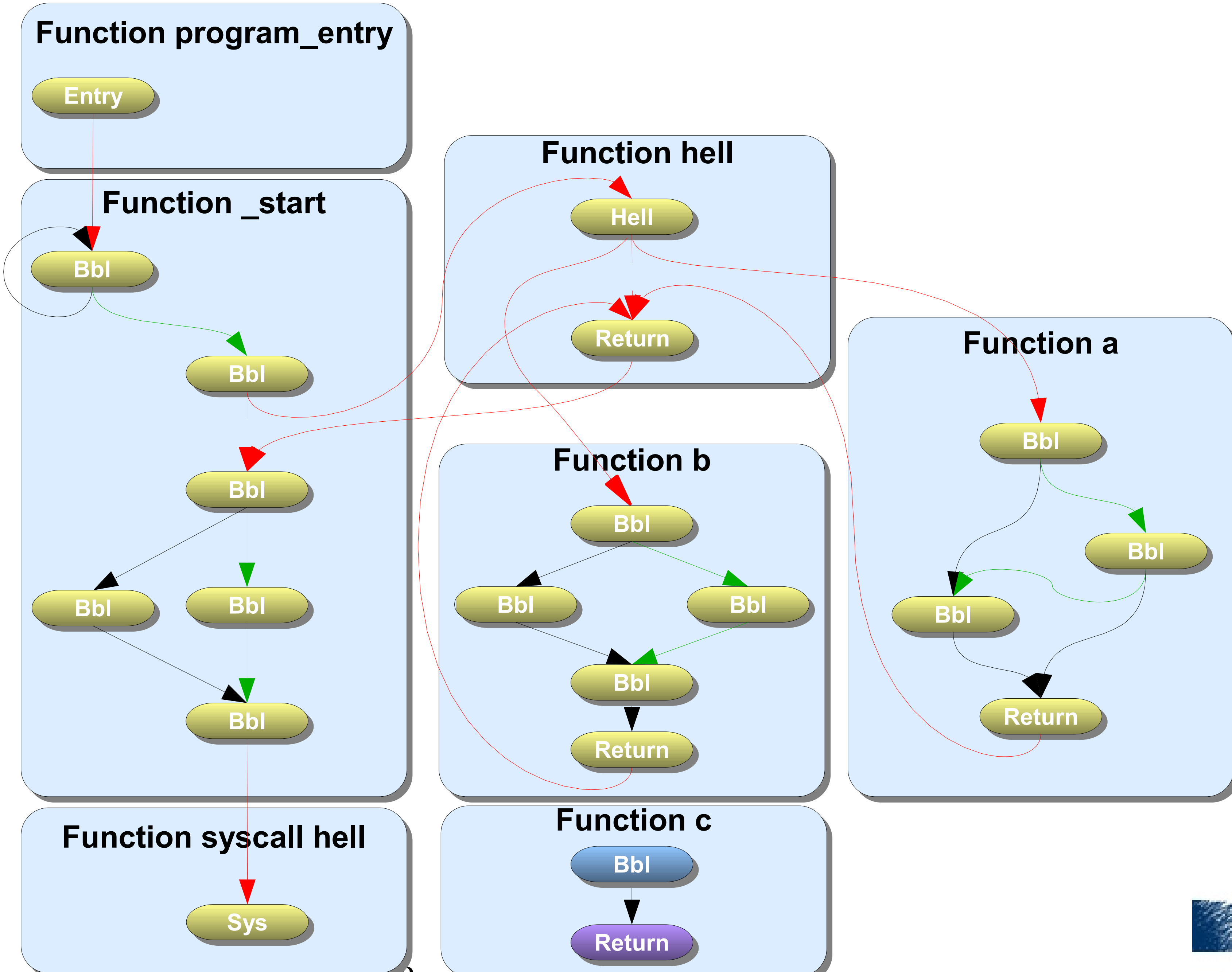
Part 3: Analyses and transformations

- Data flow analyses
  - Liveness Analysis
  - Constant Propagation
- Control flow analyses
  - Dead Code and Data Detection
  - Inteprocedural Dominators
- Some high-level transformations
- Instrumentation Support
- A graphical interface

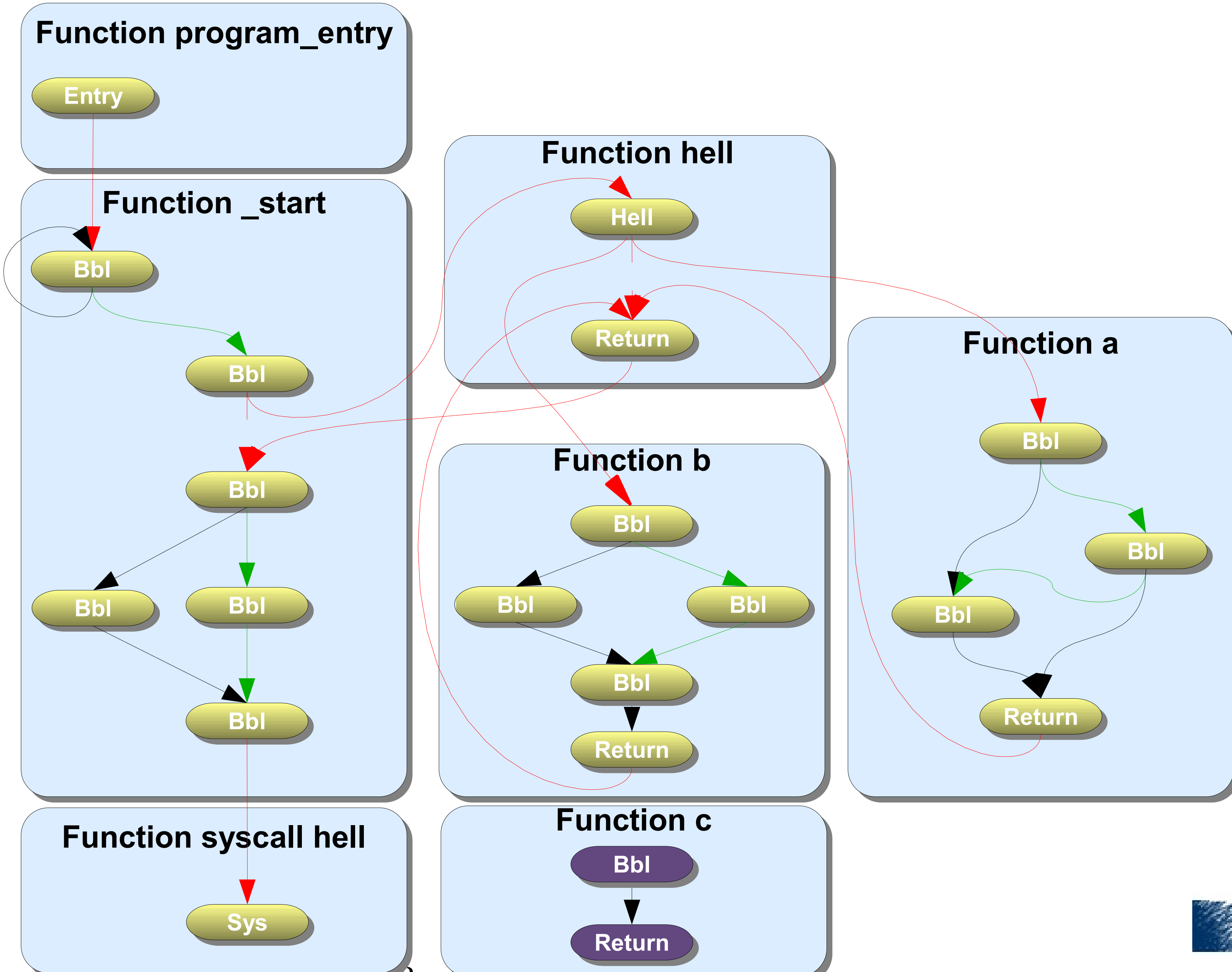
Dead Code Detection on ICFG



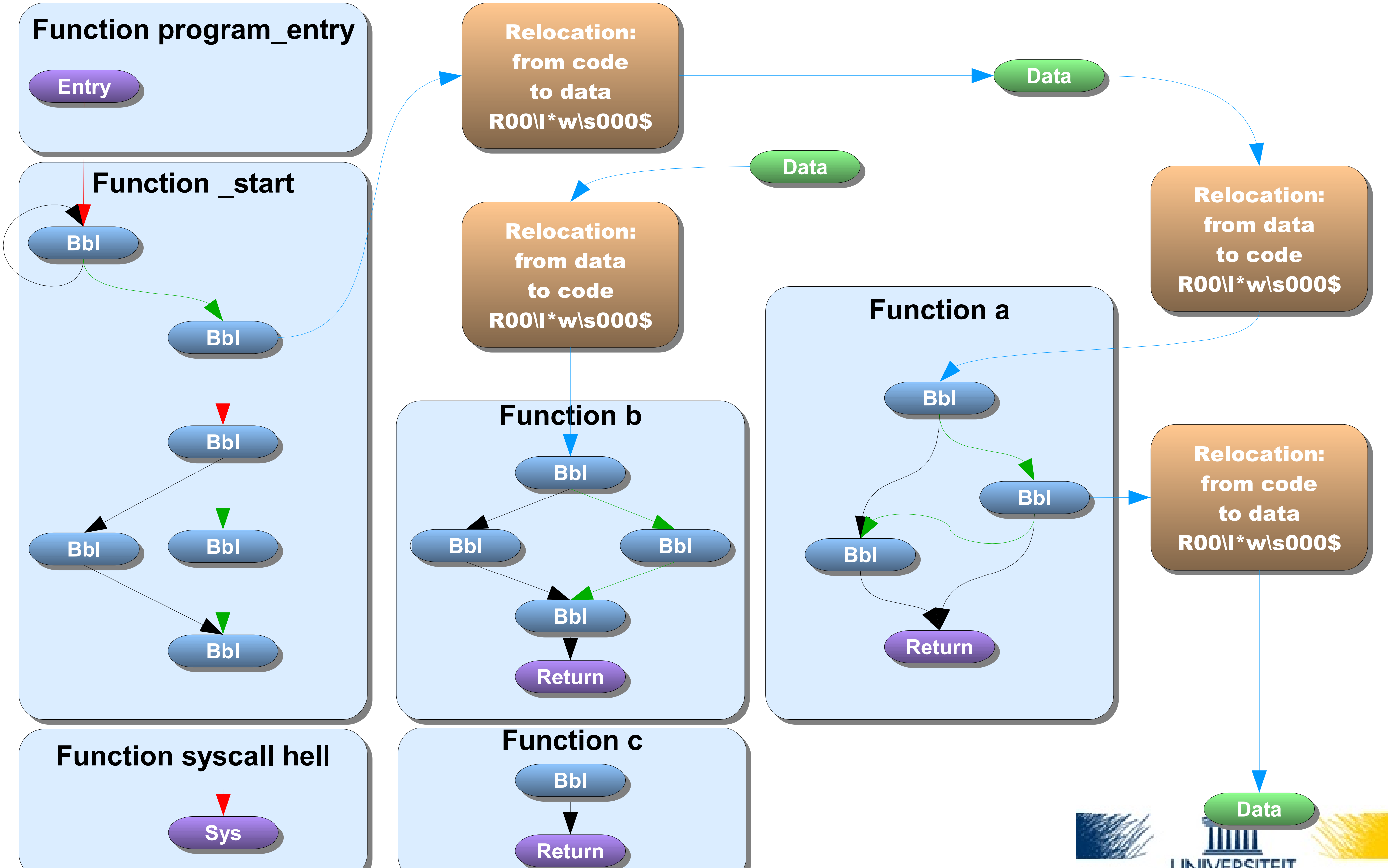
Dead Code Detection on ICFG



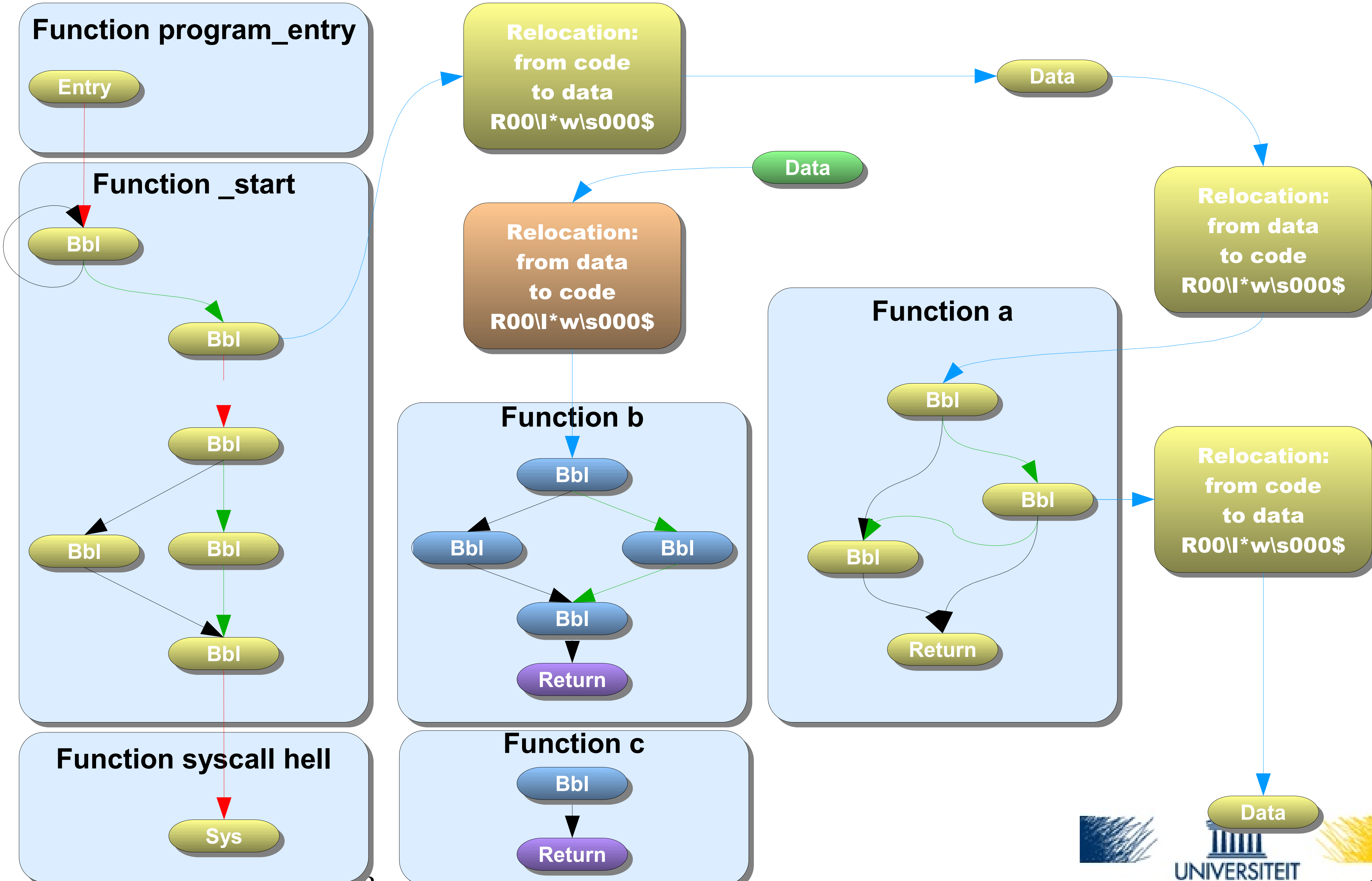
Dead Code Detection on ICFG



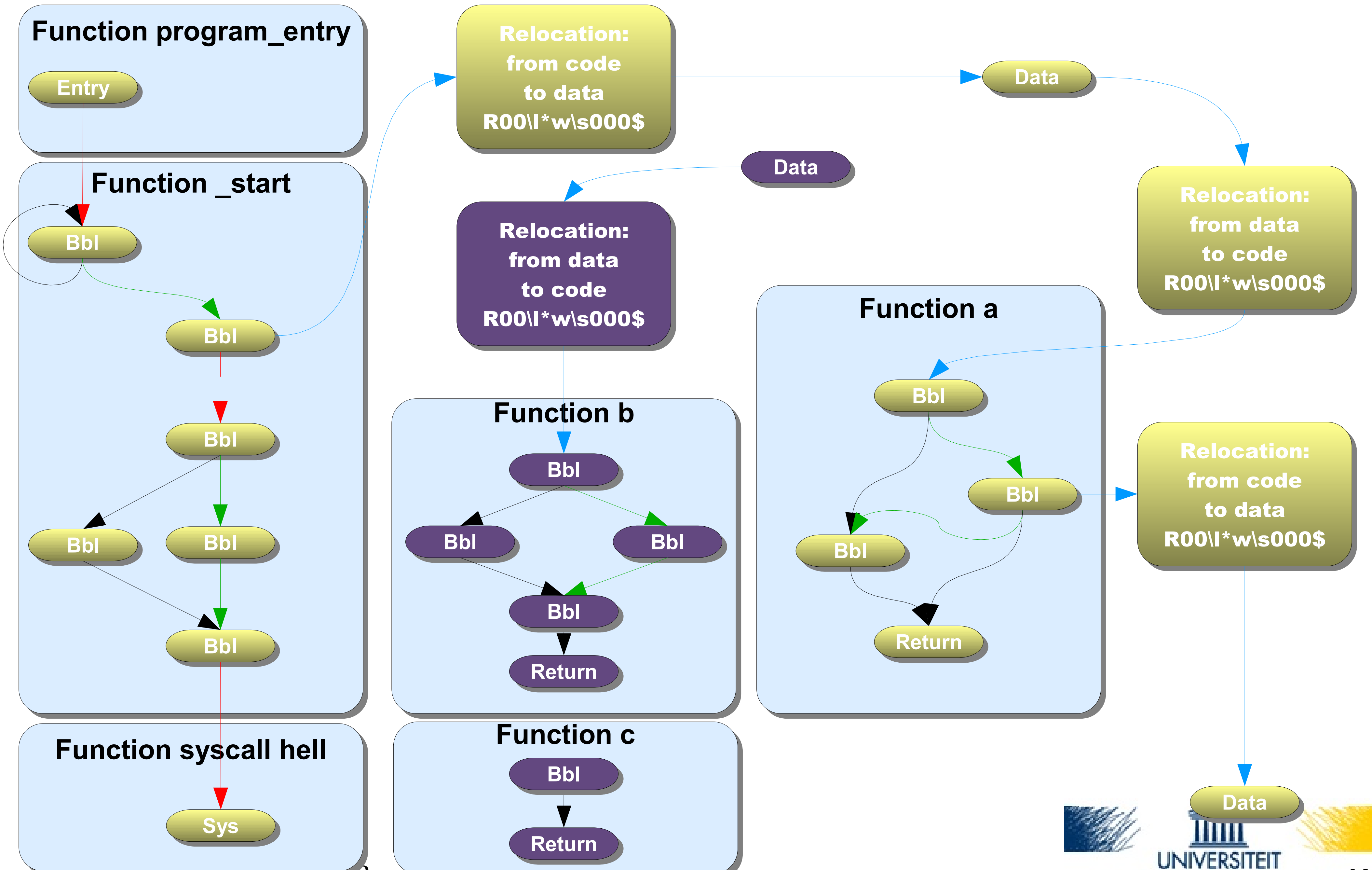
Dead Code and Data Detection on AWPCFG



Dead Code and Data Detection on AWPCFG

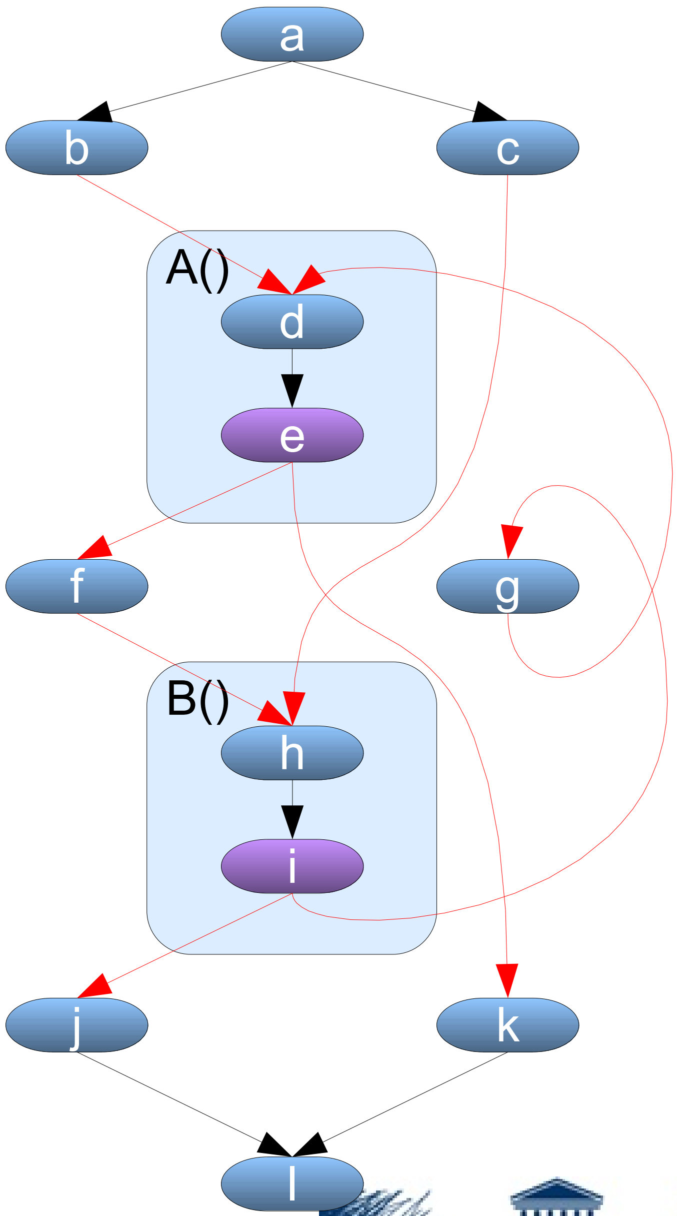


Dead Code and Data Detection on AWPCFG



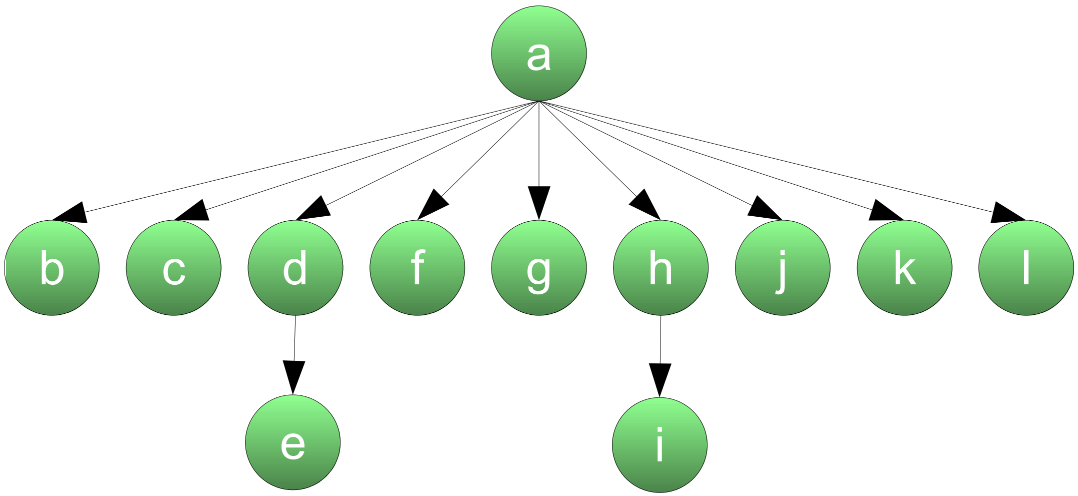
Interprocedural Dominator Analysis

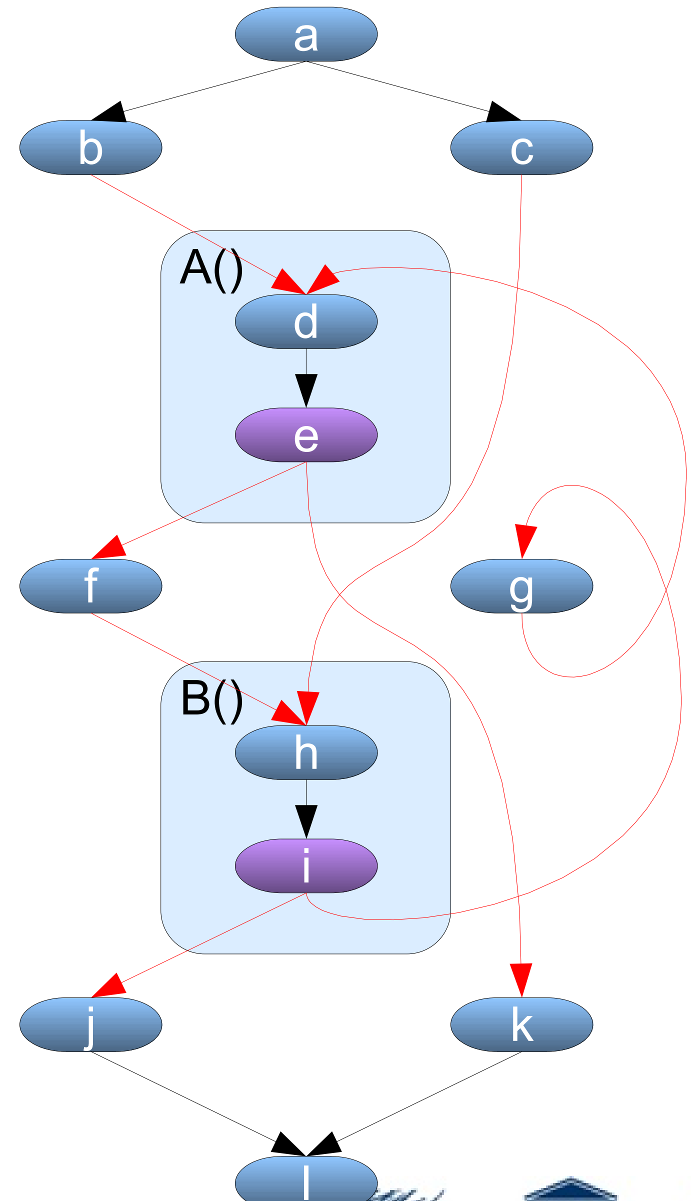
```
void F(int x)
{
   if (x > 0)
   {
      A();
      B();
   }
   else
   {
      B();
      A();
   }
}
```



Interprocedural Dominator Analysis

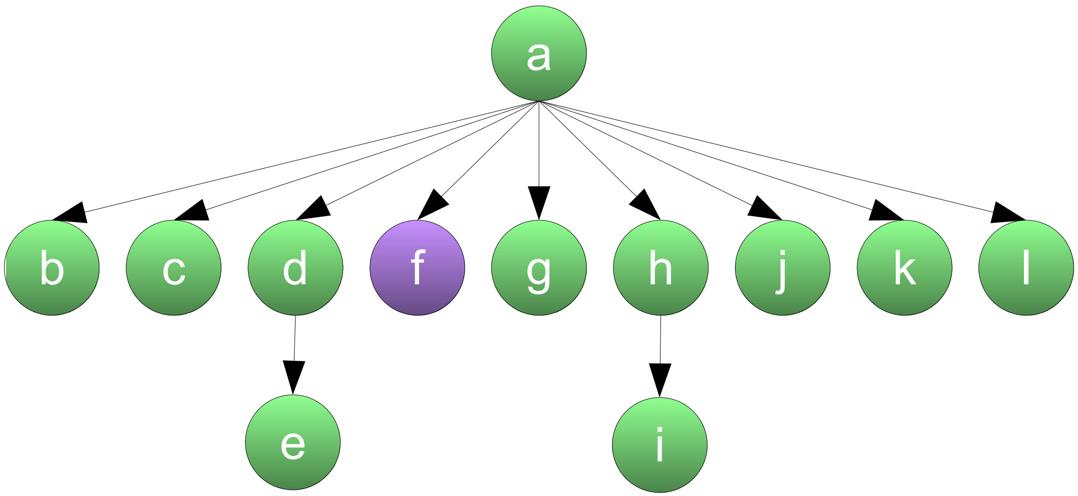
Classical dominator analysis

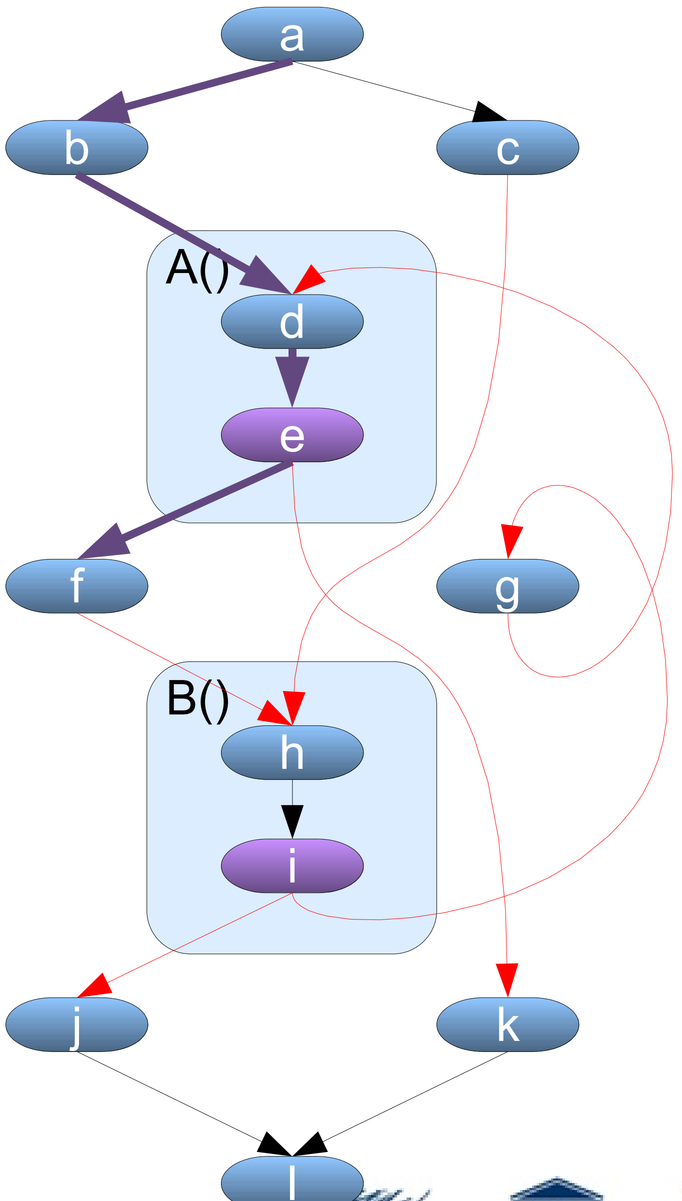




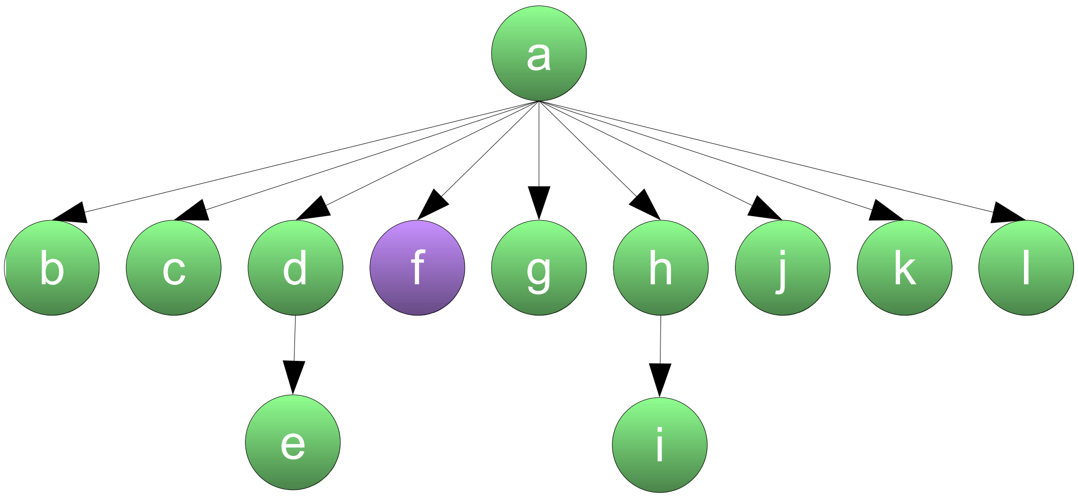
Interprocedural Dominator Analysis

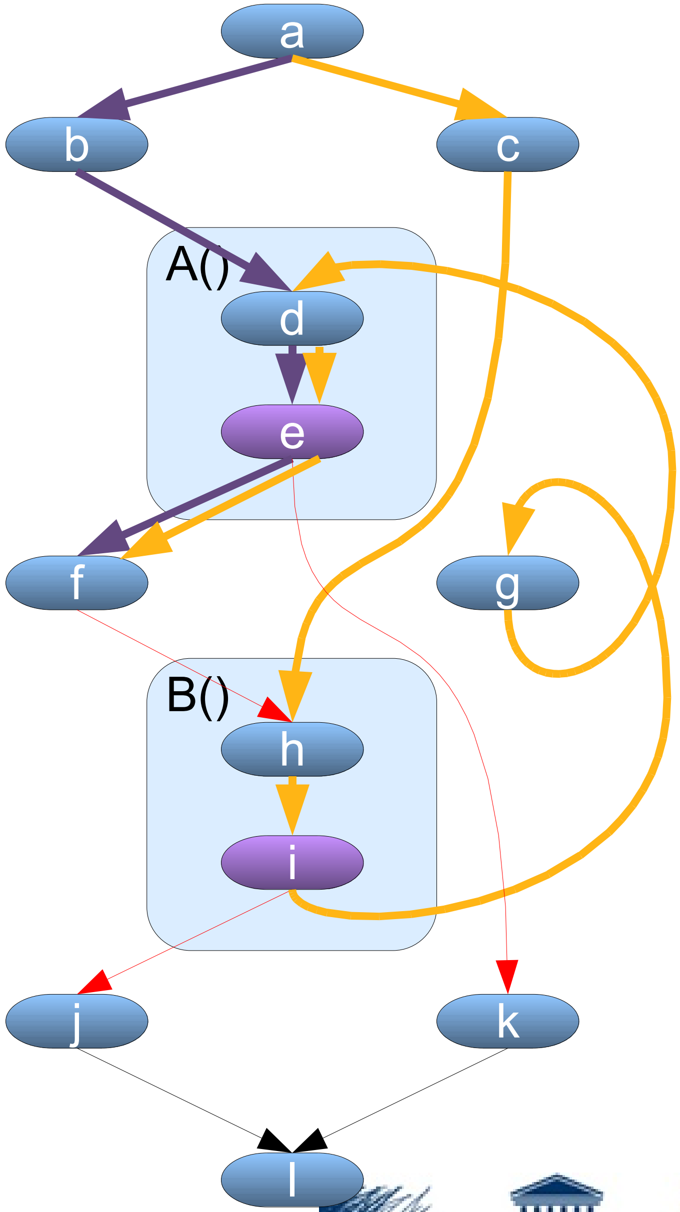
Classical dominator analysis





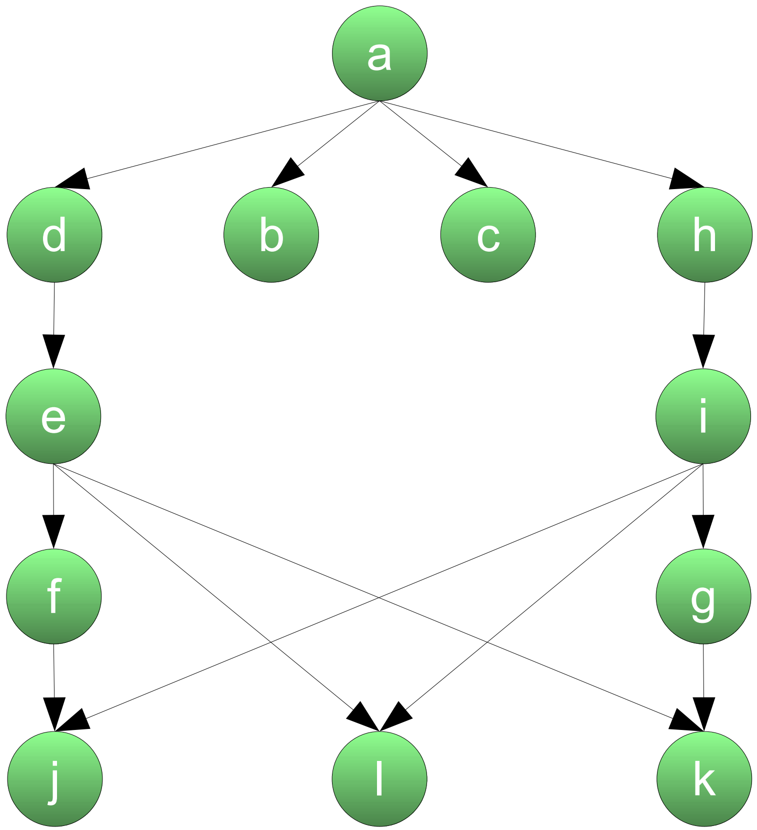
Classical dominator analysis

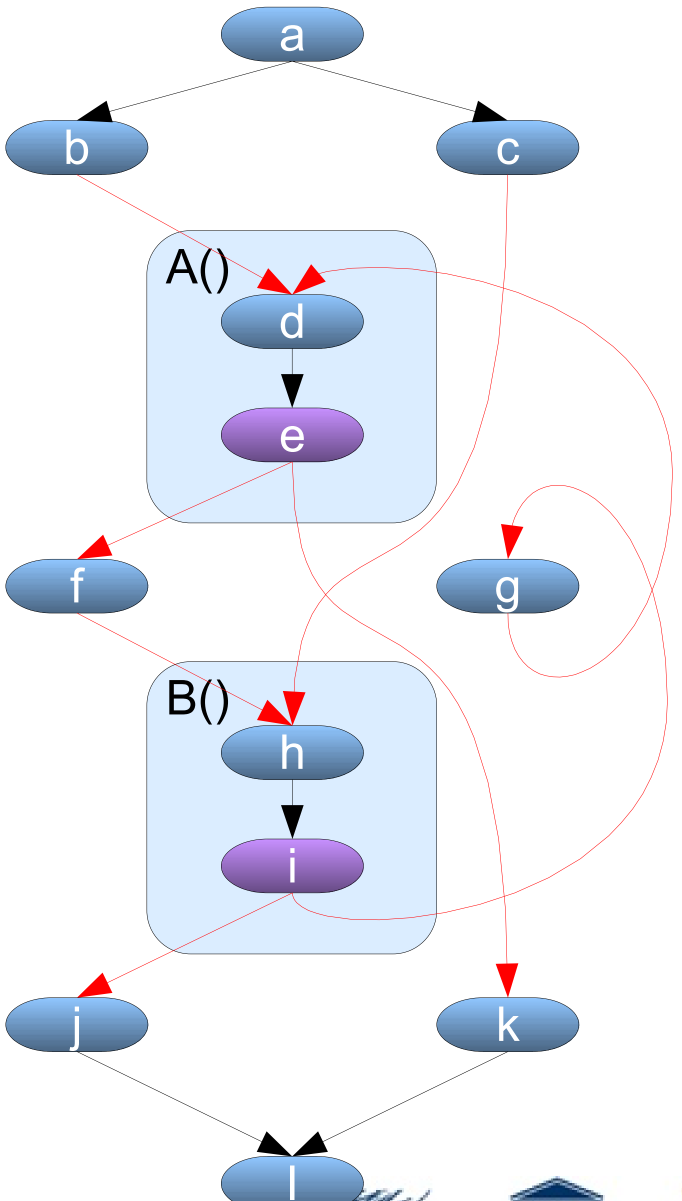




Interprocedural Dominator Analysis

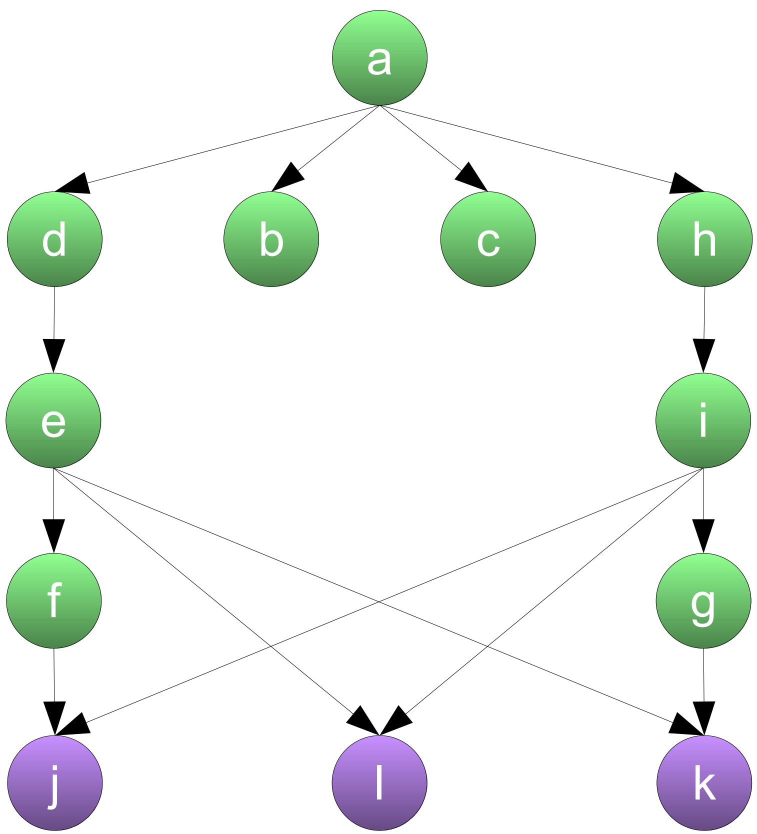
Interprocedural dominator analysis

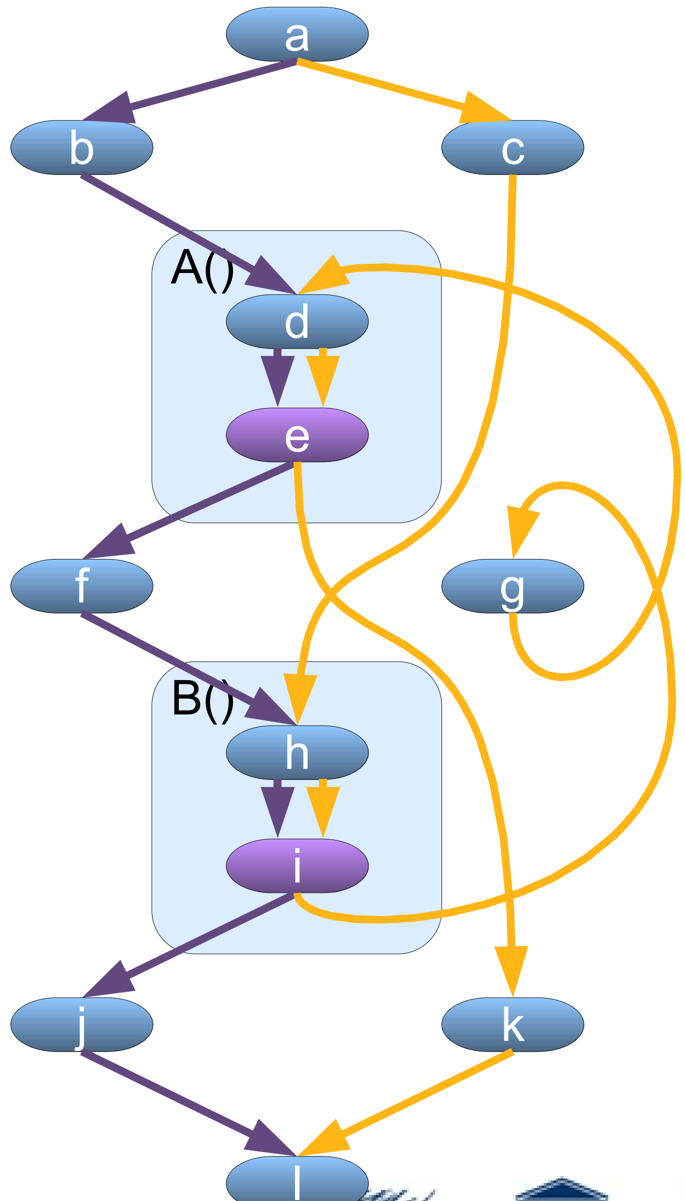




Interprocedural Dominator Analysis

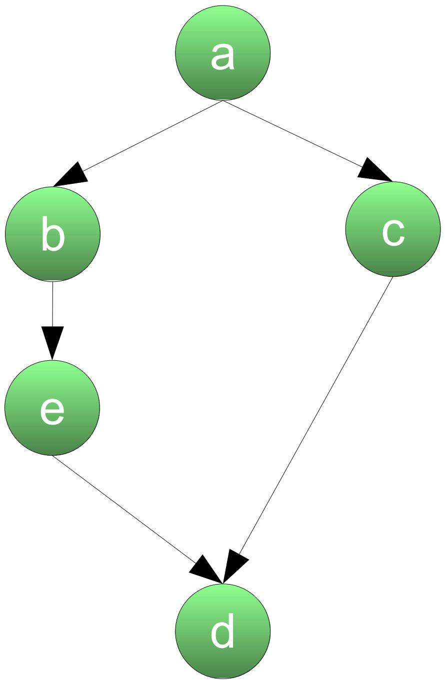
Interprocedural dominator analysis

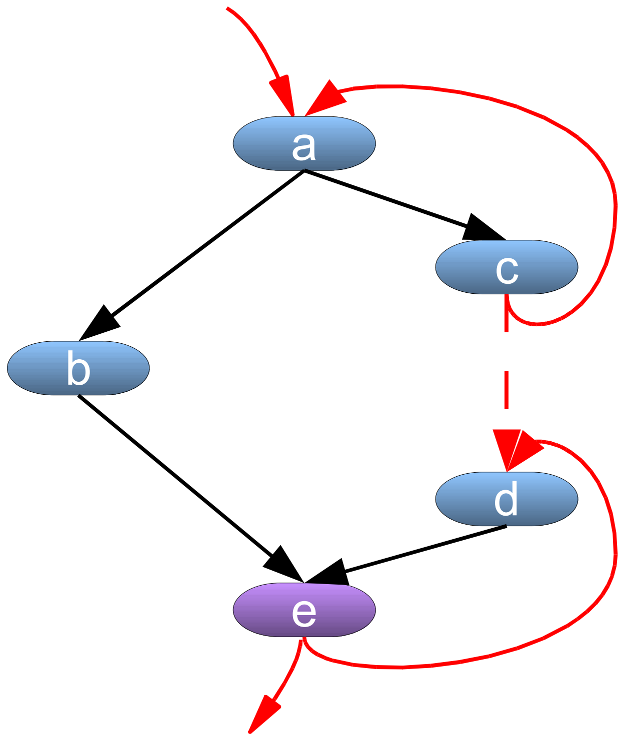




No longer a dominator tree
[De Sutter et al. 2006???]

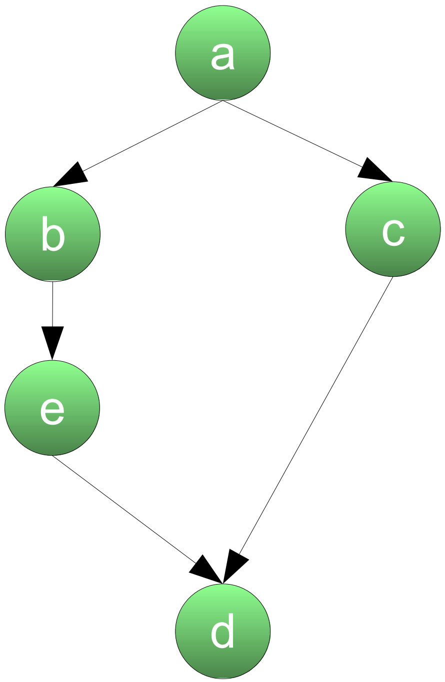
Recursive function

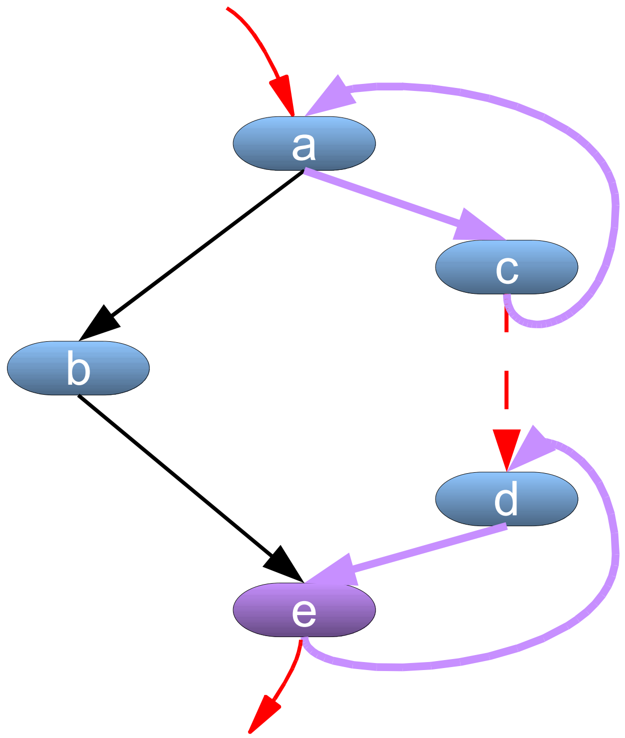




Interprocedural Dominator Analysis

Recursive function



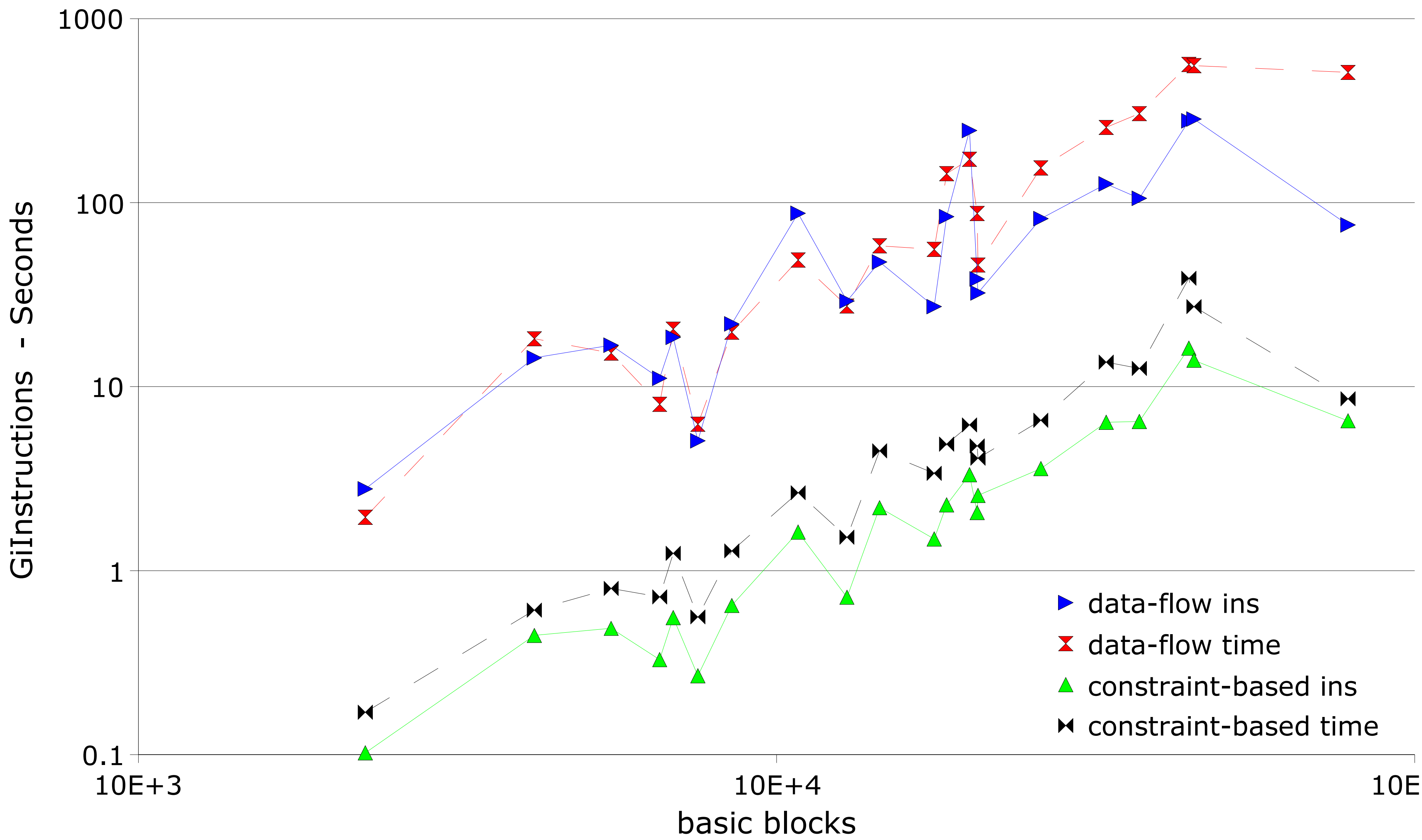


2 natural loops, crossing function header/footer

Interprocedural Dominator Analysis

- Classical algorithms (e.g. Lengauer and Tarjan) cannot be used to compute the interprocedural dominance relation
- Pure data flow solution is slow: $O(n^2m)$, with n = #nodes, m = #edges
- ICFG: typically thousands of nodes and edges!
- Implementation in Diablo: fast enough for practical purposes.
  Up to 20x faster due to optimized algorithm

Interprocedural Dominator Analysis



Using Dominator Analysis in Diablo

- Compute interprocedural dominator information: `ComDominators(cfg)`
- `BBL_DOMINATED_BY(bbl)`: linked list of immediate dominators of a basic block
- Global variable `dominator_info_correct`: avoid unnecessary recomputation of dominator information.
- Any transformation that changes the ICFG's structure should assign `FALSE` to this variable.

Loop Information in Diablo

- `ComDominators` also computes the natural loops in the ICFG
- iterate over the loops:
  ```
  CFG_FOREACH_LOOP(cfg,loop) { ... }
  ```
- retrieve loop header: `LOOP_HEADER(loop)`
- iterate over blocks in loop:
  ```
  t_loopiterator *iterator;
  LOOP_FOREACH_BBL(loop,iterator,bbl) { ... }
  ```

Part 3: Analyses and transformations

- Data flow analyses
  - Liveness Analysis
  - Constant Propagation
- Control flow analyses
  - Dead Code and Data Detection
  - Inteprocedural Dominators
- Some high-level transformations
- Instrumentation Support
- A graphical interface

Complex Transformations

- Function inlining
- Function outlining
- Architecture-dependent peephole optimizations
- Loop-invariant code motion (ARM only)
- Partial redundancy elimination (ARM only)
  → uses profile information to guide the optimization

Part 3: Analyses and transformations

- Data flow analyses
  - Liveness Analysis
  - Constant Propagation
- Control flow analyses
  - Dead Code and Data Detection
  - Inteprocedural Dominators
- Some high-level transformations
- Instrumentation Support
- A graphical interface

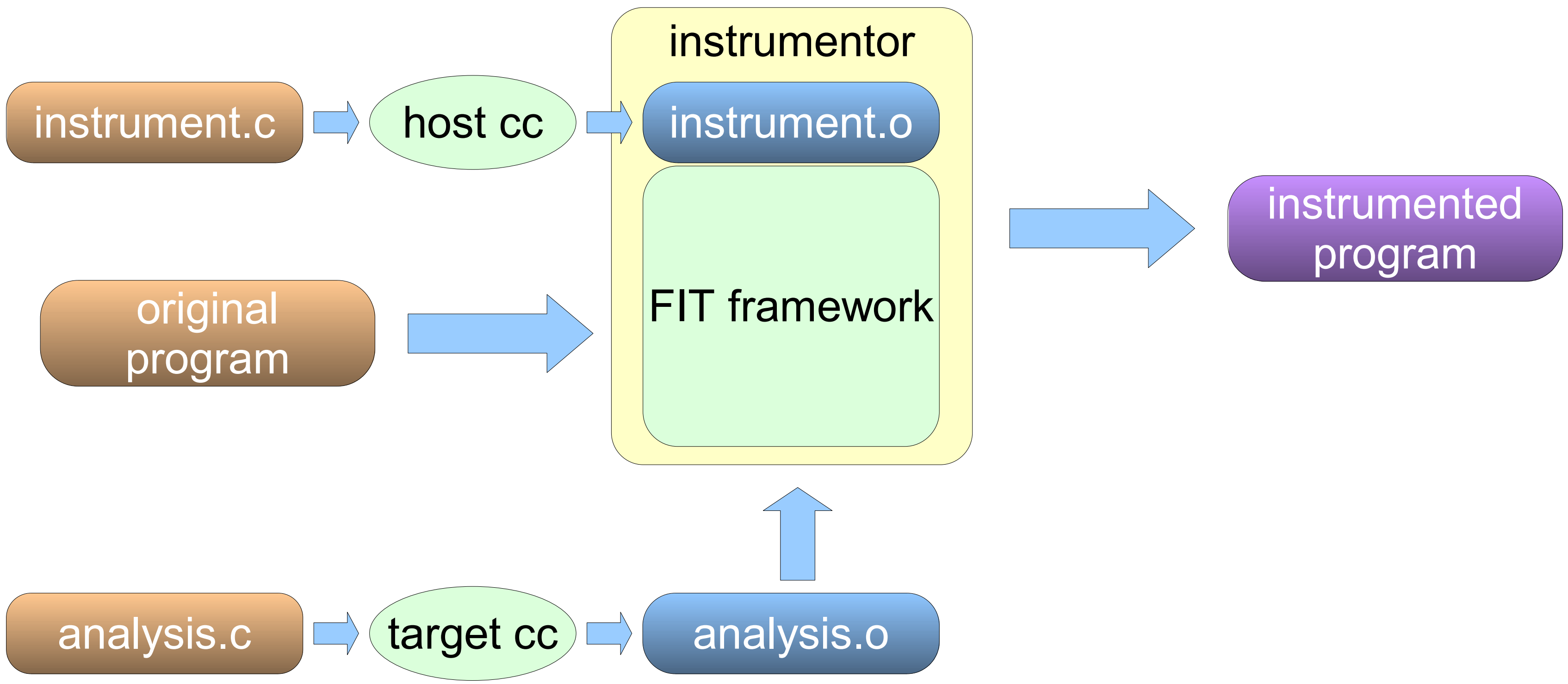
the Flexible Instrumentation Toolkit

- creates binary instrumentors for i386 and ARM
- allows insertion of arbitrary analysis routines before or after
  - instructions
  - basic blocks
  - procedures
  - the program
- ATOM-compatible

FIT

- Built on the Diablo framework
- FIT and Diablo have identical program representation
  - partitioning in functions
  - function names
  - ...
- FIT results can easily be fed back to other Diablo-based tools
  e.g. FIT-generated profile information is used to guide some optimizations provided by Diablo

How does it work?



Example: basic block profiling

Data structures

```
extern int count[];
```

Call upon entry of each basic block

```
void CountBlock(int index) {
  count[index]++;
}
```

basic block index

Call upon program termination

```
void WriteProfile(int nblocks) {
  int i;
  FILE *f = fopen("prof.out","w");
  for (i = 0; i < nblocks; i++)
    fprintf(f, "%d\n", count[i]);
  fclose(f);
}
```

total # of basic blocks

Analysis file

Example: basic block profiling

```
void InstrumentInit(int argc, char **argv) {
    AddCallProto("WriteProfile(int)");
    AddCallProto("CountBlock(int)");
    AddArrayProto("count");
}
void Instrument(int argc, char **argv, Obj *obj) {
    Proc *p; Block *b; int nblocks=0;
    FILE *fp = fopen(“addr.out”,”w”);

    for (p = GetFirstObjProc(obj); p != NULL; p = GetNextProc(p))
        for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b))
            nblocks++;
    SetArraySize("count",nblocks,4);

    nblocks=0;
    for (p = GetFirstObjProc(obj); p != NULL; p = GetNextProc(p)) {
        for (b = GetFirstBlock(p); b != NULL; b = GetNextBlock(b)) {
            SetArrayElem32("count",nblocks,1,0);
            AddCallBlock(b,BlockBefore,"CountBlock",nblocks++);
            fprintf (fp, “%x\n”, BlockPC(b));
        }
    }
    AddCallProgram(ProgramAfter,"WriteProfile",nblocks);
    fclose(fp);
}
```

Declare analysis functions and data structures

Compute data structure size

Initialize data structure

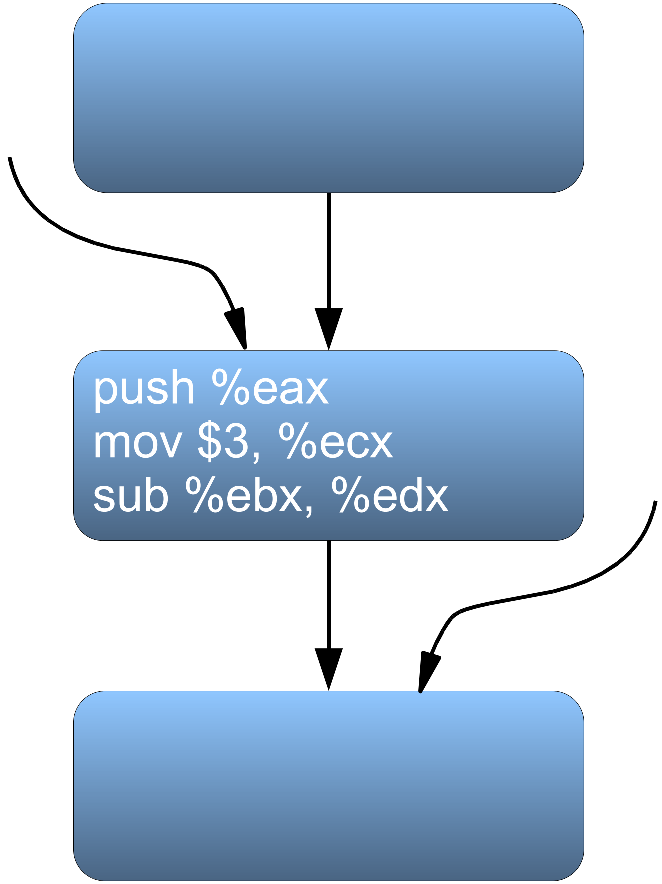
Call analysis function

Basic block addresses known at instrumentation time

Call analysis function

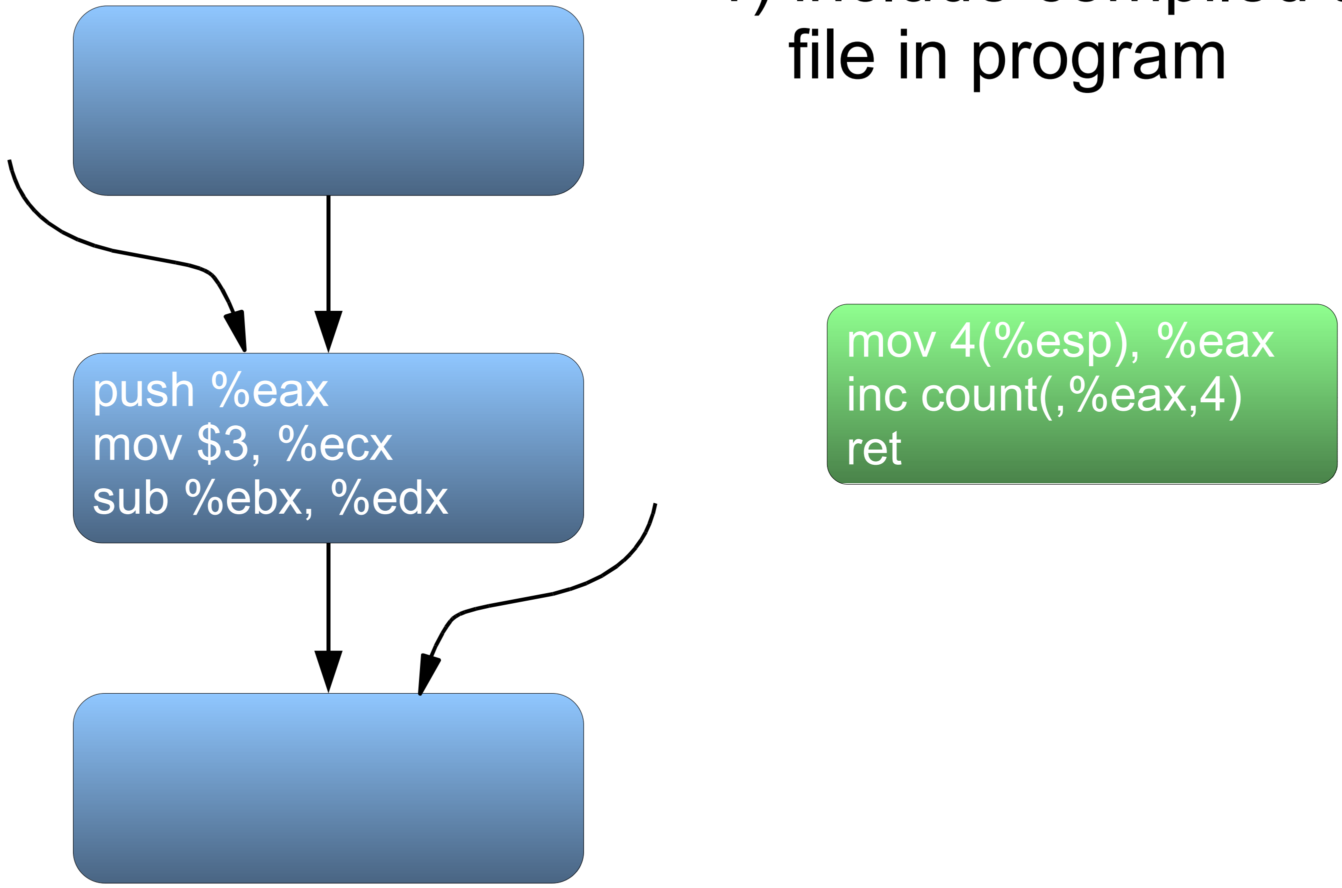
Instrumentation file

How does the instrumentation work?



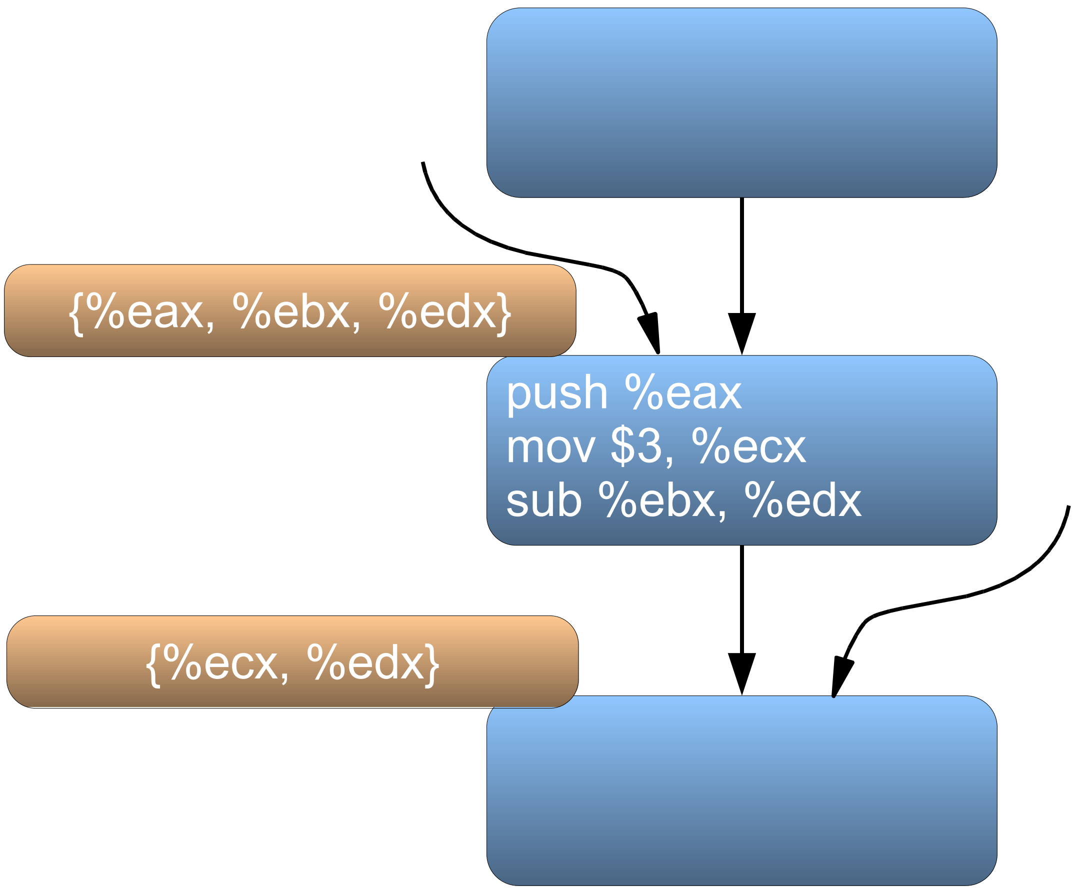
How does the instrumentation work?

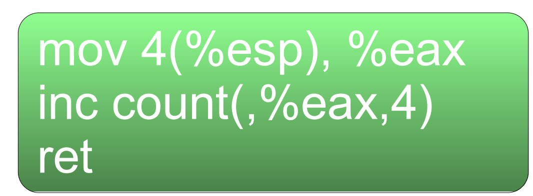
1) include compiled analysis file in program



How does the instrumentation work?

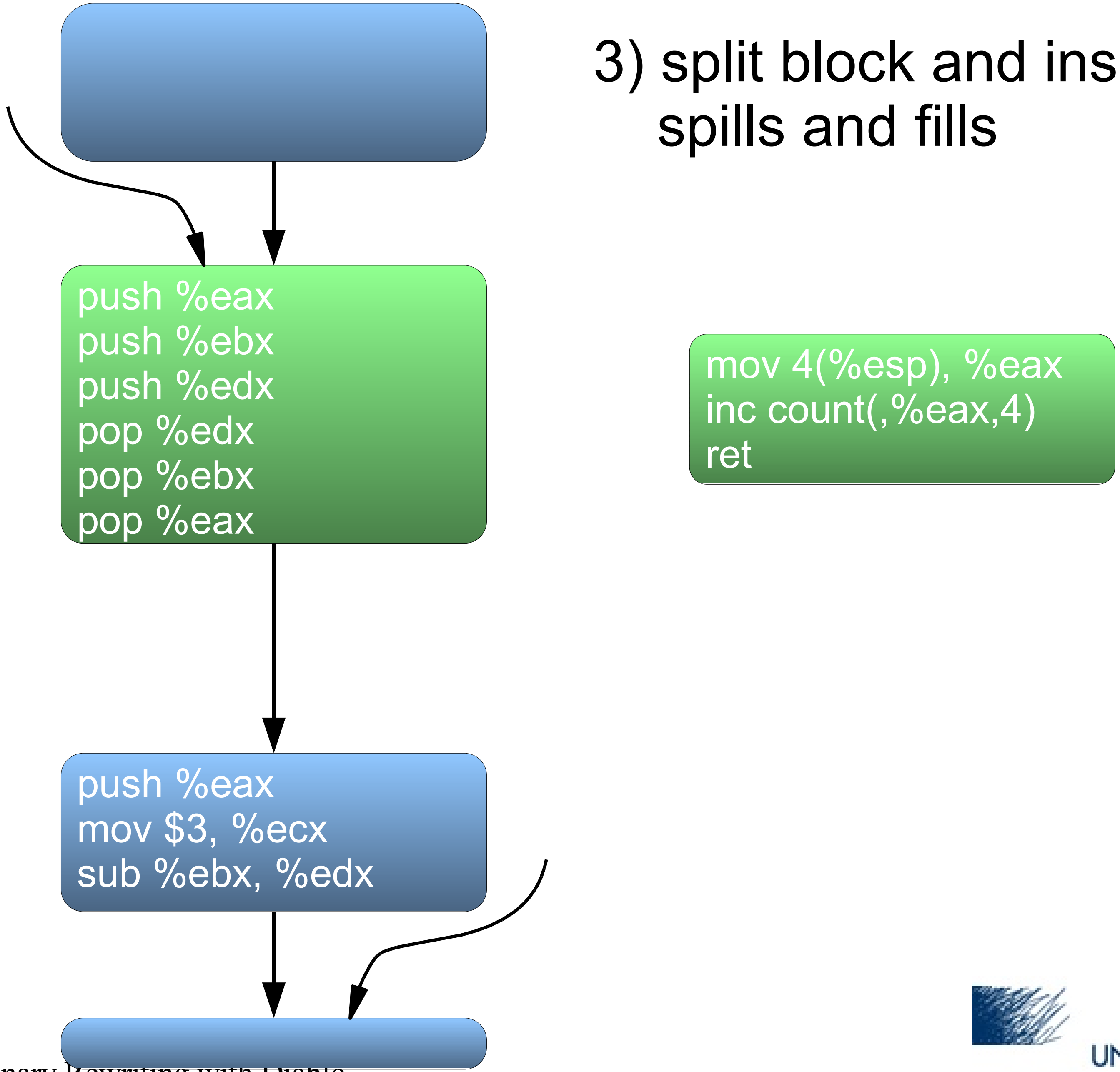
2) detect used registers
with liveness analysis



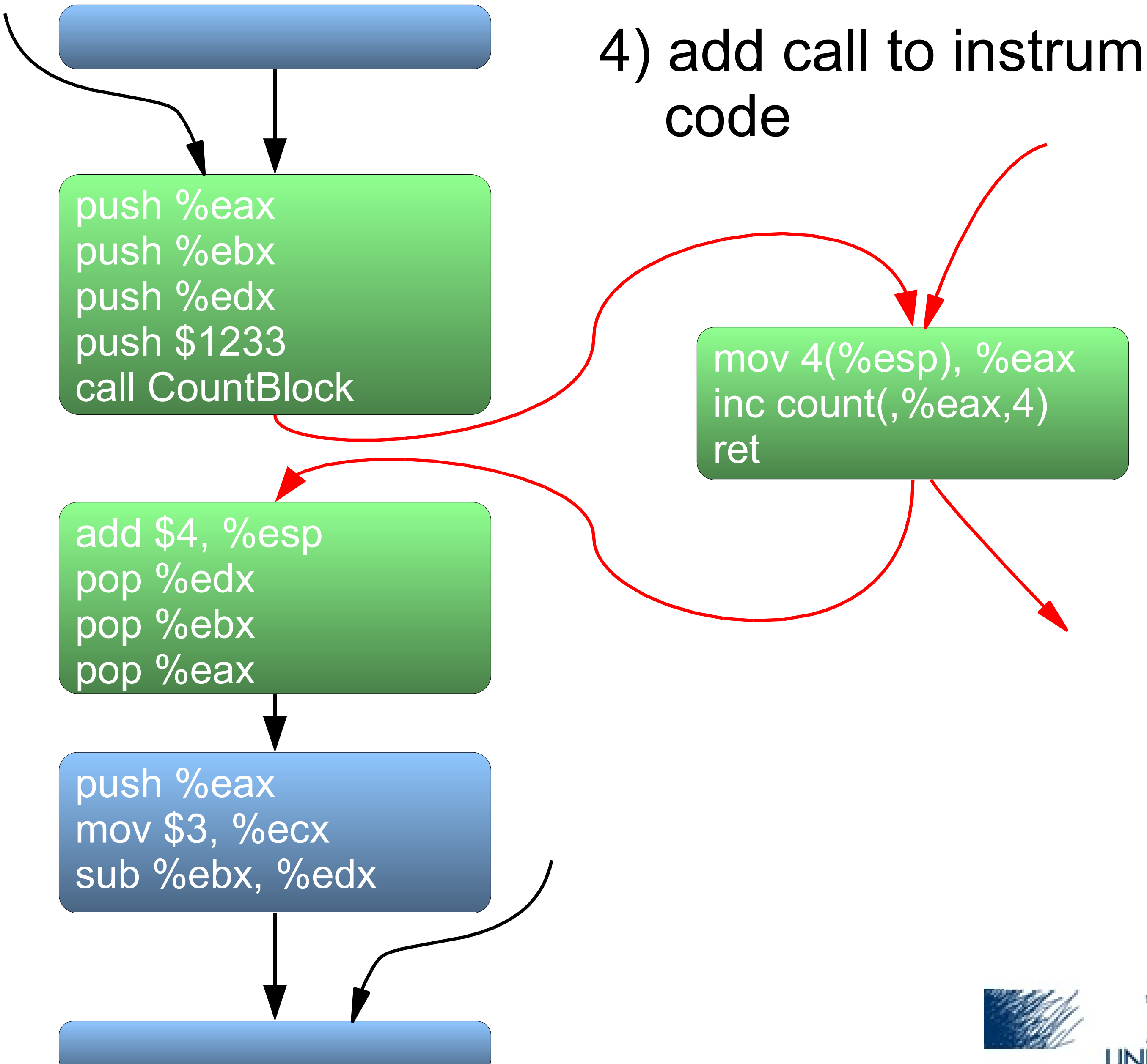


How does the instrumentation work?

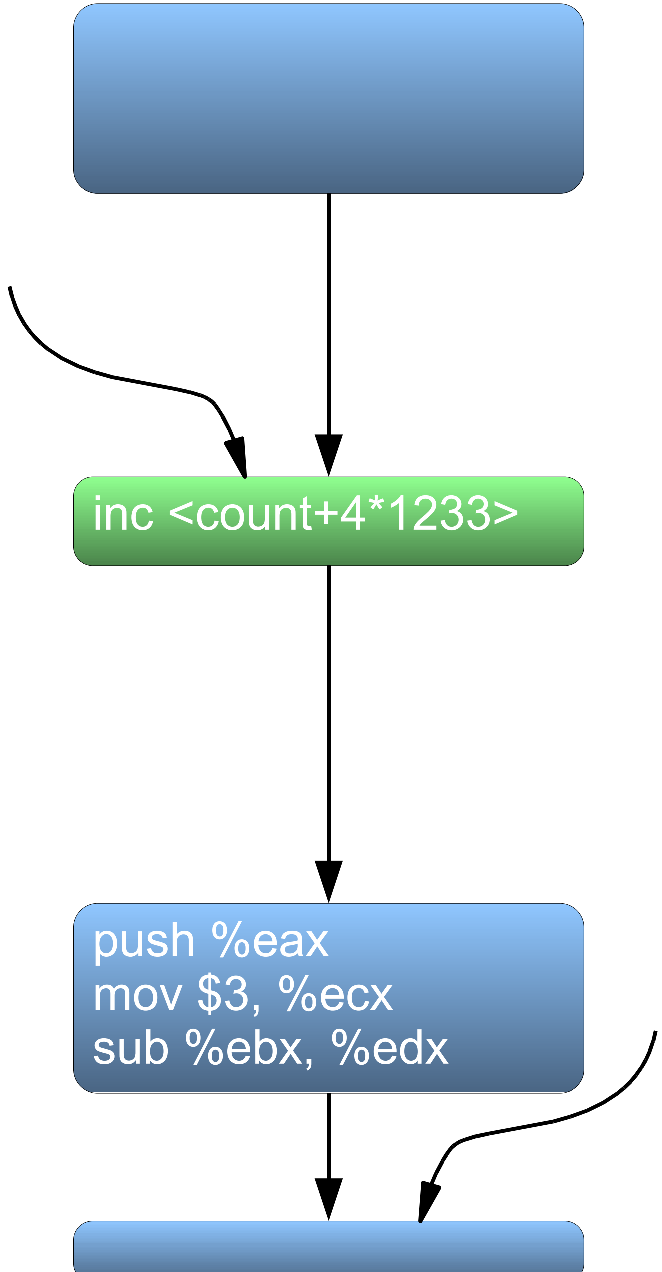
3) split block and insert spills and fills



How does the instrumentation work?



4) add call to instrumentation code



5) use standard optimizations in Diablo

Part 3: Analyses and transformations

- Data flow analyses
  - Liveness Analysis
  - Constant Propagation
- Control flow analyses
  - Dead Code and Data Detection
  - Inteprocedural Dominators
- Some high-level transformations
- Instrumentation Support
- A graphical interface

LANCET: A Nifty Code Editing Tool

- How does one use Diablo?
  - First, try to find optimization opportunities
  - produce dot-files and run dot, study CFGs and grep 120 kilo-lines of analysis output
  - come up with some great optimization, implement it
  - Debug it: study many CFGs with corner cases and adapt implementation

→ you develop great shell scripting skills to automate dot and grep

LANCET: A Nifty Code Editing Tool

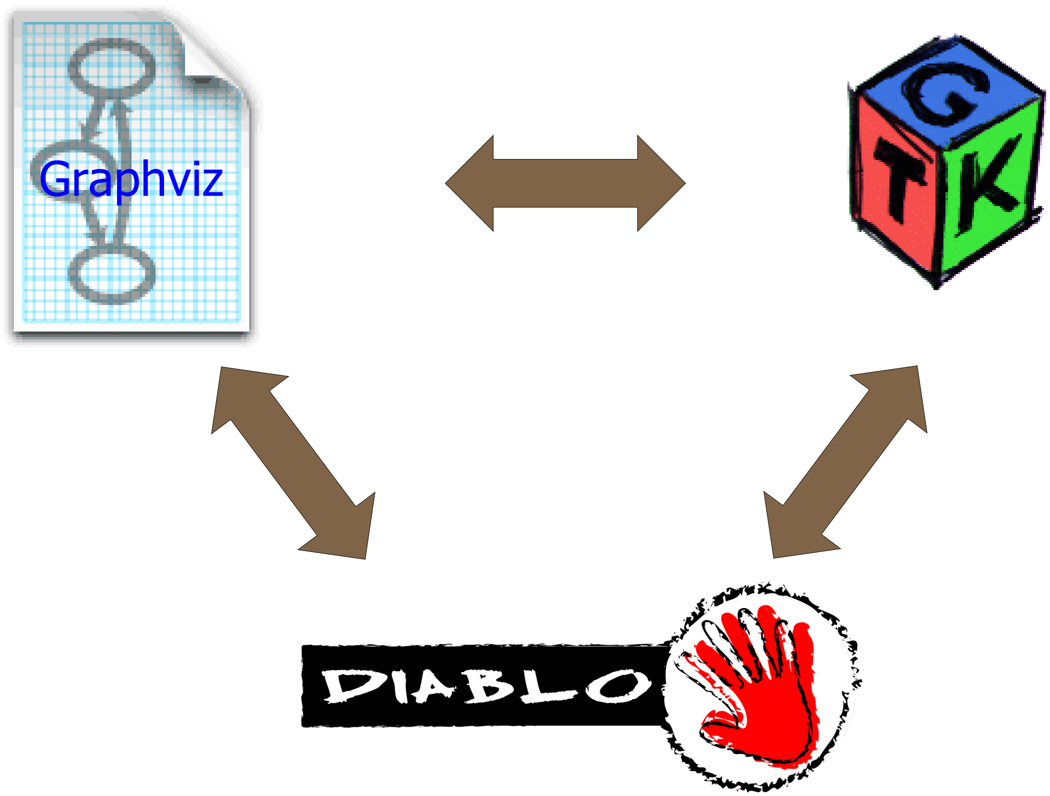
- Alternative: Lancet
  - Study graphs interactively with GUI
  - Show data flow analysis information while hoovering
  - Let Lancet show cases where conditions hold
  - Instead of just FATALing when things go wrong, GFATALBBL with immediate feedback and inquiry options

→ you don't develop the scripting skills, but save a lot of time

Architecture of LANCET

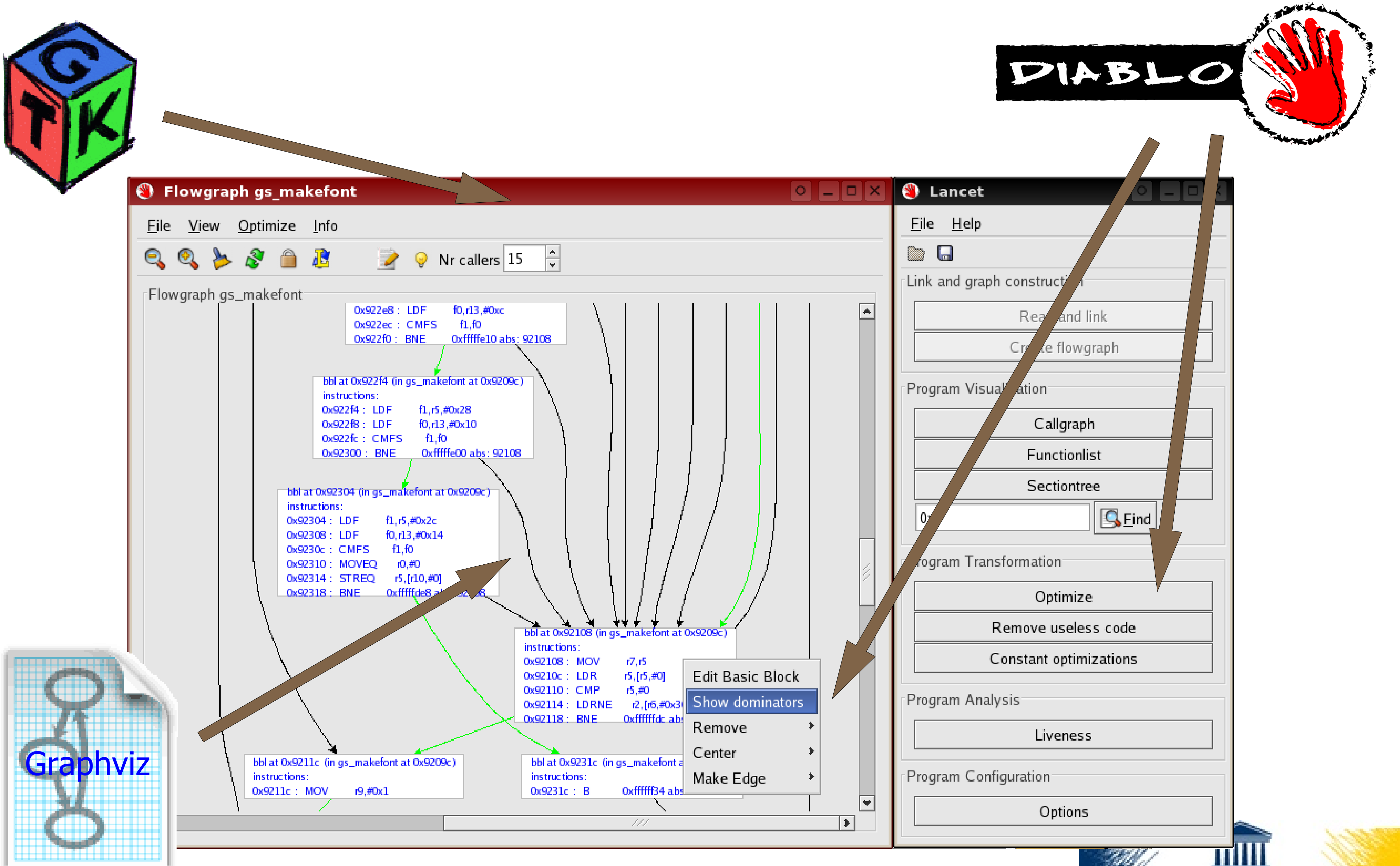
Graphviz: a powerful graph layout engine

GTK+: a toolkit to create a graphical users interface

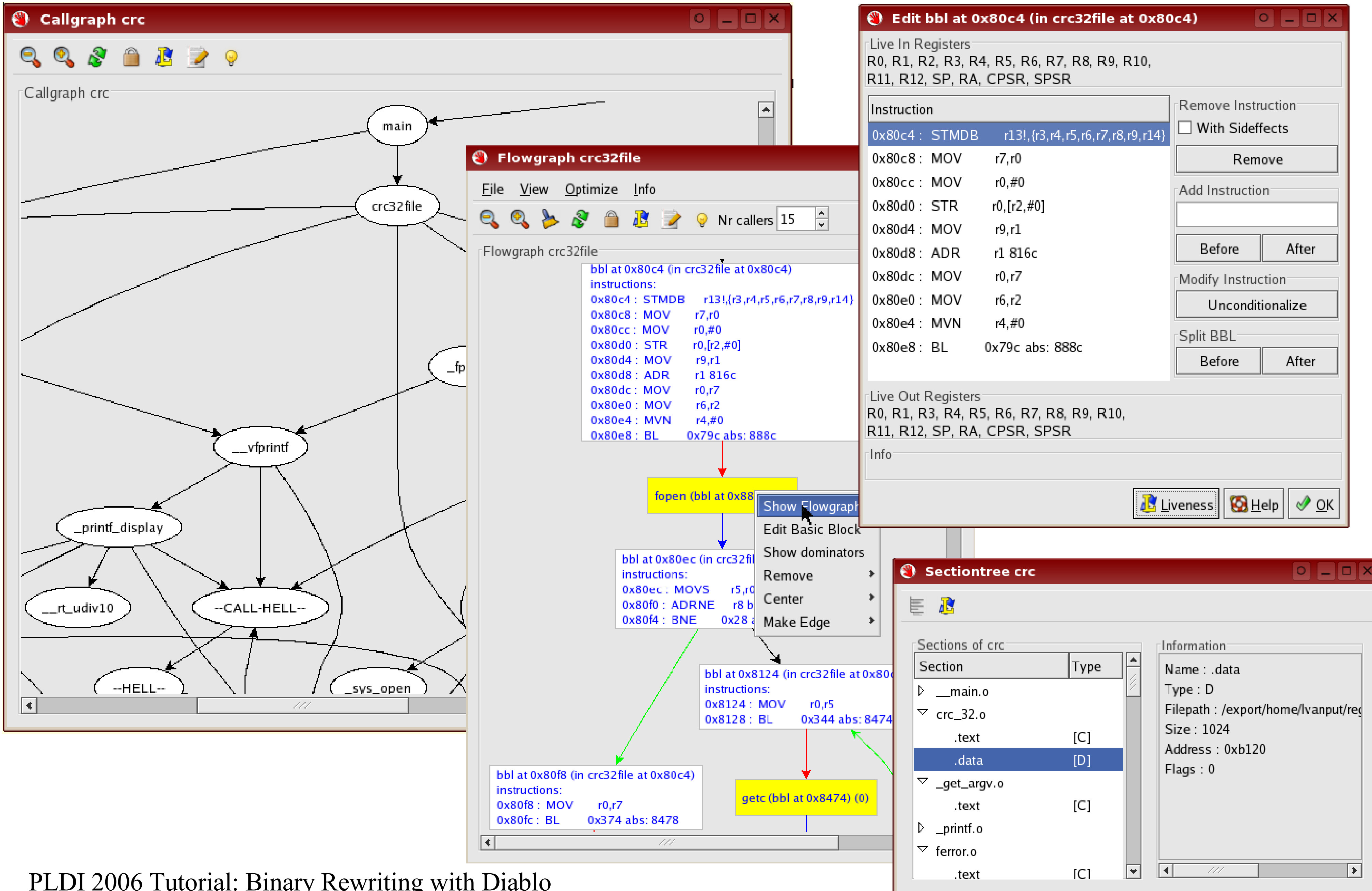


DIABLO: a link-time binary rewriting framework

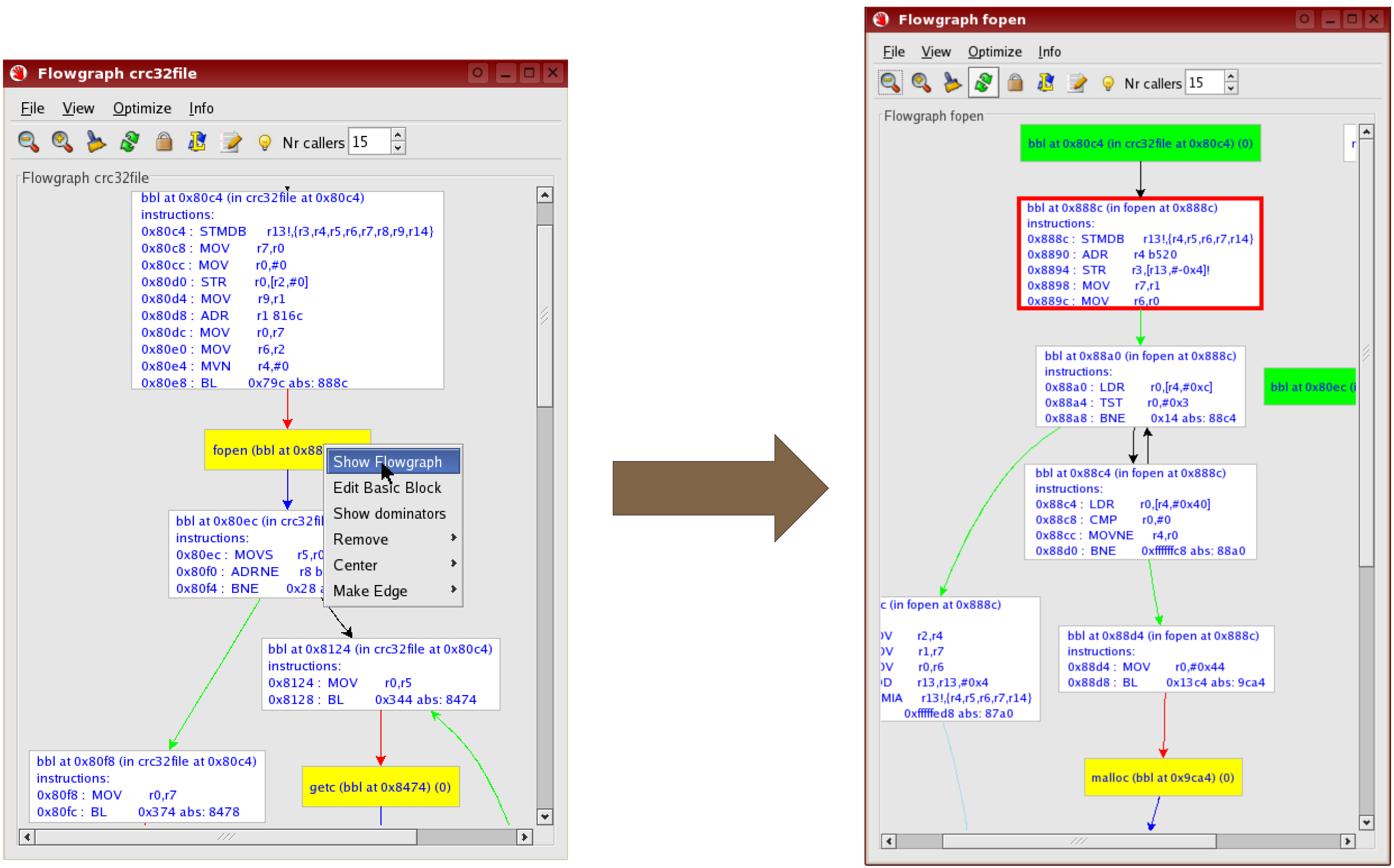
Cooperation of libraries



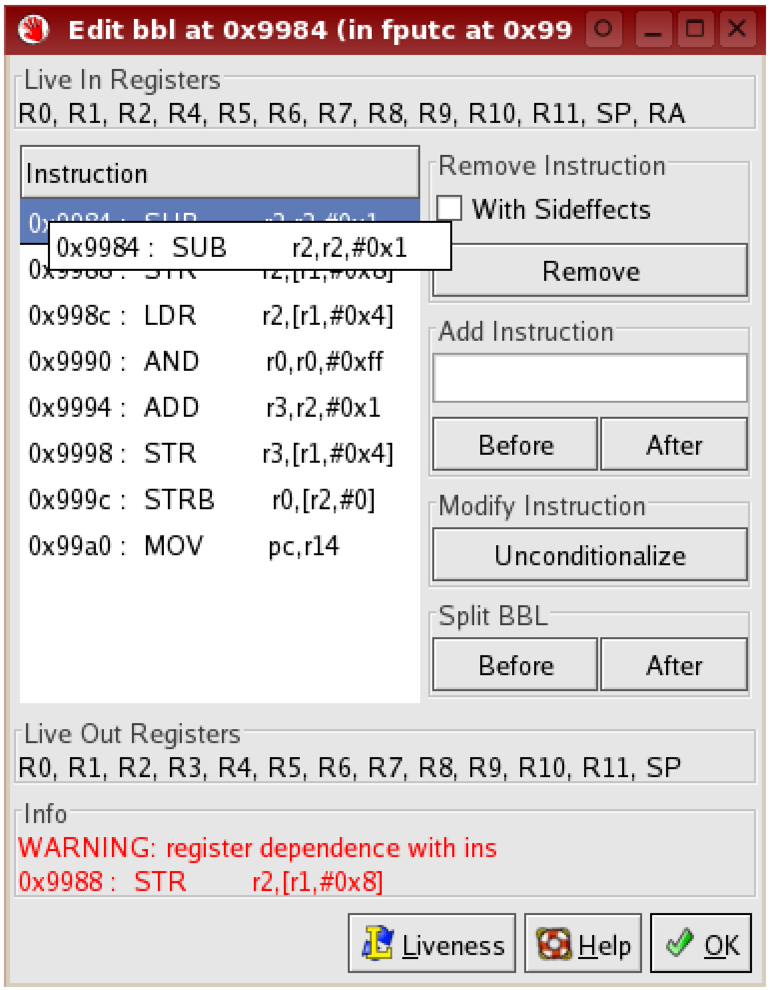
Visualization of code



Easy navigation through the program



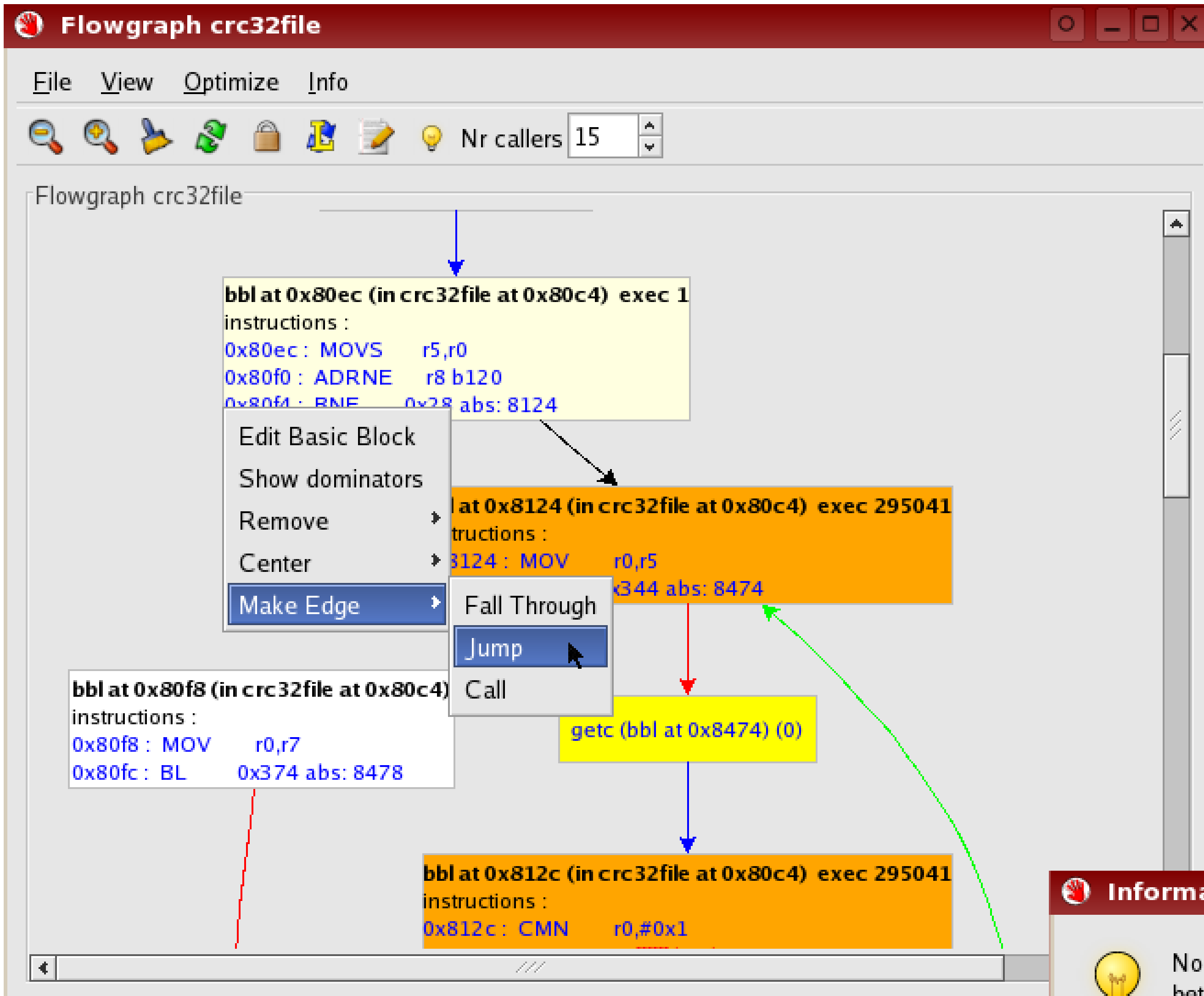
Manually change the instructions



Add/remove/modify instructions, reorder instructions, split basic blocks, ...

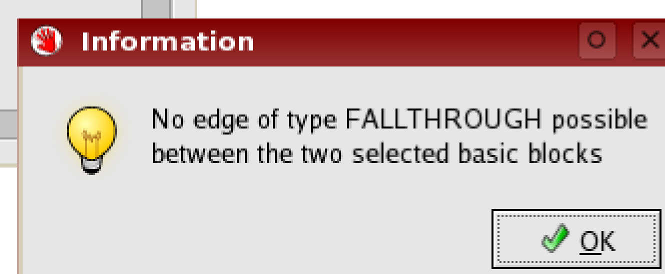
Built-in or user-provided analyses can look over your shoulder and provide feedback

Manually change control flow

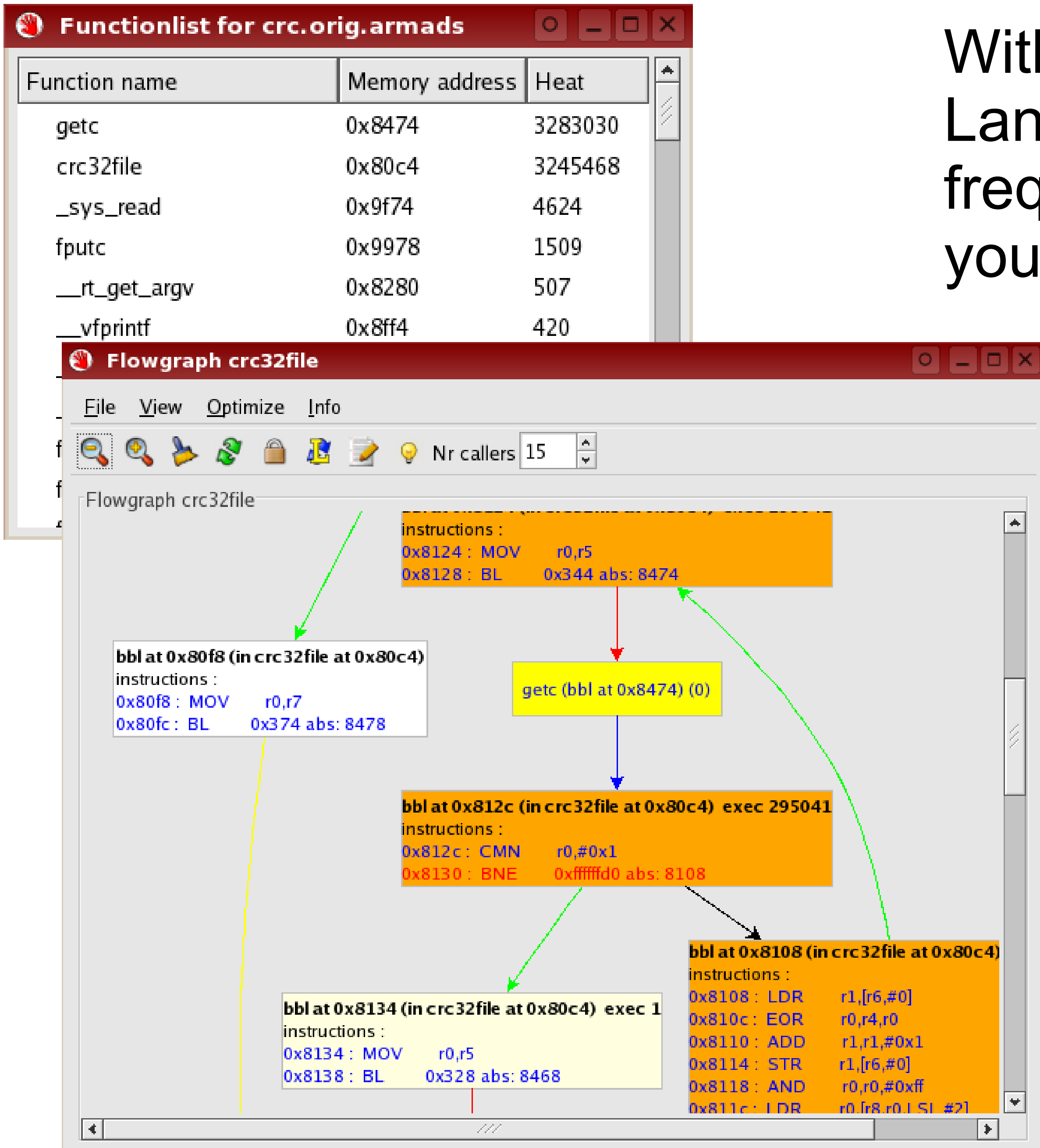


Add, remove, modify, delete edges...

Diablo can check inconsistencies (e.g. circular fallthrough paths) or handle side-effects (e.g. make a jump unconditional when the fallthrough path is removed).



Focus on hot code

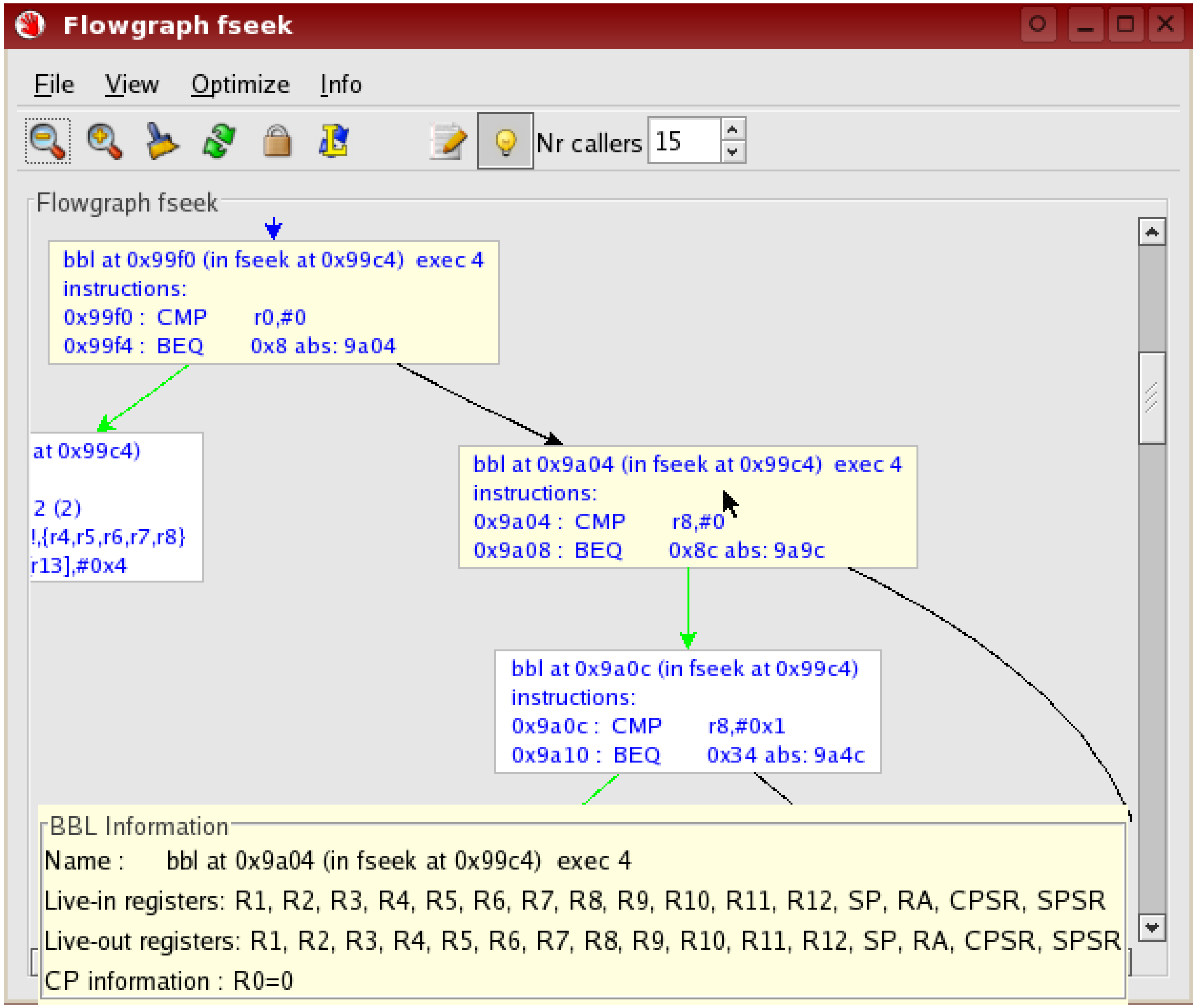


With profile information, Lancet lets you find the most frequently executed parts of your program immediately.

Profile-view uses color information to show execution frequency.

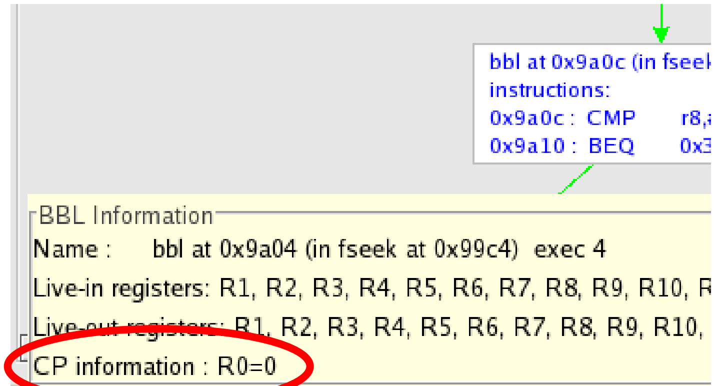
Conditional instructions having different heat are emphasized.

Display analysis results



Show liveness info, constant propagation info, dominator results or your own analysis results

How to display analysis results?



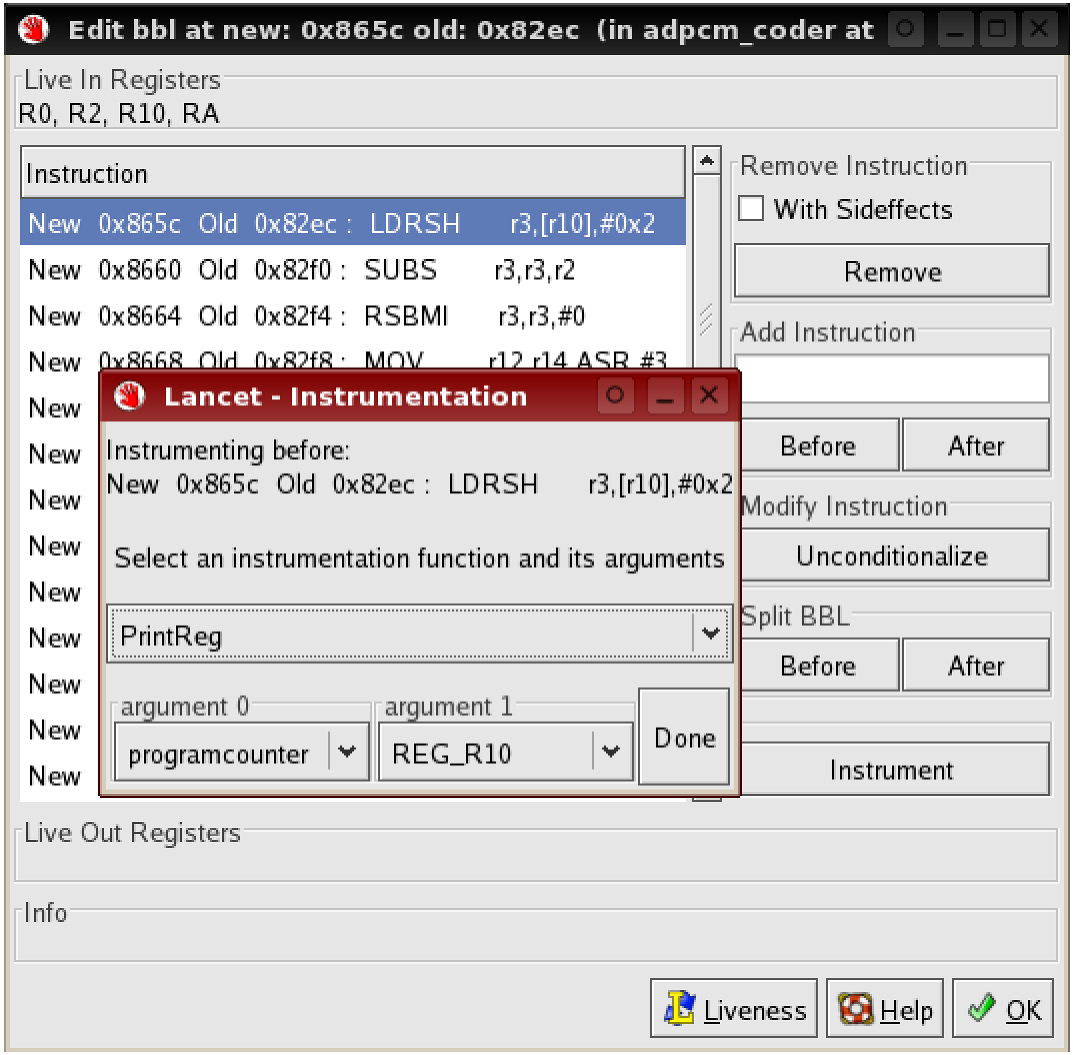

```
typedef t_string
  (*LancetBblInfoCallback)
   (t_bbl * bbl);
```

In your Lancet frontend e.g. `diablo_gui_main.c`:

```
t_string PrintCPInfoAfter (t_bbl * bbl)
{
   return
     StringIo ("CP information : %s",
       BblConstantsAfterTextual (bbl));
}
...

int main (int argc, char **argv)
{
   ...
   LancetInit(argc, argv);
   ...
   LancetRegisterBblInfoCallback (
     PrintCPInfoAfter);
   ...
   GuiLaunch (...);
   ...
}
```

Point-wise instrumentation



If instrumentation support is turned on, you can use Lancet to selectively add user-provided instrumentation routines.

In lancet_instrument_analyses_user.c:

```
void PrintReg(int address,
              int reg) {
   ...
}
```

In frontend e.g. diablo_gui_main.c:

```
void InstrumentInit()
{
  FitProtoNew
      ("PrintReg(REGV,REGV)");
}
```

From a Diablo users point of view...

- implementation and debugging aid for new analysis and optimizations
  - e.g. inspect the graphs to find out why an optimization was not applied
  - pop up the cfg containing a bbl when a corner case is encountered: GFATALBBL((t_bbl*, t_string message))
  - ...
- detection of new optimization opportunities, using profile information, point-wise instrumentation, visualization of analysis information