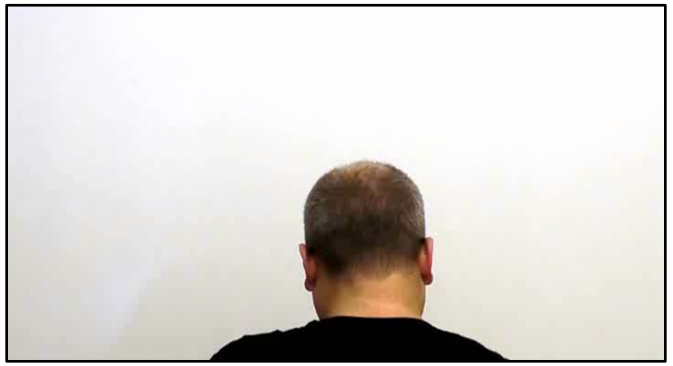
VERACODE

Lessons In Static Binary Analysis

Christien Rioux, Chief Scientist



Declaration of Vast Oversimplifications

- ✓ We take an Intel 32-bit specific view of the world for convenience. Nothing presented here is truly platform specific however.
- ✓ We take a C/C++ bias in these slides. Java is similar but usually simpler. C# is similar.
- ✓ The research to build the system described has taken over 5 years of effort to develop and 5 more to implement correctly.
- ✓ We've built the system to be retargetable, resourceable, and pluggable. We don't go into architecture specifically here, focusing specifically on design and algorithms.

Boring Stuff First: Yes, It's Patented

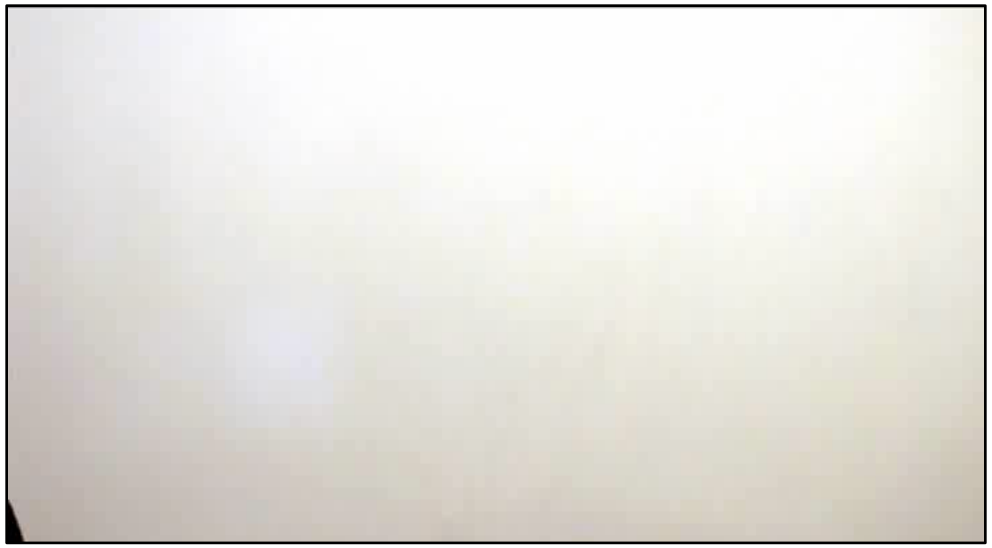
What I'm describing here is effectively covered by U.S. Patent #7051322 and #7752609.

Patented material is published by design, don't expect any 'trade secrets' in this discussion.

If we didn't patent it, it's publically available from other places, and there are references to those works.

Components of Static Binary Analysis

- ✓Binary Modeler
- ✓Intermediate Representation
- ✓Model Querying and Condition Searching System



VERACODE

Intermediate Representation

What is an Intermediate Representation?

An intermediate representation is a data structure that is transformable, and represents language and architectural elements used to build software.

Many compilers use multiple intermediate representations such as:

- Abstract Syntax Trees (ASTs)
- Register Transfer Languages (RTLs)
- Single Static Assignment Trees (SSAs)
- Compiler-specific IRs, such as LLVM IR, GCC Trees, MSVC internal IR
- Research IRs from various educational institutions, SPARK, SCORE, Pegasus, etc.
- Reverse-Engineering oriented IRs such as REIL

Designing a Good IR Is Really Important.

You may not want to pick up someone else's IR.

Building your own will help you ensure that what you have can represent everything you want it to represent.

If you choose to build your own, pay attention to every little detail, because it's going to matter down the line. Every little bit.

Lesson #1:

Most IRs are not built for transformation, or optimized for memory usage. Expect this to be custom work.

Desirable Characteristics for an IR

- **Attention to Size and Speed**
  - Don't over-optimize too soon, but expect to have to do this, so ensure your design is optimizable at some point once you have things working!
- **Debuggability And Readability**
  - You're going to be staring at this thing wonder what went wrong. Make sure you can export, search and read pieces of the IR manually.
- **Complete Language Support**
  - Every language element
  - Every architectural element
  - Type information
  - Debug information
  - Superset of all languages
- **Single Uniform IR**
  - Represent everything only one way
  - Represent low and high level elements in the same IR

Why a Single, Uniform IR?

Remember, you're transforming the IR as you go from low to high level representation. If your transformation process is required to 'discover' all of the possible places in the binary that are code and decode them, then you are going to have high level stuff sitting along side low level stuff, likely in the same procedure, adjacent to each other. If you have one IR, you don't have to 'revert' the high level stuff to incrementally decode the program.

Lesson #2:

Without a single uniform IR, you end up throwing away all of your work when you discover new code.

Hierarchical Scoped Model

The IR should look like a tree with internal linkage and a direct parent to child relationship. This allows language elements to contain other language elements, and ‘refer’ to other elements via pointer or ‘id’, minimizing duplication.

Low Level Language Elements

- **Registers/Flags**
- **Sequences**
- **Expressions**
  - Immediates
  - Register/Object References
  - Memory Dereferences
- **Statements**
  - ‘Call/Return’ semantics
  - Evaluations
- **Low-Level Types And Data Formats**
  - Spans
  - Ints
  - Floats
- **Idiomatic Operations**
  - Block copies
  - Indexing

High Level Language Elements

- Types
  - Classes
  - Prototypes
- Objects
  - Variables
  - Procedures
- Namespaces
  - Named
  - Anonymous
- Templates
  - More useful than you'd think!

Expressions and Statements

Statements are sequentially executed pieces of code. They perform expression evaluations and are the root of control flow in the system. They should perform a minimum of data flow, leaving much of that to the expressions:

EVAL, CALL, RETURN, TRAP, TRAPRET, THROW, LOOP, SWITCH, IFELSE, etc

Expressions are hierarchical and represent arithmetic and perform dataflow operations, minimizing the amount of control flow:

((a+b)+foo(3)) is an expression

Immediates, Operations, Object References, etc.

Expressions

1 8bit → int<8>

1 32bit → int<32>

\+ add "operation"

(1+1) add operation w/arguments

Namespaces And Sequences

Namespaces are used to keep collections of objects in an unordered fashion.

Sequences are used to keep collections of objects in an ordered fashion.

Namespaces can be anonymous or named. Sequences in theory could be named, but using LABEL statements makes moving control flow targets around a lot easier than reparenting and splitting sequences.

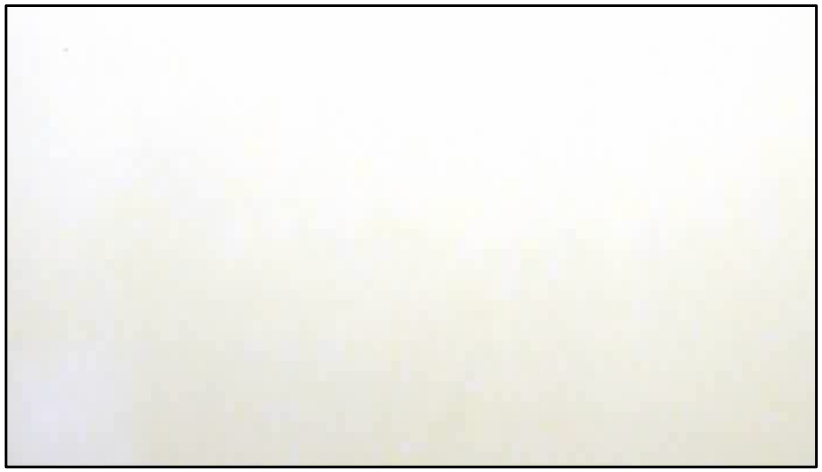
Sequences are useful for building basic block graphs.

Namespaces are useful for segmenting code by image, object file, source file, or whatever divisions are convenient.

Objects And Types

Objects are things like:

- Variables
- Procedures
- Registers



They have the property that they all have ‘types’

Also, these things can actually be referenced.

Types are things like:

- Classes, the ‘type’ of a Variable
- Prototypes, the ‘type’ of a procedure

Registers are generally considered ‘typeless’, but could be considered to be ‘spans’ of opaque bits. Either way works.

Variables

Variables are a high level construct because they can be free of low level representation.

The link between a variable and its ‘low level representation’ is called a ‘match’ expression. For a variable ‘int x’, the match expression could be *[esp-16]* or *EAX*.

Effectively, you should be able to drop in a variable’s match expression any place the variable is referenced and achieve the same meaning.

At high-level representation, we don’t care about the match expressions any more, more on that in a bit.

Procedures

A procedure is a function, lambda, block, etc… the base unit of *interprocedural control flow*, which is the mechanism by which 'CALL' statements and 'RETURN' statements operate.

Procedures have the property that they are usually named, and have a fixed or variable number of arguments and/or returns.

The arguments and returns to procedures are actually variables, whose location is shared with that of the calling procedure. This allows a 'call stack' to share variables in what is called a 'frame'.

Procedure frames are discussed later in the 'variablization' section where we describe where variables come from.

Templates

Templates are blocks of IR scopes that are made ‘generic’ and ‘instantiable’ with the usage of ‘template variables’. Instantiating a template duplicates the ‘base specialization’ and replaces the ‘template variable references’ with ‘instance expressions’. Got that?

Lesson #3:

If you use templates to represent things like integer types, you can have int<32> and int<16> be distinct objects in your hierarchy, allowing for very specific precision without complex ‘type selectors’.

What Should A Good IR Enable?

Dataflow Analysis
Controlflow Analysis
Expression And Statement Manipulation
Serialization In Parts and as a Whole
Debugging!!

Also:

Lesson #4

If you're going to make your system multithreaded, you should start with a thread-safe IR.

Dataflow Representation

A good IR enables a dataflow representation, such as a Def-Use graph or a Def-Kill graph. The IR should allow these graphs to be built easily and reference the variables and their definitions and uses.

Call graphs that show the interprocedural connections are also especially useful.

Lesson #5:

Do yourself a favor, and don't make a DU graph that incrementally updates itself. Batch your IR edits and make the DU graphs read only, it's way faster that way.

Control Flow Representation

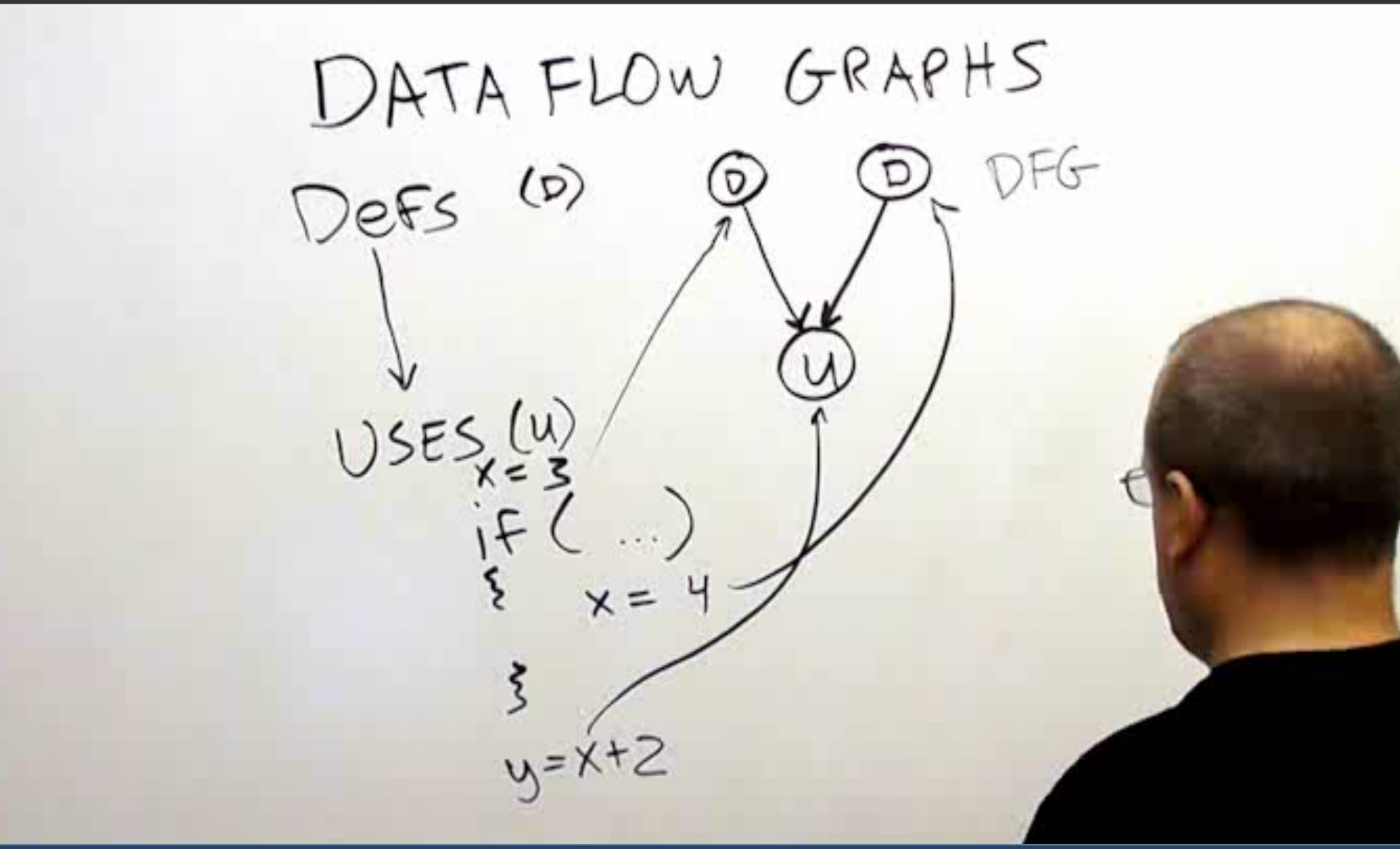
Control flow should be flexible. You're going to be editing this a lot. Make distinct interprocedural and intraprocedural control flow.

Low level control flow should be simple.

High level control flow is not going to be.

Lesson #6:

Build a 'cursor' object that works like a 'program counter'. Something that you can point into the IR and step forward and backward in execution order. This will make building your graphs way easier.


DATA FLOW GRAPHS
Defs (D)
D
D
DFG
U
USES (U)
x = 3
if (...)
{
x = 4
}
y = x + 2

Expression and Statement Manipulation

If you design your statements to be effectively linear-flow non-hierarchical entities, and use ‘sequences’ to provide hierarchy, statements are easier to manage. That’s pretty clear.

But also:

Lesson #7:

If you ensure that no expressions can be referenced outside of a parent-child relationship, you can manipulate them much easier without worrying about managing back-references.

Serialization

Make sure you can write out your IR to disk in both binary and text formats and load it back up into memory.

At various points you'll want to checkpoint the IR you've built, because this process can take a long time and you don't want to have to start all over if you run out of memory or something.

Also, debugging, environments, and just about every kind of reporting will want to use this at some point.

The Constraints Of The Real World

Vector math

Extreme use of templates

Optimization…

Lesson #8

You must implement every language element because you will encounter all of the things.

VERACODE

Binary Modeling

Compilers Make Assumptions

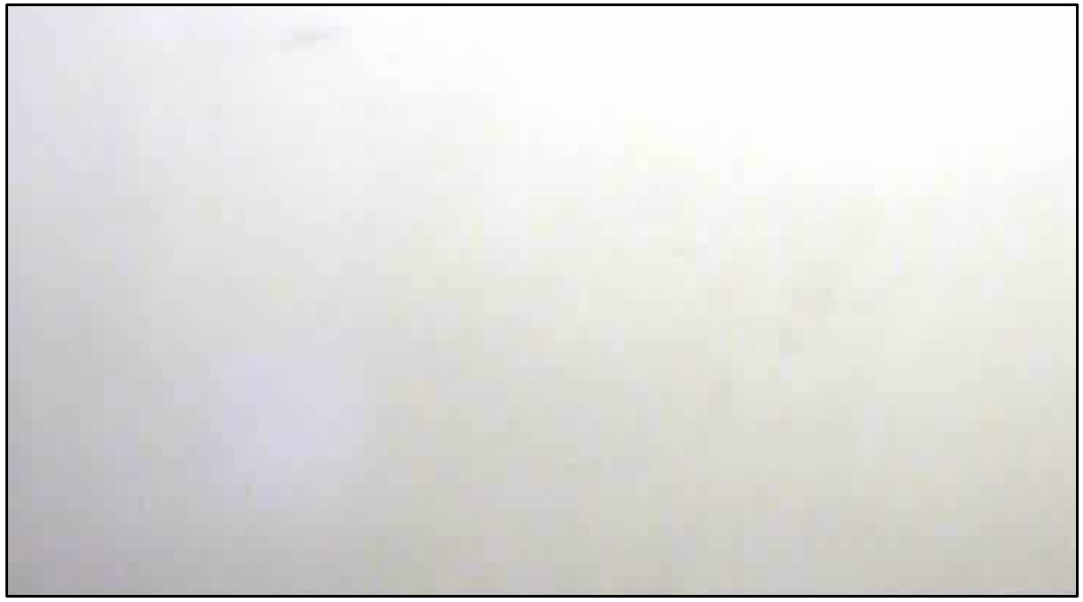
Lesson #9:

Because the modeler takes in the output of a compiler, not *arbitrary* input, you can use the assumptions the compiler makes to your advantage.

For example, on MSVC x86 you can assume the call mechanism uses the stack and the stack goes backward, and that external functions follow a set of known calling conventions, such as stdcall, fastcall, cdecl, etc, but that non-external functions can have arbitrary calling conventions including passing parameters in registers.

Inputs To The Modeler

- Executables
- Linked Libraries
- Environments
- Debug Symbols



If any of these are missing, the quality or readability of the output will be reduced. With no environments, for example, you will only have the names of operating system functions, not their parameters. With no debug symbols, you will have no line number information and original variable names.

Environments are not necessary on some platforms with extremely descriptive external linkage such as .NET and JAVA.

Describe: Environments

When performing the modeling, one must know when to stop. The output generated could include system libraries, or the kernel or other outlandish components that would normally be considered undesirable to model every single time.

Environments are a pairing of two things:

- External dependencies
- Type information for the edge of those dependencies

Environments require a 'type information compiler' of some sort to generate meaningful type information on the edge between the user code and the 'environment'.

Environment Compilation

Compiling environments for MSVC x86 C++ looks like this:

- Build a list of DLLs
- Find all of the header files that the compiler would use to allow you to link those things in your code
- Compile the header files into the IR format
- Pair the IR objects up with the DLL exports by signature
- Export the DLL IR images to an archive for later retrieval

This process is more complicated for lower level languages like C.

The Modeling Pipeline

- ✓Frontend Environment Loader
- ✓Dataflow Transformer
- ✓Dataflow Optimizer
- ✓Control Flow Transformer
- ✓Optional: Backend Source Generation

Platform-Specific Front End Loader

- Load all the images and all their dependencies and all dependent environment files
- Pull in any detected debug symbols and create IR objects from them to seed the modeling process
- Use entry points, export tables, RTTI information, ELF symbol tables, mangled names and whatever else you happen to have laying around to create even more IR objects and mark places to start instruction decoding.

The Veracode 'demangler' takes all the weird names in PE executables and creates objects:

```
?_Init@?$basic_streambuf@DU?$char_traits@D@std@@@std@@IAEXPAPAD0PAH001@Z
```

```
protected:
void __thiscall std::basic_streambuf<char,struct std::char_traits<char>
>::_Init(char * *,char * *,int *,char * *,char * *,int *))
```

Dataflow Transformer

- Import
- Variablization
- Propagation And Merging
- Code Discovery
- Call Conversion
- Reference Conversion

Import

During this phase, we iteratively decode instructions from places where we have determined the instruction pointer can reach.

Specifically, we'll start with entry points, and build basic blocks connecting all well known control flow into a basic block and procedure graph.

This will not find all of the code, since *indirect* control flow such as jumping to registers and memory dereferences can not be easily sussed out until later in the process.

That's okay. Just decode everything you can until you find you can't.

Variablization

Variablization is the process by which real variables are created to represent low level register and memory access.

We start by creating a whole variable for each def/use 'site' in the program. Sites are registers accesses, in whole or in part, and memory dereferences.

proc Foo {
EAX := ESP
EAX := EAX+16
EBX := *EAX….

Can you find all the sites?
How many variables get created at first?

Variablizing Stack Variables

The trick here is to measure all allocated variables that are on the stack with reference to the base of the stack frame. Specifically, ESP at the entry point of the procedure. The compiler lays out stack frames with regard to a particular frame 'base'. Don't rely on 'EBP' or some other register that may be optimized away.

EBP is set to ESP at the top of the procedure!

Just forward substitute and/or use range propagation to figure out what ESP offset is in your dereference, and use that to measure where the stack variables live in the frame, and what their match expression really is ☺

Propagation And Merging

The goal of variablization is to create as many variables as necessary to represent correct data flow, but no more than that if possible.

We start by creating many variable sites and then propagating definitions downward across the control flow graph, and where they meet overlapping match expressions on use sites we merge the two variables together.

This is similar to 'Single Static Assignment' form generation.

Lesson #10:

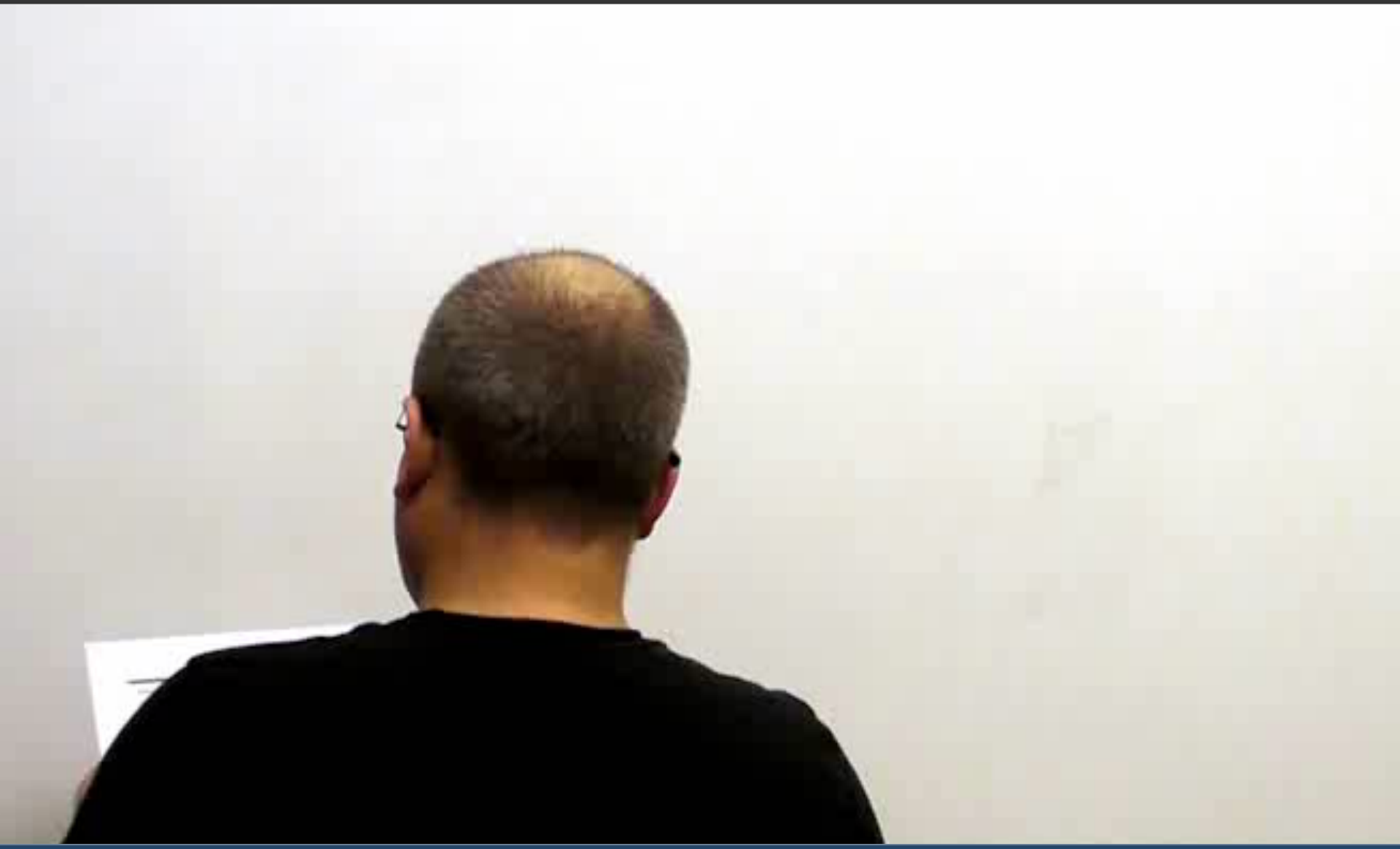
Variablization is like SSA form, except instead of generating *phi functions*, actually *merge* the variables together to form a single variable.

Interprocedural Propagation

The same kind of propagation done to link defs and uses can be used to propagate defs that reach the end of the procedure up to call sites, and uses that reach the top of the procedure up to call sites.

Across the call site, we translate the match expression using the call sites' values for things like frame bases such as ESP, such that the defs and uses are represented in the callers' contexts.

When those propagations merge with sites in the caller, they become *arguments* and *return values* for the procedures in the propagation chain.



Code Discovery

Code discovery is the process by which we take indirect jumps and calls and figure out where they could possibly go.

Indirect jumps are going to turn into switch statements most of the time. Finding their targets is an expression analysis and some table lookups, followed by simplification.

Indirect calls require type information at the call, either constructed or from debug symbols to determine how the calls go from source to destination, the calling convention.

Lesson #11:

The compiler didn't know where virtual function calls were going when it built the binary, and you don't need to know where the calls are going either, just HOW they go there.

Call Conversion

Call conversion takes all of the CALL statements and turns them into Operation expressions.

Also, it fixes the variables that are prototype arguments and returns and puts that prototype on discovered procedures.

Calls that go to many places have an expression that says where they go, and that can get handled by aggregation later in the process once we have more type information propagated around.

This effectively determines the calling convention for procedures where it is unknown.

Reference Conversion

When you have a complex expression, such as &A[100].b[2].c, it starts off life as a reference inside of 'A', such as *(&A+812)

And that starts off as *(EAX+824) perhaps.

Using a range propagator, we can put EAX in terms of the ESP at the top of the procedure and get something like this:

*((ESP+12)+812)

Determining that ESP+12 is the address of the variable A on the stack is the job of the reference conversion system. This is done after variablization not during it, since it doesn't affect how we find the variables in the first place.

Type Voting

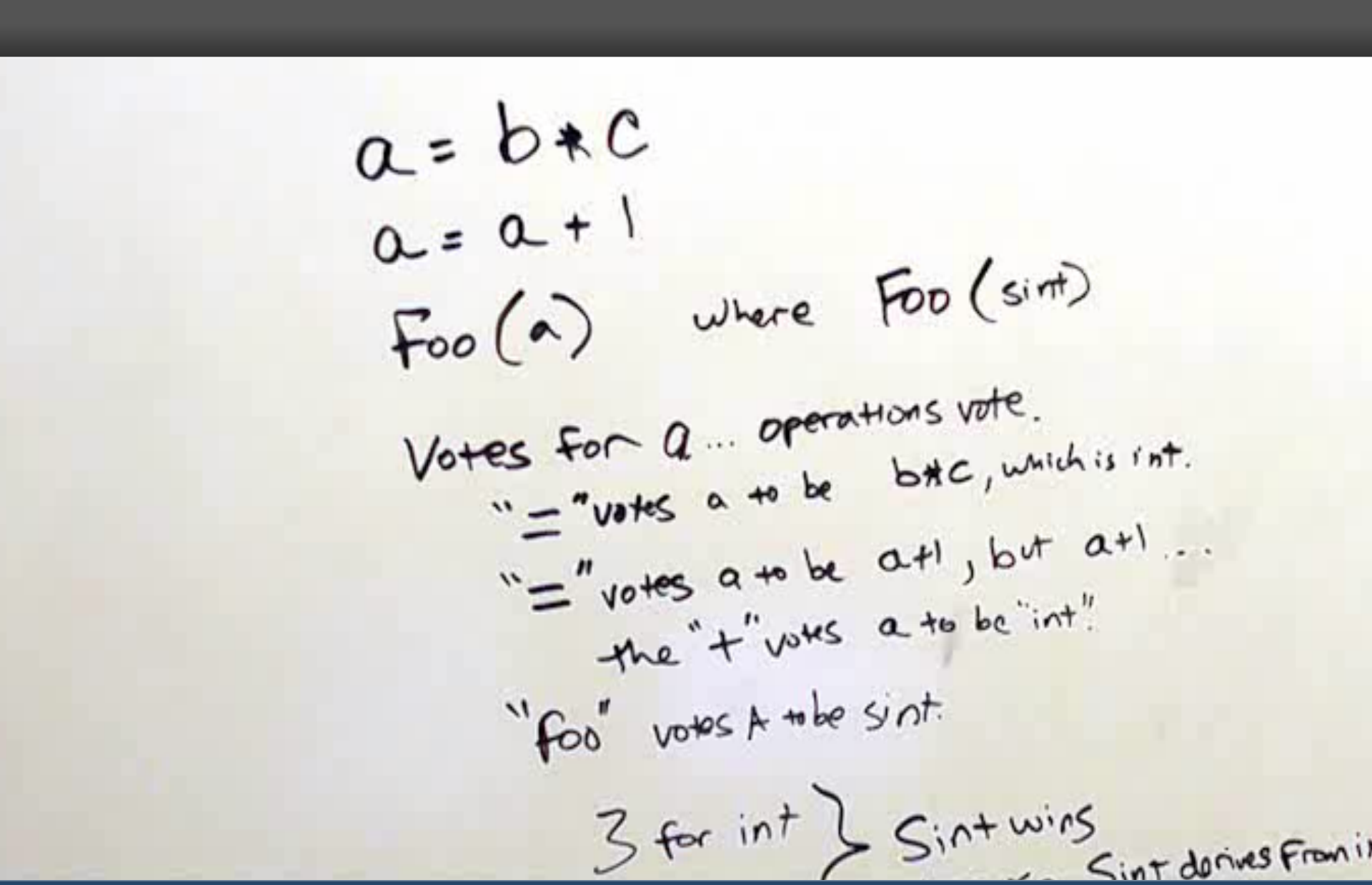
Propagation Propagation Propagation Propagation !

We take all of the type information that is loose in the system, and try to minimize the amount of casts that required to represent it.

For example, A=B+C can be represented with no casts at all if A is the same type as B and C. If A were something completely different, then we would have A=(int)(B+C), which would be suboptimal if we could change the type of A.

In this sense , casts do not perform extra operations such as bit extracts or sign extensions, they just change types from one thing of the same bit width to another.

Balancing this across the entire IR graph is an interesting problem ☺


a = b * c
a = a + 1
Foo(a) where Foo(sint)
Votes for a ... operations vote.
"=" votes a to be b*c, which is int.
"=" votes a to be a+1, but a+1 ...
the "+" votes a to be "int".
"foo" votes A to be sint.
3 for int } Sint wins
Sint derives From i

Dataflow Optimizations

- Aggregation
- Match Expression Removal
- Copy Constant Propagation
- Expression Forwarding
- Algebraic Simplification
- Dead Code Elimination
- Unused Variable Elimination
- Variable Merging
- Idiomatic Pattern Replacements

Aggregation

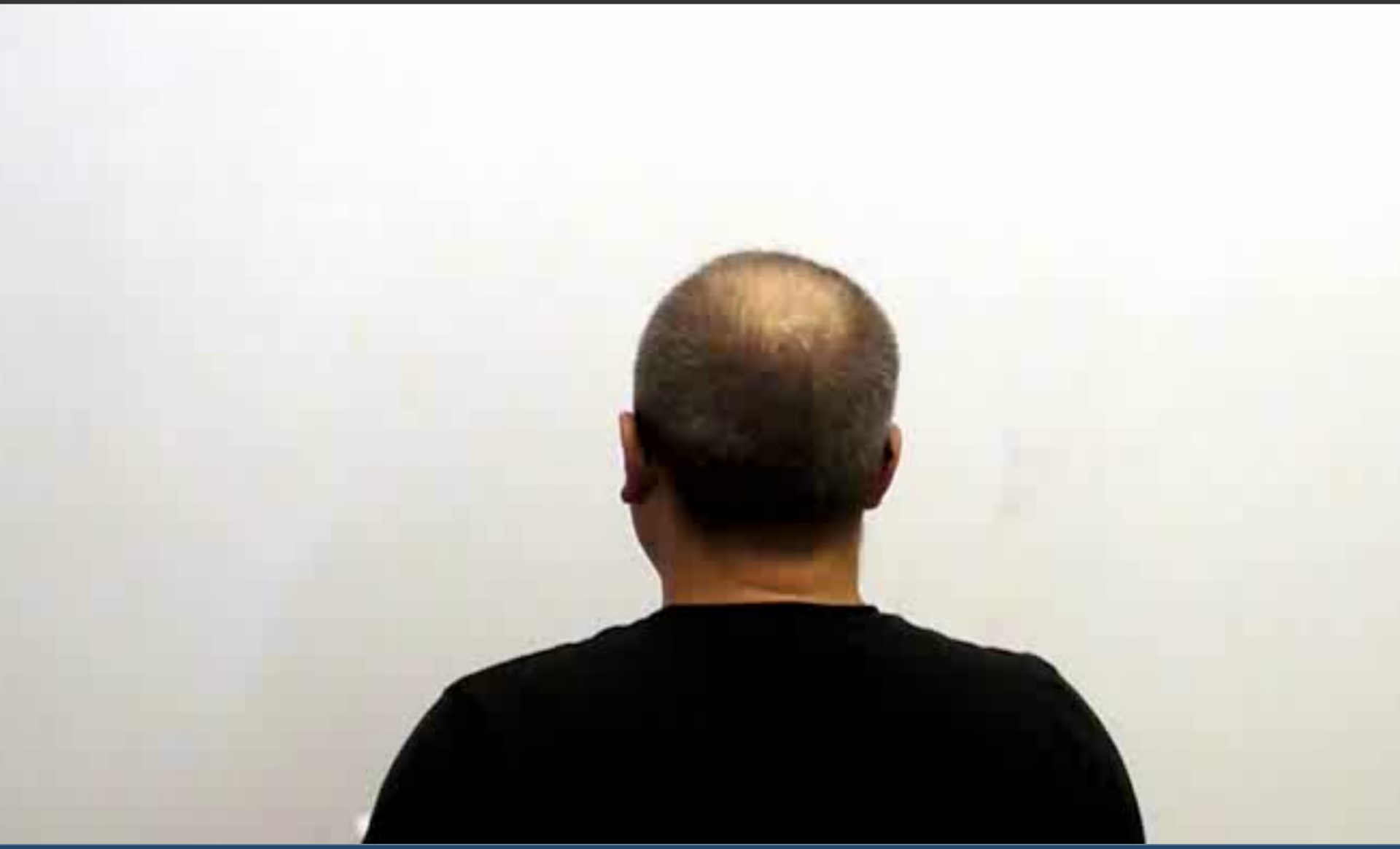
Aggregation takes low level dereference and reference expressions and turns them in to high level object references:

For example:

*(&foo+var*4+16) becomes foo[var].b

Class objects like ‘A.b’ and arrays ‘foo[var]’ must be considered in the structuring.

Open problem: How do you know when you’re referencing a variable or the first member of its class, ie &(A.a) versus &A ?



Match Expression Removal

Match expressions bind variables to their low-level representations.

To truly become high-level, we eliminate match expressions once we're fully variablized and merged because they cease to have meaning for the transformation.

This also allows us to perform further merging since the requirement of keeping specific low-level memory or registers allocated is no longer necessary.

Copy Propagation and Expression Forwarding

From Wikipedia: http://en.wikipedia.org/wiki/Copy_propagation

In compiler theory, copy propagation is the process of replacing the occurrences of targets of direct assignments with their values[1]. A direct assignment is an instruction of the form x = y, which simply assigns the value of y to x.

From the following code:

y = x

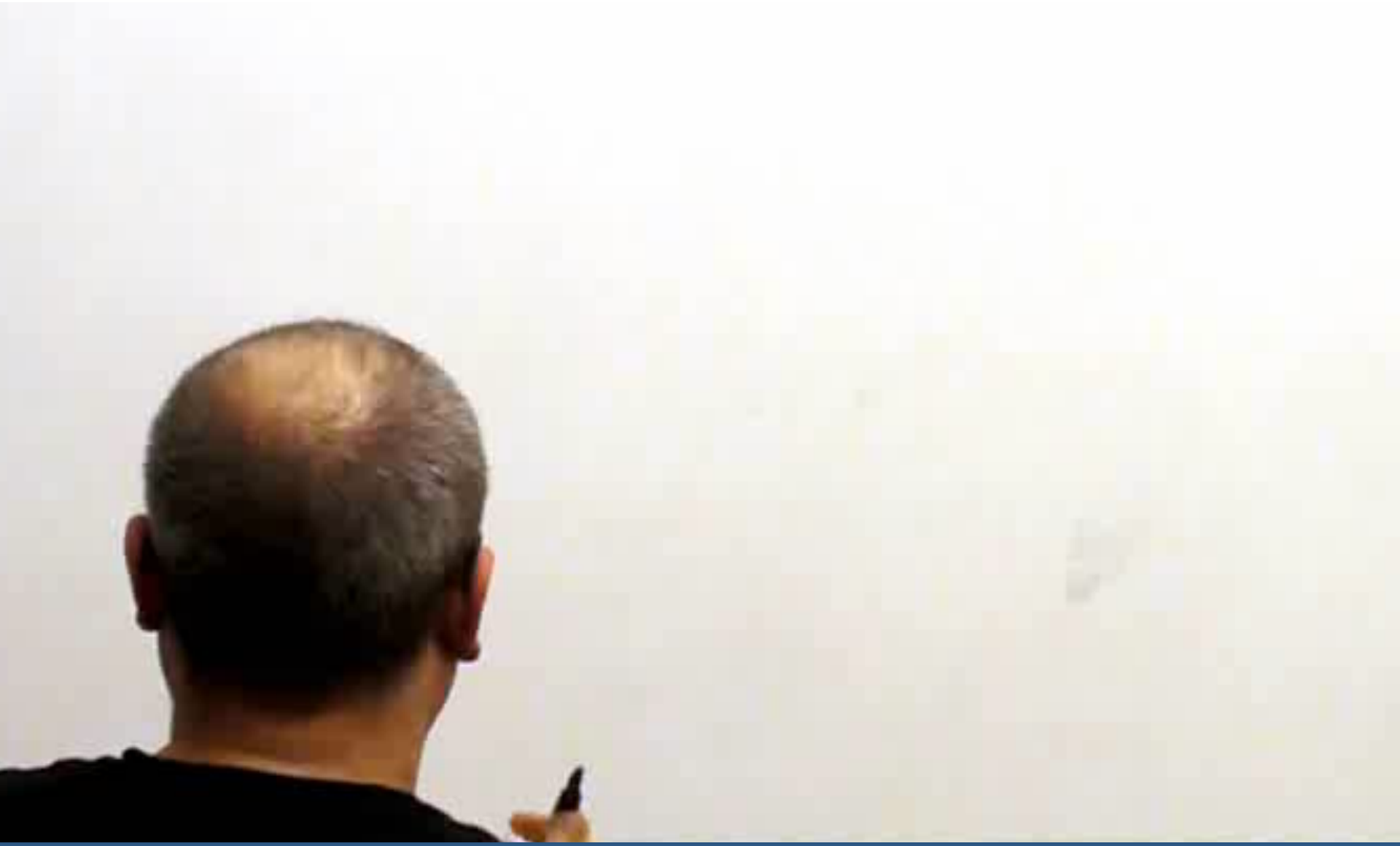
z = 3 + y

Copy propagation would yield:

z = 3 + x

Copy propagation often makes use of reaching definitions, use-def chains and def-use chains when computing which occurrences of the target may be safely replaced. If all upwards exposed uses of the target may be safely modified, the assignment operation may be eliminated.

Copy propagation is a useful "clean up" optimization frequently used after other optimizations have already been run. Some optimizations -- such as elimination of common sub expressions[1] -- *require* that copy propagation be run afterwards in order to achieve an increase in efficiency.



Algebraic Simplification

Forward substitution can lead to many complicated expressions like: (((((A+1)+1)+1)+1…, which could easily be A+4 instead.

More complicated transformations such as associativity and commutivity rules can be applied to find a better and *canonical* form for a simplification

Lesson 12:

Find a canonical representation when you simplify. If you get A+1 or 1+A, pick a sort for your output so all simplifications that can end up the same end up the same.

Variable Merging

Variables are merged in the variablizer, but some variable merging can not be done at that time unless the match expressions are released.

If you were to do the following:

A := B

C := A

And there were no references to A anywhere else, or B or C, they could all become D:= D and then be eliminated. This clears up a lot of intermediate variables.

You'll want to keep tabs on which variables are 'debug symbol' variables in the event that you wish to keep around things to make the output look as close as possible to the original input.

Idiomatic Pattern Replacements

- 64 bit arithmetic done as 32 bit chunks
- Unrolled memory block copies
- Inlined functions such as strcpy,memcpy, strlen,etc
- Architecture specific patterns
  - Complex comparisons
  - Floating point comparisons
  - Divide by power of two
- Language specific patterns
  - New/delete
  - Alloca
  - Static casts through virtual bases
  - Exception throw
  - Etc

Controlflow Transformer

- Interval Analysis
- 2-Way Conditionals
- N-Way Conditionals
- Unconditional Jumps
  - Break (‘Exit’)
  - Continue (‘Cycle’)
  - Goto
- Longjumps
- Exception Handling
  - Goto from catch
  - Re-throw from catch
  - Nested exceptions

Interval Analysis

From Cifuentes, 1994:

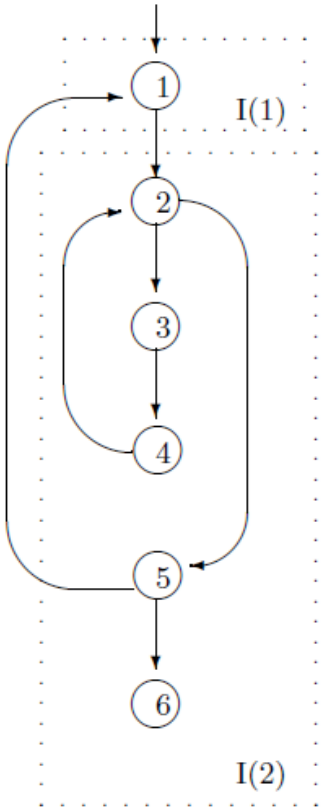
Interval:

Given a node h, an interval I(h) is the maximal, single-entry subgraph in which h is the only entry node and in which all closed paths contain h. The unique interval node h is called the interval head or simply the header node.

For a great reference on building a CFT, look at Cifuentes' doctoral thesis:

http://www.phatcode.net/res/228/files/decompilation_thesis.pdf

What does an interval look like?



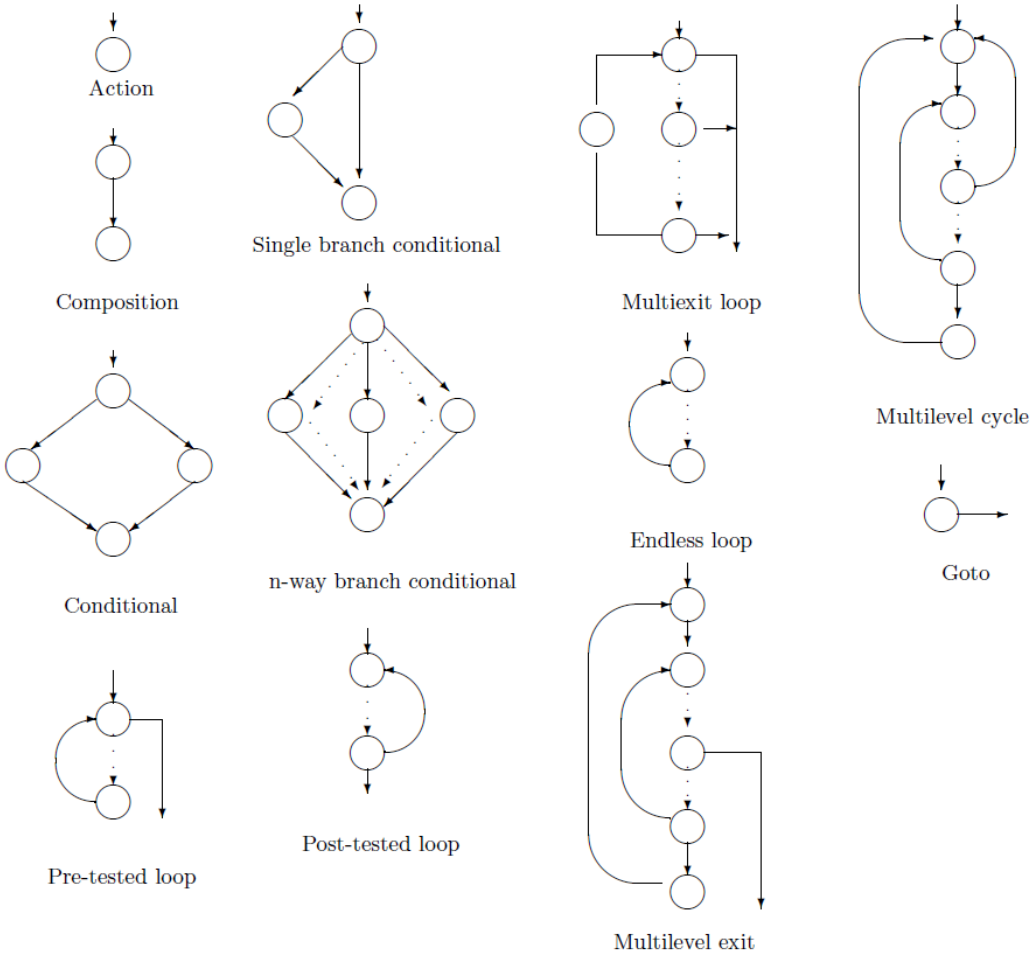
G

$I(1) = \{1\}$

$I(2) = \{2,3,4,5,6\}$

From Cifuentes, 1994

Types of Reducible Intervals



From Cifuentes, 1994

This gets very complicated.

Again, I recommend looking at the Cifuentes thesis on this subject matter if you intend to get into interval graph theory.

It's hairy stuff with a lot of edge cases and maddening to debug.

http://www.phatcode.net/res/228/files/decompilation_thesis.pdf

Exception Handling

Stack based exception handling and table based exception handling both require special compiler specific handling.

Lesson #14:

It's easiest to do most of your exception handling at the dataflow stage, not the control flow stage! You can do this right after variablization most of the time.