Announcements!

- Please share your slides! These talks are pretty unique.
  - Email me (sb54@illinois.edu) or slack me
- Also contact me if you want me to add a link to your name on the meetup webpage
- Speakers to invite
- List of speakers

A bunch of random thoughts on

Compiler IRs

Overview

- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
  - Trees
    - High-level transformations
    - Turn them into DAGs
  - SSA
    - Where is the value in a φ used?
  - Multi-Level IRs (WHIRL)
    - Trade off?
- Undefined Behavior and poison values
- Target and source independence in IRs

Overview

- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
  - Trees
    - High-level transformations
    - Turn them into DAGs
  - SSA
    - Where is the value in a φ used?
  - Multi-Level IRs (WHIRL)
    - Trade off?
- Undefined Behavior and Poison
- Target and source independence in IRs

- There is simply no metric to evaluate IRs
- It’s all empirical

- There is simply no metric to evaluate IRs
- It’s all empirical
- People’s intuitions may be wrong

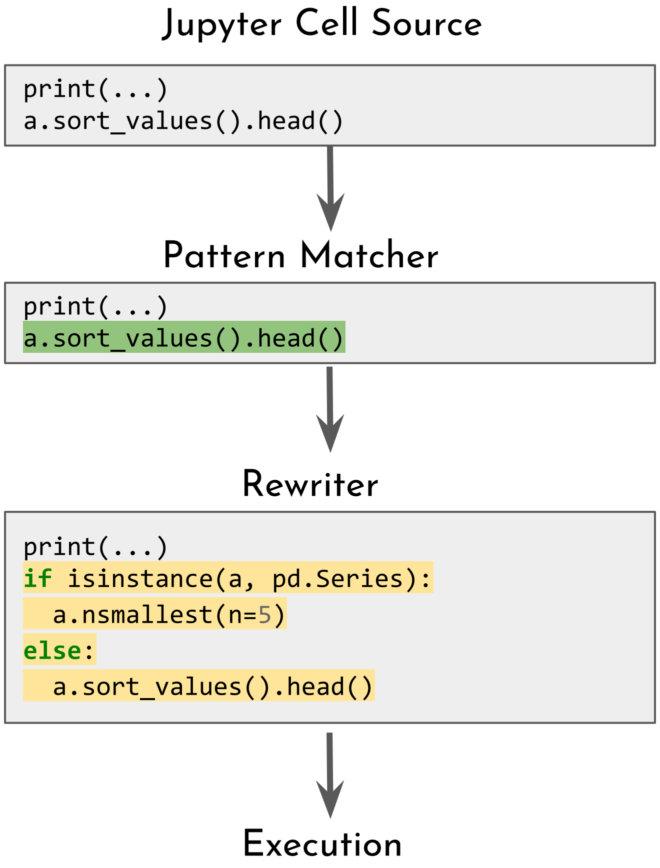
Overview

- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
  - Trees
    - High-level transformations
    - Turn them into DAGs
  - SSA
    - Where is the value in a φ used?
  - Multi-Level IRs (WHIRL)
    - Trade off?
- Undefined Behavior and Poison
- Target and source independence in IRs

Overview

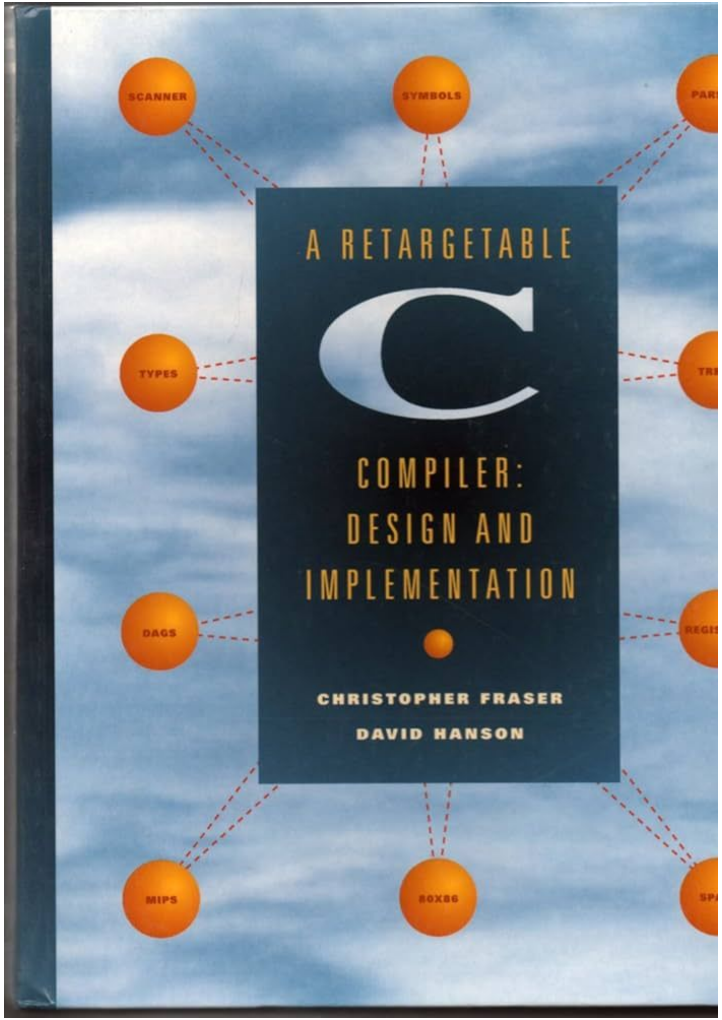
- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
  - Trees
    - High-level transformations
    - Turn them into DAGs
  - SSA
    - Where is the value in a φ used?
  - Multi-Level IRs (WHIRL)
    - Trade off?
- Undefined Behavior and Poison
- Target and source independence in IRs

High-Level Transformations in Dias



Overview

- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
    - Trees
        - High-level transformations
        - Turn them into DAGs
    - SSA
        - Where is the value in a φ used?
    - Multi-Level IRs (WHIRL)
        - Trade off?
- Undefined Behavior and Poison
- Target and source independence in IRs



- Whole ANSI C compiler explained in a book
- Badly written
- Still very educational

LCC DAGs

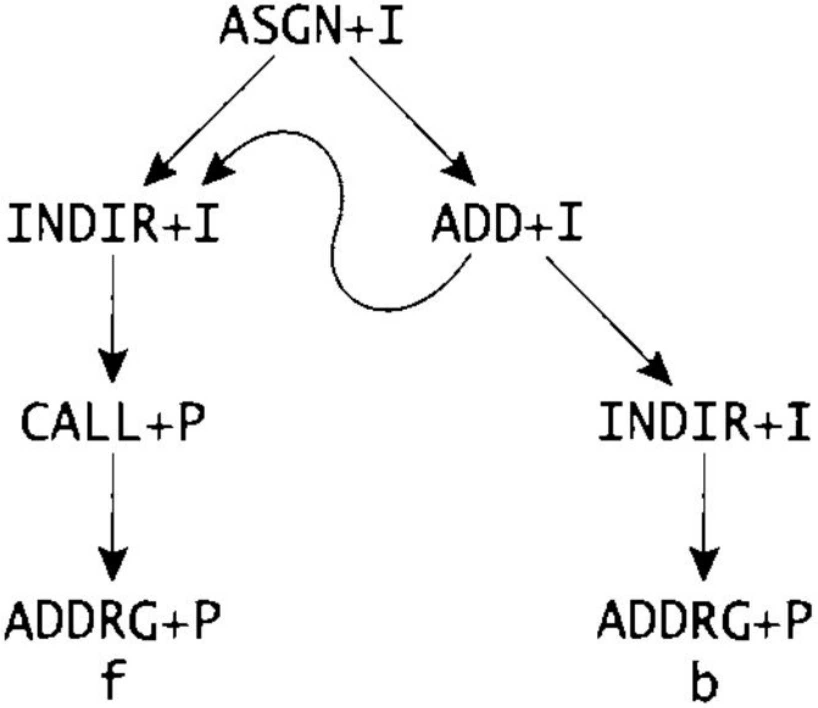


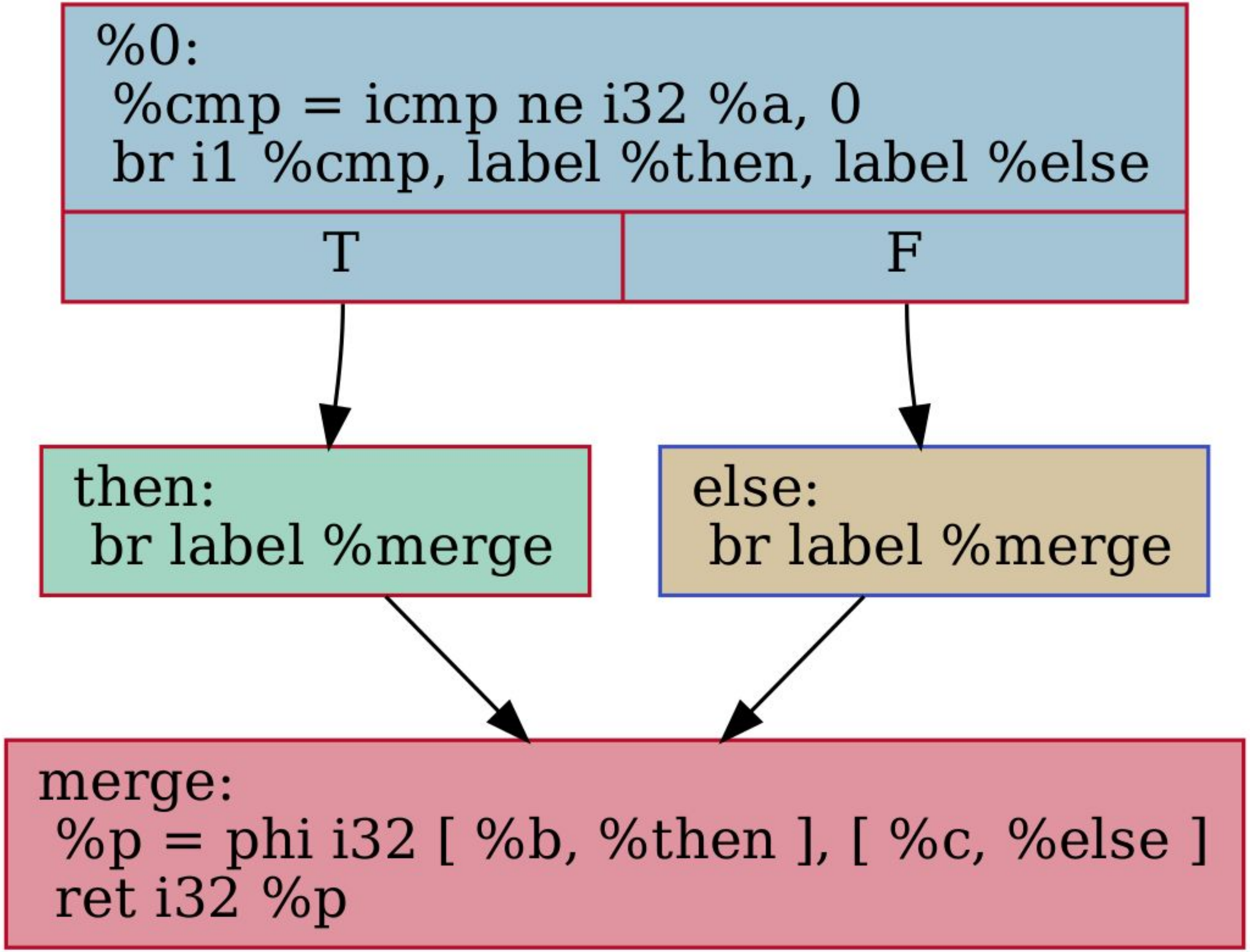
FIGURE 8.2 Tree for *f() += b.

Clang AST

```
`-CompoundAssignOperator <line:4:5, col:15> 'int' lvalue '+=' ComputeLHSTy='int' ComputeResultTy='int'
  |-UnaryOperator <col:5, col:10> 'int' lvalue prefix '*' cannot overflow
  | `-CallExpr <col:6, col:10> 'int *'
  |   `-ImplicitCastExpr <col:6> 'int *(*)()' <FunctionToPointerDecay>
  |     `-DeclRefExpr <col:6> 'int *()' lvalue Function 0xc448be8 'log' 'int *()'
 `-IntegerLiteral <col:15> 'int' 3
```

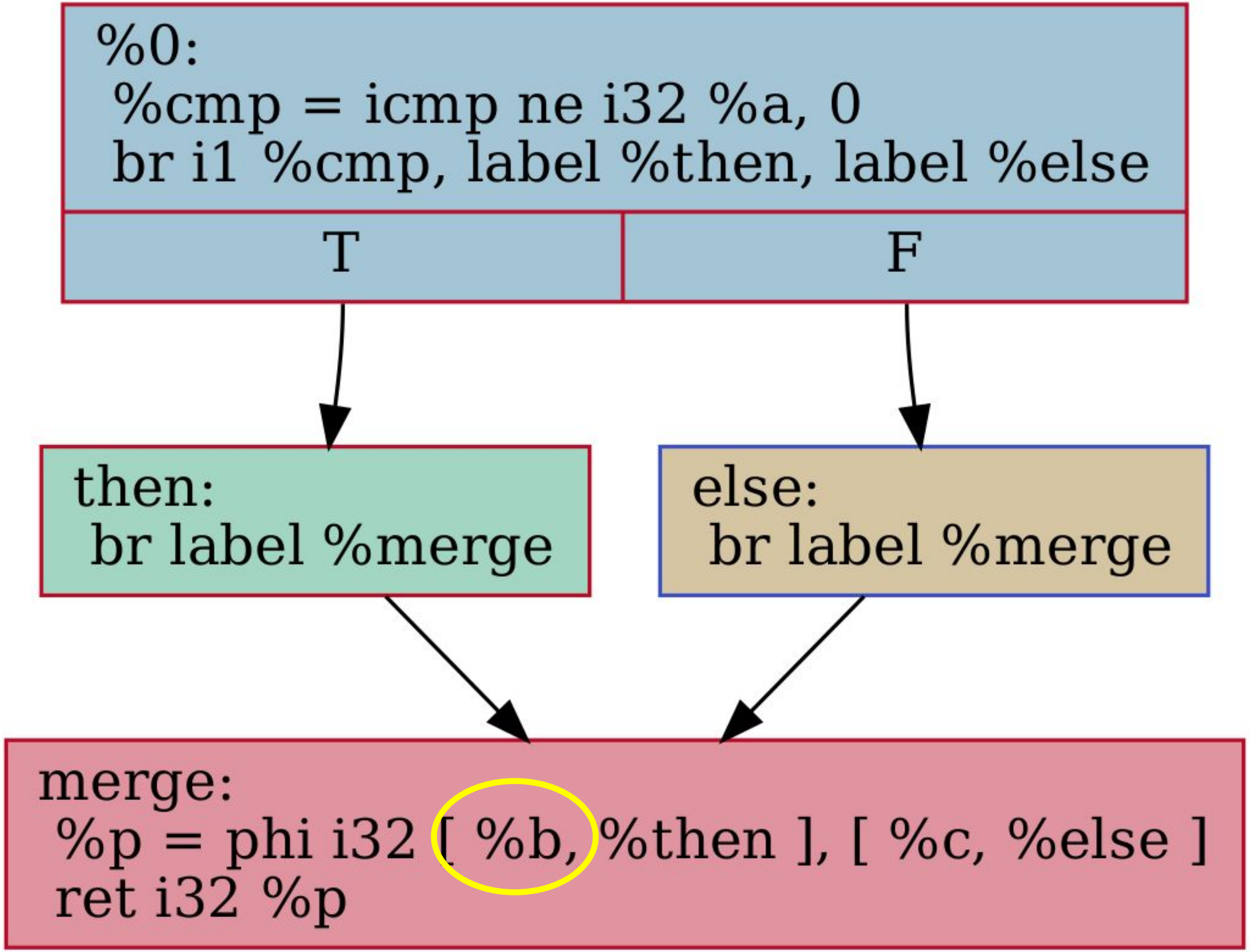
Overview

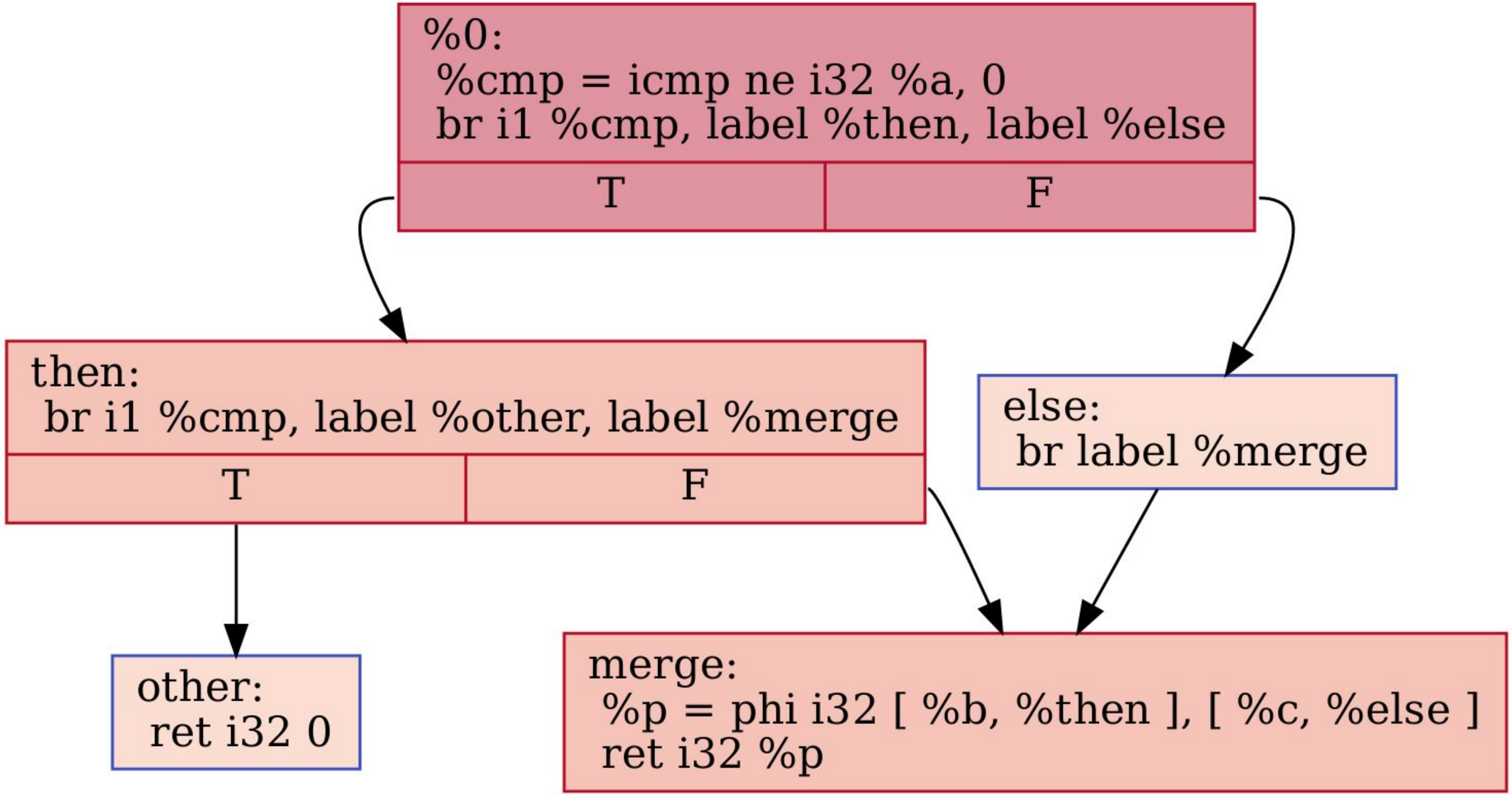
- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
  - Trees
    - High-level transformations
    - Turn them into DAGs
  - SSA
    - Where is the value in a φ used?
  - Multi-Level IRs (WHIRL)
    - Trade off?
- Undefined Behavior and Poison
- Target and source independence in IRs



%0:
%cmp = icmp ne i32 %a, 0
br i1 %cmp, label %then, label %else
T
F
then:
br label %merge
else:
br label %merge
merge:
%p = phi i32 [%b, %then], [%c, %else]
ret i32 %p

What is the usage point of %b ?





CFG for 'foo' function

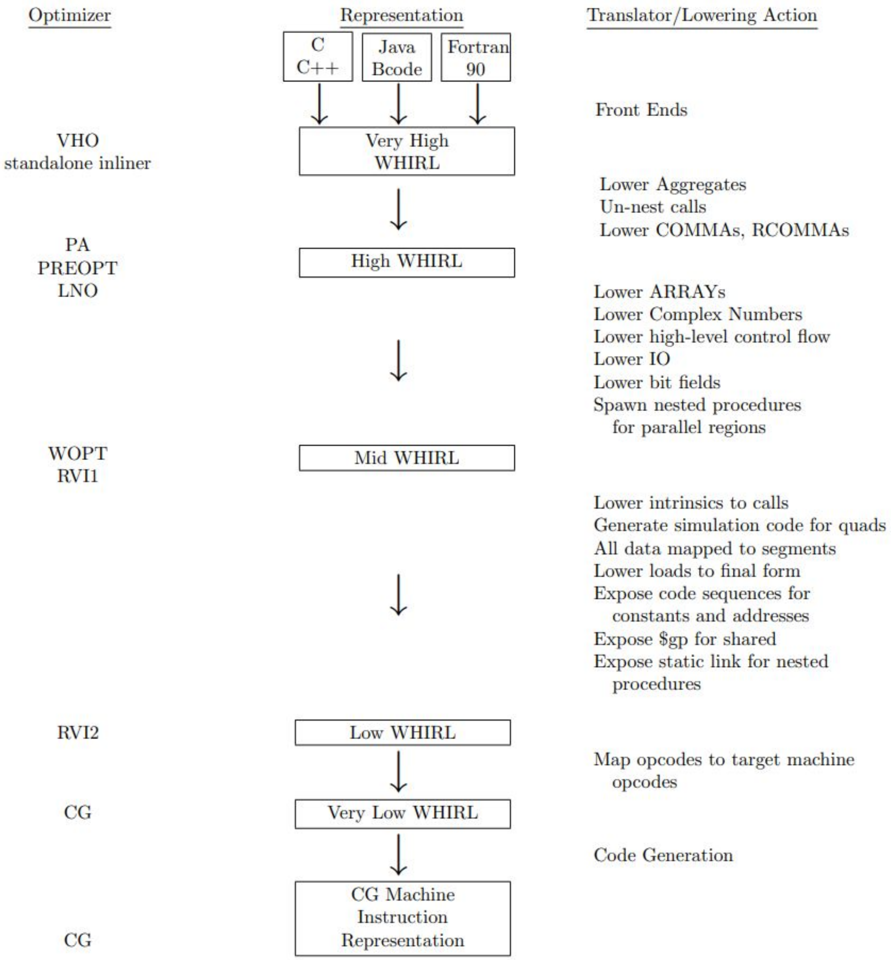
The Big Idea

`φ’s turn control flow into data flow`

Overview

- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
  - Trees
    - High-level transformations
    - Turn them into DAGs
  - SSA
    - Where is the value in a φ used?
  - Multi-Level IRs (WHIRL)
    - Trade off?
- Undefined Behavior and Poison
- Target and source independence in IRs

Link



Optimizer
Representation
Translator/Lowering Action
C
C++
Java
Bcode
Fortran
90
Front Ends
VHO
standalone inliner
Very High
WHIRL
Lower Aggregates
Un-nest calls
Lower COMMAs, RCOMMAs
PA
PREOPT
LNO
High WHIRL
Lower ARRAYs
Lower Complex Numbers
Lower high-level control flow
Lower IO
Lower bit fields
Spawn nested procedures
for parallel regions
WOPT
RVI1
Mid WHIRL
Lower intrinsics to calls
Generate simulation code for quads
All data mapped to segments
Lower loads to final form
Expose code sequences for
constants and addresses
Expose $gp for shared
Expose static link for nested
procedures
RVI2
Low WHIRL
Map opcodes to target machine
opcodes
CG
Very Low WHIRL
Code Generation
CG
CG Machine
Instruction
Representation

Where is the catch?

- Cognitive loaded
- Optimizations that cross levels
  - Vectorization in LLVM

Overview

- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
  - Trees
    - High-level transformations
    - Turn them into DAGs
  - SSA
    - Where is the value in a φ used?
  - Multi-Level IRs (WHIRL)
    - Trade off?
- Undefined Behavior and Poison
- Target and source independence in IRs

Undefined Behavior

Undefined Behavior is just a *design* choice!

Undefined Behavior

```
if (a + c < b + c)
```

Correct ?

```
if (a < b)
```

Undefined Behavior

```
if (INT_MAX + 1 < 0 + 1)
```

Correct ?

```
if (INT_MAX < 0)
```

Undefined Behavior

```
if (INT_MIN < 1)
```

TRUE

INCORRECT ?

```
if (INT_MAX < 0)
```

FALSE

Undefined Behavior

```
if (INT_MIN < 1)
```

TRUE

~~INCORRECT ?~~ NO!

Signed Overflow is UB!

```
if (INT_MAX < 0)
```

FALSE

Undefined Behavior *Enabling* Transformations

Assume that the program does <u>not</u> exhibit Undefined Behavior!

Inhibiting Undefined Behavior

```
int b, c;
…
for (int i = 0; i < N; ++i) {
   *p = b + c;
}
```

Loop-invariant!

Inhibiting Undefined Behavior

```
int b, c;
...
for (int i = 0; i < N; ++i) {
  *p = b + c;
}
```

Can we hoist?

Inhibiting Undefined Behavior

```
int b, c;
…
for (int i = 0; i < N; ++i) {
  *p = b + c;
}
```

N <= 0 ?

Inhibiting Undefined Behavior

```
int b, c;
…
for (int i = 0; i < N; ++i) {
   *p = b + c;
}
```

~~Can we hoist?~~ NO!

`N <= 0` ?

Undefined Behavior *Disabling* Transformations

The compiler can’t make the program more undefined!

Workaround ?

But it can make it more defined...

Define Signed Overflow ?

Define signed overflow as

2’s complement

Problems ?

Problems ?

The first example is disabled

Problems ?

```
for (int i = 0; i < N; ++i) {
  p[i] = …;
}
```

Iteration count ?

Problems ?

```
for (int i = 0; i < N; ++i) {
   p[i] = …;
}
```

`N == INT_MAX` ?

Iteration count ?

Problems ?

i32

```
for (int i = 0; i < N; ++i) {
  p[i] = …;
}
```

In 64-bit machine, `sext` in every iteration

Problems ?

```
for (int i = 0; i < N; ++i) {
  p[i] = …;
}
```

Widen to `i64` ?

Problems ?

Other peephole optimizations:

- `X + 1 > X` → `true`
- `X*2/2` → `X`
- `...`

Define Signed Overflow ?

☠

Define signed overflow as poison

Poison

most math ops

Poison either poisons or causes immediate Undefined Behavior

- `load`, `store`
- `sdiv`, `udiv`
- `call`, `invoke`
- ...

Inhibiting Undefined Behavior

```
int b, c;
...
for (int i = 0; i < N; ++i) {
   *p = b + c;
}
```

Can we hoist?

Let’s do it!

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
  *p = tmp;
}
// Assume `tmp` is never used again
```

Case 1

Does not overflow

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
  *p = tmp;
}
// Assume `tmp` is never used again
```

Case 1

Does not overflow

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
   *p = tmp;
}
// Assume `tmp` is never used again
```

We don't care

Case 2a

Does overflow

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
  *p = tmp;
}
// Assume `tmp` is never used again
```

Case 2a

Does overflow

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
   *p = tmp;
}
// Assume `tmp` is never used again
```

N <= 0

Case 2a

Does overflow

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
  *p = tmp;
}
// Assume `tmp` is never used again
```

N <= 0

We never get in

Case 2a

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
  *p = tmp;
}
// Assume `tmp` is never used again
```

Does overflow

N <= 0

We never get in

We never use tmp

Case 2b

Does overflow

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
  *p = tmp;
}
// Assume `tmp` is never used again
```

Case 2b

Does overflow

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
  *p = tmp;
}
// Assume `tmp` is never used again
```

N > 0

Case 2b

Does overflow

```
int b, c;
…
int tmp = b + c;
for (int i = 0; i < N; ++i) {
  *p = tmp;
}
// Assume `tmp` is never used again
```

N > 0

Do we care ?

Note

```
int b, c;
...
for (int i = 0; i < N; ++i) {
   *p = b + c;
}
```

Can we hoist?

Bonus!

Assume a target P:

- **Signed addition:** `padd`

Bonus!

Assume a target P:

- Signed addition: `padd`
- Explodes on SW

Codegen of res = add <nsw> a, b

```
res = padd a, b
```

CORRECT ?

Codegen of res = add <nsw> a, b

```
res = padd a, b
```

~~CORRECT ?~~

Codegen of `res = add <nsw> a, b`

```
if (a + b overflows) {
  res = <undefined value>
} else {
  res = padd a, b
}
```

Codegen of `res = add a, b`

```
if (a + b overflows) {
  res = <undefined value>
} else {
  res = padd a, b
}
```

No <nsw>!

Codegen of `res = add a, b`

```
if (a + b overflows) {
  res = <undefined value>
} else {
  res = padd a, b
}
```

CORRECT ?

Codegen of `res = add a, b`

```
if (a + b overflows) {
  res = <undefined value>
} else {
  res = padd a, b
}
```

~~CORRECT~~ ?

Codegen of `res = add a, b`

```
if (a + b overflows) {
  res = <Actual 2's complement result>
} else {
  res = padd a, b;
}
```

Codegen of `res = add a, b`

```
if (a + b overflows) {
  res = <Actual 2's complement result>
} else {
  res = padd a, b;
}
```

Must do it without `padd`

Adding Definedness

Conflicts Between Optimizations

How do we define branch-on-poison ?

```
if (poison) {
   ...
} else {
   ...
}
```

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

Loop-Unswitching

```
while (foo) {
   if (bar) {
      <body 1>
   } else {
      <body 2>
   }
}
```

Loop-invariant

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

Wrap around

Loop-invariant

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

```
if (bar) {
  while (foo) {
    <body 1>
  }
} else {
  while (foo) {
    <body 2>
  }
}
```

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

```
if (bar) {
  while (foo) {
    <body 1>
  }
} else {
  while (foo) {
    <body 2>
  }
}
```

What if `bar` is poison ...

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

…foo is **false** upon entering …

```
if (bar) {
  while (foo) {
    <body 1>
  }
} else {
  while (foo) {
    <body 2>
  }
}
```

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

…`foo` **is false upon entering …**

```
if (bar) {
  while (foo) {
    <body 1>
  }
} else {
  while (foo) {
    <body 2>
  }
}
```

… and branch-on-poison is UB ?

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

We never reach that!

No UB

```
if (bar) {
  while (foo) {
    <body 1>
  }
} else {
  while (foo) {
    <body 2>
  }
}
```

UB!

Case 1: Define it Non-Deterministically

```
if (poison) {
   ...
} else {
   ...
}
```

Non-deterministic choice

Case 1: Define it Non-Deterministically

```
if (poison) {
   ...
} else {
   ...
}
```

Non-deterministic choice

i.e. Assume we take both paths

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

We never reach that!

No UB

```
if (bar) {
  while (foo) {
    <body 1>
  }
} else {
  while (foo) {
    <body 2>
  }
}
```

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

No UB

```
if (bar) {
  while (foo) {
    <body 1>
  }
} else {
  while (foo) {
    <body 2>
  }
}
```

Non-deterministic choice

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

No UB

```
if (bar) {
  while (foo) {
    <body 1>
  }
} else {
  while (foo) {
    <body 2>
  }
}
```

Both are dead!

Loop-Unswitching

```
while (foo) {
  if (bar) {
    <body 1>
  } else {
    <body 2>
  }
}
```

No UB

```
if (bar) {
  while (foo) {
    <body 1>
  }
} else {
  while (foo) {
    <body 2>
  }
}
```

No UB

Global Value Numbering (GVN)

```
int foo = a + b;
if (foo == bar) {
  tar = a + b;
  *p = tar;
}
```

Global Value Numbering (GVN)

```
int foo = a + b;
if (foo == bar) {
  tar = a + b;
  *p = tar;
}
```

foo is now the same as bar

Global Value Numbering (GVN)

```
int foo = a + b;
if (foo == bar) {
  tar = a + b;
  *p = tar;
}
```

`tar` **is the same as** `foo`

`foo` **is now the same as** `bar`

GVN could potentially replace `tar` **with** `bar`

Global Value Numbering (GVN)

```
int foo = a + b;
if (foo == bar) {
  tar = a + b;
  *p = tar;
}
```

```
int foo = a + b;
if (foo == bar) {
  *p = bar;
}
```

Global Value Numbering (GVN)

```c
int foo = a + b;
if (foo == bar) {
   tar = a + b;
   *p = tar;
}
```

```c
int foo = a + b;
if (foo == bar) {
   *p = bar;
}
```

What if `bar` is poison ?

Global Value Numbering (GVN)

```
int foo = a + b;
if (foo == bar) {
  tar = a + b;
  *p = tar;
}
```

```
int foo = a + b;
if (foo == bar) {
  *p = bar;
}
```

It poisons ==

Global Value Numbering (GVN)

```c
int foo = a + b;
if (foo == bar) {
   tar = a + b;
   *p = tar;
}
```

```c
int foo = a + b;
if (foo == bar) {
   *p = bar;
}
```

Branch-on-poison

Global Value Numbering (GVN)

```
int foo = a + b;
if (foo == bar) {
  tar = a + b;
  *p = tar;
}
```

Non-deterministic choice

```
int foo = a + b;
if (foo == bar) {
  *p = bar;
}
```

Global Value Numbering (GVN)

```
int foo = a + b;
if (foo == bar) {
  tar = a + b;
  *p = tar;
}
```

```
int foo = a + b;
if (foo == bar) {
  *p = bar;
}
```

No UB

Global Value Numbering (GVN)

```
int foo = a + b;
if (foo == bar) {
  tar = a + b;
  *p = tar;
}
```

```
int foo = a + b;
if (foo == bar) {
  *p = bar;
}
```

No UB

Non-deterministic choice

Global Value Numbering (GVN)

```c
int foo = a + b;
if (foo == bar) {
  tar = a + b;
  *p = tar;
}
```

```c
int foo = a + b;
if (foo == bar) {
  *p = bar;
}
```

No UB

UB!

Overview

- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
    - Trees
        - High-level transformations
        - Turn them into DAGs
    - SSA
        - Where is the value in a φ used?
    - Multi-Level IRs (WHIRL)
        - Trade off?
- Undefined Behavior and Poison
- Target and source independence in IRs

Transformations vs Cost Models

- The transformation may be target independent but the cost model may not be

Transformations vs Cost Models

- The transformation may be target independent but the cost model may not be
- **Example**: Loop unrolling
  - You can do it in Rust, but to do it *effectively*, you need to know the target

Transformations vs Cost Models

- The transformation may be target independent but the cost model may not be
- **Example**: Loop unrolling
  - You can do it in Rust, but to do it *effectively*, you need to know the target
- **Result**: Target-independent IRs but target-*aware* information flowing (e.g., TargetInfo)

How Target-Independent is LLVM IR?

- **Conventional Wisdom**: LLVM IR is target-independent

How Target-Independent is LLVM IR?

- **Conventional Wisdom**: LLVM IR is target-independent
- **Reality pt1**: Attributes like inreg and ton of intrinsics

A Front-End-Based Definition of Target Independence

"An IR is target independent if any front-end lowering to it does not need to know the target"

Reality pt2

- **Example**: 3 different IRs for 3 different target

Reality pt2

- Example: 3 different IRs for 3 different target
- **Why? → ABIs and calling conventions**

Reality pt2

- Example: 3 different IRs for 3 different target
- **Why? → ABIs and calling conventions**
- But wait, LLVM IR abstracts away functions!

Reality pt2

- Example: 3 different IRs for 3 different target
- **Why? → ABIs and calling conventions**
- But wait, LLVM IR abstracts away functions!
  - Yes, but it doesn't have classes

Reality pt2

- Example: 3 different IRs for 3 different target
- **Why? → ABIs and calling conventions**
- But wait, LLVM IR abstracts away functions!
  - Yes, but it doesn't have classes
  - X86-64 ABI: "If a C++ object has either a non-trivial copy constructor or a non-trivial destructor, it is passed by invisible reference …"

Reality pt2

- More obvious example: int
- LLVM IR doesn't have the bit-agnostic int
- You need to know the target to generate LLVM IR

But wait, at least it's source independent right?

Overview

- IRs are not a science (yet)
- Why do we create IRs?
- Types of IRs
  - Trees
    - High-level transformations
    - Turn them into DAGs
  - SSA
    - Where is the value in a φ used?
  - Multi-Level IRs (WHIRL)
    - Trade off?
- Undefined Behavior and poison values
- Target and source independence in IRs