CSC 6580
Spring 2020

Instructor: Stacy Prowell

Homework:

Structuring

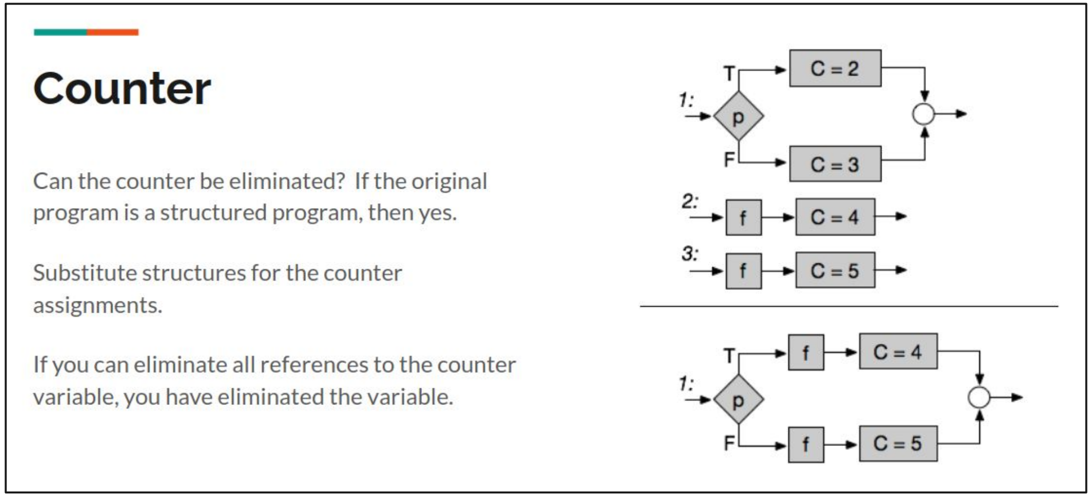
Now that you have the program and some basic data structures, create two new data structures:

- An if-then-else data structure that holds a basic block ending in true / false branching
- A sequence data structure that holds a series of (sequential) basic blocks.

Now for each basic block you found, create a prime (if-then-else or sequence). Let the exit be label 0. For now treat any block that ends with an unknown destination as an exit. Keep track of the number of times a label is referenced and, for any label referenced only once (except the entry point and the exit) substitute the label setting block with the corresponding prime. Be careful to avoid an infinite recursion!

Structuring

The **structure theorem** was covered on 2/11, and an example was given on 2/13.



Counter

Can the counter be eliminated? If the original program is a structured program, then yes.

Substitute structures for the counter assignments.

If you can eliminate all references to the counter variable, you have eliminated the variable.

Structuring

Simple output for an if-then-else.

```
if
  lea rcx, [rax]
  jz 0x2132
then
  mov edi, 1
  jmp 0x214f
  L := 0x214f
else
  call 0x215a
  L := 0x214f
fi
```

Simple output for a sequence.

```
mov edi, 1
jmp 0x214f
L := 0x214f
```

Structuring

Simple output for an if-then-else.

```
if
  lea rcx, [rax]
  jz 0x2132
then
  mov edi, 1
  jmp 0x214f
  L := 0x214f
else
  call 0x215a
  L := 0x214f
fi
```

Simple output for a sequence.

```
mov edi, 1
jmp 0x214f
L := 0x214f
```

Note that these necessarily end with an assignment to the label. This allows the new program to simulate the control flow of the original program.

Structuring

Simple output for an if-then-else.

```
if
  lea rcx, [rax]
  jz 0x2132
then
  mov edi, 1
  jmp 0x214f
  L := 0x214f
else
  call 0x215a
  L := 0x214f
fi
```

Simple output for a sequence.

```
mov edi, 1
jmp 0x214f
L := 0x214f
```

The predicate has side-effects. Don't worry about this now. Splitting the predicate apart to isolate the side-effect-free test requires the full semantics of the operation. Think about `loop`, for example.

A Little Philosophy

What are we trying to do?

$$\mathcal{C} : \mathbb{P} \rightarrow \mathbb{B}$$

Is the compiler a total function?

What are we trying to do?

$\mathcal{C} : \mathbb{P} \rightarrow \mathbb{B}$

Is the compiler a total function? No. Some programs cannot be compiled.

What are we trying to do?

$$\mathcal{C} : \mathbb{P} \to \mathbb{B}$$

Is the compiler surjective (onto)?

What are we trying to do?

$\mathcal{C} : \mathbb{P} \to \mathbb{B}$

Is the compiler surjective (onto)? No. Some binaries have no corresponding source code.

What are we trying to do?

We can:

- try to *decompile* the program.

That is, produce source code from the binary.

Is such source code a *faithful representation* of the program?

Is source code easier to understand than the original program? Can you derive the necessary answers from it?

What are we trying to do?

We can:

- try to *derive a representation of* the program.

That is, produce expressions in some mathematical language, possibly not compilable, that is a *faithful representation* of the program… in some sense.

What are we trying to do?

We can:

- try to *answer specific questions about* the program.

That is, produce answers to specific questions about a program.

What are we trying to do?

All of these are *models* of the program; none of them is the program itself.

In fact, we can think of the binary program itself as an (executable) model for a physical computation that takes place on some particular processor… and that’s sometimes useful.

Mycroft
Type-Based Decompilation

Decompilation

Decompiling a program requires that we reconstruct a lot of semantic information, many times by *guessing* to arrive at a *plausible* answer.

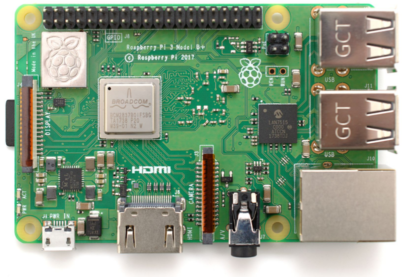
This is especially true for memory shapes. That is, what a portion of memory represents.

Often we can get close to this with *type reconstruction*, where we decide that part of memory is an integer, a pointer, or a structure. We can only do this by observing how that memory is *used*.

Type-Based Decompilation*
(or Program Reconstruction via Type Reconstruction)

Alan Mycroft

Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK
`http://www.cl.cam.ac.uk/users/am`



Co-Founder of the
Raspberry Pi Foundation

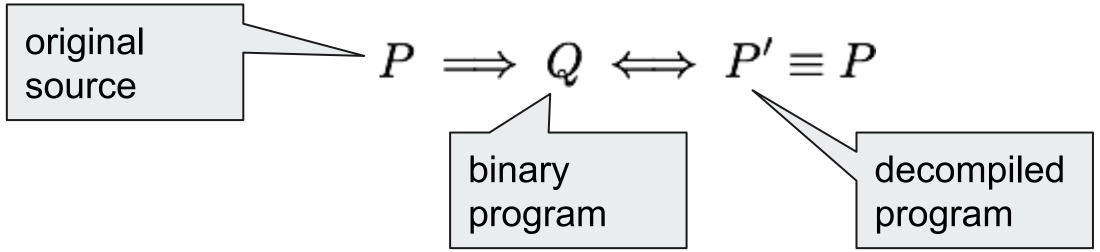
Abstract. We describe a system which decompiles (reverse engineers) C programs from target machine code by type-inference techniques. This extends recent trends in the converse process of compiling high-level languages whereby type information is preserved during compilation. The algorithms remain independent of the particular architecture by virtue of treating target instructions as register-transfer specifications. Target code expressed in such RTL form is then transformed into SSA form (undoing register colouring etc.); this then generates a set of type constraints. Iteration and recursion over data-structures causes synthesis of appropriate recursive C `struct`s; this is triggered by and resolves occurs-check constraint violation. Other constraint violations are resolved by C's casts and `union`s. In the limit we use heuristics to select between equally suitable C code—a good GUI would clearly facilitate its professional use.

Many-to-One

A correct optimizing compiler is going to transform many valid input *source* programs to the same output *binary* code: **many to one**.

For this reason, you cannot recover the original input source program, but you may be able to recover a **semantically equivalent** source program. This is **decompilation**, and it can be *very difficult*.

In the *ideal* case you could “round trip” the decompiled source. In practice… not always.



$$P \Longrightarrow Q \Longleftrightarrow P' \equiv P$$

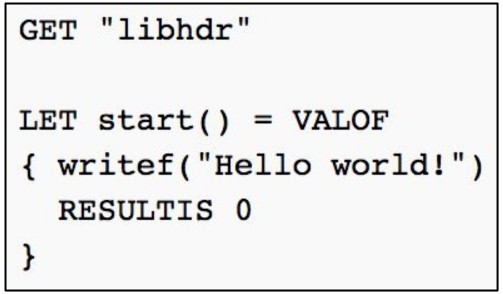
Aside: BCPL

The folks at Cambridge were busy creating the Combined (or Cambridge) Programming Language (CPL). This language had lots of features, some of which were hard to compile.

A *step* in creating CPL was to create a Basic CPL, or BCPL. This language **worked**, worked **well**, was **easy to port**, and was therefore incredibly influential. CPL is a dead language, but BCPL lives on.

Aside: BCPL

- The first “brace” language `{ … }` or `$( … $)` for limited keyboards.
- The prototypical “Hello world” was first written in BCPL.
- Introduced the `//` comment form.
- Introduced the virtual machine / bytecode approach.
- No types! Everything is a word… and platform dependent…
- ...and first major use of Hungarian notation.
- Some compilers fit in 16 KiB.
- Spread to support 25 different architectures before gradually giving way to C.
- First language used on the Xerox Alto… which was also the first computer designed for a GUI
- Was succeeded by B, and then by C, and then by C++ and everything else.
- Is (just barely) older than Stacy; first compiler was written in 1967.

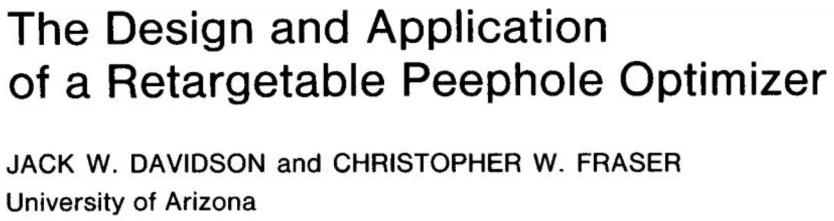


```
GET "libhdr"

LET start() = VALOF
{ writef("Hello world!")
  RESULTIS 0
}
```

Aside: RTL

Register Transfer Langauge (RTL) originates in:
ACM Transactions on Programming Languages and Systems, Vol. 2, No. 2, April 1980, Pages 191-20



The Design and Application
of a Retargetable Peephole Optimizer

JACK W. DAVIDSON and CHRISTOPHER W. FRASER
University of Arizona

Aside: RTL

Register Transfer Language (RTL) “lifts” processor semantics to a more abstract level. This is commonly used in compilers as an intermediate representation (IR) to allow a separation of concerns:

- Compile source code to RTL
- Translate RTL into target architecture

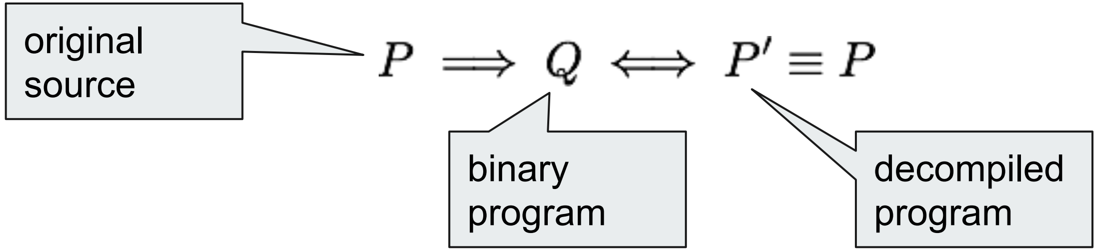
See GCC RTL, LLVM Target Independent Code, or Ghidra’s P-code.

Many-to-One

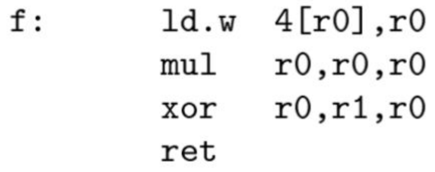
A correct optimizing compiler is going to transform many valid input *source* programs to the same output *binary* code: **many to one**.

For this reason, you cannot recover the original input source program, but you may be able to recover a **semantically equivalent** source program. This is **decompilation**, and it can be *very difficult*.

In the *ideal* case you could "round trip" the decompiled source. In practice... not always.



The Intuitive Example



```
f:      ld.w   4[r0],r0
        mul    r0,r0,r0
        xor    r0,r1,r0
        ret
```

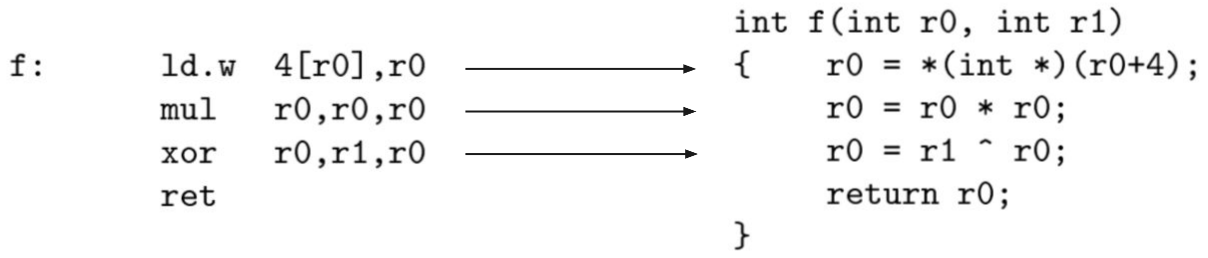
We need to know what his "intuitive" example means to follow along.

- `r0, r1, r2, …`: Registers
- `ld.w`: Load a word
- `4[r0]`: The *content* of the location `(r0 + 4)`
- `mul, xor, ret`: Multiply, exclusive or, return

The order is *source*, *destination*. So `add r0,r1,r2` adds `r0` and `r1` and stores the result in `r2`.

Values are passed to and from functions in registers, in order.

The Intuitive Example



```
f:      ld.w  4[r0],r0
        mul   r0,r0,r0
        xor   r0,r1,r0
        ret
```

```
int f(int r0, int r1)
{    r0 = *(int *)(r0+4);
     r0 = r0 * r0;
     r0 = r1 ^ r0;
     return r0;
}
```

Next we can consider *live* ranges for each variable, and give a new name to each value. This converts the program to **Single Static Assignment (SSA)**, a convenient form for further analysis.

This undoes **register coloring**.

Aside: Register Coloring

Source code is compiled to some intermediate representation (IR). RTL is an example of an IR. The next step is to map IR operations to machine operations, and to map IR resources (registers) to finite machine resources.

In the IR we may use thousands of values, but the machine might provide only a few registers, some of which may not be available for use at all (RIP, RSP, RBP, etc.). The assignment of values to registers is **register allocation**, often called **register coloring** because the same register must be used for multiple values.

We will depend on *live value analysis*.

Aside: Register Coloring

- Registers are scarce
  - Lots more IR variables than machine registers so…
  - ...we have to either reuse registers or work with memory… and memory is slow.
- Some registers aren't really different registers. `RAX, EAX, AX, AH`
- Some instructions must use specific registers. `loop` uses `RCX`, for instance
- Some registers may be reserved for the operating system or the calling convention. `rsp, rbp`

Aside: Register Coloring

Consider:

```
x = y+z;
a = x;
y = x+a;
```

Assume values are stored as an offset from `.vars`.

```
x: .vars+0
y: .vars+8
z: .vars+16
a: .vars+24
```

We initially have to fetch the values we need, and have to put the changed values back at the end.

Aside: Register Coloring

Consider:

```
x = y+z;
a = x;
y = x+a;
```

Let’s just work with stuff in memory, and allocate new storage (on the stack) if we need it.

```
mov rsi, .vars
; x = y+z
mov rax, [rsi+8]
mov rbx, [rsi+16]
add rax, rbx
mov [rsi+0], rax
; a = x
...
```

Aside: Register Coloring

Consider:

```
x = y+z;
a = x;
y = x+a;
```

This is *really easy* but *really inefficient.*

Let's just work with stuff in memory, and allocate new storage (on the stack) if we need it.

```
mov rsi, .vars
; x = y+z
mov rax, [rsi+8]
mov rbx, [rsi+16]
add rax, rbx
mov [rsi+0], rax
; a = x
...
```

Aside: Register Coloring

Consider:

```
x = y+z;
a = x;
y = x+a;
```

Idea: At each program point a register holds at most one live variable.

So we need to do live variable analysis.

Aside: Register Coloring

	x	y	z	a
	dead	live	live	dead
`x = y+z;`	live	dead	dead	dead
`a = x;`	live	dead	dead	live
`y = x+a;`	dead	live	dead	dead

Each entry shows the status of the variable *after* the corresponding line completes.

At the end y is *dirty* and has to be written back. Thus it is *live*. We ignore both x and a and note they are both dead on entry.

At most two registers are live at any time, so we can get away with two registers.

Aside: Register Coloring

	x	y	z	a	rax	rbx
	dead	live	live	dead	y	z
x = y+z;	live	dead	dead	dead	x	-
a = x;	live	dead	dead	live	x	a
y = x+a;	dead	live	dead	dead	y	-

```
mov rsi, .vars
mov rax, [rsi+8]
mov rbx, [rsi+16]
; x = y+z
add rax, rbx
; a = x
mov rbx, rax
; y = x+a
add rax, rbx
mov [rsi+8], rax
```

Aside: Register Coloring

	x	y	z	a	rax	rbx
	dead	live	live	dead	y	z
`x = y+z;`	live	dead	dead	dead	x	-
`a = x;`	live	dead	dead	live	x	a
`y = x+a;`	dead	live	dead	dead	y	-

```
mov rsi, .vars
mov rax, [rsi+8]
mov rbx, [rsi+16]
; x = y+z
add rax, rbx
; a = x
mov rbx, rax
; y = x+a
add rax, rbx
mov [rsi+8], rax
```

Much better!

Aside: Register Coloring

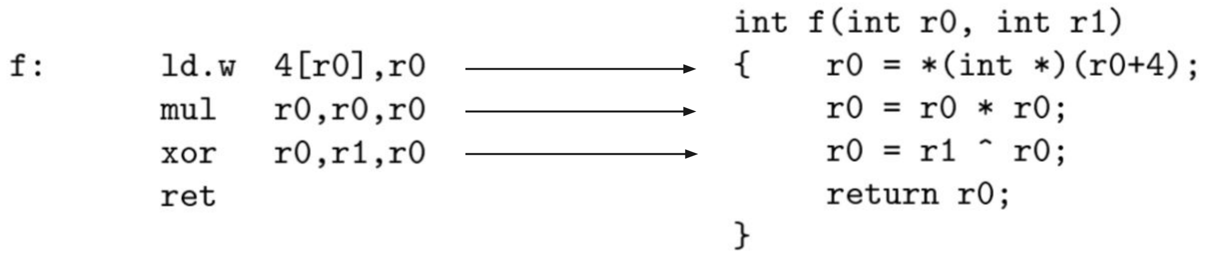
	x	y	z	a	rax	rbx
	dead	live	live	dead	y	z
`x = y+z;`	live	dead	dead	dead	x	-
`a = x;`	live	dead	dead	live	x	a
`y = x+a;`	dead	live	dead	dead	y	-

```
mov rsi, .vars
mov rax, [rsi+8]
mov rbx, [rsi+16]
; x = y+z
add rax, rbx
; a = x
mov rbx, rax
; y = x+a
add rax, rbx
mov [rsi+8], rax
```

Much better!

We can do better still, but that's enough for now.

The Intuitive Example



```
f:      ld.w   4[r0],r0    ──────────▶   int f(int r0, int r1)
        mul    r0,r0,r0    ──────────▶   {     r0 = *(int *)(r0+4);
        xor    r0,r1,r0    ──────────▶         r0 = r0 * r0;
        ret                                    r0 = r1 ^ r0;
                                               return r0;
                                         }
```

Next we can consider *live* ranges for each variable, and give a new name to each value. This converts the program to **Single Static Assignment (SSA)**, a convenient form for further analysis.

This undoes **register coloring**.

The Intuitive Example

```
int f(int r0, int r1)
{     int r0a = *(int *)(r0+4);
      int r0b = r0a * r0a;
      int r0c = r1 ^ r0b;
      return r0c;
}
```

```
int f(int r0, int r1)
{     r0 = *(int *)(r0+4);
      r0 = r0 * r0;
      r0 = r1 ^ r0;
      return r0;
}
```

Single Static Assignment (SSA)

Suppose we treat `r0` as an `int*`, which is how it is used.

Then `*(int *)(r0+4) = *((int *)r0+1)` becomes `*(r0+1) = r0[1]`.

The Intuitive Example

```
int f(int r0, int r1)
{     int r0a = *(int *)(r0+4);
      int r0b = r0a * r0a;
      int r0c = r1 ^ r0b;
      return r0c;
}
```

```
int f(int *r0, int r1)
{     int r0a = r0[1];            ;
      int r0b = r0a * r0a;
      int r0c = r1 ^ r0b;
      return r0c;
}
```

Now `r0` is being used differently (`r0a` is an `int`). Since we have named each value, we can substitute (paying attention to side-effects and *sequence points*).

```
return r0c = return r1 ^ r0b = return r1 ^ (r0a * r0a) = return r1 ^ r0[1] * r0[1]
```

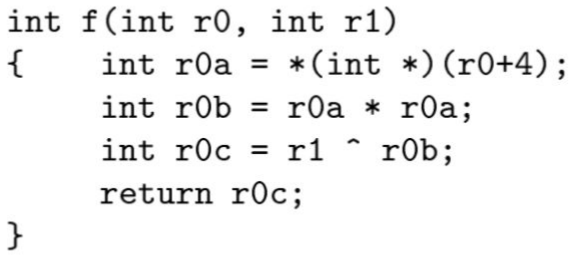
Aside: Sequence Point

A **sequence point** is a place in the program flow where computation is *stable*.

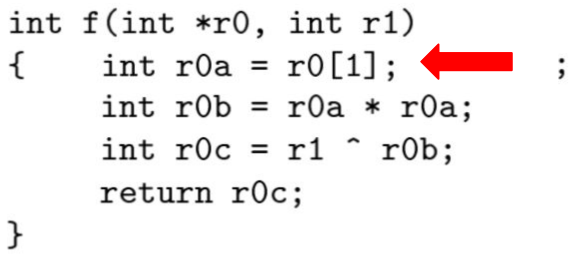
- All side effects of prior computations have completed.
- No side effects of subsequent computations have yet occurred.

This is an important concept in the semantics of C, C++, and some other languages.

The Intuitive Example



```
int f(int r0, int r1)
{     int r0a = *(int *)(r0+4);
      int r0b = r0a * r0a;
      int r0c = r1 ^ r0b;
      return r0c;
}
```



```
int f(int *r0, int r1)
{     int r0a = r0[1];          ;
      int r0b = r0a * r0a;
      int r0c = r1 ^ r0b;
      return r0c;
}
```

Now `r0` is being used differently (`r0a` is an `int`). Since we have named each value, we can substitute (paying attention to side-effects and *sequence points*).

```
return r0c = return r1 ^ r0b = return r1 ^ (r0a * r0a) = return r1 ^ r0[1] * r0[1]
```

The Intuitive Example

```
int f(int *r0, int r1);
{     return r1 ^ (r0[1] * r0[1]);
}
```

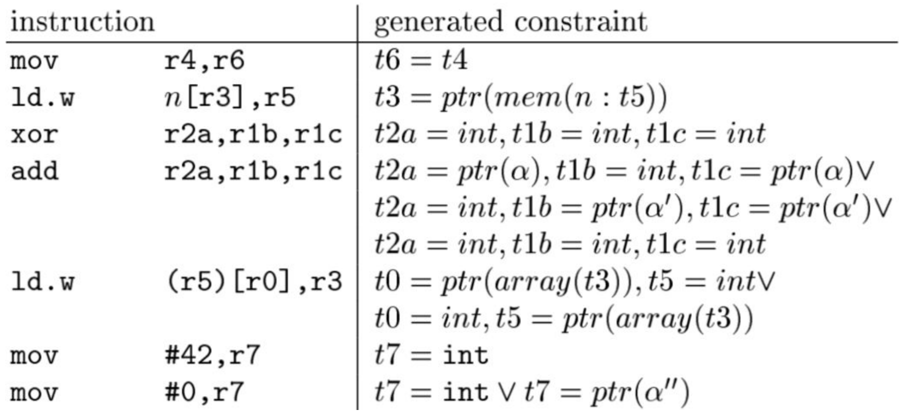
```
int f(int *r0, int r1)
{     int r0a = r0[1];          ;
      int r0b = r0a * r0a;
      int r0c = r1 ^ r0b;
      return r0c;
}
```

We have decompiled this code into *plausible* source.

Type “Reconstruction”

Back to type theory!

This is really the assignment of a *plausible* type. The original types are lost in compilation.

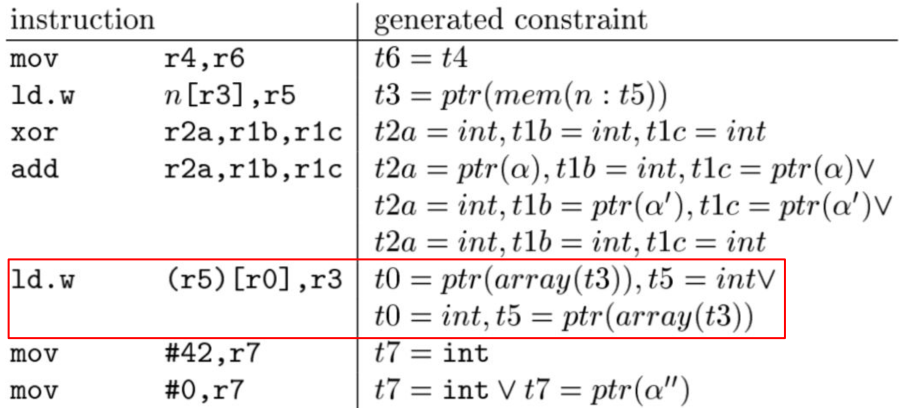


instruction		generated constraint
`mov`	`r4,r6`	$t6 = t4$
`ld.w`	n`[r3],r5`	$t3 = ptr(mem(n : t5))$
`xor`	`r2a,r1b,r1c`	$t2a = int, t1b = int, t1c = int$
`add`	`r2a,r1b,r1c`	$t2a = ptr(\alpha), t1b = int, t1c = ptr(\alpha) \vee$
		$t2a = int, t1b = ptr(\alpha'), t1c = ptr(\alpha') \vee$
		$t2a = int, t1b = int, t1c = int$
`ld.w`	`(r5)[r0],r3`	$t0 = ptr(array(t3)), t5 = int \vee$
		$t0 = int, t5 = ptr(array(t3))$
`mov`	`#42,r7`	$t7 = \texttt{int}$
`mov`	`#0,r7`	$t7 = \texttt{int} \vee t7 = ptr(\alpha'')$

Type “Reconstruction”

Back to type theory!

This is really the assignment of a *plausible* type. The original types are lost in compilation.



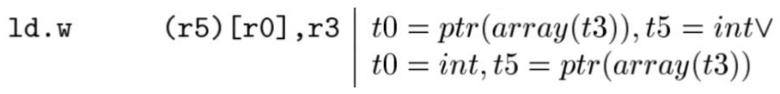
instruction		generated constraint
`mov`	`r4,r6`	$t6 = t4$
`ld.w`	n`[r3],r5`	$t3 = \mathit{ptr}(\mathit{mem}(n : t5))$
`xor`	`r2a,r1b,r1c`	$t2a = \mathit{int}, t1b = \mathit{int}, t1c = \mathit{int}$
`add`	`r2a,r1b,r1c`	$t2a = \mathit{ptr}(\alpha), t1b = \mathit{int}, t1c = \mathit{ptr}(\alpha) \vee$
		$t2a = \mathit{int}, t1b = \mathit{ptr}(\alpha'), t1c = \mathit{ptr}(\alpha') \vee$
		$t2a = \mathit{int}, t1b = \mathit{int}, t1c = \mathit{int}$
`ld.w`	`(r5)[r0],r3`	$t0 = \mathit{ptr}(\mathit{array}(t3)), t5 = \mathit{int} \vee$
		$t0 = \mathit{int}, t5 = \mathit{ptr}(\mathit{array}(t3))$
`mov`	`#42,r7`	$t7 = \texttt{int}$
`mov`	`#0,r7`	$t7 = \texttt{int} \vee t7 = \mathit{ptr}(\alpha'')$

Type “Reconstruction”

The trick:

```
(r5)[r0] = (r0)[r5]
```

Thus `r0` (or `r5`) is a pointer to an array, and `r5` (or `r0`) is the index into the array.



ld.w	(r5)[r0],r3	$t0 = ptr(array(t3)), t5 = int \vee$ $t0 = int, t5 = ptr(array(t3))$

Next Time: SSA