1 Optimization (Continued)

1.1 Flow Analysis

Let's consider the following code,

```
(let (curr lst)
    (let (total 0)
        (loop
            (if (= lst nil) (break total)
                (block
                (set! total (+ total (fst lst)))
                (set! lst (snd lst)))))))
```

How can we use flow analysis to reduce the amount of tag checking generated in the final assembly?

1.1.1 The `check` Instruction

An idea we want to do is to make tag checks explicit with `check` steps: which checks can we remove? One new step we can introduce in the intermediate representation is

```
check <some bool expr>
```

The semantics are simple: if the check is true, then everything continues as normal. Otherwise, an error is thrown.

(Example.) `check sametag(curr, nil)` checks to see if `curr` has the same tag as `nil`. Something we've incorporated into our compiler is the `isnum(x)` and `isbool(x)` checks, which checks to see if the expression x is a number or boolean, respectively.

With this said, the corresponding intermediate representation of the above code is

```
sum(lst) {
    start0:     curr <- lst
    start1:     total <- 0
    loop_0:     check sametag(curr, nil)
    loop_1:     %t_0 <- curr == nil
    loop_2:     if %t_0 thn_0 els_0
    thn_0:      rax <- total
    thn_1:      goto end_0
    thn_2:      goto ifend_0
    els_0:      check isnonnilpair(curr)
    els_1:      %t_1 <- fst curr
    els_2:      check isnum(total)
    els_3:      check isnum(%t_1)
    els_4:      %t_2 <- total + %t_1
    els_5:      total <- %t_2
    els_6:      check isnonnilpair(curr)
    els_7:      %t_3 <- snd curr
    els_8:      curr <- %t_3
    els_9:      rax <- curr
    els_10:     goto ifend_0
    ifend_0:    goto loop_0
    end_0:      return rax
}
```

We can use flow analysis to analyze how data flows through a program. We can use this information to identify variables that hold values at different points in the program, and how these values change over time. For our purposes, we wish to use flow analysis to reduce the amount of unncessary tag checking. Using the `check` instruction that was mentioned, we can do just this.

1.1.2 A Flow Analysis Walkthrough

The flow analysis we'll do starts from the beginning and goes to the end (this is known as *forward analysis*). The information we'll keep track of are the *potential* tags. Let's analyze each line of the intermediate representation. For each line executed, we consider what possible tag value each variable can represent. The potential tags are **N**umbers, **B**ooleans, **Nil**, and **P**airs. At any point in the program, each variable can hold a set of these possible types. Let $A = \{N, B, \text{Nil}, P\}$ be the set of all types.

IR	`lst`	`curr`	`total`	t_0	t_1	t_2	t_3	`rax`
`start0: curr <- lst`	A	$\to A$						
`start1: total <- 0`	A	A	$\to N$					
`loop_0: check sametag(curr, nil)`	A	$\to \{\text{Nil}, P\}$	N					
`loop_1: %t_0 <- curr == nil`	A	$\{\text{Nil}, P\}$	N	$\to B$				
`loop_2: if %t_0 thn_0 els_0`	A	$\{\text{Nil}, P\}$	N	B				
`thn_0: rax <- total`	A	{Nil, P}	N	B				N
`thn_1: goto end_0`	A	{Nil, P}	N	B				N
`thn_2: goto ifend_0`[1]								
`els_0: check isnonnilpair(curr)`[2]	A	$\{\text{Nil}, P\} \to P$	N	B				
`els_1: %t_1 <- fst curr`	A	P	N	B	$\to A$			
`els_2: check isnum(total)`	A	P	$N \to N$	B	$\to A$			
`els_3: check isnum(%t_1)`	A	P	N	B	$A \to N$			
`els_4: %t_2 <- total + %t_1`	A	P	N	B	N	$\to N$		
`els_5: total <- %t_2`	A	P	$N \to N$	B	N	N		
`els_6: check isnonnilpair(curr)`	A	$P \to P$	N	B	N	N		
`els_7: %t_3 <- snd curr`	A	P	N	B	N	N	$\to A$	
`els_8: curr <- %t_3`	A	$P \to A$	N	B	N	N	A	
`els_9: rax <- curr`	A	A	N	B	N	N	A	$\to A$
`els_10: goto ifend_0`	A	A	N	B	N	N	A	A
`ifend_0: goto loop_0`	A	A	N	B	N	N	A	A
`end_0: return rax`								

Remarks:

- At (1), we have dead code. So, nothing needs to be filled out.
- At (2), we can copy the tag information we have from the `goto` instruction which jumps to this line. In this case, we copied this information from the line `loop_2`.
- In general, we can copy the information from the `goto` to the target label. This is especially important when we have a `goto` that goes to a label that's *before* where the `goto` occurred.

At `goto loop_0`, we now perform a backwards jump back to the label `loop_0` and perform additional forward analysis with the information we found prior to the `goto`. These tags are denoted by red.

IR	`lst`	`curr`	`total`	t_0	t_1	t_2	t_3	`rax`
`start0: curr <- lst`	A	$\to A$						
`start1: total <- 0`	A	A	$\to N$					
`loop_0: check sametag(curr, nil)`	A	$A \to \{\text{Nil}, P\}$	N	$\to B$	$\to N$	$\to N$	$\to A$	$\to A$
`loop_1: %t_0 <- curr == nil`	A	$\{\text{Nil}, P\}$	N	$\to B$	N	N	A	A
`loop_2: if %t_0 thn_0 els_0`	A	$\{\text{Nil}, P\}$	N	B	N	N	A	A
`thn_0: rax <- total`	A	{Nil, P}	N	B	N	N	A	$A \to N$
`thn_1: goto end_0`	A	{Nil, P}	N	B	N	N	A	N
`thn_2: goto ifend_0`								
`els_0: check isnonnilpair(curr)`[3]	A	$\{\text{Nil}, P\} \to P$	N	B	N	N	A	A
`els_1: %t_1 <- fst curr`	A	P	N	B	$\to A$			
`els_2: check isnum(total)`	A	P	$N \to N$	B	$\to A$			
`els_3: check isnum(%t_1)`	A	P	N	B	$A \to N$			
`els_4: %t_2 <- total + %t_1`	A	P	N	B	N	$\to N$		
`els_5: total <- %t_2`	A	P	$N \to N$	B	N	N		
`els_6: check isnonnilpair(curr)`	A	$P \to P$	N	B	N	N		
`els_7: %t_3 <- snd curr`	A	P	N	B	N	N	$\to A$	
`els_8: curr <- %t_3`	A	$P \to A$	N	B	N	N	A	
`els_9: rax <- curr`	A	A	N	B	N	N	A	$\to A$
`els_10: goto ifend_0`	A	A	N	B	N	N	A	A
`ifend_0: goto loop_0`	A	A	N	B	N	N	A	A
`end_0: return rax`								

Remark:

- At (4), note that we're not directly copying N from `thn_1` to `els_0`. Rather, we're copying the tag information from the `goto` instruction that jumps to this line.

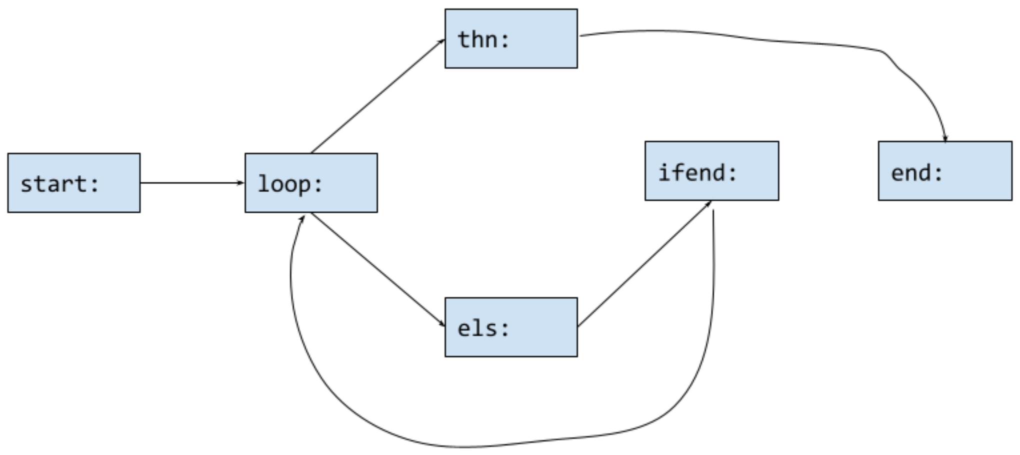
Let's consider the lines `els_2: check isnum(total)` and `els_6: check isnonnilpair(curr)`. Based on the forward analysis, these two lines of code are useless. Likewise, `els_0: check isnonnilpair(curr)` could be *optimized* (not removed) to check if `curr` is `nil`.

1.1.3 In Summary

In summary, the idea behind flow analysis is that there's really two steps:

1. Do the analysis and gather information
2. Rescan the program with that information and change the program to remove/optimize any code as needed.

The corresponding **control flow graph** looks like



At the start, we have a bunch of instructions. This eventually leads to a loop. In the `thn` branch, we go straight to the end since we have the dead code. In the `els` branch, we eventually get to the `ifend` statement where we end up going to the loop.