Overview

**Frontend:** Lexer/Parser & AST Builder & Semantic Analyzer (CW 1 & 2)

**Middle End:** Optimizations

**Backend:** RISC-V
Instruction Selection

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces
- Automation has been less successful in the back end
Instruction Selection

- Mapping the IR into assembly code (in our case AST to RISCV-V assembly)
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes
Register Allocation

- Deciding which value reside in a register
- Minimise amount of spilling
Instruction Scheduling

- Avoid hardware stalls and interlocks
- Reordering operations to hide latencies
- Use all functional units productively
The Big Picture

How hard are these problems?

- Instruction selection
  - Can make locally optimal choices, with automated tool
  - Global optimality is NP-Complete

- Instruction scheduling
  - Single basic block ⇒ heuristic work quickly
  - General problem, with control flow ⇒ NP-Complete

- Register allocation
  - Single basic block, no spilling ⇒ linear time
  - Whole procedure is NP-Complete (graph colouring algorithm)

These problems are tightly coupled

However, conventional wisdom says we lose little by solving these problems independently.
How to solve these problems?

- **Instruction selection**
  - Use some form of pattern matching
  - Assume enough registers or target “important” values

- **Instruction scheduling**
  - Within a block, list scheduling is “close” to optimal
  - Across blocks, build framework to apply list scheduling

- **Register allocation**
  - Start from virtual registers & map “enough” into k
  - With targeting, focus on “good” priority heuristic

### Approximate Solutions

Will be important to define good metrics for “close”, “good”, “enough”, . . . .
Generating Code for a Register-Based Machine

The key code quality issue is holding values in registers

- when can a value be safely allocated to a register?
  - When only 1 name can reference its value
  - Pointers, parameters, aggregates & arrays all cause trouble

- when should a value be allocated to a register?
  - when it is both safe & profitable

Encoding this knowledge into the IR:

- assign a virtual register to anything that go into one
- load or store the others at each reference
Representing Values in Memory/Registers

Statically Typed Languages (C/C++)

Plain (Unboxed) Values

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>bool</td>
</tr>
<tr>
<td>c</td>
<td>int</td>
</tr>
</tbody>
</table>

Boxed Values

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>?</td>
</tr>
<tr>
<td>b</td>
<td>?</td>
</tr>
<tr>
<td>c</td>
<td>?</td>
</tr>
</tbody>
</table>

Dynamically Typed Languages (Python)

ChocoPy

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>bool</td>
</tr>
<tr>
<td>c</td>
<td>int</td>
</tr>
</tbody>
</table>

Plain (Unboxed) Values

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>36</td>
</tr>
<tr>
<td>b</td>
<td>T</td>
</tr>
<tr>
<td>c</td>
<td>12</td>
</tr>
</tbody>
</table>

Boxed Values

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>bool</td>
</tr>
<tr>
<td>c</td>
<td>int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
</tr>
<tr>
<td>b</td>
<td>bool</td>
</tr>
<tr>
<td>c</td>
<td>int</td>
</tr>
</tbody>
</table>

10
Properties of Boxed Values

**Advantages**

- Better for
  - Dynamically Typed Languages
  - Dynamic Dispatch
- Can be Treated More Uniformly in The Compiler

**Disadvantages**

- Reduced Performance (Runtime)
- Higher Memory Cost
- Requires Complex Runtime System
Instruction Selection: SSA Rewrites Across Dialects

%0 : !choco.ir.named_type<"int"> = choco.ir.literal() ["value" = 42 : !i32]

%0 : !riscv_ssa.reg = riscv_ssa.li() ["immediate" = 42 : !i32]
def rewrite_literal(self, op: Literal, rewriter: PatternRewriter):
    value = op.value

    if isinstance(value, IntegerAttr):
        constant = RISCVSSA.LIOp.get(op.value)

    if isinstance(value, BoolAttr):
        if value.data == True:
            constant = RISCVSSA.LIOp.get(1)
        elif value.data == False:
            constant = RISCVSSA.LIOp.get(0)

    rewriter.replace_op(op, [constant])
PatternRewriter: insert\|erase\_op - the low-level interface

def insert\_op\_before\_matched\_op(self, op: Union[Operation, List[Operation]])

def insert\_op\_after\_matched\_op(self, op: Union[Operation, List[Operation]])

def insert\_op\_at\_pos(self, op: Union[Operation, List[Operation]],
    block: Block, pos: int)

def insert\_op\_before(self, op: Union[Operation, List[Operation]], target\_op: Operation)

def insert\_op\_after(self, op: Union[Operation, List[Operation]], target\_op: Operation)

def erase\_op(self, op: Operation, safe\_erase: bool = True)
PatternRewriter: The default interface

```python
def replace_matched_op(self, 
    new_ops: Union[Operation, List[Operation]], 
    new_results: Optional[List[Optional[OpResult]]] = None)
```

*Replace an operation with a list of new operations*

*The users of the old operation are automatically connected to use the new operation.*
What about Control Flow

```python
if 1 < 0:
    print(1)
else:
    print(2)
```

```assembly
%0 = riscv_ssa.li() ["immediate" = 1 : !i32]
%1 = riscv_ssa.li() ["immediate" = 0 : !i32]
%2 = riscv_ssa.slt(%0, %1)
%3 = riscv_ssa.li() ["immediate" = 0 : !i32]
riscv_ssa.beq(%2, %3) ["offset" = !riscv.label<if_else_1>]
```

```assembly
riscv_ssa.label() ["label" = !riscv.label<if_then_1>]
%4 = riscv_ssa.li() ["immediate" = 1 : !i32]
riscv_ssa.call(%4) ["func_name" = "_print_int"]
riscv_ssa.j() ["offset" = !riscv.label<if_after_1>]
```

```assembly
else:
    print(2)
```

```assembly
%5 = riscv_ssa.li() ["immediate" = 2 : !i32]
riscv_ssa.call(%5) ["func_name" = "_print_int"]
```

```assembly
riscv_ssa.label() ["label" = !riscv.label<if_after_1>]
```
Basic Blocks: Linear Sequences of Operations

if i < 0

fallthrough

if then 1
print(2)

beq (%2, %3)

if else 1
print(1)

if after 1
...

...
Do we need basic blocks

**Advantages**

- Facilitate reordering of control flow

**Disadvantages**

- Pattern rewrites become more difficult
- Not as easy to reason about
PatternRewriter: Moving Blocks

def inline_block_before_matched_op(self, block: Block)

def inline_block_before(self, block: Block, op: Operation)

def inline_block_after(self, block: Block, op: Operation):