Spiglet is simplified Piglet
Piglet for expressions

Represents a computation that returns a single value

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT i</td>
<td>An integer constant i</td>
</tr>
<tr>
<td>LABEL n</td>
<td>Symbolic constant n (a code label)</td>
</tr>
<tr>
<td>TEMP i</td>
<td>Temporary i (a “register”)</td>
</tr>
<tr>
<td>BIN e1 e2</td>
<td>Application of a binary operator to integer operands</td>
</tr>
<tr>
<td>HALLOCATE e</td>
<td>Allocate e bytes on the heap</td>
</tr>
<tr>
<td>CALL f e1 ... en</td>
<td>Procedure call</td>
</tr>
<tr>
<td>ESEQ s e</td>
<td>Expression sequence; evaluate s for side effects</td>
</tr>
</tbody>
</table>
## Piglet for statements

Represents a computation that returns no value

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE (TEMP t) e</td>
<td>Evaluate e and put result in a temporary t</td>
</tr>
<tr>
<td>HLOAD (TEMP t) e1 (INT i)</td>
<td>Evaluate e1 to an address a, load *(a+i) and put result in t</td>
</tr>
<tr>
<td>HSTORE e1 (INT i) e2</td>
<td>Evaluate e1 to an address a, store e2 in *(a+i)</td>
</tr>
<tr>
<td>EXP e</td>
<td>Evaluate e and discard result</td>
</tr>
<tr>
<td>JUMP (LABEL n)</td>
<td>Jump to code address n</td>
</tr>
<tr>
<td>CJUMP e1 (LABEL n)</td>
<td>Evaluate e1, jump to n if true; otherwise fall through</td>
</tr>
<tr>
<td>BEGIN s1 ... sn END</td>
<td>Evaluate s1, then s2, ..., then sn</td>
</tr>
<tr>
<td>LABEL n</td>
<td>Define a constant value of name n as current code address.</td>
</tr>
<tr>
<td>PRINT e</td>
<td>Evaluate integer e, print result</td>
</tr>
<tr>
<td>ERROR</td>
<td>Report an error and exit</td>
</tr>
<tr>
<td>NOP</td>
<td>Do nothing</td>
</tr>
</tbody>
</table>
**Spiglet for expressions**

Distinguish between *expressions* e and *simple expressions* p

<table>
<thead>
<tr>
<th>p</th>
<th>Simple expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN (TEMP t1) p2</td>
<td>Application of a binary operator to integer operands</td>
</tr>
<tr>
<td>HALLOCATE p</td>
<td>Allocate e bytes on the heap</td>
</tr>
<tr>
<td>CALL p (TEMP t1) ... (TEMP tn)</td>
<td>Procedure call</td>
</tr>
</tbody>
</table>
Distinguish between *expressions* \( e \) and *simple expressions* \( p \)

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT ( i )</td>
<td>An integer constant ( i )</td>
</tr>
<tr>
<td>LABEL ( n )</td>
<td>Symbolic constant ( n ) (a code label)</td>
</tr>
<tr>
<td>TEMP ( i )</td>
<td>Temporary ( i ) (a “register”)</td>
</tr>
</tbody>
</table>
Spiglet for statements

Represents a computation that returns no value

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE (TEMP t) e</td>
<td>Evaluate e and put result in a temporary t</td>
</tr>
<tr>
<td>HLOAD (TEMP t1) (TEMP t2) (INT i)</td>
<td>Evaluate t2 to an address a, load *(a+i) and put result in t1</td>
</tr>
<tr>
<td>HSTORE (TEMP t1) (INT i) (TEMP t2)</td>
<td>Evaluate t1 to an address a, store t2 in *(a+i)</td>
</tr>
<tr>
<td>JUMP (LABEL n)</td>
<td>Jump to code address n</td>
</tr>
<tr>
<td>CJUMP (TEMP t) (LABEL n)</td>
<td>Evaluate e1, jump to n if true; otherwise fall through</td>
</tr>
<tr>
<td>BEGIN s1 ... sn END</td>
<td>Evaluate s1, then s2, ..., then sn</td>
</tr>
<tr>
<td>LABEL n</td>
<td>Define a constant value of name n as current code address.</td>
</tr>
<tr>
<td>PRINT p</td>
<td>Evaluate integer e, print result</td>
</tr>
<tr>
<td>ERROR</td>
<td>Report an error and exit</td>
</tr>
<tr>
<td>NOP</td>
<td>Do nothing</td>
</tr>
</tbody>
</table>
No EXP statement
No ESEQ expressions
Non-simple expressions can occur only as RHS of MOVE
Generating Spiglet

Simple version:

Visit each expression, building a list of MOVEs to TEMPs, replacing subexpressions with TEMPs

\[ E[e] \rightarrow t \]

evaluates e, storing result in the TEMP t

\[ S[e] \]

flattens statement e
$E[\text{TEMP } t_1 ] \ t =$

MOVE t (TEMP t1)
$E[\text{INT } n ] \mathbf{t} =$

MOVE t (INT n)
Piglet to Spiglet

\[ E[\text{PLUS } e_1 e_2 ] t = \]

BEGIN

\[ E[ e_1 ] (\text{TEMP } t_1) \]

\[ E[ e_2 ] (\text{TEMP } t_2) \]

MOVE t (PLUS (TEMP t_1) (TEMP t_2))

END

Pass down temporary in which to write the result.
\[ \begin{align*} S[\text{HSTORE } a \ i \ b ] \ t & = \\
\text{BEGIN} & \\
E[\ a \ ] \ t1 & \\
E[\ b \ ] \ t2 & \\
\text{HSTORE } t1 \ i \ t2 & \\
\text{END} \end{align*} \]
\[ S[\text{HLOAD a b i }] t = \]
BEGIN
\[ E[\text{a }] t1 \]
\[ E[\text{n}] t2 \]
HLOAD t1 t2 i
END
\[ S[\text{CALL } e_0 \ e_1 \ ... \ e_n ] = \]
\[ E[\ e_0 \ ] \ t_0 \]
\[ E[\ e_1 \ ] \ t_1 \]
\[ ... \]
\[ E[\ e_n \ ] \ t_n \]
\[ \text{CALL } t_0 \ t_1 \ ... \ t_n \]
$E[^{\text{BEGIN } s_1 \ldots s_n \text{ END } e}] \ t =$
BEGIN
  s_1
  ...
  s_n
  $E[^{e}] \ t$
END
$S[\text{MOVE (TEMP t) e}] = E[[e]] t$
Tiling

Above translation introduces a lot of temporaries

Can reduce number of temporaries by *tiling*
- Tile = fragment of IR tree (“tree pattern”)
- Each tile is translated into a short sequence of instructions that put result of the expression rooted at that node into a register

Idea is to cover IR tree in disjoint tiles, using the smallest number of tiles possible

Used mainly for instruction selection (i.e., generating assembly code from IR trees)
Can also be used for Piglet -> Spiglet
To implement tiling, need a library of tiles

Tiny tiles --> can always find a covering
Large tiles --> produce fewer instructions

Need both to cover all cases
Tiling example

HLOAD (TEMP t) (PLUS (TEMP base) (MUL (PLUS (INT 3) (INT 1)) (INT 4)))
Tiling example

HLOAD (TEMP t) (PLUS (TEMP base) (MUL (PLUS (INT 3) (INT 1)) (INT 4)))

\[
\begin{align*}
  t1 &= 3 + 1 \\
  t2 &= t1 \times 4 \\
  t3 &= \text{base} + t2 \\
  t &= [t3]
\end{align*}
\]
Tiling example

HLOAD (TEMP t) (PLUS (TEMP base) (MUL (PLUS (INT 3) (INT 1)) (INT 4)))

t = 16[base]
Greedy algorithm:

Start at root, and use largest tile possible for that node

Work down the tree
Maximal munch

visit(MOVE m) {
    // order by size of tile
    // recurse on descendants not included in
    // the tile
    if (m.dst instanceof PLUS &&
        ((PLUS) m.dst).right instanceof INT) {
        m.dst.left.accept(this);
        m.src.accept(this);
        emit("store");
    } else ...
    } else {
        m.dst.accept(this);
        m.src.accept(this);
        emit("store");
    }
}
Dynamic programming

Start at leaves and work up

Memoize cheapest tile at each root

At each node, choose tile whose cost plus cost of tiles for its descendants is cheapest
Tiling effectiveness

Works well for generating assembly from IR trees

Works better for RISC instruction sets than CISC

Not as effective for Piglet -> Spiglet
Next steps

Next steps in project:
- instruction selection
- register allocation

Instruction selection:
- map IR trees to assembly instructions

Register allocation:
- replace temporaries with references to registers
- if not enough registers, *spill* to the stack
Usually, do instruction selection first (with pseudo-registers)

Then, do register allocation

Project:
- do register allocation on Spiglet (producing Kanga),
- then do very simple instruction selection
Register allocation

Need to have values in register before use
Limited number of registers (MIPS: 31, x86: 8, x86-64: 16)
Register allocation changes instruction choices
Can move loads and stores
Optimal allocation is NP-complete for >= 1 register
Liveness analysis

Problem:
- IR has unbounded number of temporaries
- Target machine has fixed number of registers

Approach:
- Temporaries that are not used at the same time can map to the same register
- Not used at the same time = disjoint live ranges
- If not enough registers, spill to memory

Compiler does liveness analysis for each temporary
- live if holds a value that may be needed in the future
Control-flow analysis

Before doing liveness analysis, need to understand control flow of the program

Build control-flow graph (CFG)
int max(int x, int y) {
    int z;
    if (x > y)
        z = x;
    else
        z = y;
    return z;
}

For now: each node is a single instruction.
int max(int x, int y) {
    int z;
    if (x > y)
        z = x;
    else
        z = y;
    return z;
}

The live range of a variable extends from its last use to its definition.
int max(int x, int y) {
    int z;
    if (x > y)
        z = x;
    else
        z = y;
    return z;
}

**Note**: important to distinguish between LHS and RHS of MOVE instructions: z and x are not live at the same time!
**Terminology**

*out-edges* lead to *successor* nodes

*in-edges* come from *predecessor* nodes

pred(n) is the set of predecessors of node n

succ(n) is the set of successors
Liveness analysis

An assignment to $x$ defines $x$. This is called a \textit{def}.

An occurrence of $x$ on the RHS of an assignment uses $x$. A \textit{use}.

Variable is live on an edge if there is a path from that edge to a \textit{use} of the variable that does not go through a \textit{def}.

To decide whether variable $x$ is live at instruction $I$:

for each \textit{use} of $x$:
- trace backward through the control flow graph
- stop at a \textit{def} of $x$
Set-based algorithm

Compute liveness for all variables at once.

Let use(n) be the set of variables used in n
Let def(n) be set of variables defined in n

- If x is in use(n), x is live-in at n
- If x is live-in at n, x is live-out at all predecessors of n
- If x is live-out at n, and not in def(n), x is live-in at n

Solve the following equations:
- in(n) = use(n) U (out(n) - def(n))
- out(n) = U_{s \in succ(n)} in(s)
Set-based algorithm

for each n:
   \( \text{in}(n) = \{ \} \)
   \( \text{out}(n) = \{ \} \)
repeat:
   for each n:
      \( \text{in}'(n) = \text{in}(n) \)
      \( \text{out}'(n) = \text{out}(n) \)
      \( \text{in}(n) = \text{use}(n) \cup (\text{out}(n) - \text{def}(n)) \)
      \( \text{out}(n) = \bigcup_{s \in \text{succ}(n)} \text{in}(s) \)
until \( \text{in}'(n) = \text{in}(n) \) and \( \text{out}'(n) = \text{out}(n) \) for all n
Speeding things up

Can merge nodes of the flow-graph into basic blocks
  - fewer nodes, fewer edges, smaller graph
  - need to adjust the equations to handle this (later!)

One variable-at-a-time can often be faster
  - why? most variables have very short live-ranges

Set representation
Basic blocks

Given linear sequence of instructions

Divide instructions into *basic blocks*
- first statement is a label (or an instruction following a jump)
- last statement is a jump (JUMP or CJUMP)
- there are no other labels or jumps

Edge between two basic blocks if:
- block 1 jumps to label L
- block 2 starts with label L
- block 1 CJUMPs
- block 2 starts with the instruction after the CJUMP
Set representation

Represent sets of variables as bit vectors

\[ S \cup T = S | T \]
\[ S \setminus T = S \& \sim T \]

can implement to have good cache locality

- represent set as linked list of bit vectors, where each node fills a cache line
Complexity

O(N) nodes
O(N) variables

union takes O(N) time
for loop takes \(O(N^2)\) time
repeat loop runs \(O(N^2)\) times

\[\Rightarrow O(N^4) \text{ worst-case}\]

In practice: repeat loop runs 2-3 times, sets are sparse
\[\Rightarrow O(N) \text{ to } O(N^2) \text{ in practice}\]
Control-flow analysis to build CFG

Liveness analysis to compute live ranges for each temporary
- liveness analysis is a *dataflow analysis*
- dataflow analyses are also used for optimization

Use live ranges to construct interference graph

Register allocation to assign temporaries to fewest number of registers possible
Interference graphs

An undirected graph
- Node for each variable
- Edges connect variables whose live ranges overlap

Register allocation *colors* the interference graph
- one color per register
- no two adjacent nodes can have the same color
Questions?