Intermediate Representations

Akim Demaille Étienne Renault Roland Levillain
first.last@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

April 29, 2019

Intermediate Representations

1. Intermediate Representations
   - Compilers Structure
   - Intermediate Representations
   - Tree

2. Memory Management

3. Translation to Intermediate Language

4. The Case of the Tiger Compiler

5. lir: Low Level Intermediate Representation

Compilers Structure

So many ends...

Ends:

front end
: analysis

middle end
: generic synthesis

back end
: specific synthesis

The gcc team suggests

front end
: name ("a front end").

front-end
: adjective ("the front-end interface").

Front Ends...

The front end is dedicated to analysis:

- lexical analysis (scanning)
- syntactic analysis (parsing)
- ast generation
- static semantic analysis (type checking, context sensitive checks)
- source language specific optimizations
- hir generation

… Back Ends

The back end is dedicated to specific synthesis:

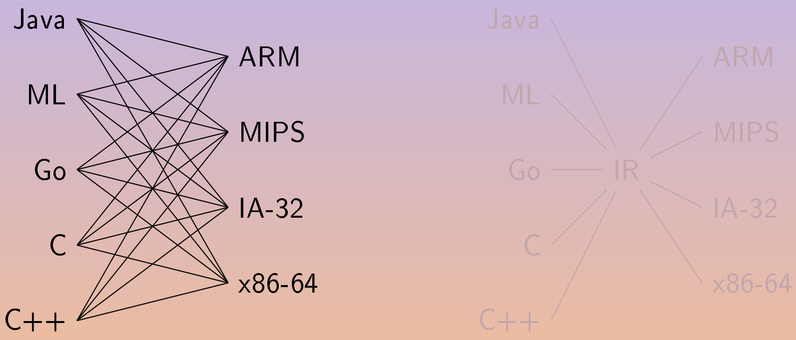
- instruction selection (mir to lir)
- register allocation
- assembly specific optimizations
- assembly code emission

… Middle Ends…

The middle end is dedicated to generic synthesis:

- stepwise refinement of hir to mir
- generic optimizations

Retargetable Compilers



Retargetable Compilers

Other Compiling Strategies

- Intermediate language-based strategy: SmartEiffel, GHC
- Bytecode strategy: Java bytecode (JVM), CIL (.NET)
- Hybrid approaches: GCJ (Java bytecode or native code)
- Retargetable optimizing back ends: MLRISC, VPO (Very Portable Optimizer), and somehow C-- (Quick C--).
- Modular systems: LLVM (compiler as a library, centered on a typed IR). Contains the LLVM core libraries, Clang, LLDB, etc. Also:
  - VMKit: a substrate for virtual machines (JVM, etc.).
  - Emscripten: an LLVM-to-JavaScript compiler. Enables C/C++ to JS compilation.

Intermediate Representations (IR) are fundamental.

1. Intermediate Representations
   - Compilers Structure
   - Intermediate Representations
   - Tree
2. Memory Management
3. Translation to Intermediate Language
4. The Case of the Tiger Compiler
5. lir: Low Level Intermediate Representation

Format? Representation? Language?

Intermediate representation:

- a faithful model of the source program
- "written" in an abstract language, the *intermediate language*
- may have an external syntax
- may be interpreted/compiled
  (havm, byte code)
- may have different levels
  (gcc's Tree is very much like C).

What Language Flavor?

- Imperative?
  - Stack Based? (Java Byte-code)
  - Register Based? (gcc's rtl, `tc`'s `Tree`)
- Functional?

  Most functional languages are compiled into a lower level language, eventually a simple λ-calculus.
- Other?

What Level?

A whole range of expressivities, typically aiming at making some optimizations easier:

- Keep array expressions?

  Yes: adequate for dependency analysis and related optimizations,

  No: Good for constant folding, strength reduction, loop invariant code motion, etc.

- Keep loop constructs?

What level of machine independence?

- Explicit register names?

Designing an Intermediate Representation

> *Intermediate-language design is largely an art, not a science.*
>
> — [Muchnick, 1997]

Different Levels [Muchnick, 1997]

```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

```
r1 <- [fp - 4]
r2 <- r1 + 2
r3 <- [fp - 8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp - 216
f1 <- [r7 + r6]
```

Different Levels [Muchnick, 1997]

```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

```
r1 <- [fp - 4]
r2 <- r1 + 2
r3 <- [fp - 8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp - 216
f1 <- [r7 + r6]
```

Different Levels [Muchnick, 1997]

```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

```
r1 <- [fp - 4]
r2 <- r1 + 2
r3 <- [fp - 8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp - 216
f1 <- [r7 + r6]
```

Different Levels [Muchnick, 1997]

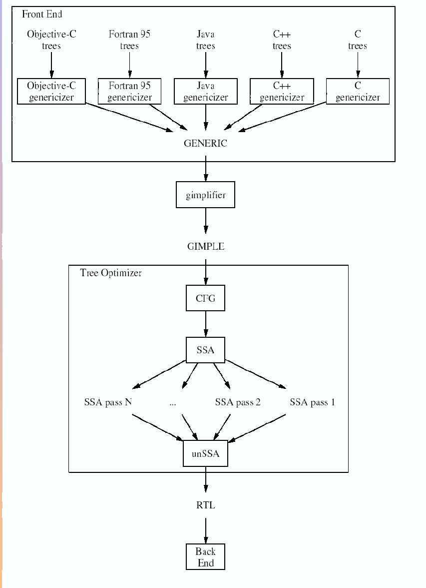
```
float a[20][10];
...
a[i][j+2];
```

```
t1 <- a[i,j+2]
```

```
t1 <- j + 2
t2 <- i * 20
t3 <- t1 + t2
t4 <- 4 * t3
t5 <- addr a
t6 <- t5 + t4
t7 <- *t6
```

```
r1 <- [fp - 4]
r2 <- r1 + 2
r3 <- [fp - 8]
r4 <- r3 * 20
r5 <- r4 + r2
r6 <- 4 * r5
r7 <- fp - 216
f1 <- [r7 + r6]
```

Different Levels: The GCC Structure



Stack Based: Java Byte-Code [Edwards, 2003]

```java
class Gcd
{
  static public int gcd(int a, int b)
  {
    while (a != b)
      {
        if (a > b)
          a -= b;
        else
          b -= a;
      }
    return a;
  }

  static public int main(String[] arg)
  {
    return gcd(12, 34);
  }
}
```

Stack Based: Java Byte-Code

```
% gcj-3.3 -c gcd.java
% jcf-dump-3.3 -c gcd
...
Method name:"gcd" public static
Signature: 5=(int,int)int
Attribute "Code", length:66,
max_stack:2, max_locals:2,
code_length:26
  0: iload_0
  1: iload_1
  2: if_icmpeq 24
  5: iload_0
  6: iload_1
  7: if_icmple 17
 10: iload_0
 11: iload_1
 12: isub
 13: istore_0
 14: goto 0
```

```
 17: iload_1
 18: iload_0
 19: isub
 20: istore_1
 21: goto 0
 24: iload_0
 25: ireturn
Attribute "LineNumberTable",
          length:22, count: 5
  line: 5 at pc: 0
  line: 7 at pc: 5
  line: 8 at pc: 10
  line: 10 at pc: 17
  line: 12 at pc: 24
...
```

Stack Based [Edwards, 2003]

Advantages

- Trivial translation of expressions
- Trivial interpreters
- No pressure on registers
- Often compact

Disadvantages

- Does not fit with today's architectures
- Hard to analyze
- Hard to optimize

Stack Based [Edwards, 2003]

Advantages

- Trivial translation of expressions
- Trivial interpreters
- No pressure on registers
- Often compact

Disadvantages

- Does not fit with today's architectures
- Hard to analyze
- Hard to optimize

Stack Based: Examples

ucode, used in hp pa-risk, and mips, was designed for stack evaluation (HP 3000 is stack based).

Today it is less adequate.

mips translates it back and forth to triples for optimization.

hp converts it into sllic (Spectrum Low Level ir) [Muchnick, 1997].

Register Based: tc's Tree

```
let function gcd(a: int, b: int) : int =
    (
      while a <> b
        do if a > b then a := a - b
                    else b := b - a;
      a
    )
in
  print_int(gcd(42, 51))
end
```

Register Based: tc's Tree (1/4)

```
/* == High Level Intermediate representation. == */
# Routine: gcd
label l0
# Prologue
move temp t0 temp fp
move temp fp temp sp
move
  temp sp
  binop sub temp sp const 12
move
mem temp fp
    temp i0
move
  mem binop add temp fp const -4
  temp i1
move
  mem binop add temp fp  const -8
  temp i2
```

Register Based: tc's Tree (2/4)

```
# Body
move temp rv
  eseq
  seq
    label l2
    cjump ne mem binop add temp fp const -4
             mem binop add temp fp const -8
             name l3 name l1
    label l3
    seq
      cjump gt mem binop add temp fp const -4
               mem binop add temp fp const -8
               name l4 name l5
      label l4
      move mem binop add temp fp const -4
           binop sub mem binop add temp fp const -4
                     mem binop add temp fp const -8
      jump name l6
```

Register Based: tc's Tree (3/4)

```
      label l5
      move mem binop add temp fp const -8
           binop sub mem binop add temp fp const -8
                     mem binop add temp fp const -4
      label l6
    seq end
    jump name l2
    label l1
  seq end
  mem  binop add temp fp const -4

# Epilogue
move temp sp temp fp
move temp fp temp t0
label end
```

Register Based: tc's Tree (4/4)

```
# Routine: _main
label main
# Prologue
# Body
seq
  sxp
    call
      name print_int
      call name l0 temp fp const 42 const 51
      call end
    call end
  sxp
    const 0
seq end
# Epilogue
label end
```

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).

- 2 address instructions? (*triples*)
- 3 address instructions? (*quadruples*)

Tree Expressions and instructions are unnamed, related to each other as nodes of trees

dag Compact, good for local value numbering, but that's all.

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).

- 2 address instructions? (*triples*)
- 3 address instructions? (*quadruples*)

Tree Expressions and instructions are unnamed, related to each other as nodes of trees

dag Compact, good for local value numbering, but that's all.

Register Based: What Structure?

How is the structure coded?

Addresses Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).

- 2 address instructions? (*triples*)
- 3 address instructions? (*quadruples*)

Tree Expressions and instructions are unnamed, related to each other as nodes of trees

dag Compact, good for local value numbering, but that's all.

Register Based: What Structure?

How is the structure coded?

Addresses
: Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).
  - 2 address instructions? (*triples*)
  - 3 address instructions? (*quadruples*)

Tree
: Expressions and instructions are unnamed, related to each other as nodes of trees

dag
: Compact, good for local value numbering, but that's all.

Register Based: What Structure?

How is the structure coded?

Addresses
: Expressions and instructions have names, or (absolute) addresses. (Stack based is a bit like a relative address).
  - 2 address instructions? (*triples*)
  - 3 address instructions? (*quadruples*)

Tree
: Expressions and instructions are unnamed, related to each other as nodes of trees

dag
: Compact, good for local value numbering, but that's all.

Quadruples vs. Triples [Muchnick, 1997]

```
L1:  i  <- i + 1
                              (1)  i + 1
                              (2)  i sto (1)
     t1 <- i + 1              (3)  i + 1
     t2 <- p + 4              (4)  p + 4
     t3 <- *t2                (5)  *(4)
     p  <- t2                 (6)  p sto (4)
     t4 <- t1 < 10            (7)  (3) < 10
     *r <- t3                 (8)  *r sto (5)
     if t4 goto L1            (9)  if (7), (1)
```

Quadruples vs. Triples [Muchnick, 1997]

```
L1:  i  <- i + 1

     t1 <- i + 1
     t2 <- p + 4
     t3 <- *t2
     p  <- t2
     t4 <- t1 < 10
     *r <- t3
     if t4 goto L1
```

```
(1)   i + 1
(2)   i sto (1)
(3)   i + 1
(4)   p + 4
(5)   *(4)
(6)   p sto (4)
(7)   (3) < 10
(8)   *r sto (5)
(9)   if (7), (1)
```

Register Based: gcc's rtl

```c
int
gcd(int a, int b)
{
  while (a != b)
    {
      if (a > b)
        a -= b;
      else
        b -= a;
    }
  return a;
}
```

Register Based: gcc's rtl

```
;; Function gcd
(note 1 0 2 ("gcd.c") 3)
(note 2 1 3 NOTE_INSN_DELETED)
(note 3 2 4 NOTE_INSN_FUNCTION_BEG)
(note 4 3 5 NOTE_INSN_DELETED)
(note 5 4 6 NOTE_INSN_DELETED)
(note 6 5 7 NOTE_INSN_DELETED)
(insn 7 6 8 (const_int 0 [0x0]) -1 (nil)
    (nil))
```

Register Based: gcc's rtl cont'd

```
(note 8 7 9 ("gcd.c") 4)
(note 9 8 40 NOTE_INSN_LOOP_BEG)
(note 40 9 10 NOTE_INSN_LOOP_CONT)
(code_label 10 40 13 2 "" "" [0 uses])
(insn 13 10 14 (set (reg:SI 59)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
     -1 (nil) (nil))
(insn 14 13 15 (set (reg:CCZ 17 flags)
        (compare:CCZ (reg:SI 59)
            (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32]))) -1 (nil)
    (nil))
```

Register Based: gcc's rtl cont'd

```
(jump_insn 15 14 16 (set (pc)
        (if_then_else (ne (reg:CCZ 17 flags)
                (const_int 0 [0x0]))
            (label_ref 18)
            (pc))) -1 (nil)
    (nil))
(jump_insn 16 15 17 (set (pc)
        (label_ref 44)) -1 (nil)
    (nil))
(barrier 17 16 18)
(code_label 18 17 19 4 "" "" [0 uses])
(note 19 18 20 NOTE_INSN_LOOP_END_TOP_COND)
(note 20 19 21 NOTE_INSN_DELETED)
(note 21 20 22 NOTE_INSN_DELETED)
```

Register Based: gcc's rtl cont'd

```
(note 22 21 25 ("gcd.c") 6)
(insn 25 22 26 (set (reg:SI 60)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
     -1 (nil) (nil))
(insn 26 25 27 (set (reg:CCGC 17 flags)
        (compare:CCGC (reg:SI 60)
            (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32]))) -1 (nil)
    (nil))
(jump_insn 27 26 28 (set (pc)
        (if_then_else (le (reg:CCGC 17 flags)
                (const_int 0 [0x0]))
            (label_ref 34)
            (pc))) -1 (nil)
    (nil))
```

Register Based: gcc's rtl cont'd

```
(note 28 27 30 ("gcd.c") 7)
(insn 30 28 31 (set (reg:SI 61)
        (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                (const_int 4 [0x4])) [0 b+0 S4 A32])) -1 (nil)
    (nil))
(insn 31 30 32 (parallel[
            (set (mem/f:SI (reg/f:SI 53 virtual-incoming-args)
                [0 a+0 S4 A32])
                (minus:SI (mem/f:SI (reg/f:SI 53 virtual-incoming-args)
                [0 a+0 S4 A32])
                    (reg:SI 61)))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (expr_list:REG_EQUAL (minus:SI (mem/f:SI (reg/f:SI 53
                                virtual-incoming-args) [0 a+0 S4 A32])
            (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32]))
        (nil)))
(jump_insn 32 31 33 (set (pc)
        (label_ref 39)) -1 (nil)
    (nil))
(barrier 33 32 34)
(code_label 34 33 35 5 "" "" [0 uses])
```

Register Based: gcc's rtl cont'd

```
(note 35 34 37 ("gcd.c") 9)
(insn 37 35 38 (set (reg:SI 62)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
     -1 (nil) (nil))
(insn 38 37 39 (parallel[
            (set (mem/f:SI (plus:SI (reg/f:SI 53 virtual-incoming-args)
                        (const_int 4 [0x4])) [0 b+0 S4 A32])
                (minus:SI (mem/f:SI (plus:SI (reg/f:SI 53
                                             virtual-incoming-args)
                            (const_int 4 [0x4])) [0 b+0 S4 A32])
                    (reg:SI 62)))
            (clobber (reg:CC 17 flags))
        ] ) -1 (nil)
    (expr_list:REG_EQUAL (minus:SI (mem/f:SI (plus:SI (reg/f:SI
                                            53 virtual-incoming-args)
                    (const_int 4 [0x4])) [0 b+0 S4 A32])
            (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
        (nil)))
(code_label 39 38 41 6 "" "" [0 uses])
(jump_insn 41 39 42 (set (pc)
        (label_ref 10)) -1 (nil)
    (nil))
(barrier 42 41 43)
(note 43 42 44 NOTE_INSN_LOOP_END)
```

Register Based: gcc's rtl cont'd

```
(note 45 44 46 ("gcd.c") 11)
(note 46 45 47 NOTE_INSN_DELETED)
(note 47 46 49 NOTE_INSN_DELETED)
(insn 49 47 51 (set (reg:SI 64)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32])) -1 (nil)
    (nil))
(insn 51 49 52 (set (reg:SI 58)
        (reg:SI 64)) -1 (nil)
    (nil))
(jump_insn 52 51 53 (set (pc)
        (label_ref 56)) -1 (nil)
    (nil))
(barrier 53 52 54)
(note 54 53 55 NOTE_INSN_FUNCTION_END)
(note 55 54 59 ("gcd.c") 12)
(insn 59 55 60 (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))
(insn 60 59 56 (clobber (reg:SI 58)) -1 (nil)
    (nil))
(code_label 56 60 58 1 "" "" [0 uses])
(insn 58 56 61 (set (reg/i:SI 0 eax)
        (reg:SI 58)) -1 (nil)
    (nil))
(insn 61 58 0 (use (reg/i:SI 0 eax)) -1 (nil)
```

Register Based [Edwards, 2003]

Advantages

- Suits today's architectures
- Clearer data flow

Disadvantages

- Harder to synthesize
- Less compact
- Harder to interpret

Register Based [Edwards, 2003]

Advantages

- Suits today's architectures
- Clearer data flow

Disadvantages

- Harder to synthesize
- Less compact
- Harder to interpret

Tree

Tree [Appel, 1998]

A simple intermediate language:

- Tree structure (no kidding...)
- Unbounded number of registers (temporaries)
- Two way conditional jump

Tree: Grammar

```
⟨Exp⟩  ::= "const" int
         | "name" ⟨Label⟩
         | "temp" ⟨Temp⟩
         | "binop" ⟨Oper⟩ ⟨Exp⟩ ⟨Exp⟩
         | "mem" ⟨Exp⟩
         | "call" ⟨Exp⟩ [{⟨Exp⟩}] "call end"
         | "eseq" ⟨Stm⟩ ⟨Exp⟩

⟨Stm⟩  ::= "move" ⟨Exp⟩ ⟨Exp⟩
         | "sxp" ⟨Exp⟩
         | "jump" ⟨Exp⟩ [{⟨Label⟩}]
         | "cjump" ⟨Relop⟩ ⟨Exp⟩ ⟨Exp⟩ ⟨Label⟩ ⟨Label⟩
         | "seq" [{⟨Stm⟩}] "seq end"
         | "label" ⟨Label⟩

⟨Oper⟩  ::= "add" | "sub" | "mul" | "div" | "mod"
⟨Relop⟩ ::= "eq" | "ne" | "lt" | "gt" | "le" | "ge"
```

Tree Samples

```
% echo '1 + 2 * 3' | tc -H -
/* == High Level Intermediate representation. == */
# Routine: Main Program
label Main
# Prologue
# Body
sxp
    binop add
        const 1
        binop mul
            const 2
            const 3
# Epilogue
label end
```

Tree Samples

```
% echo 'if 1 then print_int (1)' | tc -H -
# Routine: Main Program
label Main
# Prologue
# Body
seq
    cjump ne, const 1, const 0, name l1, name l2
    label l1
    sxp call name print_int, const 1
    jump name l3
    label l2
    sxp const 0
    label l3
seq end
# Epilogue
label end
```

Memory Management

Memory Management

Memory Hierarchy [Appel, 1998]

Different kinds of memory in a computer, with different performances:

Registers	Small memory units built on the cpu (bytes, 1 cycle)
L1 Cache	Last main memory access results (kB, 2-3 cycles)
L2 Cache	(MB, 10 cycles)
Memory	The usual ram (GB, 100 cycles)
Storage	Disks (100GB, TB, $>$ 1Mcycles)

Use the registers as much as possible.

Register Overflow

What if there are not enough registers? Use the main memory, but how? Recursion:

Without Each name is bound once. It can be statically allocated a single unit of main memory. (Cobol, Concurrent Pascal, Fortran (unless `recursive`)).

With A single name can be part of several concurrent bindings. Memory allocation must be dynamic.

Register Overflow

What if there are not enough registers? Use the main memory, but how?
Recursion:

Without
: Each name is bound once. It can be statically allocated a single unit of main memory. (Cobol, Concurrent Pascal, Fortran (unless `recursive`)).

With
: A single name can be part of several concurrent bindings. Memory allocation must be dynamic.

Dynamic Memory Allocation

Depending on the persistence, several models:

Global Global objects, whose liveness is equal to that of the program, are statically allocated
(e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function
(e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

- User Controlled
  - `malloc`/`free` (C), `new`/`dispose` (Pascal), `new`/`delete` (C++) etc.
- Garbage Collected
  - With or without `new`
  (`lisp`, Smalltalk, ML, Haskell, Tiger, Perl etc.).

Dynamic Memory Allocation

Depending on the persistence, several models:

Global Global objects, whose liveness is equal to that of the program, are statically allocated
(e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function
(e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

- **User Controlled**
  `malloc`/`free` (C), `new`/`dispose` (Pascal), `new`/`delete` (C++) etc.
- **Garbage Collected**
  With or without `new`
  (`lisp`, Smalltalk, ML, Haskell, Tiger, Perl etc.).

Dynamic Memory Allocation

Depending on the persistence, several models:

Global Global objects, whose liveness is equal to that of the program, are statically allocated
(e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function
(e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

- User Controlled
  - `malloc`/`free` (C), `new`/`dispose` (Pascal), `new`/`delete` (C++) etc.
- Garbage Collected
  - With or without `new`
    (`lisp`, Smalltalk, ML, Haskell, Tiger, Perl etc.).

Dynamic Memory Allocation

Depending on the persistence, several models:

Global Global objects, whose liveness is equal to that of the program, are statically allocated
(e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function
(e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

User Controlled

`malloc`/`free` (C), `new`/`dispose` (Pascal),
`new`/`delete` (C++) etc.

Garbage Collected

With or without `new`
(`lisp`, Smalltalk, ML, Haskell, Tiger, Perl etc.).

Dynamic Memory Allocation

Depending on the persistence, several models:

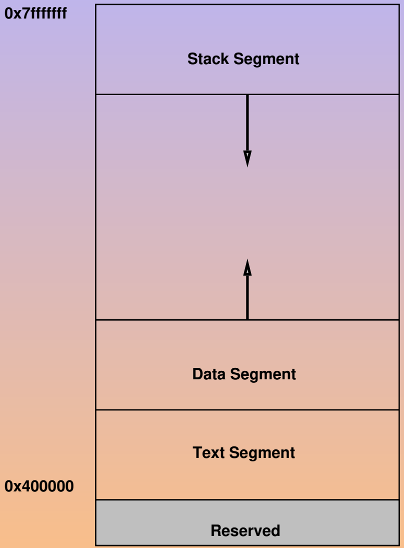
Global Global objects, whose liveness is equal to that of the program, are statically allocated (e.g., `static` variables in C)

Automatic Liveness is bound to that of the host function (e.g., `auto` variables in C)

Heap Liveness is independent of function liveness:

- **User Controlled**
  `malloc`/`free` (C), `new`/`dispose` (Pascal), `new`/`delete` (C++) etc.
- **Garbage Collected**
  With or without `new` (`lisp`, Smalltalk, `ML`, Haskell, Tiger, Perl etc.).

spim Memory Model [Larus, 1990]



Stack Management

Function calls is a last-in first-out process, hence, it is properly represented by a stack.

Or...

"Call tree": the complete history of calls.
The execution of the program is its depth first traversal.
Depth-first walk requires a stack.

Activation Blocks

Activation Blocks

- In recursive languages, a single routine can be "opened" several times concurrently.
- An *activation* designates one single instance of execution.
- Automatic variables are bound to the liveness of the activation.
- Their location is naturally called *activation block*, or *stack frame*.

Activation Blocks

- In recursive languages, a single routine can be "opened" several times concurrently.
- An *activation* designates one single instance of execution.
- Automatic variables are bound to the liveness of the activation.
- Their location is naturally called *activation block*, or *stack frame*.

Activation Blocks

- In recursive languages, a single routine can be "opened" several times concurrently.
- An *activation* designates one single instance of execution.
- Automatic variables are bound to the liveness of the activation.
- Their location is naturally called *activation block*, or *stack frame*.

Activation Blocks

- In recursive languages, a single routine can be “opened” several times concurrently.
- An *activation* designates one single instance of execution.
- Automatic variables are bound to the liveness of the activation.
- Their location is naturally called *activation block*, or *stack frame*.

Activation Blocks Contents

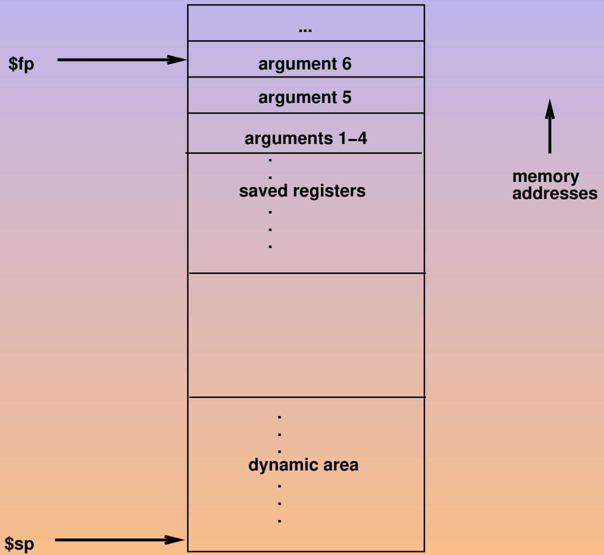
Data to store on the stack:

arguments	incoming
local variables	user automatic variables
return address	where to `return`
saved registers	the caller's environment to restore
temp	compiler automatic variables, spills
static link	when needed

Activation Blocks Layout

The layout is suggested by the constructor.
Usually the layout is from earliest known, to latest.

Activation Blocks Layout on mips [Larus, 1990]



Frame & Stack Pointers

The stack of activation blocks is implemented as an array with

frame pointer
: the inner frontier of the activation block

stack pointer
: the outer frontier

Usually the stack is represented growing towards the bottom.

Flexible Automatic Memory

`auto` Static size, automatic memory.

`malloc` Dynamic size, persistent memory.

Automatic memory is extremely convenient...

```
int
open2(char* str1, char* str2, int flags, int mode)
{
  char name[strlen(str1) + strlen(str2) + 1];
  stpcpy(stpcpy(name, str1), str2);
  return open(name, flags, mode);
}
```

Flexible Automatic Memory

malloc is a poor replacement.

```
int
open2(char* str1, char* str2, int flags, int mode)
{
  char* name
    = (char*) malloc(strlen(str1) + strlen(str2) + 1);
  if (name == 0)
    fatal("virtual memory exceeded");
  stpcpy(stpcpy(name, str1), str2);
  int fd = open(name, flags, mode);
  free(name);
  return fd;
}
```

Flexible Automatic Memory

`alloca` is a good replacement.

```
int
open2(char *str1, char *str2, int flags, int mode)
{
  char *name
    = (char *) alloca(strlen(str1) + strlen(str2) + 1);
  stpcpy(stpcpy(name, str1), str2);
  return open(name, flags, mode);
}
```

Advantages of alloca [Loosemore et al., 2003]

- Using `alloca` wastes very little space and is very fast.
  (It is open-coded by the GNU C compiler.)
- `alloca` does not cause memory fragmentation.
  Since `alloca` does not have separate pools for different sizes of block, space used for any size block can be reused for any other size.
- Automatically freed.
  Nonlocal exits done with `longjmp` automatically free the space allocated with `alloca` when they exit through the function that called `alloca`. This is the most important reason to use `alloca`.

Advantages of `alloca` [Loosemore et al., 2003]

- Using `alloca` wastes very little space and is very fast.
  (It is open-coded by the GNU C compiler.)
- `alloca` does not cause memory fragmentation.
  Since `alloca` does not have separate pools for different sizes of block, space used for any size block can be reused for any other size.
- Automatically freed.
  Nonlocal exits done with `longjmp` automatically free the space allocated with `alloca` when they exit through the function that called `alloca`. This is the most important reason to use `alloca`.

Advantages of `alloca` [Loosemore et al., 2003]

- Using `alloca` wastes very little space and is very fast.
  (It is open-coded by the GNU C compiler.)
- `alloca` does not cause memory fragmentation.
  Since `alloca` does not have separate pools for different sizes of block, space used for any size block can be reused for any other size.
- Automatically freed.
  Nonlocal exits done with `longjmp` automatically free the space allocated with `alloca` when they exit through the function that called `alloca`. This is the most important reason to use `alloca`.

Disadvantages of `alloca` [Loosemore et al., 2003]

- If you try to allocate more memory than the machine can provide, you don't get a clean error message.
  Instead you get a fatal signal like the one you would get from an infinite recursion; probably a segmentation violation.
- Some non-GNU systems fail to support `alloca`, so it is less portable. However, a slower emulation of `alloca` written in C is available for use on systems with this deficiency.

Disadvantages of `alloca` [Loosemore et al., 2003]

- If you try to allocate more memory than the machine can provide, you don't get a clean error message.
  Instead you get a fatal signal like the one you would get from an infinite recursion; probably a segmentation violation.
- Some non-GNU systems fail to support `alloca`, so it is less portable. However, a slower emulation of `alloca` written in C is available for use on systems with this deficiency.

Arrays vs. Alloca [Loosemore et al., 2003]

- A variable size array's space is freed at the end of the scope of the name of the array.
  The space allocated with `alloca` remains until the end of the function.
- It is possible to use `alloca` within a loop, allocating an additional block on each iteration.
  This is impossible with variable-sized arrays.

Arrays vs. Alloca [Loosemore et al., 2003]

- A variable size array's space is freed at the end of the scope of the name of the array.
  The space allocated with `alloca` remains until the end of the function.
- It is possible to use `alloca` within a loop, allocating an additional block on each iteration.
  This is impossible with variable-sized arrays.

Implementing Dynamic Arrays & Alloca

- Playing with `$sp` which makes `$fp` mandatory.
- An additional stack (as with the C emulation of `alloca`).

Nonlocal Variables

escapes-n-recursion

```
let function trace(fn: string, val: int) =
      (print(fn); print("("); print_int(val); print(") "))

    function one(input : int) =
    let function two() =
        (trace("two", input); one(input - 1))
    in
      if input > 0 then
        (two(); trace("one", input))
    end
in
  one(3); print("\n")
end
```

escapes-n-recursion

```
let function trace(fn: string, val: int) =
      (print(fn); print("("); print_int(val); print(") "))

    function one(input : int) =
    let function two() =
        (trace("two", input); one(input - 1))
    in
      if input > 0 then
        (two(); trace("one", input))
    end
in
  one(3); print("\n")
end
```

```
% tc -H escapes-n-recursion.tig > f.hir && havm f.hir
```

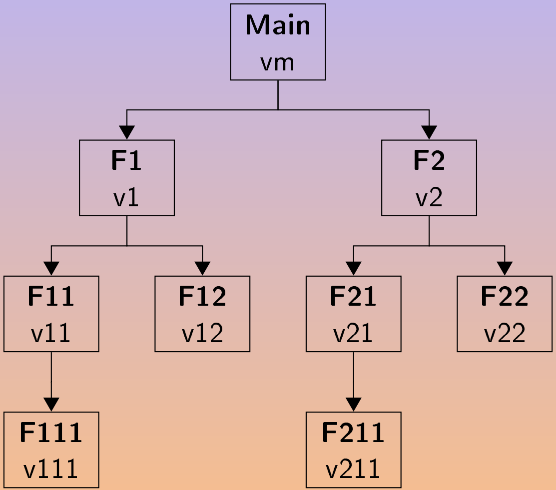
escapes-n-recursion

```
let function trace(fn: string, val: int) =
      (print(fn); print("("); print_int(val); print(") "))

    function one(input : int) =
    let function two() =
        (trace("two", input); one(input - 1))
    in
      if input > 0 then
        (two(); trace("one", input))
    end
in
  one(3); print("\n")
end
```

```
% tc -H escapes-n-recursion.tig > f.hir && havm f.hir
two(3) two(2) two(1) one(1) one(2) one(3)
```

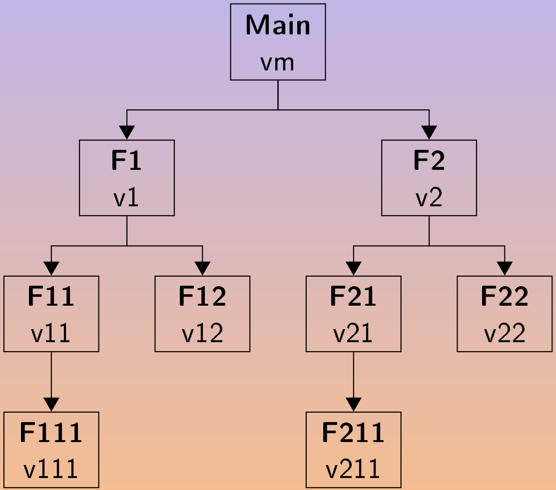
Deep Static Function Hierarchies



What if:

1. `Main uses vm`
2. `Main calls F1`
3. `F1 uses v1`
4. `F1 uses vm`, non local
5. `F1 calls F11`
6. `F11 uses v11`
7. `F11 uses v1`
8. `F11 uses vm`
9. `F11 calls F12`
10. `F12 calls F1`

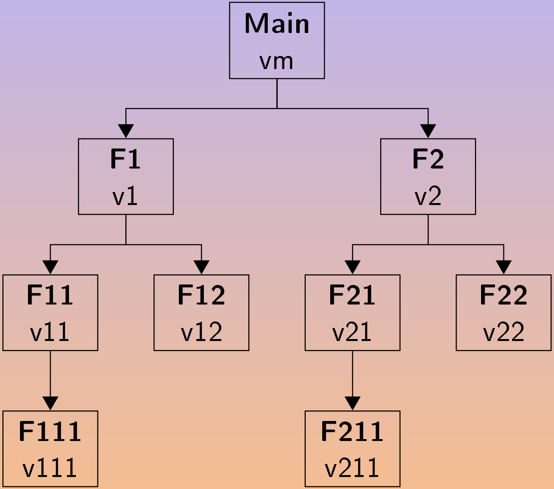
Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

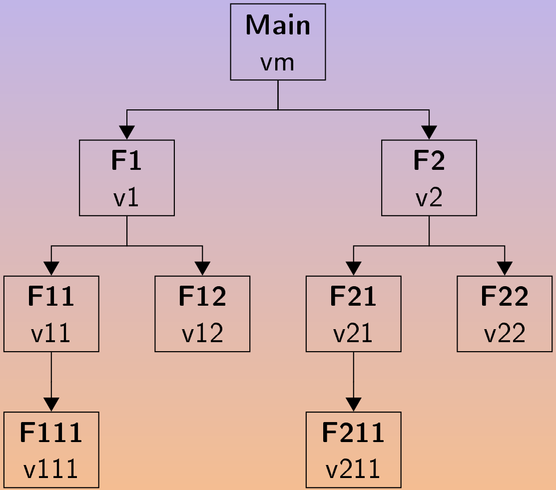
Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

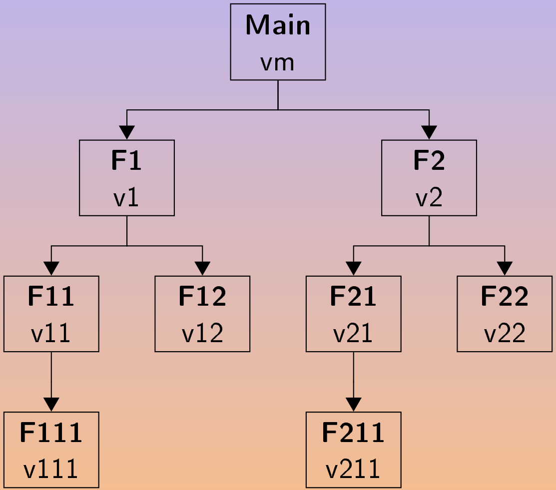
Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

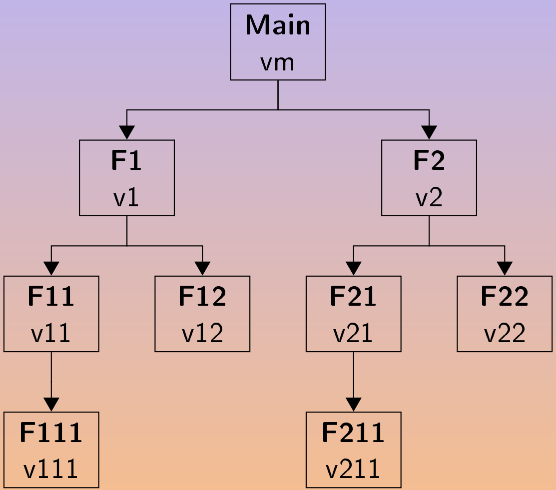
Deep Static Function Hierarchies



What if:

1. `Main uses vm`
2. `Main calls F1`
3. `F1 uses v1`
4. `F1 uses vm`, non local
5. `F1 calls F11`
6. `F11 uses v11`
7. `F11 uses v1`
8. `F11 uses vm`
9. `F11 calls F12`
10. `F12 calls F1`

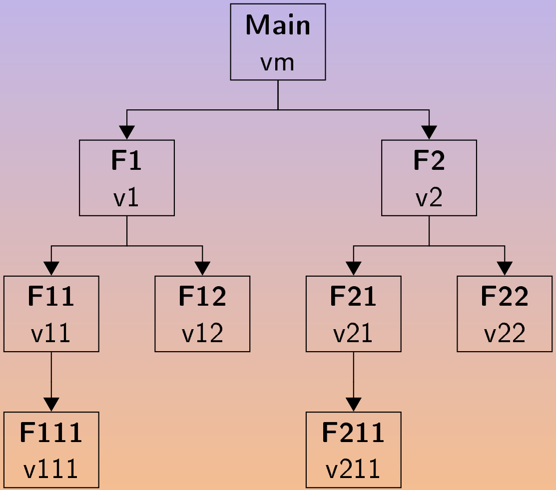
Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

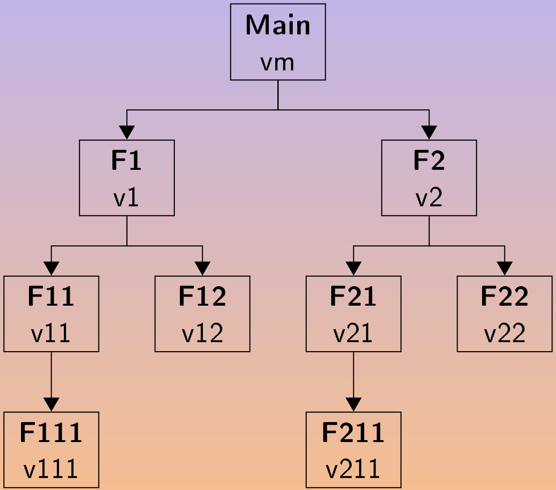
Deep Static Function Hierarchies



What if:

1. `Main uses vm`
2. `Main calls F1`
3. `F1 uses v1`
4. `F1 uses vm`, non local
5. `F1 calls F11`
6. `F11 uses v11`
7. `F11 uses v1`
8. `F11 uses vm`
9. `F11 calls F12`
10. `F12 calls F1`

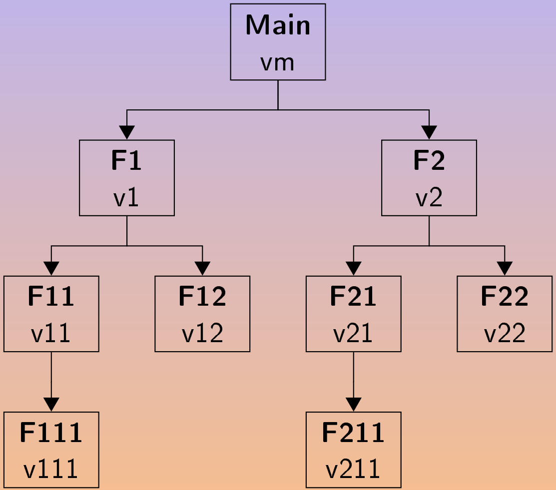
Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

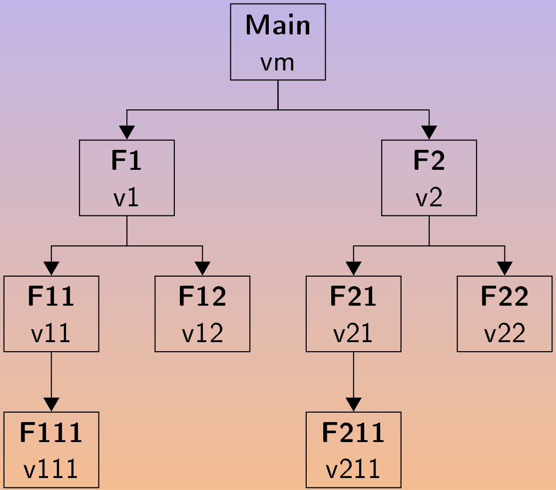
Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

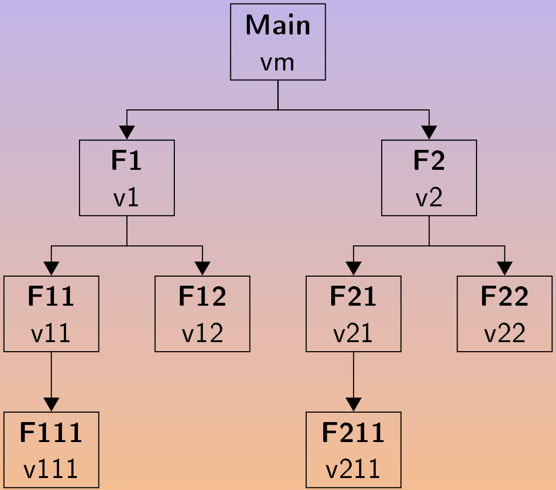
Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

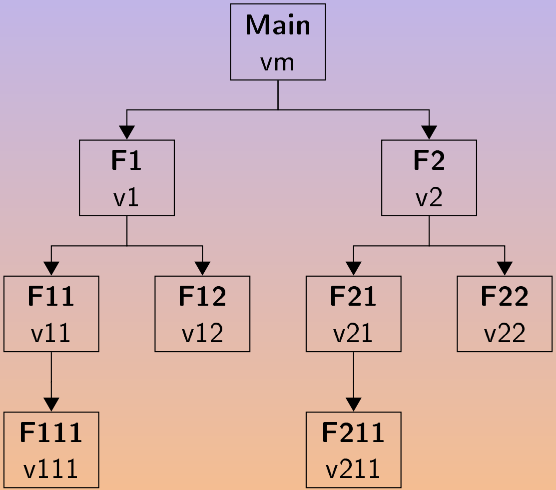
Deep Static Function Hierarchies



What if:

1. `Main uses vm`
2. `Main calls F1`
3. `F1 uses v1`
4. `F1 uses vm`, non local
5. `F1 calls F11`
6. `F11 uses v11`
7. `F11 uses v1`
8. `F11 uses vm`
9. `F11 calls F12`
10. `F12 calls F1`

Deep Static Function Hierarchies



What if:

1. Main uses vm
2. Main calls F1
3. F1 uses v1
4. F1 uses vm, non local
5. F1 calls F11
6. F11 uses v11
7. F11 uses v1
8. F11 uses vm
9. F11 calls F12
10. F12 calls F1

Deep Static Function Hierarchies

The caller must provide the callee with its static link.

Caller	Callee	Static Link
Main	F1	$\texttt{fp}_{\texttt{Main}}$ = fp
F1	F11	$\texttt{fp}_{\texttt{F1}}$ = fp
F11	F12	$\texttt{fp}_{\texttt{F1}}$ = $\texttt{sl}_{\texttt{F11}}$ = $*\texttt{fp}_{\texttt{F11}}$ = *fp
F12	F2	$\texttt{fp}_{\texttt{Main}}$ = $\texttt{sl}_{\texttt{F1}}$ = $*\texttt{sl}_{\texttt{F12}}$ = $**\texttt{fp}_{\texttt{F12}}$ = **fp
F2	F22	$\texttt{fp}_{\texttt{F2}}$ = fp
F22	F11	$\texttt{fp}_{\texttt{F1}}$ = ???

Assuming that the static link is stored at fp.

Higher Order Functions

```
let
  function addgen (a: int) : int -> int =
    let
      function res (b: int) : int =
        a + b
    in
      res
    end
  var add50 := addgen (50)
in
  add50 (1)
end
```

Translation to Intermediate Language

Calling Conventions

Calling Conventions at hir Level

You must:

- Preserve *some* registers (`fp`, `sp`)
- Allocate the frame
- Handle the static link (`i0`)
- Receive the (other) arguments (`i1`, `i2`...)

You don't:

- Save temporaries (`havm` has magic for recursion)
- Jump to the `ra` (this is not nice feature from `havm`)

havm Calling Conventions

```
let function gcd (a: int, b: int) : int = (...)
in print_int (gcd (42, 51)) end
```

```
# Routine: gcd
label l0
# Prologue
move temp t2, temp fp
move temp fp, temp sp
move temp sp, temp sp - const 4
move mem temp fp, temp i0
move temp t0, temp i1
move temp t1, temp i2
```

```
# Body
move temp rv
     eseq
       ...
       temp t0
# Epilogue
move temp sp, temp fp
move temp fp, temp t2
label end
# Routine: Main Program
label Main
sxp call name print_int
         call name l0 temp fp
              const 42 const 51
label end
```

Clever Translations

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1.
```
cjump (α < β, ltrue, lfalse)
```

2.
```
eseq (seq (cjump (α < β, ltrue, lfalse),
           label ltrue
           move temp t, const 1
           jump lend
           label lfalse
           move temp t, const 0
           label lend),
      temp t)
```

3.
```
seq (sxp (α)
     sxp (β))
```

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
             label ltrue
             move temp t, const 1
             jump lend
             label lfalse
             move temp t, const 0
             label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. cjump (α < β, ltrue, lfalse)

2. eseq (seq (cjump (α < β, ltrue, lfalse),
```
               label ltrue
               move temp t, const 1
               jump lend
               label lfalse
               move temp t, const 0
               label lend),
          temp t)
```

3. seq (sxp (α)
   sxp (β))

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α $<$ β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2.
```
eseq (seq (cjump (α < β, ltrue, lfalse),
           label ltrue
           move temp t, const 1
           jump lend
           label lfalse
           move temp t, const 0
           label lend),
      temp t)
```

3.
```
seq (sxp (α)
     sxp (β))
```

It depends on the *use*:

1. `if` $\alpha < \beta$ `then ...`
2. `a :=` $\alpha < \beta$
3. `(`$\alpha < \beta$`, ()).`

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
              label ltrue
              move temp t, const 1
              jump lend
              label lfalse
              move temp t, const 0
              label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

It depends on the *use*:

1. `if` $\alpha < \beta$ `then ...`
2. `a :=` $\alpha < \beta$
3. `(`$\alpha < \beta$`, ()).`

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. cjump (α < β, ltrue, lfalse)

2. eseq (seq (cjump (α < β, ltrue, lfalse),
```
             label ltrue
             move temp t, const 1
             jump lend
             label lfalse
             move temp t, const 0
             label lend),
        temp t)
```

3. seq (sxp (α)
```
       sxp (β))
```

It depends on the *use*:

1. if $\alpha < \beta$ then ...
2. a := $\alpha < \beta$
3. ($\alpha < \beta$, ()).

Translating Conditions

What is the right translation for $\alpha < \beta$, with α and β two arbitrary expressions?

1. `cjump (`α `<` β`, ltrue, lfalse)`

2. ```
   eseq (seq (cjump (α < β, ltrue, lfalse),
                label ltrue
                move temp t, const 1
                jump lend
                label lfalse
                move temp t, const 0
                label lend),
         temp t)
   ```

3. ```
   seq (sxp (α)
        sxp (β))
   ```

It depends on the *use*:

1. `if` $\alpha < \beta$ `then ...`
2. `a :=` $\alpha < \beta$
3. `(`$\alpha < \beta$`, ())`.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Context Sensitive Translation

- The right translation depends upon the *use*.
  This is context sensitive!
- How to implement this?
  - When entering an `IfExp`, warn "I want a condition",
  - then, depending whether it is an expression or a statement, warn "I want an expression" or "I want a statement".
- Don't forget to preserve the demands of higher levels...
- Eek.

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

`Exp`	un_nx	un_ex	un_cx (t, f)
Ex(e)			
Cx(a < b)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

`Exp`	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)		
Cx($a < b$)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

`Exp`	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	
Cx($a < b$)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

`Exp`	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)			
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- Ex Expression shell, encapsulation of a proto value,
- Nx Statement shell, encapsulating a wannabe statement,
- Cx Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)	seq(sxp(a), sxp(b))		
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

`Exp`	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)	seq(sxp(a), sxp(b))	eseq($t \leftarrow (a < b)$, t)	
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- Ex Expression shell, encapsulation of a proto value,
- Nx Statement shell, encapsulating a wannabe statement,
- Cx Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

`Exp`	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$(a $<$ b), t)	cjump(a $<$ b, t, f)
Nx(s)			

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

`Exp`	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$(a $<$ b), t)	cjump(a $<$ b, t, f)
Nx(s)	s		

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

Exp	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump($e \neq 0$, t, f)
Cx($a < b$)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$($a < b$), t)	cjump($a < b$, t, f)
Nx(s)	s	???	

Prototranslation, Expression Shells

Rather, delay the translation until the use is known (`translate::Exp`):

- **Ex** Expression shell, encapsulation of a proto value,
- **Nx** Statement shell, encapsulating a wannabe statement,
- **Cx** Condition shell, encapsulating a wannabe condition.

Then, ask them to finish their translation according to the use:

`Exp`	un_nx	un_ex	un_cx (t, f)
Ex(e)	sxp(e)	e	cjump(e $\neq$ 0, t, f)
Cx(a $<$ b)	seq(sxp(a), sxp(b))	eseq(t $\leftarrow$(a $<$ b), t)	cjump(a $<$ b, t, f)
Nx(s)	s	???	???

if 11 < 22 | 22 < 33 then print_int(1) else print_int(0)

```
cjump ne
    eseq seq cjump 11 < 22 name l0 name l1
               label l0  move temp t0 const 1
                         jump name l2
               label l1  move temp t0
                         eseq seq move temp t1 const 1
                                  cjump 22 < 33 name l3 name l4
                                  label l4
                                  move temp t1 const 0
                                  label l3
                              seq end
                              temp t1
                        jump name l2
               label l2
         seq end
         temp t0
    const 0
    name l5
    name l6
label l5   sxp call name print_int const 1
           jump name l7
label l6   sxp call name print_int const 0
           jump name l7
label l7
```

A Better Translation: Ix

```
  seq
    cjump l1 < 22 name l3 name l4
    label l3
      cjump 1 <> 0 name l0 name l1
    label l4
      cjump 22 < 33 name l0 name l1
  seq end
label l0
  sxp call name print_int const 1
  jump name l2
label l1
  sxp call name print_int const 0
label l2
```

Complex Expressions

Complex Expressions

- Array creation
- Record creation
- String comparison
- While loops
- For loops

While Loops

```
while condition
  do body
```

While Loops

```
while condition
  do body
```

```
test:
  if not (condition)
    goto done
  body
  goto test
done:
```

For Loops

```
for i := min to max
  do body
```

```
let i := min
    limit := max
in
  while i <= limit
  do
    (body; ++i)
end
```

For Loops

```
for i := min to max
  do body
```

```
let i := min
    limit := max
in
  if (i > limit)
    goto end
loop:
    body
    if (i >= limit)
      goto end
    ++i
    goto loop
end:
```

Additional Features

- Bounds checking
- Nil checking
- ...

The Case of the Tiger Compiler

Translation in the Tiger Compiler

Actors: The `temp` Module

`temp::Temp`	*temporaries* are pseudo-registers. Generation of fresh temporaries.
`temp::Label`	Pseudo addresses, both for data and code. Generation of fresh labels.
`misc::endo_map<T>`	Mapping from `T` to `T`. Used during register allocation.

Actors: The `tree` Module

Implementation of hir and lir.

```
/Tree/  /Exp/   Const (int)
                Name  (temp::Label)
                Temp  (temp::Temp)
                Binop (Oper, Exp, Exp)
                Mem   (Exp)
                Call  (Exp, list<Exp*>)
                Eseq  (Stm, Exp)
        /Stm/   Move  (Exp, Exp)
                Sxp   (Exp)
                Jump  (Exp, list<temp::Label>)
                CJump (Relop, Exp, Exp, Label, Label)
                Seq   (list<Stm *>)
                Label (temp::Label)
```

Actors: The `tree` Module: Warnings

- `temp::Temp` is not `tree::Temp`.
  The latter aggregates one of the former.
  Similarly with `Label`.
- n-ary `seq`.
  (Unlike [Appel, 1998]).
- `Sxp` instead of `Exp`.

Actors: The `frame` Module

`Access` How to reach a "variable".
Abstract class with two concrete subclasses.

`frame::In_Register`

`frame::In_Frame`

`Frame` What "variables" a frame contains.

`local_alloc(bool escapes_p) -> Access`

Frames and (`frame::`) accesses are not aware of static links.

Actors: The `translate` Module

`Access` Static link aware version of `frame::Access`:
how to reach a variable, including non local: a `frame::Access` and a `translate::Level`.

`exp(Level use) -> Exp` Tree expression
The location of `this` Access, from the use point of view.

`Level` Static link aware version of `frame::Frame`:
what variables a frame contains, and *where is its parent level*.

`fp(Level use) -> Exp` Tree expression
The frame pointer of `this` Level, from the use point of view.
Used for `calls`, and reaching frame resident temporaries.

`translate::Exp`
: Prototranslation wrappers (`Ex`, `Nx`, `Cx`, `Ix`).

`translate/translation.hh`
: Auxiliary functions used by the `Translator`.

`translate::Translator`
: The translator.

lir: Low Level Intermediate Representation

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no “expression”.
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide “jump or continue” instructions.
- Limited Number of Registers
  From temps to actual registers.

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no "expression".
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide "jump or continue" instructions.
- Limited Number of Registers
  From temps to actual registers.

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no "expression".
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide "jump or continue" instructions.
- Limited Number of Registers
  From temps to actual registers.

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no "expression".
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide "jump or continue" instructions.
- Limited Number of Registers
  From temps to actual registers.

Inadequacy of hir

hir constructs not supported in assembly complicate the back end:

- Structure
  No nested sequences.
- Expressions
  Assembly is imperative: there is no “expression”.
- Calling Conventions
  A (high-level) call is a delicate operation, not a simple instruction.
- Two Way Conditional Jumps
  Machines provide “jump or continue” instructions.
- Limited Number of Registers
  From `temps` to actual registers.

Linearization: Principle

- eseq and seq must be eliminated (except the outermost seq).
- Similar to cut-elimination: permute inner eseq and seq to lift them higher, until they vanish.
- A simple rewriting system.

```
eseq (s1, eseq (s2, e)) ⇝ eseq (seq (s1, s2), e)
sxp (eseq (s, e))       ⇝ seq (s, sxp (e))
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3)
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es)
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
binop (+, eseq (s, e1), e2)
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
binop (+, eseq (s, e1), e2) ⇝ eseq (s, binop (+, e1, e2))
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
binop (+, eseq (s, e1), e2) ⇝ eseq (s, binop (+, e1, e2))
binop (+, e1, eseq (s, e2))
```

Linearization: More Rules

```
seq (ss1, seq (ss2), ss3) ⇝ seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) ⇝ eseq (s, call (f, e, es))
binop (+, eseq (s, e1), e2) ⇝ eseq (s, binop (+, e1, e2))
binop (+, e1, eseq (s, e2)) ⇝ eseq (s, binop (+, e1, e2))
```

Linearization: Incorrect Changes

```
binop (+, e1, eseq (s, e2)) ⇝ eseq (s, binop (+, e1, e2))
```

- But what if s modifies the value of e1?

```
binop (+, temp t,
          eseq (move (temp t, const 42),
                const 0))
  ⇝ eseq (move (temp t, const 42),
          binop (+, temp t, const 0))
```

- This transformation is invalid: it changes the semantics.
- How can it be solved?

Linearization: Incorrect Changes

`binop (+, e1, eseq (s, e2))` $\rightsquigarrow$ `eseq (s, binop (+, e1, e2))`

- But what if s modifies the value of e1?

  ```
  binop (+, temp t,
            eseq (move (temp t, const 42),
                  const 0))
    ~> eseq (move (temp t, const 42),
             binop (+, temp t, const 0))
  ```

- This transformation is invalid: it changes the semantics.
- How can it be solved?

Linearization: Incorrect Changes

```
binop (+, e1, eseq (s, e2)) ⇝ eseq (s, binop (+, e1, e2))
```

- But what if s modifies the value of e1?

  ```
  binop (+, temp t,
              eseq (move (temp t, const 42),
                    const 0))
    ⇝ eseq (move (temp t, const 42),
             binop (+, temp t, const 0))
  ```

- This transformation is invalid: it changes the semantics.
- How can it be solved?

Linearization: Incorrect Changes

t + (t := 42, 0)

```
binop (+,
       temp t,
       eseq (move (temp t, const 42),
             const 0))
```

Wrong

```
eseq (move (temp t,
            const 42),
      binop (+,
             temp t,
             const 0))
```

Linearization: Incorrect Changes

$t + (t := 42, 0)$

```
binop (+,
       temp t,
       eseq (move (temp t, const 42),
             const 0))
```

Wrong

```
eseq (move (temp t,
            const 42),
      binop (+,
             temp t,
             const 0))
```

Linearization: Incorrect Changes

t + (t := 42, 0)

```
binop (+,
       temp t,
       eseq (move (temp t, const 42),
             const 0))
```

Wrong

```
eseq (move (temp t,
            const 42),
      binop (+,
             temp t,
             const 0))
```

Right

```
eseq (seq (move (temp t0, temp t)
           move (temp t, const 42)),
      binop (+,
             temp t0,
             const 0))
```

Linearization: More Temporaries

- When "de-expressioning" fresh temporaries are needed

```
binop (+, e1, eseq (s, e2))
 ⇝ eseq (seq (move (temp t, e1), s),
         binop (+, temp t, e2))
```

- More generally

```
call (f, es1, eseq (s, e), es2)
 ⇝ eseq (seq (move (temp t1, e1),
              move (temp t2, e2),
              move (temp t3, e3),
              ...,
              s),
         call (f, ts, e, es2))
```

- This is extremely inefficient when not needed. . .

Linearization: Commutativity

- Save useless extra temporaries and moves.
- Problem: commutativity cannot be known statically.
  E.g., `move (mem (t1), e)` and `mem (t2)`
  commute iff `t1` $\neq$ `t2`.
- We need a *conservative* approximation,
  i.e., never say "commute" when they don't.
  E.g., "if `e` is a `const` then `s` and `e` definitely commute".

Linearization: Commutativity

- Save useless extra temporaries and moves.
- Problem: commutativity cannot be known statically.
  E.g., `move (mem (t1), e)` and `mem (t2)`
  commute iff `t1` $\neq$ `t2`.
- We need a *conservative* approximation,
  i.e., never say "commute" when they don't.
  E.g., "if `e` is a `const` then `s` and `e` definitely commute".

Linearization: Commutativity

- Save useless extra temporaries and moves.
- Problem: commutativity cannot be known statically.
  E.g., `move (mem (t1), e)` and `mem (t2)`
  commute iff `t1` $\neq$ `t2`.
- We need a *conservative* approximation,
  i.e., never say "commute" when they don't.
  E.g., "if `e` is a `const` then `s` and `e` definitely commute".

Call Normalization

Normalization of a `call` depends on the kind of the routine:

procedure
: then its parent must be an `sxp`

function
: then its parent must be a `move (temp t, .)`

This normalization is performed simultaneously with linearization.

Two Way Jumps

Obviously, to enable the translation of a `cjump` into actual assembly instructions, the "false" label must follow the `cjump`.
How?

Two Way Jumps: Basic Blocks

Split the long outer `seq` into "basic blocks":

- a single entry: the first instruction
- a single (maybe multi-) exit: the last instruction

It may require

- a new label as first instruction, to which the prologue `jumps`
- new labels after `jumps` or `cjumps`
- a new `jump` from the last instruction to the epilogue.

Two Way Jumps: Traces

Start from the initial block, and "sew" each remaining basic block to this growing "trace".

- If the last instruction is a `jump`
  - if the "destination block" is available, add it
  - otherwise, fetch any other remaining block.
- If the last instruction is a `cjump`
  - If the false destination is available, push it
  - If the true destination is available, flip the `cjump` and push it,
  - otherwise, change the `cjump` to go to a fresh label, attach this label, and finally `jump` to the initial false destination.

Two Way Jumps: Optimizing Traces

Many jumps should be removable, but sometimes there are choices to make.

```
label prologue
  Prologue.
jump name test
```

```
label test
cjump i <= N, body, done
```

```
label body
  Body.
jump name test
```

```
label done
  Epilogue
jump name end
```

Two Way Jumps: Optimizing Traces

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i > N,
   done, body
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i <= N,
   body, done
```

```
label done
  Epilogue
jump name end
```

```
label body
  Body
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label body
  Body
jump name test
```

```
label test
cjump i <= N,
  body, done
```

```
label done
  Epilogue
jump name end
```

Two Way Jumps: Optimizing Traces

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i > N,
   done, body
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i <= N,
   body, done
```

```
label done
  Epilogue
jump name end
```

```
label body
  Body
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label body
  Body
jump name test
```

```
label test
cjump i <= N,
   body, done
```

```
label done
  Epilogue
jump name end
```

Two Way Jumps: Optimizing Traces

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i > N,
   done, body
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i <= N,
   body, done
```

```
label done
  Epilogue
jump name end
```

```
label body
  Body
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label body
  Body
jump name test
```

```
label test
cjump i <= N,
  body, done
```

```
label done
  Epilogue
jump name end
```

Two Way Jumps: Optimizing Traces

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i > N,
   done, body
```

```
label body
  Body
jump name test
```

```
label done
  Epilogue
jump name end
```

```
label prologue
  Prologue
jump name test
```

```
label test
cjump i <= N,
   body, done
```

```
label done
  Epilogue
jump name end
```

```
label body
  Body
jump name test
```

```
label prologue
  Prologue
jump name test
```

```
label body
  Body
jump name test
```

```
label test
cjump i <= N,
  body, done
```

```
label done
  Epilogue
jump name end
```

Bibliography I

Appel, A. W. (1998).
Modern Compiler Implementation in C, Java, ML.
Cambridge University Press.

Edwards, S. (2003).
COMS W4115 Programming Languages and Translators.
`http://www.cs.columbia.edu/~sedwards/classes/2003/w4115/`.

Larus, J. R. (1990).
SPIM S20: A MIPS R2000 simulator.
Technical Report TR966, Computer Sciences Department, University of Wisconsin–Madison.

Bibliography II

Loosemore, S., Stallman, R. M., McGrath, R., Oram, A., and Drepper, U. (2003).
The GNU C Library Reference Manual.
Free Software Foundation, 59 Temple Place – Suite 330, Boston, MA 02111-1307 USA, 0.10 edition.

Muchnick, S. (1997).
Advanced Compiler Design and Implementation.
Morgan Kaufmann Publishers.