Intermediate Representations

A. Demaille, R. Levillain

EPITA — École Pour l'Informatique et les Techniques Avancées

May 6, 2014
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”).

... 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
Intermediate Representations

- 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.

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 $\lambda$-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. [5]

```
f1 <- [fp - 216]
r7 <- f1
r6 <- 4 * r5
r5 <- r4 + r2
r4 <- r3 + 20
r3 <- [fp - 8]
r2 <- r1 + 2
r1 <- [fp - 4]
t7 <- *t6
```

Different Levels [5]

```
float a[20][10];
...  
a[i][j+2];
```
Different Levels: The GCC Structure

Stack Based: Java Byte-Code

```java
public 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);
    }
}
```

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
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) [5].

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

Register Based: tc's Tree (1/3)

```plaintext
/* == High Level Intermediate representation. == */
# Routine: gcd
label 10
# Prologue
move temp t2, temp fp
move temp fp, temp sp
move
  temp sp
  binop (-)
  temp sp
  const 4
move
temp
nem
  temp fp
temp t0
move temp t0, temp t1
move temp t1, temp t2
```

Register Based: tc's Tree (2/3)

```plaintext
# Body
move temp rv
  eseq seq
    label 12
cjump ne temp t0 temp t1 name l3 name l1
    label 13
    seq
cjump gt temp t0 temp t1 name l4 name l5
    label l4
    move temp t0
    binop (-) temp t0 temp t1
    jump name l6
    label l5
    move temp t1
    binop (-) temp t1 temp t0
    label l6
    seq end
    jump name l2
    label l1
    seq end
    temp t0
```
Register Based: tc’s Tree (3/3)

```
# Epilogue
move temp sp, temp fp
move temp fp, temp t2
label end
```

Register Based: tc’s Tree (4/3)

```
# Routine: Main Program
label Main
# Prologue
# Body
sxp
call name print_int
call name l0
        temp fp
        const 42
        const 51
call end
call 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.

Quadruples vs. Triples [5]

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

int
gcd (int a, int b)
{
    while (a != b)
    {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
    return a;
}
Register Based: GCC’s RTL cont’d

(note 22 21 25 (“gcd.c”) 6)
(note 22 21 25 (insn 22 21 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)
       (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))

A. Demaille, R. Levillain
Intermediate Representations
33 / 107
Register Based [2]

Advantages
- Suits today’s architectures
- Clearer data flow

Disadvantages
- Harder to synthesize
- Less compact
- Harder to interpret

Tree [1]

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"
```
Intermediate Representations

Memory Management

Translation to intermediate language

The Case of the Tiger Compiler

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
```

```
% 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

1 Intermediate Representations
2 Memory Management
   a Memory Management
      1 Memory Management
      2 Activation Blocks
      3 Nonlocal Variables
3 Translation to intermediate language
4 The Case of the Tiger Compiler
Memory Hierarchy [1]

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.

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 (1isp, Smalltalk, ML, Haskell, Tiger, Perl etc.).

spim Memory Model [3]
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.

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.

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.

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...

```c
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.

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

alloca is a good replacement.

```c
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 [4]

- 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 [4]

- 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 [4]

- 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`).

escapes-n-recursion

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

let function one (input : int) =
  (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`
```ml
two (3) two (2) two (1) one (1) one (2) one (3)
```
Deep Static Function Hierarchies

What if:
- Main uses $\text{vm}$
- Main calls $F_1$
- $F_1$ uses $v_1$
- $F_1$ uses $\text{vm}$, non local
- $F_1$ calls $F_11$
- $F_11$ uses $v_{11}$
- $F_11$ uses $v_1$
- $F_11$ uses $\text{vm}$
- $F_11$ calls $F_12$
- $F_12$ calls $F_1$

The caller must provide the callee with its static link.

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
<th>Static Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>$F_1$</td>
<td>$f_{\text{Main}} = f_p$</td>
</tr>
<tr>
<td>$F_1$</td>
<td>$F_{11}$</td>
<td>$f_{F_1} = f_p$</td>
</tr>
<tr>
<td>$F_{11}$</td>
<td>$F_{12}$</td>
<td>$f_{F_{11}} = f_p$</td>
</tr>
<tr>
<td>$F_{12}$</td>
<td>$F_2$</td>
<td>$f_{F_{12}} = f_p$</td>
</tr>
<tr>
<td>$F_2$</td>
<td>$F_{22}$</td>
<td>$f_{F_2} = f_p$</td>
</tr>
<tr>
<td>$F_{22}$</td>
<td>$F_{11}$</td>
<td>$f_{F_{22}} = ????$</td>
</tr>
</tbody>
</table>

Assuming that the static link is stored at $f_p$.

Higher Order Functions

```ocaml
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

1. Intermediate Representations
2. Memory Management
3. Translation to intermediate language
   - Calling Conventions
   - Clever Translations
   - Complex Expressions
4. The Case of the Tiger Compiler
Intermediate Representations

Memory Management

Translation to intermediate language

Calling Conventions

Clever Translations

Complex Expressions

The Case of the Tiger Compiler

Calling Conventions

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

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

```hs
# 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

# 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
call name 10 temp fp
call name 42 const 51
temp end
```

Clever Translations
Translating Conditions

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

1. $cjump (\alpha < \beta, ltrue, lfalse)$
2. $eseq (seq (cjump (\alpha < \beta, ltrue, lfalse), label ltrue$
   $\quad move temp t, const 1$
   $\quad jump lend$
   $\quad label lfalse$
   $\quad move temp t, const 0$
   $\quad label lend), temp t)$
3. $seq (sxp (\alpha)$
   $\quad sxp (\beta))$

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.

Prototranslation, Expression Shells

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

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

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

<table>
<thead>
<tr>
<th>Exp</th>
<th>unNx</th>
<th>unEx</th>
<th>unCx (t, f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex(e)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cx(a &lt; b)</td>
<td>seq(sxp(a), sxp(b))</td>
<td>eseq(t := (a &lt; b), t)</td>
<td>cjump(a &lt; b, t, f)</td>
</tr>
<tr>
<td>Nx(s)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Translating if

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

cjump no
eseq seq cjump 11 < 22 name 10 name 11
label name 12
move temp 10
label name 13
move temp 10
eseq seq move temp 11 const 1
3jump 22 < 33 name 13 name 14
label name 15
move temp 11 const 0
label name 16
seq seq end
move temp 13
jump name 12
temp 12
label name 17
jump name 17
label name 16
seq call name print_int const 1
jump name 17
label name 15
jump name 17
jump name 17

A Better Translation: \texttt{Ix}

```plaintext
seq
cjump 11 < 22 name 13 name 14
label 13
cjump 1 <> 0 name 10 name 11
label 14
cjump 22 < 33 name 10 name 11
seq end
label 10
sxp call name print_int const 1
jump name 12
label 11
sxp call name print_int const 0
label 12
```

Complex Expressions

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

While Loops

```plaintext
while condition
  do body
  test:
    if not (condition)
      goto done
    body
  goto test
done:
```
For Loops

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

The Case of the Tiger Compiler

Additional Features

- Bounds checking
- Nil checking
- ...

Intermediate Representations

Memory Management

Translation to intermediate language

The Case of the Tiger Compiler

Calling Conventions

Clever Translations

Complex Expressions

Intermediate Representations

Memory Management

Translation to intermediate language

The Case of the Tiger Compiler

Calling Conventions

Clever Translations

Complex Expressions

For Loops

```plaintext
let i := min
limit := max
in
  if (i > limit)
  goto end
loop:
  body
  if (i >= limit)
  goto end
  ++i
  goto loop
end:
```
Translation in the Tiger Compiler

1. Intermediate Representations
2. Memory Management
3. Translation to intermediate language

The Case of the Tiger Compiler
- Translation in the Tiger Compiler
  - LIR: Low Level Intermediate Representation

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)

- temp::Temp is not tree::Temp.
  - The latter aggregates one of the former.
  - Similarly with Label.
- n-ary seq.
  - (Unlike [1]).
- 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.

- `Access local_alloc (bool escapes_p)`

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 exp (Level use)**

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.

**Exp fp (Level use)**

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.
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

```plaintext
seq (ss1, seq (ss2), ss3) => seq (ss1, ss2, ss3)
call (f, eseq (s, e), es) => eseq (s, call (f, e, es))
binop (+, eseq (s, eseq (s1, e2))) => eseq (s, binop (+, e1, e2))
binop (+, e1, eseq (s1, e2)) => eseq (s, binop (+, e1, e2))
```

Linearization: Incorrect Changes

```plaintext
binop (+, e1, eseq (s, e2)) => eseq (s, binop (+, e1, e2))
```

- But what if s modifies the value of e1?
  ```plaintext
  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)
```

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

Correct:
```
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 ≠ 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
  ```
  procedure then its parent must be an sxp
  ```
- **function** then its parent must be a move (temp t, .)
  ```
  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 \texttt{cjump} into actual assembly instructions, the “false” label must follow the \texttt{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 \texttt{cjump}s
- 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 \texttt{jump}
  - if the “destination block” is available, add it
  - otherwise, fetch any other remaining block.
- If the last instruction is a \texttt{cjump}
  - If the false destination is available, push it
  - If the true destination is available, flip the \texttt{cjump} and push it,
  - otherwise, change the \texttt{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 test

label test
  cjump i <= N, body, done

label body
  Body.
  jump 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

Bibliography I


Bibliography II
