Computer Languages

Transformations of the IR

Verónica Gaspes

School of Information Science, Computer and Electrical Engineering

Mars 2

www2.hh.se/staff/vero/languages

Another overview of the compiler

Source: unstructured text

```
class A {
  public static void ...
     System.out.print(3);}
class B {
  int  x;
  int f(int y){
    return x+y;}}
```

Abstract syntax: strucutured

Intermediate representation

Target: unstructured text

```
main:
            subu $sp, $sp, 32
            sw $ra, 20($sp)
            sd $a0, 32($sp)
            sw $0, 24($sp)
            sw $0, 28($sp)
```

Another overview of the compiler

Source: unstructured text

```
class A {
  public static void ...
     System.out.print(3);}
class B {
  int  x;
  int f(int y){
    return x+y;}}
```

Abstract syntax: strucutured

Intermediate representation

Target: unstructured text

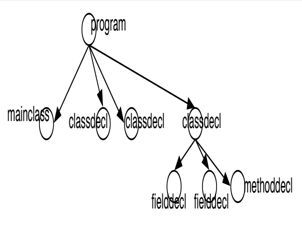
```
main:
        subu $sp, $sp, 32
        sw $ra, 20($sp)
        sd $a0, 32($sp)
        sw $0, 24($sp)
        sw $0, 28($sp)
```

Another overview of the compiler

Source: unstructured text

```
class A {
  public static void ...
     System.out.print(3);}
class B {
  int  x;
  int f(int y){
    return x+y;}}
```

Abstract syntax: structucured



Intermediate representation

Target: unstructured text

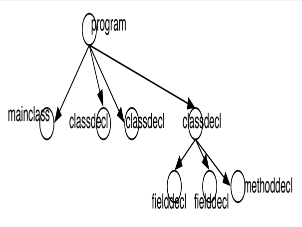
```
main:
            subu $sp, $sp, 32
            sw $ra, 20($sp)
            sd $a0, 32($sp)
            sw $0, 24($sp)
            sw $0, 28($sp)
```

Another overview of the compiler

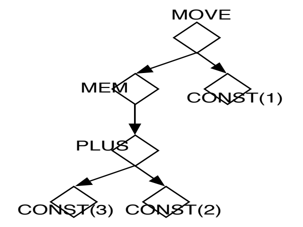
Source: unstructured text

```
class A {
  public static void ...
     System.out.print(3);}
class B {
  int  x;
  int f(int y){
    return x+y;}}
```

Abstract syntax: strucutured



Intermediate representation



Target: unstructured text

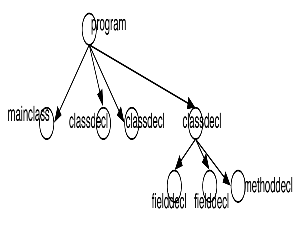
```
main:
        subu $sp, $sp, 32
        sw $ra, 20($sp)
        sd $a0, 32($sp)
        sw $0, 24($sp)
        sw $0, 28($sp)
```

Another overview of the compiler

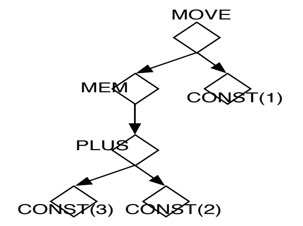
Source: unstructured text

```
class A {
  public static void ...
     System.out.print(3);}
class B {
  int  x;
  int f(int y){
    return x+y;}}
```

Abstract syntax: structured



Intermediate representation



Target: unstructured text

```
main:
        subu $sp, $sp, 32
        sw $ra, 20($sp)
        sd $a0, 32($sp)
        sw $0, 24($sp)
        sw $0, 28($sp)
```

Today I will explain things that you will not have to implement. You will be provided with the code for doing this.

However, we need to explain what kind of modifications compilers do to the IR so that it is easier to translate to assembler!

You find the details in chapter 8 of the course book.

Today I will explain things that you will not have to implement. You will be provided with the code for doing this.

However, we need to explain what kind of modifications compilers do to the IR so that it is easier to translate to assembler!

You find the details in chapter 8 of the course book.

Today I will explain things that you will not have to implement. You will be provided with the code for doing this.

However, we need to explain what kind of modifications compilers do to the IR so that it is easier to translate to assembler!

You find the details in chapter 8 of the course book.

Mismatches between IR trees and machine-language

- $\mathrm{CJUMP}(op, e_1, e_2, l_1, l_2)$! In real machines conditional jumps fall to the next instruction!
- $\mathrm{ESEQ}(s, e)$ makes different evaluation orders yield different results!
- $\mathrm{CALL}(f, e_1, \ldots, e_n)$ too!
- $\mathrm{CALL}(f, e_1, \ldots, e_n)$ within other $\mathrm{CALL}()$ disturb using dedicated registers for parameters.

Solution

Transform the IR-trees preserving meaning but making them easier to translate to machine language!

Mismatches between IR trees and machine-language

- $\mathrm{CJUMP}(op,e_1,e_2,l_1,l_2)$! In real machines conditional jumps fall to the next instruction!
- $\mathrm{ESEQ}(s,e)$ makes different evaluation orders yield different results!
- $\mathrm{CALL}(f,e_1,\ldots,e_n)$ too!
- $\mathrm{CALL}(f,e_1,\ldots,e_n)$ within other $\mathrm{CALL}()$ disturb using dedicated registers for parameters.

Solution

Transform the IR-trees preserving meaning but making them easier to translate to machine language!

Mismatches between IR trees and machine-language

- $\mathrm{CJUMP}(op,e_1,e_2,l_1,l_2)$! In real machines conditional jumps fall to the next instruction!
- $\mathrm{ESEQ}(s,e)$ makes different evaluation orders yield different results!
- $\mathrm{CALL}(f,e_1,\ldots,e_n)$ too!
- $\mathrm{CALL}(f,e_1,\ldots,e_n)$ within other $\mathrm{CALL}()$ disturb using dedicated registers for parameters.

Solution

Transform the IR-trees preserving meaning but making them easier to translate to machine language!

Mismatches between IR trees and machine-language

- $\mathrm{CJUMP}(op, e_1, e_2, l_1, l_2)$! In real machines conditional jumps fall to the next instruction!
- $\mathrm{ESEQ}(s, e)$ makes different evaluation orders yield different results!
- $\mathrm{CALL}(f, e_1, \ldots, e_n)$ too!
- $\mathrm{CALL}(f, e_1, \ldots, e_n)$ within other $\mathrm{CALL}()$ disturb using dedicated registers for parameters.

Solution

Transform the IR-trees preserving meaning but making them easier to translate to machine language!

Mismatches between IR trees and machine-language

- $\mathrm{CJUMP}(op, e_1, e_2, l_1, l_2)$! In real machines conditional jumps fall to the next instruction!
- $\mathrm{ESEQ}(s, e)$ makes different evaluation orders yield different results!
- $\mathrm{CALL}(f, e_1, \ldots, e_n)$ too!
- $\mathrm{CALL}(f, e_1, \ldots, e_n)$ within other $\mathrm{CALL}()$ disturb using dedicated registers for parameters.

Solution

Transform the IR-trees preserving meaning but making them easier to translate to machine language!

Mismatches between IR trees and machine-language

- $\mathrm{CJUMP}(op,e_1,e_2,l_1,l_2)$! In real machines conditional jumps fall to the next instruction!
- $\mathrm{ESEQ}(s,e)$ makes different evaluation orders yield different results!
- $\mathrm{CALL}(f,e_1,\ldots,e_n)$ too!
- $\mathrm{CALL}(f,e_1,\ldots,e_n)$ within other $\mathrm{CALL}()$ disturb using dedicated registers for parameters.

Solution

Transform the IR-trees preserving meaning but making them easier to translate to machine language!

Overview of the transformation

Canonical Trees

The original IR-tree is transformed into a list of canonical trees without $\mathrm{SEQ}()$ or $\mathrm{ESEQ}()$ nodes

Basic Blocks

The list of canonical trees is grouped into a set of basic blocks such that there are no jumps or labels inside a block.

Traces

Finally, the basic blocks are reordered into a set of traces where each $\mathrm{CJUMP}()$ is immediatelly followed by its `false` label.

Overview of the transformation

Canonical Trees

The original IR-tree is transformed into a list of canonical trees without $\mathrm{SEQ}()$ or $\mathrm{ESEQ}()$ nodes

Basic Blocks

The list of canonical trees is grouped into a set of basic blocks such that there are no jumps or labels inside a block.

Traces

Finally, the basic blocks are reordered into a set of traces where each $\mathrm{CJUMP}()$ is immediatelly followed by its `false` label.

Overview of the transformation

Canonical Trees

The original IR-tree is transformed into a list of canonical trees without $\mathrm{SEQ}()$ or $\mathrm{ESEQ}()$ nodes

Basic Blocks

The list of canonical trees is grouped into a set of basic blocks such that there are no jumps or labels inside a block.

Traces

Finally, the basic blocks are reordered into a set of traces where each $\mathrm{CJUMP}()$ is immediatelly followed by its `false` label.

Overview of the transformation

Canonical Trees

The original IR-tree is transformed into a list of canonical trees without $\mathrm{SEQ}()$ or $\mathrm{ESEQ}()$ nodes

Basic Blocks

The list of canonical trees is grouped into a set of basic blocks such that there are no jumps or labels inside a block.

Traces

Finally, the basic blocks are reordered into a set of traces where each $\mathrm{CJUMP}()$ is immediatelly followed by its `false` label.

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

$\mathrm{CONST}(1)$
$\mathrm{TEMP}(t_{98})$
$\mathrm{BINOP}(+,\mathrm{CONST}(1),\mathrm{TEMP}(t_{98}))$
$\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))$
$\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98})))$
$\mathrm{ESEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1)),\mathrm{TEMP}(t_{98}))$

tree.Stm

$\mathrm{JUMP}(l_1)$
$\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1))$
$\mathrm{MOVE}(\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))),\mathrm{CONST}(1))$

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

CONST(*1*)

tree.Exp

CONST(*1*)
TEMP(t_{98})
BINOP(+,CONST(*1*),TEMP(t_{98}))
CALL(f,TEMP(t_{98}))
MEM(CALL(f,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(*1*)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(*1*))
MOVE(MEM(CALL(f,TEMP(t_{98}))),CONST(*1*))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

CONST(*1*)
TEMP(t_{98})
BINOP(+,CONST(*1*),TEMP(t_{98}))
CALL(*f*,TEMP(t_{98}))
MEM(CALL(*f*,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(*1*)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(*1*))
MOVE(MEM(CALL(*f*,TEMP(t_{98}))),CONST(*1*))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

$\mathrm{TEMP}(t_{98})$

tree.Exp

$\mathrm{CONST}(1)$
$\mathrm{TEMP}(t_{98})$
$\mathrm{BINOP}(+,\mathrm{CONST}(1),\mathrm{TEMP}(t_{98}))$
$\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))$
$\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98})))$
$\mathrm{ESEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1)),\mathrm{TEMP}(t_{98}))$

tree.Stm

$\mathrm{JUMP}(l_1)$
$\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1))$
$\mathrm{MOVE}(\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))),\mathrm{CONST}(1))$

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

CONST(*1*)
TEMP(t_{98})
BINOP(+,CONST(*1*),TEMP(t_{98}))
CALL(*f*,TEMP(t_{98}))
MEM(CALL(*f*,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(*1*)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(*1*))
MOVE(MEM(CALL(*f*,TEMP(t_{98}))),CONST(*1*))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

$\text{JUMP}(l_1)$

tree.Exp

$\text{CONST}(1)$
$\text{TEMP}(t_{98})$
$\text{BINOP}(+,\text{CONST}(1),\text{TEMP}(t_{98}))$
$\text{CALL}(f,\text{TEMP}(t_{98}))$
$\text{MEM}(\text{CALL}(f,\text{TEMP}(t_{98})))$
$\text{ESEQ}(\text{MOVE}(\text{TEMP}(t_{98}),\text{CONST}(1)),\text{TEMP}(t_{98}))$

tree.Stm

$\text{JUMP}(l_1)$
$\text{MOVE}(\text{TEMP}(t_{98}),\text{CONST}(1))$
$\text{MOVE}(\text{MEM}(\text{CALL}(f,\text{TEMP}(t_{98}))),\text{CONST}(1))$

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

CONST(*1*)
TEMP(t_{98})
BINOP(+,CONST(*1*),TEMP(t_{98}))
CALL(f,TEMP(t_{98}))
MEM(CALL(f,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(*1*)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(*1*))
MOVE(MEM(CALL(f,TEMP(t_{98}))),CONST(*1*))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

BINOP(+,CONST(1),TEMP(t_{98}))

tree.Exp

CONST(1)
TEMP(t_{98})
BINOP(+,CONST(1),TEMP(t_{98}))
CALL(f,TEMP(t_{98}))
MEM(CALL(f,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(1)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(1))
MOVE(MEM(CALL(f,TEMP(t_{98}))),CONST(1))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

CONST(*1*)
TEMP(t_{98})
BINOP(+,CONST(*1*),TEMP(t_{98}))
CALL(f,TEMP(t_{98}))
MEM(CALL(f,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(*1*)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(*1*))
MOVE(MEM(CALL(f,TEMP(t_{98}))),CONST(*1*))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

$\text{MOVE}(\text{TEMP}(t_{98}),\text{CONST}(1))$

tree.Exp

$\text{CONST}(1)$
$\text{TEMP}(t_{98})$
$\text{BINOP}(+,\text{CONST}(1),\text{TEMP}(t_{98}))$
$\text{CALL}(f,\text{TEMP}(t_{98}))$
$\text{MEM}(\text{CALL}(f,\text{TEMP}(t_{98})))$
$\text{ESEQ}(\text{MOVE}(\text{TEMP}(t_{98}),\text{CONST}(1)),\text{TEMP}(t_{98}))$

tree.Stm

$\text{JUMP}(l_1)$
$\text{MOVE}(\text{TEMP}(t_{98}),\text{CONST}(1))$
$\text{MOVE}(\text{MEM}(\text{CALL}(f,\text{TEMP}(t_{98}))),\text{CONST}(1))$

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

$\text{CONST}(1)$
$\text{TEMP}(t_{98})$
$\text{BINOP}(+,\text{CONST}(1),\text{TEMP}(t_{98}))$
$\text{CALL}(f,\text{TEMP}(t_{98}))$
$\text{MEM}(\text{CALL}(f,\text{TEMP}(t_{98})))$
$\text{ESEQ}(\text{MOVE}(\text{TEMP}(t_{98}),\text{CONST}(1)),\text{TEMP}(t_{98}))$

tree.Stm

$\text{JUMP}(l_1)$
$\text{MOVE}(\text{TEMP}(t_{98}),\text{CONST}(1))$
$\text{MOVE}(\text{MEM}(\text{CALL}(f,\text{TEMP}(t_{98}))),\text{CONST}(1))$

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

$\textsc{call}(f,\textsc{temp}(t_{98}))$

tree.Exp

$\textsc{const}(1)$
$\textsc{temp}(t_{98})$
$\textsc{binop}(+,\textsc{const}(1),\textsc{temp}(t_{98}))$
$\textsc{call}(f,\textsc{temp}(t_{98}))$
$\textsc{mem}(\textsc{call}(f,\textsc{temp}(t_{98})))$
$\textsc{eseq}(\textsc{move}(\textsc{temp}(t_{98}),\textsc{const}(1)),\textsc{temp}(t_{98}))$

tree.Stm

$\textsc{jump}(l_1)$
$\textsc{move}(\textsc{temp}(t_{98}),\textsc{const}(1))$
$\textsc{move}(\textsc{mem}(\textsc{call}(f,\textsc{temp}(t_{98}))),\textsc{const}(1))$

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

$\mathrm{CONST}(1)$
$\mathrm{TEMP}(t_{98})$
$\mathrm{BINOP}(+,\mathrm{CONST}(1),\mathrm{TEMP}(t_{98}))$
$\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))$
$\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98})))$
$\mathrm{ESEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1)),\mathrm{TEMP}(t_{98}))$

tree.Stm

$\mathrm{JUMP}(l_1)$
$\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1))$
$\mathrm{MOVE}(\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))),\mathrm{CONST}(1))$

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

$\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98})))$

tree.Exp

$\mathrm{CONST}(1)$
$\mathrm{TEMP}(t_{98})$
$\mathrm{BINOP}(+,\mathrm{CONST}(1),\mathrm{TEMP}(t_{98}))$
$\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))$
$\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98})))$
$\mathrm{ESEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1)),\mathrm{TEMP}(t_{98}))$

tree.Stm

$\mathrm{JUMP}(l_1)$
$\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1))$
$\mathrm{MOVE}(\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))),\mathrm{CONST}(1))$

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

CONST(*1*)
TEMP(t_{98})
BINOP(+,CONST(*1*),TEMP(t_{98}))
CALL(*f*,TEMP(t_{98}))
MEM(CALL(*f*,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(*1*)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(*1*))
MOVE(MEM(CALL(*f*,TEMP(t_{98}))),CONST(*1*))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine instruction language*!

Example

MOVE(MEM(CALL(f,TEMP(t_{98}))),CONST(1))

tree.Exp

CONST(1)
TEMP(t_{98})
BINOP(+,CONST(1),TEMP(t_{98}))
CALL(f,TEMP(t_{98}))
MEM(CALL(f,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(1)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(1))
MOVE(MEM(CALL(f,TEMP(t_{98}))),CONST(1))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

CONST(*1*)
TEMP(t_{98})
BINOP(+,CONST(*1*),TEMP(t_{98}))
CALL(*f*,TEMP(t_{98}))
MEM(CALL(*f*,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(*1*)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(*1*))
MOVE(MEM(CALL(*f*,TEMP(t_{98}))),CONST(*1*))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

ESEQ(MOVE(TEMP(t_{98}),CONST(1)),TEMP(t_{98}))

tree.Exp

CONST(1)
TEMP(t_{98})
BINOP(+,CONST(1),TEMP(t_{98}))
CALL(f,TEMP(t_{98}))
MEM(CALL(f,TEMP(t_{98})))
ESEQ(MOVE(TEMP(t_{98}),CONST(1)),TEMP(t_{98}))

tree.Stm

JUMP(l_1)
MOVE(TEMP(t_{98}),CONST(1))
MOVE(MEM(CALL(f,TEMP(t_{98}))),CONST(1))

IR-trees (short reminder)

IR-trees are the *abstract syntax* for the *abstract machine* instruction language!

Example

tree.Exp

$\mathrm{CONST}(1)$
$\mathrm{TEMP}(t_{98})$
$\mathrm{BINOP}(+,\mathrm{CONST}(1),\mathrm{TEMP}(t_{98}))$
$\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))$
$\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98})))$
$\mathrm{ESEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1)),\mathrm{TEMP}(t_{98}))$

tree.Stm

$\mathrm{JUMP}(l_1)$
$\mathrm{MOVE}(\mathrm{TEMP}(t_{98}),\mathrm{CONST}(1))$
$\mathrm{MOVE}(\mathrm{MEM}(\mathrm{CALL}(f,\mathrm{TEMP}(t_{98}))),\mathrm{CONST}(1))$

IR-trees (remarks)

- $\mathrm{ESEQ}(s,e)$is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions.To exploit this, the order of evaluation of expressions should not influence the result!$\mathrm{ESEQ}(s,e)$ spoils this!
- We want to get rid of $\mathrm{ESEQ}(s,e)$ before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- $\mathrm{ESEQ}(s,e)$ is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions. To exploit this, the order of evaluation of expressions should not influence the result! $\mathrm{ESEQ}(s,e)$ spoils this!
- We want to get rid of $\mathrm{ESEQ}(s,e)$ before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- $\mathrm{ESEQ}(s,e)$ is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions. To exploit this, the order of evaluation of expressions should not influence the result! $\mathrm{ESEQ}(s,e)$ spoils this!
- We want to get rid of $\mathrm{ESEQ}(s,e)$ before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- $\mathrm{ESEQ}(s,e)$is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions.To exploit this, the order of evaluation of expressions should not influence the result!$\mathrm{ESEQ}(s,e)$ spoils this!
- We want to get rid of $\mathrm{ESEQ}(s,e)$ before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- $\mathrm{ESEQ}(s,e)$is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions. To exploit this, the order of evaluation of expressions should not influence the result! $\mathrm{ESEQ}(s,e)$ spoils this!
- We want to get rid of $\mathrm{ESEQ}(s,e)$ before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- $\mathrm{ESEQ}(s,e)$ is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions. To exploit this, the order of evaluation of expressions should not influence the result! $\mathrm{ESEQ}(s,e)$ spoils this!
- We want to get rid of $\mathrm{ESEQ}(s,e)$ before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- ESEQ(s,e)is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions. To exploit this, the order of evaluation of expressions should not influence the result! ESEQ(s,e) spoils this!
- We want to get rid of ESEQ(s,e) before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- $\mathrm{ESEQ}(s,e)$is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions. To exploit this, the order of evaluation of expressions should not influence the result! $\mathrm{ESEQ}(s,e)$ spoils this!
- We want to get rid of $\mathrm{ESEQ}(s,e)$ before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- ESEQ(s,e)is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions. To exploit this, the order of evaluation of expressions should not influence the result! ESEQ(s,e) spoils this!
- We want to get rid of ESEQ(s,e) before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- $\text{ESEQ}(s,e)$is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions. To exploit this, the order of evaluation of expressions should not influence the result! $\text{ESEQ}(s,e)$ spoils this!
- We want to get rid of $\text{ESEQ}(s,e)$ before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

IR-trees (remarks)

- ESEQ(s,e)is very useful but introduces side effects when evaluating expressions!
- Many computer architectures support parallel evaluation of expressions. To exploit this, the order of evaluation of expressions should not influence the result! ESEQ(s,e) spoils this!
- We want to get rid of ESEQ(s,e) before generating code!
- We will subject each fragment to transformations that preserve meaning and result in IR-trees that are easier to translate to assembler!

Getting rid of $\mathrm{ESEQ}(s,e)$

- If $\mathrm{ESEQ}(s,e)$ is the main expression in a statement, as in

$$\mathrm{JUMP}(\mathrm{ESEQ}(s,e))$$

we might replace it by

$$\mathrm{SEQ}(s,\mathrm{JUMP}(e))$$

- So, if we manage to shift all $\mathrm{ESEQ}(s,e)$ to *top level*, then we can eliminate them! (we *transform* an IR-tree with an $\mathrm{ESEQ}(s,e)$ into another IR-tree *that achieves the same results* and does not contain the $\mathrm{ESEQ}(s,r)$)
- We have to provide the transformations that do the *shift* of subexpressions $\mathrm{ESEQ}(s,e)$ to top level!

Getting rid of $\mathrm{ESEQ}(s,e)$

- If $\mathrm{ESEQ}(s,e)$ is the main expression in a statement, as in

$$\mathrm{JUMP}(\mathrm{ESEQ}(s,e))$$

we might replace it by

$$\mathrm{SEQ}(s,\mathrm{JUMP}(e))$$

- So, if we manage to shift all $\mathrm{ESEQ}(s,e)$ to *top level*, then we can eliminate them! (we *transform* an IR-tree with an $\mathrm{ESEQ}(s,e)$ into another IR-tree *that achieves the same results* and does not contain the $\mathrm{ESEQ}(s,r)$)
- We have to provide the transformations that do the *shift* of subexpressions $\mathrm{ESEQ}(s,e)$ to top level!

Getting rid of $\text{ESEQ}(s,e)$

- If $\text{ESEQ}(s,e)$ is the main expression in a statement, as in

$$\text{JUMP}(\text{ESEQ}(s,e))$$

we might replace it by

$$\text{SEQ}(s,\text{JUMP}(e))$$

- So, if we manage to shift all $\text{ESEQ}(s,e)$ to *top level*, then we can eliminate them! (we *transform* an IR-tree with an $\text{ESEQ}(s,e)$ into another IR-tree *that achieves the same results* and does not contain the $\text{ESEQ}(s,r)$)
- We have to provide the transformations that do the *shift* of subexpressions $\text{ESEQ}(s,e)$ to top level!

Getting rid of $\mathrm{ESEQ}(s,e)$

- If $\mathrm{ESEQ}(s,e)$ is the main expression in a statement, as in
$$\mathrm{JUMP}(\mathrm{ESEQ}(s,e))$$
we might replace it by
$$\mathrm{SEQ}(s,\mathrm{JUMP}(e))$$
- So, if we manage to shift all $\mathrm{ESEQ}(s,e)$ to *top level*, then we can eliminate them! (we *transform* an IR-tree with an $\mathrm{ESEQ}(s,e)$ into another IR-tree *that achieves the same results* and does not contain the $\mathrm{ESEQ}(s,r)$)
- We have to provide the transformations that do the *shift* of subexpressions $\mathrm{ESEQ}(s,e)$ to top level!

Getting rid of $\mathrm{ESEQ}(s,e)$

- If $\mathrm{ESEQ}(s,e)$ is the main expression in a statement, as in

$$\mathrm{JUMP}(\mathrm{ESEQ}(s,e))$$

  we might replace it by

$$\mathrm{SEQ}(s,\mathrm{JUMP}(e))$$

- So, if we manage to shift all $\mathrm{ESEQ}(s,e)$ to *top level*, then we can eliminate them! (we *transform* an IR-tree with an $\mathrm{ESEQ}(s,e)$ into another IR-tree *that achieves the same results* and does not contain the $\mathrm{ESEQ}(s,r)$)
- We have to provide the transformations that do the *shift* of subexpressions $\mathrm{ESEQ}(s,e)$ to top level!

Getting rid of $\mathrm{ESEQ}(s,e)$

- If $\mathrm{ESEQ}(s,e)$ is the main expression in a statement, as in
$$\mathrm{JUMP}(\mathrm{ESEQ}(s,e))$$
we might replace it by
$$\mathrm{SEQ}(s,\mathrm{JUMP}(e))$$
- So, if we manage to shift all $\mathrm{ESEQ}(s,e)$ to *top level*, then we can eliminate them! (we *transform* an IR-tree with an $\mathrm{ESEQ}(s,e)$ into another IR-tree *that achieves the same results* and does not contain the $\mathrm{ESEQ}(s,r)$)
- We have to provide the transformations that do the *shift* of subexpressions $\mathrm{ESEQ}(s,e)$ to top level!

Getting rid of $\mathrm{ESEQ}(s,e)$

- If $\mathrm{ESEQ}(s,e)$ is the main expression in a statement, as in

$$\mathrm{JUMP}(\mathrm{ESEQ}(s,e))$$

  we might replace it by

$$\mathrm{SEQ}(s,\mathrm{JUMP}(e))$$

- So, if we manage to shift all $\mathrm{ESEQ}(s,e)$ to *top level*, then we can eliminate them! (we *transform* an IR-tree with an $\mathrm{ESEQ}(s,e)$ into another IR-tree *that achieves the same results* and does not contain the $\mathrm{ESEQ}(s,r)$)
- We have to provide the transformations that do the *shift* of subexpressions $\mathrm{ESEQ}(s,e)$ to top level!

Getting rid of $\text{ESEQ}(s,e)$

- If $\text{ESEQ}(s,e)$ is the main expression in a statement, as in

$$\text{JUMP}(\text{ESEQ}(s,e))$$

  we might replace it by

$$\text{SEQ}(s,\text{JUMP}(e))$$

- So, if we manage to shift all $\text{ESEQ}(s,e)$ to *top level*, then we can eliminate them! (we *transform* an IR-tree with an $\text{ESEQ}(s,e)$ into another IR-tree *that achieves the same results* and does not contain the $\text{ESEQ}(s,r)$)
- We have to provide the transformations that do the *shift* of subexpressions $\text{ESEQ}(s,e)$ to top level!

Getting rid of $\text{ESEQ}(s,e)$

- If $\text{ESEQ}(s,e)$ is the main expression in a statement, as in

$$\text{JUMP}(\text{ESEQ}(s,e))$$

  we might replace it by

$$\text{SEQ}(s,\text{JUMP}(e))$$

- So, if we manage to shift all $\text{ESEQ}(s,e)$ to *top level*, then we can eliminate them! (we *transform* an IR-tree with an $\text{ESEQ}(s,e)$ into another IR-tree *that achieves the same results* and does not contain the $\text{ESEQ}(s,r)$)
- We have to provide the transformations that do the *shift* of subexpressions $\text{ESEQ}(s,e)$ to top level!

Getting rid of ESEQ(s,e)

Some transformations that move ESEQ(s,e) to top level

ESEQ$(s_1,$ESEQ$(s_2,e))$ $\Rightarrow$ ESEQ(SEQ$(s_1,s_2),e)$

BINOP$(op,$ESEQ$(s,e_1),e_2)$ $\Rightarrow$ ESEQ$(s,$BINOP$(op,e_1,e_2))$

BINOP$(op,e_1,$ESEQ$(s,e_2))$ $\Rightarrow$ ESEQ$(s,$BINOP$(op,e_1,e_2))$ IF e_1 *and* s *commute*

BINOP$(op,e_1,$ESEQ$(s,e_2))$ $\Rightarrow$
ESEQ(SEQ(MOVE(TEMP$(t),e_1),s),$BINOP$(op,$TEMP$(t),e_2))$

And there are some more . . .

Getting rid of ESEQ(s,e)

Some transformations that move ESEQ(s,e) to top level

ESEQ$(s_1,$ESEQ$(s_2,e))$ $\Rightarrow$ ESEQ(SEQ$(s_1,s_2),e)$

BINOP$(op,$ESEQ$(s,e_1),e_2)$ $\Rightarrow$ ESEQ$(s,$BINOP$(op,e_1,e_2))$

BINOP$(op,e_1,$ESEQ$(s,e_2))$ $\Rightarrow$ ESEQ$(s,$BINOP$(op,e_1,e_2))$ IF e_1 *and s commute*

BINOP$(op,e_1,$ESEQ$(s,e_2))$ $\Rightarrow$
ESEQ(SEQ(MOVE(TEMP$(t),e_1),s),$BINOP$(op,$TEMP$(t),e_2))$

And there are some more . . .

Getting rid of $\mathrm{ESEQ}(s,e)$

Some transformations that move $\mathrm{ESEQ}(s,e)$ to top level

$\mathrm{ESEQ}(s_1,\mathrm{ESEQ}(s_2,e)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$

$\mathrm{BINOP}(op,\mathrm{ESEQ}(s,e_1),e_2) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$ IF e_1 *and* s *commute*

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow$
$\mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t),e_1),s),\mathrm{BINOP}(op,\mathrm{TEMP}(t),e_2))$

And there are some more . . .

Getting rid of $\mathrm{ESEQ}(s,e)$

Some transformations that move $\mathrm{ESEQ}(s,e)$ to top level

$\mathrm{ESEQ}(s_1,\mathrm{ESEQ}(s_2,e)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$

$\mathrm{BINOP}(op,\mathrm{ESEQ}(s,e_1),e_2) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$ IF *e_1 and s commute*

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow$
$\mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t),e_1),s),\mathrm{BINOP}(op,\mathrm{TEMP}(t),e_2))$

And there are some more ...

Getting rid of $\text{ESEQ}(s,e)$

Some transformations that move $\text{ESEQ}(s,e)$ to top level

$\text{ESEQ}(s_1,\text{ESEQ}(s_2,e)) \Rightarrow \text{ESEQ}(\text{SEQ}(s_1,s_2),e)$

$\text{BINOP}(op,\text{ESEQ}(s,e_1),e_2) \Rightarrow \text{ESEQ}(s,\text{BINOP}(op,e_1,e_2))$

$\text{BINOP}(op,e_1,\text{ESEQ}(s,e_2)) \Rightarrow \text{ESEQ}(s,\text{BINOP}(op,e_1,e_2))$ IF *e_1 and s commute*

$\text{BINOP}(op,e_1,\text{ESEQ}(s,e_2)) \Rightarrow$
$\text{ESEQ}(\text{SEQ}(\text{MOVE}(\text{TEMP}(t),e_1),s),\text{BINOP}(op,\text{TEMP}(t),e_2))$

And there are some more ...

Getting rid of $\mathrm{ESEQ}(s,e)$

Some transformations that move $\mathrm{ESEQ}(s,e)$ to top level

$\mathrm{ESEQ}(s_1,\mathrm{ESEQ}(s_2,e)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$

$\mathrm{BINOP}(op,\mathrm{ESEQ}(s,e_1),e_2) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$ IF e_1 *and* s *commute*

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t),e_1),s),\mathrm{BINOP}(op,\mathrm{TEMP}(t),e_2))$

And there are some more . . .

Getting rid of $\mathrm{ESEQ}(s,e)$

Some transformations that move $\mathrm{ESEQ}(s,e)$ to top level

$\mathrm{ESEQ}(s_1,\mathrm{ESEQ}(s_2,e)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$

$\mathrm{BINOP}(op,\mathrm{ESEQ}(s,e_1),e_2) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$ IF e_1 *and* s *commute*

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow$
$\mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t),e_1),s),\mathrm{BINOP}(op,\mathrm{TEMP}(t),e_2))$

And there are some more . . .

Getting rid of $\mathrm{ESEQ}(s,e)$

Some transformations that move $\mathrm{ESEQ}(s,e)$ to top level

$\mathrm{ESEQ}(s_1,\mathrm{ESEQ}(s_2,e)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$

$\mathrm{BINOP}(op,\mathrm{ESEQ}(s,e_1),e_2) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$ IF e_1 *and* s *commute*

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow$
$\mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t),e_1),s),\mathrm{BINOP}(op,\mathrm{TEMP}(t),e_2))$

And there are some more ...

Getting rid of $\mathrm{ESEQ}(s,e)$

Some transformations that move $\mathrm{ESEQ}(s,e)$ to top level

$\mathrm{ESEQ}(s_1,\mathrm{ESEQ}(s_2,e)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$

$\mathrm{BINOP}(op,\mathrm{ESEQ}(s,e_1),e_2) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$ IF e_1 *and* s *commute*

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t),e_1),s),\mathrm{BINOP}(op,\mathrm{TEMP}(t),e_2))$

And there are some more ...

Getting rid of $\mathrm{ESEQ}(s,e)$

Some transformations that move $\mathrm{ESEQ}(s,e)$ to top level

$\mathrm{ESEQ}(s_1,\mathrm{ESEQ}(s_2,e)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$

$\mathrm{BINOP}(op,\mathrm{ESEQ}(s,e_1),e_2) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$ IF *e_1 and s commute*

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow$
$\mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t),e_1),s),\mathrm{BINOP}(op,\mathrm{TEMP}(t),e_2))$

And there are some more . . .

Getting rid of $\mathrm{ESEQ}(s,e)$

Some transformations that move $\mathrm{ESEQ}(s,e)$ to top level

$\mathrm{ESEQ}(s_1,\mathrm{ESEQ}(s_2,e)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$

$\mathrm{BINOP}(op,\mathrm{ESEQ}(s,e_1),e_2) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$ IF e_1 *and* s *commute*

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow$
$\mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t),e_1),s),\mathrm{BINOP}(op,\mathrm{TEMP}(t),e_2))$

And there are some more ...

Getting rid of $\mathrm{ESEQ}(s,e)$

Some transformations that move $\mathrm{ESEQ}(s,e)$ to top level

$\mathrm{ESEQ}(s_1,\mathrm{ESEQ}(s_2,e)) \Rightarrow \mathrm{ESEQ}(\mathrm{SEQ}(s_1,s_2),e)$

$\mathrm{BINOP}(op,\mathrm{ESEQ}(s,e_1),e_2) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow \mathrm{ESEQ}(s,\mathrm{BINOP}(op,e_1,e_2))$ IF *e_1 and s commute*

$\mathrm{BINOP}(op,e_1,\mathrm{ESEQ}(s,e_2)) \Rightarrow$
$\mathrm{ESEQ}(\mathrm{SEQ}(\mathrm{MOVE}(\mathrm{TEMP}(t),e_1),s),\mathrm{BINOP}(op,\mathrm{TEMP}(t),e_2))$

And there are some more . . .

The algorithm

Extract subexpressions

Given a *Tree.Exp* or a *Tree.Stm* the subexpressions can be extracted calling method

abstract public LinkedList kids();

that is implemented adequately in each subclass of *Tree.Exp* and of *Tree.Stm*.

Example

CALL(NAME(*f*),CONST(*3*),MEM(CONST(*4*))).*kids()*

will yield the list

[NAME(*f*),CONST(*3*),MEM(CONST(*4*))].

The algorithm

Extract subexpressions

Given a *Tree.Exp* or a *Tree.Stm* the subexpressions can be extracted calling method

abstract public LinkedList kids();

that is implemented adequately in each subclass of *Tree.Exp* and of *Tree.Stm*.

Example

CALL(NAME(*f*),CONST(*3*),MEM(CONST(*4*))).*kids()*

will yield the list

[NAME(*f*),CONST(*3*),MEM(CONST(*4*))].

The algorithm

Reorder

A list of expressions can be reordered using the *rewrite rules* to one statement that does all the side effects and a list of expressions without $\mathrm{ESEQ}()$s

Example

For the list

$[e_1, e_2, \mathrm{ESEQ}(s, e_3)]$

the statement s must be pulled to the left past e_1 and e_2.

If s *commutes* with e_1 and e_2 reordering will yield

$(s, [e_1, e_2, e_3])$.

If s does not commute with e_1 and e_2 reordering will yield

$(\mathrm{SEQ}(\mathrm{MOVE}(t_1, e_1), \mathrm{SEQ}(\mathrm{MOVE}(t_2, e_2), s)), [\mathrm{TEMP}(t_1), \mathrm{TEMP}(t_2), e_3])$

The algorithm

Reorder

A list of expressions can be reordered using the *rewrite rules* to one statement that does all the side effects and a list of expressions without $\mathrm{ESEQ}()$s

Example

For the list

$[e_1, e_2, \mathrm{ESEQ}(s, e_3)]$

the statement s must be pulled to the left past e_1 and e_2.

If s *commutes* with e_1 and e_2 reordering will yield

$(s, [e_1, e_2, e_3])$.

If s does not commute with e_1 and e_2 reordering will yield

$(\mathrm{SEQ}(\mathrm{MOVE}(t_1, e_1), \mathrm{SEQ}(\mathrm{MOVE}(t_2, e_2), s)), [\mathrm{TEMP}(t_1), \mathrm{TEMP}(t_2), e_3])$

The algorithm

Building up

Once the expressions have been reordered they must be used to form back the corresponding *Tree.Exp* or *Tree.Stm*. That is why we find methods

- *public abstract Tree.Exp build(ExpList kids)* in class *Tree.Exp*
- *public abstract Tree.Stm build(ExpList kids)* in class *Tree.Stm*

That are implemented adequtely in each of their subclasses.

Example

```
class BINOP extends Exp{
   public Exp build(ExpList kids){
       return new BINOP(binop,kids.head,kids.tail.head);}
}
```

The algorithm

Building up

Once the expressions have been reordered they must be used to form back the corresponding *Tree.Exp* or *Tree.Stm*. That is why we find methods

- *public abstract Tree.Exp build(ExpList kids)* in class *Tree.Exp*
- *public abstract Tree.Stm build(ExpList kids)* in class *Tree.Stm*

That are implemented adequtely in each of their subclasses.

Example

```
class BINOP extends Exp{
   public Exp build(ExpList kids){
       return new BINOP(binop,kids.head,kids.tail.head);}
}
```

The algorithm

Building up

Once the expressions have been reordered they must be used to form back the corresponding *Tree.Exp* or *Tree.Stm*. That is why we find methods

- *public abstract Tree.Exp build(ExpList kids)* in class *Tree.Exp*
- *public abstract Tree.Stm build(ExpList kids)* in class *Tree.Stm*

That are implemented adequtely in each of their subclasses.

Example

```
class BINOP extends Exp{
  public Exp build(ExpList kids){
     return new BINOP(binop,kids.head,kids.tail.head);}
}
```

The algorithm

Building up

Once the expressions have been reordered they must be used to form back the corresponding *Tree.Exp* or *Tree.Stm*. That is why we find methods

- *public abstract Tree.Exp build(ExpList kids)* in class *Tree.Exp*
- *public abstract Tree.Stm build(ExpList kids)* in class *Tree.Stm*

That are implemented adequtely in each of their subclasses.

Example

```
class BINOP extends Exp{
  public Exp build(ExpList kids){
     return new BINOP(binop,kids.head,kids.tail.head);}
}
```

The algorithm

Building up

Once the expressions have been reordered they must be used to form back the corresponding *Tree.Exp* or *Tree.Stm*. That is why we find methods

- *public abstract Tree.Exp build(ExpList kids)* in class *Tree.Exp*
- *public abstract Tree.Stm build(ExpList kids)* in class *Tree.Stm*

That are implemented adequtely in each of their subclasses.

Example

```
class BINOP extends Exp{
   public Exp build(ExpList kids){
      return new BINOP(binop,kids.head,kids.tail.head);}
}
```

The algorithm

Building up

Once the expressions have been reordered they must be used to form back the corresponding *Tree.Exp* or *Tree.Stm*. That is why we find methods

- *public abstract Tree.Exp build(ExpList kids)* in class *Tree.Exp*
- *public abstract Tree.Stm build(ExpList kids)* in class *Tree.Stm*

That are implemented adequtely in each of their subclasses.

Example

```
class BINOP extends Exp{
   public Exp build(ExpList kids){
      return new BINOP(binop,kids.head,kids.tail.head);}
}
```

IR-trees (remarks)

- Another issue is the possibility of using $\textsc{call}(f,args)$ as subexpressions.
- All functions return their result in the dedicated register $\mathtt{RV}$.Thus, in

$$\textsc{binop}(op,\textsc{call}(\ldots),\textsc{call}(\ldots))$$

  the second call will overwrite $\mathtt{RV}$ before the operation can be executed!
- All $\textsc{call}(f,args)$ are replaced by

$$\textsc{eseq}(\textsc{move}(\textsc{temp}(t),\textsc{call}(f,args)),\textsc{temp}(t))$$

  before eliminating $\textsc{eseq}()$s

IR-trees (remarks)

- Another issue is the possibility of using $\text{CALL}(f, args)$ as subexpressions.
- All functions return their result in the dedicated register $\texttt{RV}$. Thus, in

  $$\text{BINOP}(op, \text{CALL}(\ldots), \text{CALL}(\ldots))$$

  the second call will overwrite $\texttt{RV}$ before the operation can be executed!
- All $\text{CALL}(f, args)$ are replaced by

  $$\text{ESEQ}(\text{MOVE}(\text{TEMP}(t), \text{CALL}(f, args)), \text{TEMP}(t))$$

  before eliminating $\text{ESEQ}()$s

IR-trees (remarks)

- Another issue is the possibility of using $\text{CALL}(f{,}args)$ as subexpressions.
- All functions return their result in the dedicated register `RV`. Thus, in

$$\text{BINOP}(op,\text{CALL}(\ldots),\text{CALL}(\ldots))$$

  the second call will overwrite `RV` before the operation can be executed!
- All $\text{CALL}(f{,}args)$ are replaced by

$$\text{ESEQ}(\text{MOVE}(\text{TEMP}(t),\text{CALL}(f,args)),\text{TEMP}(t))$$

  before eliminating $\text{ESEQ}()$s

IR-trees (remarks)

- Another issue is the possibility of using CALL(*f*,*args*) as subexpressions.
- All functions return their result in the dedicated register `RV`. Thus, in

  $$\text{BINOP}(op,\text{CALL}(\ldots),\text{CALL}(\ldots))$$

  the second call will overwrite `RV` before the operation can be executed!
- All CALL(*f*,*args*) are replaced by

  $$\text{ESEQ}(\text{MOVE}(\text{TEMP}(t),\text{CALL}(f,args)),\text{TEMP}(t))$$

  before eliminating ESEQ()s

IR-trees (remarks)

- Another issue is the possibility of using $\text{CALL}(f,args)$ as subexpressions.
- All functions return their result in the dedicated register `RV`.Thus, in

$$\text{BINOP}(op,\text{CALL}(\dots),\text{CALL}(\dots))$$

  the second call will overwrite `RV` before the operation can be executed!
- All $\text{CALL}(f,args)$ are replaced by

$$\text{ESEQ}(\text{MOVE}(\text{TEMP}(t),\text{CALL}(f,args)),\text{TEMP}(t))$$

  before eliminating $\text{ESEQ}()$s

IR-trees (remarks)

- Another issue is the possibility of using $\textsc{call}(f,args)$ as subexpressions.
- All functions return their result in the dedicated register `RV`. Thus, in

  $$\textsc{binop}(op,\textsc{call}(\ldots),\textsc{call}(\ldots))$$

  the second call will overwrite `RV` before the operation can be executed!
- All $\textsc{call}(f,args)$ are replaced by

  $$\textsc{eseq}(\textsc{move}(\textsc{temp}(t),\textsc{call}(f,args)),\textsc{temp}(t))$$

  before eliminating $\textsc{eseq}()$s

IR-trees (remarks)

- After these transformations, $\textsc{seq}()$ nodes can only appear as children to other $\textsc{seq}()$ nodes. As no expressions remain that have statements as children, and only the $\textsc{seq}()$ statement has statements as children!
- The transformation

  $$\textsc{seq}(\textsc{seq}(a,b),c) \Rightarrow \textsc{seq}(a,\textsc{seq}(b,c))$$

  can be used to linearize the sequence structure.
- Now $\textsc{seq}()$ nodes are redundant! we can as well have a list with the statements in the sequence!

IR-trees (remarks)

- After these transformations, SEQ() nodes can only appear as children to other SEQ() nodes. As no expressions remain that have statements as children, and only the SEQ() statement has statements as children!
- The transformation

$$\mathrm{SEQ}(\mathrm{SEQ}(a,b),c) \Rightarrow \mathrm{SEQ}(a,\mathrm{SEQ}(b,c))$$

  can be used to linearize the sequence structure.
- Now SEQ() nodes are redundant! we can as well have a list with the statements in the sequence!

IR-trees (remarks)

- After these transformations, $\textsc{seq}()$ nodes can only appear as children to other $\textsc{seq}()$ nodes. As no expressions remain that have statements as children, and only the $\textsc{seq}()$ statement has statements as children!
- The transformation

$$\textsc{seq}(\textsc{seq}(a,b),c) \Rightarrow \textsc{seq}(a,\textsc{seq}(b,c))$$

can be used to linearize the sequence structure.
- Now $\textsc{seq}()$ nodes are redundant! we can as well have a list with the statements in the sequence!

IR-trees (remarks)

- After these transformations, $\textsc{seq}()$ nodes can only appear as children to other $\textsc{seq}()$ nodes. As no expressions remain that have statements as children, and only the $\textsc{seq}()$ statement has statements as children!
- The transformation

$$\textsc{seq}(\textsc{seq}(a,b),c) \Rightarrow \textsc{seq}(a,\textsc{seq}(b,c))$$

  can be used to linearize the sequence structure.
- Now $\textsc{seq}()$ nodes are redundant! we can as well have a list with the statements in the sequence!

IR-trees (remarks)

- After these transformations, $\textsc{seq}()$ nodes can only appear as children to other $\textsc{seq}()$ nodes. As no expressions remain that have statements as children, and only the $\textsc{seq}()$ statement has statements as children!
- The transformation

$$\textsc{seq}(\textsc{seq}(a,b),c) \Rightarrow \textsc{seq}(a,\textsc{seq}(b,c))$$

can be used to linearize the sequence structure.

- Now $\textsc{seq}()$ nodes are redundant! we can as well have a list with the statements in the sequence!

IR-trees (remarks)

- After these transformations, $\textsc{seq}()$ nodes can only appear as children to other $\textsc{seq}()$ nodes. As no expressions remain that have statements as children, and only the $\textsc{seq}()$ statement has statements as children!
- The transformation

  $$\textsc{seq}(\textsc{seq}(a,b),c) \Rightarrow \textsc{seq}(a,\textsc{seq}(b,c))$$

  can be used to linearize the sequence structure.
- Now $\textsc{seq}()$ nodes are redundant! we can as well have a list with the statements in the sequence!

IR-trees (remarks)

- After these transformations, $\textsc{seq}()$ nodes can only appear as children to other $\textsc{seq}()$ nodes. As no expressions remain that have statements as children, and only the $\textsc{seq}()$ statement has statements as children!
- The transformation

  $$\textsc{seq}(\textsc{seq}(a,b),c) \Rightarrow \textsc{seq}(a,\textsc{seq}(b,c))$$

  can be used to linearize the sequence structure.
- Now $\textsc{seq}()$ nodes are redundant! we can as well have a list with the statements in the sequence!

IR-trees (remarks)

- After these transformations, $\textsc{seq}()$ nodes can only appear as children to other $\textsc{seq}()$ nodes. As no expressions remain that have statements as children, and only the $\textsc{seq}()$ statement has statements as children!
- The transformation

  $$\textsc{seq}(\textsc{seq}(a,b),c) \Rightarrow \textsc{seq}(a,\textsc{seq}(b,c))$$

  can be used to linearize the sequence structure.
- Now $\textsc{seq}()$ nodes are redundant! we can as well have a list with the statements in the sequence!

IR-trees (remarks)

- To resume,
  - the translation phase resulted in an IR-tree statement for every method body,
  - the transformations sketched turned each such statement into a list of atomic statements without $\mathrm{ESEQ}()$ expression nodes.
- Before producing assembler we will rearange the lists of statements so that all

$$\mathrm{CJUMP}(op, e_1, e_2, l_t, l_f)$$

are followed by the statement under l_f
(so that they can be translated with the more common jumps!)

IR-trees (remarks)

- To resume,
  - the translation phase resulted in an IR-tree statement for every method body,
  - the transformations sketched turned each such statement into a list of atomic statements without $\mathrm{ESEQ}()$ expression nodes.
- Before producing assembler we will rearange the lists of statements so that all

  $$\mathrm{CJUMP}(op, e_1, e_2, l_t, l_f)$$

  are followed by the statement under l_f
  (so that they can be translated with the more common jumps!)

IR-trees (remarks)

- To resume,
  - the translation phase resulted in an IR-tree statement for every method body,
  - the transformations sketched turned each such statement into a list of atomic statements without $\mathrm{ESEQ}()$ expression nodes.
- Before producing assembler we will rearange the lists of statements so that all

$$\mathrm{CJUMP}(op, e_1, e_2, l_t, l_f)$$

  are followed by the statement under l_f
  (so that they can be translated with the more common jumps!)

IR-trees (remarks)

- To resume,
  - the translation phase resulted in an IR-tree statement for every method body,
  - the transformations sketched turned each such statement into a list of atomic statements without $\mathrm{ESEQ}()$ expression nodes.
- Before producing assembler we will rearange the lists of statements so that all

$$\mathrm{CJUMP}(op, e_1, e_2, l_t, l_f)$$

are followed by the statement under l_f
(so that they can be translated with the more common jumps!)

IR-trees (remarks)

- To resume,
  - the translation phase resulted in an IR-tree statement for every method body,
  - the transformations sketched turned each such statement into a list of atomic statements without $\mathrm{ESEQ}()$ expression nodes.
- Before producing assembler we will rearange the lists of statements so that all

  $$\mathrm{CJUMP}(op, e_1, e_2, l_t, l_f)$$

  are followed by the statement under l_f
  (so that they can be translated with the more common jumps!)

IR-trees (remarks)

- To resume,
  - the translation phase resulted in an IR-tree statement for every method body,
  - the transformations sketched turned each such statement into a list of atomic statements without $\mathrm{ESEQ}()$ expression nodes.
- Before producing assembler we will rearange the lists of statements so that all

$$\mathrm{CJUMP}(op, e_1, e_2, l_t, l_f)$$

are followed by the statement under l_f
(so that they can be translated with the more common jumps!)

IR-trees (remarks)

- To resume,
  - the translation phase resulted in an IR-tree statement for every method body,
  - the transformations sketched turned each such statement into a list of atomic statements without $\mathrm{ESEQ}()$ expression nodes.
- Before producing assembler we will rearange the lists of statements so that all

  $$\mathrm{CJUMP}(op, e_1, e_2, l_t, l_f)$$

  are followed by the statement under l_f

  (so that they can be translated with the more common jumps!)

IR-trees (remarks)

- To resume,
  - the translation phase resulted in an IR-tree statement for every method body,
  - the transformations sketched turned each such statement into a list of atomic statements without $\mathrm{ESEQ}()$ expression nodes.
- Before producing assembler we will rearange the lists of statements so that all

  $$\mathrm{CJUMP}(op, e_1, e_2, l_t, l_f)$$

  are followed by the statement under l_f
  (so that they can be translated with the more common jumps!)

Basic blocks and traces

- To implement this last transformation of the list of statements,a bit of control flow analysis is done.
- First, basic blocks of statements are identified and put together. A basic block is a list of statements where
  - the first statement is a LABEL()
  - the last statement is a JUMP() or CJUMP()
  - there are no other LABEL(), JUMP() or CJUMP() in it.

  a basic block is entered at the begining and exited at the end!
- Basic blocks can be rearanged in any order without altering results!

Basic blocks and traces

- To implement this last transformation of the list of statements,a bit of control flow analysis is done.
- First, basic blocks of statements are identified and put together. A basic block is a list of statements where
  - the first statement is a LABEL()
  - the last statement is a JUMP() or CJUMP()
  - there are no other LABEL(), JUMP() or CJUMP() in it.

  a basic block is entered at the begining and exited at the end!
- Basic blocks can be rearanged in any order without altering results!

Basic blocks and traces

- To implement this last transformation of the list of statements,a bit of control flow analysis is done.
- First, basic blocks of statements are identified and put together. A basic block is a list of statements where
  - the first statement is a LABEL()
  - the last statement is a JUMP() or CJUMP()
  - there are no other LABEL(), JUMP() or CJUMP() in it.

  a basic block is entered at the begining and exited at the end!
- Basic blocks can be rearanged in any order without altering results!

Basic blocks and traces

- To implement this last transformation of the list of statements,a bit of control flow analysis is done.
- First, basic blocks of statements are identified and put together. A basic block is a list of statements where
  - the first statement is a LABEL()
  - the last statement is a JUMP() or CJUMP()
  - there are no other LABEL(), JUMP() or CJUMP() in it.

  a basic block is entered at the begining and exited at the end!
- Basic blocks can be rearanged in any order without altering results!

Basic blocks and traces

- To implement this last transformation of the list of statements,a bit of control flow analysis is done.
- First, basic blocks of statements are identified and put together. A basic block is a list of statements where
  - the first statement is a LABEL()
  - the last statement is a JUMP() or CJUMP()
  - there are no other LABEL(), JUMP() or CJUMP() in it.

  a basic block is entered at the begining and exited at the end!
- Basic blocks can be rearanged in any order without altering results!

Basic blocks and traces

- To implement this last transformation of the list of statements,a bit of control flow analysis is done.
- First, basic blocks of statements are identified and put together. A basic block is a list of statements where
  - the first statement is a LABEL()
  - the last statement is a JUMP() or CJUMP()
  - there are no other LABEL(), JUMP() or CJUMP() in it.

  a basic block is entered at the begining and exited at the end!
- Basic blocks can be rearanged in any order without altering results!

Basic blocks and traces

- To implement this last transformation of the list of statements,a bit of control flow analysis is done.
- First, basic blocks of statements are identified and put together. A basic block is a list of statements where
  - the first statement is a LABEL()
  - the last statement is a JUMP() or CJUMP()
  - there are no other LABEL(), JUMP() or CJUMP() in it.

  a basic block is entered at the begining and exited at the end!
- Basic blocks can be rearanged in any order without altering results!

Basic blocks and traces

- To implement this last transformation of the list of statements,a bit of control flow analysis is done.
- First, basic blocks of statements are identified and put together. A basic block is a list of statements where
  - the first statement is a LABEL()
  - the last statement is a JUMP() or CJUMP()
  - there are no other LABEL(), JUMP() or CJUMP() in it.

  a basic block is entered at the begining and exited at the end!
- Basic blocks can be rearanged in any order without altering results!

Basic Blocks - the algorithm

Splitting

The list of canonical IR-trees corresponding to a function body is inspected from first to last element.

- On a $\mathrm{LABEL}()$, start a new block!
- On a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$, end a block!
- On any other statement, add it to the current block!

Completing

- If a block doesn't end with a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$ add to it a $\mathrm{JUMP}()$ to the label of the following block.
- If a block doesn't start with a $\mathrm{LABEL}()$, create one and stick it to it!

Basic Blocks - the algorithm

Splitting

The list of canonical IR-trees corresponding to a function body is inspected from first to last element.

- On a LABEL(), start a new block!
- On a JUMP() or CJUMP(), end a block!
- On any other statement, add it to the current block!

Completing

- If a block doesn't end with a JUMP() or CJUMP() add to it a JUMP() to the label of the following block.
- If a block doesn't start with a LABEL(), create one and stick it to it!

Basic Blocks - the algorithm

Splitting

The list of canonical IR-trees corresponding to a function body is inspected from first to last element.

- On a $\mathrm{LABEL}()$, start a new block!
- On a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$, end a block!
- On any other statement, add it to the current block!

Completing

- If a block doesn't end with a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$ add to it a $\mathrm{JUMP}()$ to the label of the following block.
- If a block doesn't start with a $\mathrm{LABEL}()$, create one and stick it to it!

Basic Blocks - the algorithm

Splitting

The list of canonical IR-trees corresponding to a function body is inspected from first to last element.

- On a $\mathrm{LABEL}()$, start a new block!
- On a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$, end a block!
- On any other statement, add it to the current block!

Completing

- If a block doesn't end with a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$ add to it a $\mathrm{JUMP}()$ to the label of the following block.
- If a block doesn't start with a $\mathrm{LABEL}()$, create one and stick it to it!

Basic Blocks - the algorithm

Splitting

The list of canonical IR-trees corresponding to a function body is inspected from first to last element.

- On a $\mathrm{LABEL}()$, start a new block!
- On a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$, end a block!
- On any other statement, add it to the current block!

Completing

- If a block doesn't end with a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$ add to it a $\mathrm{JUMP}()$ to the label of the following block.
- If a block doesn't start with a $\mathrm{LABEL}()$, create one and stick it to it!

Basic Blocks - the algorithm

Splitting

The list of canonical IR-trees corresponding to a function body is inspected from first to last element.

- On a $\mathrm{LABEL}()$, start a new block!
- On a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$, end a block!
- On any other statement, add it to the current block!

Completing

- If a block doesn't end with a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$ add to it a $\mathrm{JUMP}()$ to the label of the following block.
- If a block doesn't start with a $\mathrm{LABEL}()$, create one and stick it to it!

Basic Blocks - the algorithm

Splitting

The list of canonical IR-trees corresponding to a function body is inspected from first to last element.

- On a $\mathrm{LABEL}()$, start a new block!
- On a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$, end a block!
- On any other statement, add it to the current block!

Completing

- If a block doesn't end with a $\mathrm{JUMP}()$ or $\mathrm{CJUMP}()$ add to it a $\mathrm{JUMP}()$ to the label of the following block.
- If a block doesn't start with a $\mathrm{LABEL}()$, create one and stick it to it!

Basic blocks and traces

- Basic blocks are put together into a traces.
- A trace is a sequence of statements that *could* be executed sequentialy.
- We want to produce a set of traces such that
  - every basic block is in some trace,
  - every CJUMP() is followed by its false label
  - many JUMP()s are followed by the label they jump to (because they can then be eliminated!)
- First, a covering set of traces is built by starting with an arbitrary block and adding its *execution successors* to the trace until there are no more successors to add. Then another trace is started.

Basic blocks and traces

- Basic blocks are put together into a traces.
- A trace is a sequence of statements that *could* be executed sequentialy.
- We want to produce a set of traces such that
  - every basic block is in some trace,
  - every CJUMP() is followed by its false label
  - many JUMP()s are followed by the label they jump to (because they can then be eliminated!)
- First, a covering set of traces is built by starting with an arbitrary block and adding its *execution successors* to the trace until there are no more successors to add. Then another trace is started.

Basic blocks and traces

- Basic blocks are put together into a traces.
- A trace is a sequence of statements that *could* be executed sequentialy.
- We want to produce a set of traces such that
  - every basic block is in some trace,
  - every CJUMP() is followed by its false label
  - many JUMP()s are followed by the label they jump to (because they can then be eliminated!)
- First, a covering set of traces is built by starting with an arbitrary block and adding its *execution successors* to the trace until there are no more successors to add. Then another trace is started.

Basic blocks and traces

- Basic blocks are put together into a traces.
- A trace is a sequence of statements that *could* be executed sequentialy.
- We want to produce a set of traces such that
  - every basic block is in some trace,
  - every CJUMP() is followed by its false label
  - many JUMP()s are followed by the label they jump to (because they can then be eliminated!)
- First, a covering set of traces is built by starting with an arbitrary block and adding its *execution successors* to the trace until there are no more successors to add. Then another trace is started.

Basic blocks and traces

- Basic blocks are put together into a traces.
- A trace is a sequence of statements that *could* be executed sequentialy.
- We want to produce a set of traces such that
  - every basic block is in some trace,
  - every CJUMP() is followed by its false label
  - many JUMP()s are followed by the label they jump to (because they can then be eliminated!)
- First, a covering set of traces is built by starting with an arbitrary block and adding its *execution successors* to the trace until there are no more successors to add. Then another trace is started.

Basic blocks and traces

- Basic blocks are put together into a traces.
- A trace is a sequence of statements that *could* be executed sequentialy.
- We want to produce a set of traces such that
  - every basic block is in some trace,
  - every CJUMP() is followed by its false label
  - many JUMP()s are followed by the label they jump to (because they can then be eliminated!)
- First, a covering set of traces is built by starting with an arbitrary block and adding its *execution successors* to the trace until there are no more successors to add. Then another trace is started.

Basic blocks and traces

- Basic blocks are put together into a traces.
- A trace is a sequence of statements that *could* be executed sequentialy.
- We want to produce a set of traces such that
  - every basic block is in some trace,
  - every CJUMP() is followed by its false label
  - many JUMP()s are followed by the label they jump to (because they can then be eliminated!)
- First, a covering set of traces is built by starting with an arbitrary block and adding its *execution successors* to the trace until there are no more successors to add. Then another trace is started.

Basic blocks and traces

- Basic blocks are put together into a traces.
- A trace is a sequence of statements that *could* be executed sequentialy.
- We want to produce a set of traces such that
  - every basic block is in some trace,
  - every CJUMP() is followed by its false label
  - many JUMP()s are followed by the label they jump to (because they can then be eliminated!)
- First, a covering set of traces is built by starting with an arbitrary block and adding its *execution successors* to the trace until there are no more successors to add. Then another trace is started.

Basic blocks and traces

- Then these traces are further adjusted by considering the CJUMP()s in them:
  - A CJUMP() followed by its false label is left untouched.
  - For a CJUMP() followed by its true label, labels are switched and the condition negated.
  - For a CJUMP() followed by neither label

a new label is invented l'
the CJUMP() is replaced by:
CJUMP(op, e_1, e_2, l_t, l')
LABEL(l')
JUMP(l_f)

Basic blocks and traces

- Then these traces are further adjusted by considering the CJUMP()s in them:
  - A CJUMP() followed by its false label is left untouched.
  - For a CJUMP() followed by its true label, labels are switched and the condition negated.
  - For a CJUMP() followed by neither label

    a new label is invented l'
    the CJUMP() is replaced by:
    CJUMP(op,e_1,e_2,l_t,l')
    LABEL(l')
    JUMP(l_f)

Basic blocks and traces

- Then these traces are further adjusted by considering the CJUMP()s in them:
  - A CJUMP() followed by its false label is left untouched.
  - For a CJUMP() followed by its true label, labels are switched and the condition negated.
  - For a CJUMP() followed by neither label

a new label is invented l'
the CJUMP() is replaced by:
CJUMP(op, e_1, e_2, l_t, l')
LABEL(l')
JUMP(l_f)

Basic blocks and traces

- Then these traces are further adjusted by considering the CJUMP()s in them:
  - A CJUMP() followed by its false label is left untouched.
  - For a CJUMP() followed by its true label, labels are switched and the condition negated.
  - For a CJUMP() followed by neither label

    a new label is invented l'
    the CJUMP() is replaced by:
    CJUMP(op, e_1, e_2, l_t, l')
    LABEL(l')
    JUMP(l_f)

Basic blocks and traces

- Then these traces are further adjusted by considering the CJUMP()s in them:
  - A CJUMP() followed by its false label is left untouched.
  - For a CJUMP() followed by its true label, labels are switched and the condition negated.
  - For a CJUMP() followed by neither label

    a new label is invented l'
    the CJUMP() is replaced by:
    CJUMP(op, e_1, e_2, l_t, l')
    LABEL(l')
    JUMP(l_f)

Basic blocks and traces

- Then these traces are further adjusted by considering the CJUMP()s in them:
  - A CJUMP() followed by its false label is left untouched.
  - For a CJUMP() followed by its true label, labels are switched and the condition negated.
  - For a CJUMP() followed by neither label

a new label is invented l'

the CJUMP() is replaced by:

CJUMP(op, e_1, e_2, l_t, l')

LABEL(l')

JUMP(l_f)

Basic blocks and traces

- Then these traces are further adjusted by considering the CJUMP()s in them:
  - A CJUMP() followed by its false label is left untouched.
  - For a CJUMP() followed by its true label, labels are switched and the condition negated.
  - For a CJUMP() followed by neither label

a new label is invented l'
the CJUMP() is replaced by:
CJUMP(op, e_1, e_2, l_t, l')
LABEL(l')
JUMP(l_f)

Basic blocks and traces

- Then these traces are further adjusted by considering the CJUMP()s in them:
  - A CJUMP() followed by its false label is left untouched.
  - For a CJUMP() followed by its true label, labels are switched and the condition negated.
  - For a CJUMP() followed by neither label

a new label is invented l'
the CJUMP() is replaced by:
CJUMP(op, e_1, e_2, l_t, l')
LABEL(l')
JUMP(l_f)

The package canon

There is an implementation of these transformations that comes with the book. For it to run with the rest of the compiler I've made some small modifications (you will get it with the distribution for part five of the project)

package canon;

Canon removes ESEQ(), assigns CALL() to registers and returns a list of atomic statements

BasicBlocks puts together statements into basic blocks

TraceSchedule puts together blocks into traces and flattens back to a list of statements

The package canon

There is an implementation of these transformations that comes with the book. For it to run with the rest of the compiler I've made some small modifications (you will get it with the distribution for part five of the project)

package canon;

Canon removes ESEQ(), assigns CALL() to registers and returns a list of atomic statements

BasicBlocks puts together statements into basic blocks

TraceSchedule puts together blocks into traces and flattens back to a list of statements

The package canon

There is an implementation of these transformations that comes with the book. For it to run with the rest of the compiler I've made some small modifications (you will get it with the distribution for part five of the project)

package canon;

Canon removes ESEQ(), assigns CALL() to registers and returns a list of atomic statements

BasicBlocks puts together statements into basic blocks

TraceSchedule puts together blocks into traces and flattens back to a list of statements

The package canon

There is an implementation of these transformations that comes with the book. For it to run with the rest of the compiler I've made some small modifications (you will get it with the distribution for part five of the project)

package canon;

Canon removes ESEQ(), assigns CALL() to registers and returns a list of atomic statements

BasicBlocks puts together statements into basic blocks

TraceSchedule puts together blocks into traces and flattens back to a list of statements

The package canon

There is an implementation of these transformations that comes with the book. For it to run with the rest of the compiler I've made some small modifications (you will get it with the distribution for part five of the project)

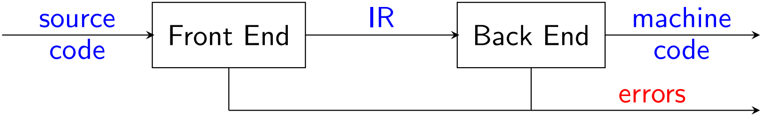
package canon;

Canon removes ESEQ(), assigns CALL() to registers and returns a list of atomic statements

BasicBlocks puts together statements into basic blocks

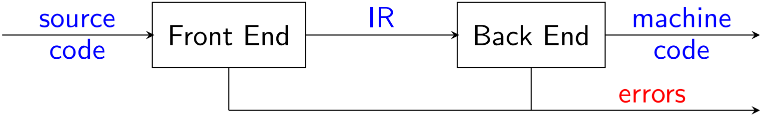
TraceSchedule puts together blocks into traces and flattens back to a list of statements

The back-end



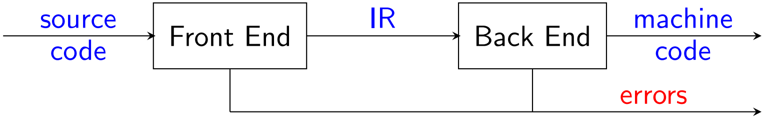
The back-end is also structured in phases!

The back-end

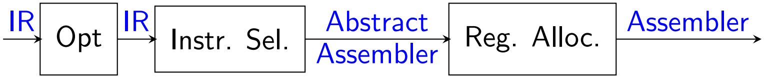


The back-end is also structured in phases!

The back-end



The back-end is also structured in phases!



Observation

For the back-end we no longer look at *minijava*! We compile IR-trees to assembler.

A small minijava program

```
class A{
   public static void main(String[] a){
      System.out.println(new B().f(3).f());
   }
}
class B{
   int x;
   int y;
   public C f(int z){
      if(x<y) x=z+1; else x=z+x;
      return new C();
   }
}
class C{
   public int f(){return 3;}
}
```

The result of translating it

```
PROCEDURE :main
EXPS(
 CALL(
  NAME _printint,
  CONST 0,
  CALL(
   NAME C_f,
   CALL(
    NAME B_f,
    CALL(
     NAME _malloc,
     CONST 0,
     CONST 8),
    CONST 3))))
```

```
PROCEDURE :B_f
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
PROCEDURE :C_f
MOVE(
 TEMP t32,
 CONST 3)
```

The result of translating it

```
PROCEDURE :main
EXPS(
 CALL(
  NAME _printint,
  CONST 0,
  CALL(
   NAME C_f,
   CALL(
    NAME B_f,
    CALL(
     NAME _malloc,
     CONST 0,
     CONST 8),
    CONST 3))))
```

```
PROCEDURE :B_f
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
PROCEDURE :C_f
MOVE(
 TEMP t32,
 CONST 3)
```

The result of translating it

```
PROCEDURE :main
EXPS(
 CALL(
  NAME _printint,
  CONST 0,
  CALL(
   NAME C_f,
   CALL(
    NAME B_f,
    CALL(
     NAME _malloc,
     CONST 0,
     CONST 8),
    CONST 3))))
```

```
PROCEDURE :B_f
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
PROCEDURE :C_f
MOVE(
 TEMP t32,
 CONST 3)
```

(almost)All of `B_f`

```
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
SEQ(
 SEQ(
  SEQ(
   LABEL L0,
   MOVE(
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0)),
    BINOP(PLUS,
     TEMP t65,
     CONST 1))),
  JUMP(
   NAME L2)),
```

```
SEQ(
 SEQ(
  LABEL L1,
  MOVE(
   MEM(
    BINOP(PLUS,
     TEMP t64,
     CONST 0)),
   BINOP(PLUS,
    TEMP t65,
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0))))),
 JUMP(
  NAME L2)))),
```

(almost)All of `B_f`

```
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
SEQ(
 SEQ(
  SEQ(
   LABEL L0,
   MOVE(
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0)),
    BINOP(PLUS,
     TEMP t65,
     CONST 1))),
  JUMP(
   NAME L2)),
```

```
SEQ(
 SEQ(
  LABEL L1,
  MOVE(
   MEM(
    BINOP(PLUS,
     TEMP t64,
     CONST 0)),
   BINOP(PLUS,
    TEMP t65,
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0))))),
 JUMP(
  NAME L2))),
```

(almost)All of `B_f`

```
MOVE(
 TEMP t32,
 ESEQ(
  SEQ(
   SEQ(
    CJUMP(LT,
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 0)),
     MEM(
      BINOP(PLUS,
       TEMP t64,
       CONST 4)),
     L0,L1),
```

```
SEQ(
 SEQ(
  SEQ(
   LABEL L0,
   MOVE(
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0)),
    BINOP(PLUS,
     TEMP t65,
     CONST 1))),
  JUMP(
   NAME L2)),
```

```
SEQ(
 SEQ(
  LABEL L1,
  MOVE(
   MEM(
    BINOP(PLUS,
     TEMP t64,
     CONST 0)),
   BINOP(PLUS,
    TEMP t65,
    MEM(
     BINOP(PLUS,
      TEMP t64,
      CONST 0))))),
 JUMP(
  NAME L2)))),
```

B_f - after the transformations

```
LABEL L6
CJUMP(LT,
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 4)),
 L0,L1)
```

```
LABEL L1
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  MEM(
   BINOP(PLUS,
    TEMP t64,
    CONST 0))))
```

```
LABEL L2
MOVE(
 TEMP t32,
 CALL(
  NAME _malloc,
  CONST 0,
  CONST 0))
JUMP(
 NAME L5)
```

```
LABEL L0
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  CONST 1))
JUMP(
 NAME L2)
LABEL L5
```

B_f - after the transformations

```
LABEL L6
CJUMP(LT,
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 4)),
 L0,L1)
```

```
LABEL L1
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  MEM(
   BINOP(PLUS,
    TEMP t64,
    CONST 0))))
```

```
LABEL L2
MOVE(
 TEMP t32,
 CALL(
  NAME _malloc,
  CONST 0,
  CONST 0))
JUMP(
 NAME L5)
```

```
LABEL L0
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  CONST 1))
JUMP(
 NAME L2)
LABEL L5
```

B_f - after the transformations

```
LABEL L6
CJUMP(LT,
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 4)),
 L0,L1)
```

```
LABEL L1
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  MEM(
   BINOP(PLUS,
    TEMP t64,
    CONST 0))))
```

```
LABEL L2
MOVE(
 TEMP t32,
 CALL(
  NAME _malloc,
  CONST 0,
  CONST 0))
JUMP(
 NAME L5)
```

```
LABEL L0
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  CONST 1))
JUMP(
 NAME L2)
LABEL L5
```

B_f - after the transformations

```
LABEL L6
CJUMP(LT,
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 4)),
 L0,L1)
```

```
LABEL L1
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  MEM(
   BINOP(PLUS,
    TEMP t64,
    CONST 0))))
```

```
LABEL L2
MOVE(
 TEMP t32,
 CALL(
  NAME _malloc,
  CONST 0,
  CONST 0))
JUMP(
 NAME L5)
```

```
LABEL L0
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t64,
   CONST 0)),
 BINOP(PLUS,
  TEMP t65,
  CONST 1))
JUMP(
 NAME L2)
LABEL L5
```

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.

What is left: code generation

Purpose: Generate a file with assembler code for a target machine

Instruction Selection

- Study the instructions of the target architecture.
- Program how to match each IR statement with machine instructions.
- For each instruction keep a list of the temporaries used.

Register Allocation

- Build a *flow graph* where instructions are nodes and edges reflect usage of temporaries.
- *Color* the graph to find independent temporaries.
- Assign registers to instructions.