Compiler Construction

Week 8: Code generation II & Extensions

Sjaak Smetsers Mart Lubbers Jordy Aaldering

2024/2025 KW3

Radboud University

Notices

Extensions Ideas

Higher order functions

Garbage collection
- Tracing Garbage Collection
- Reference Counting

Real architectures

Conclusion

Notices

Notices

- Make sure your git repo is kept up to date

Notices

- Make sure your git repo is kept up to date
- Oral exams: June 16^{th}, 17^{th} and 19^{th}

Notices

- Make sure your git repo is kept up to date
- Oral exams: June 16^{th}, 17^{th} and 19^{th}
- Pick a timeslot by visiting my office door (with a pen, [M1 01.07])

Notices

- Make sure your git repo is kept up to date
- Oral exams: June 16$^{\text{th}}$, 17$^{\text{th}}$ and 19$^{\text{th}}$
- Pick a timeslot by visiting my office door (with a pen, [M1 01.07])
- Extension proposal deadline (May 5$^{\text{th}}$ at 23:59)

Notices

- Make sure your git repo is kept up to date
- Oral exams: June 16$^{\text{th}}$, 17$^{\text{th}}$ and 19$^{\text{th}}$
- Pick a timeslot by visiting my office door (with a pen, [M1 01.07])
- Extension proposal deadline (May 5$^{\text{th}}$ at 23:59)
- Do not hesitate to contact us to discuss things

Notices

- Make sure your git repo is kept up to date
- Oral exams: June 16$^{\text{th}}$, 17$^{\text{th}}$ and 19$^{\text{th}}$
- Pick a timeslot by visiting my office door (with a pen, [M1 01.07])
- Extension proposal deadline (May 5$^{\text{th}}$ at 23:59)
- Do not hesitate to contact us to discuss things
- Presentations next week*:

Notices

- Make sure your git repo is kept up to date
- Oral exams: June 16$^{\text{th}}$, 17$^{\text{th}}$ and 19$^{\text{th}}$
- Pick a timeslot by visiting my office door (with a pen, [M1 01.07])
- Extension proposal deadline (May 5$^{\text{th}}$ at 23:59)
- Do not hesitate to contact us to discuss things
- Presentations next week*:

Notices

- Make sure your git repo is kept up to date
- Oral exams: June 16$^{\text{th}}$, 17$^{\text{th}}$ and 19$^{\text{th}}$
- Pick a timeslot by visiting my office door (with a pen, [M1 01.07])
- Extension proposal deadline (May 5$^{\text{th}}$ at 23:59)
- Do not hesitate to contact us to discuss things
- Presentations next week*: May 8$^{\text{th}}$

Extensions Ideas

Types

Types

Higher order functions

Functions as arguments (later)

Types

Higher order functions

Functions as arguments (later)

Local functions

Nested functions

- Lambda lifting (later)
- Display (array of frame pointers)
- Static link (fp as argument)

Types

Higher order functions

Functions as arguments (later)

Local functions

Nested functions

- Lambda lifting (later)
- Display (array of frame pointers)
- Static link (fp as argument)

Proper overloading

Class dictionaries

Pattern matching & User defined types

Pattern matching & User defined types

```
type IntList = INil | ICons Int IntList;
type List a  = Nil | Cons a (List a);
type Person  = {name :: [Char], age :: Int};
```

Pattern matching & User defined types

```
type IntList = INil | ICons Int IntList;
type List a  = Nil | Cons a (List a);
type Person  = {name :: [Char], age :: Int};
```

```
print (p : Person) : Void {
  print(p.name);
  print('(');
  print(p.age);
  print(')');
}
```

Pattern matching & User defined types

```
type IntList = INil | ICons Int IntList;
type List a  = Nil | Cons a (List a);
type Person  = {name :: [Char], age :: Int};
```

```
tl (l : List a) : List a {
  case l of
    Nil = return Nil
    Cons a t = return t
}
```

```
print (p : Person) : Void {
  print(p.name);
  print('(');
  print(p.age);
  print(')');
}
```

Pattern matching & User defined types

```
type IntList = INil | ICons Int IntList;
type List a  = Nil | Cons a (List a);
type Person  = {name :: [Char], age :: Int};
```

```
print (p : Person) : Void {
  print(p.name);
  print('(');
  print(p.age);
  print(')');
}
```

```
tl (l : List a) : List a {
  case l of
    Nil = return Nil
    Cons a t = return t
}

hd (l : List a) : a {
  case l of
    Cons a t = return a
    /* what happens here? */
}
```

Memory related

Garbage collection

Free space on the heap

Memory related

Garbage collection

Free space on the heap

Smart pointers

Less general solution

Optimisations

Reorder basic blocks

Last lecture

Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.

Not necessary in LLVM.

Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.

Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM

Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.

Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.

Easy for direct recursion, tricky for general TCE.

Not necessary in LLVM

```
countdown (x : Int) : Int {
  if (x == 0) { return 0; }
  else { return countdown (x-1); }
}
```

Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM

```
factorial (x : Int) : Int {
  if (x == 0) { return 1; }
  else { return x * factorial (x-1);
    }
}
```

Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM

```
factorial‘ (x : Int, acc : Int) : Int
    {
  if (x == 0) { return acc; }
  else { return factorial‘(x-1, x*acc
   ); }
}
```

```
factorial‘ (4, 1);
```

Optimisations

Reorder basic blocks

Last lecture

Register allocation

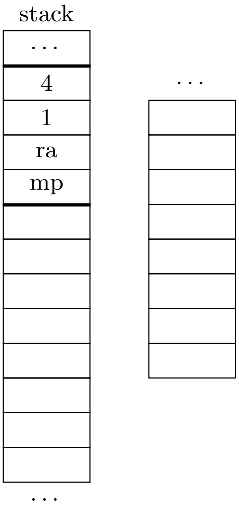
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

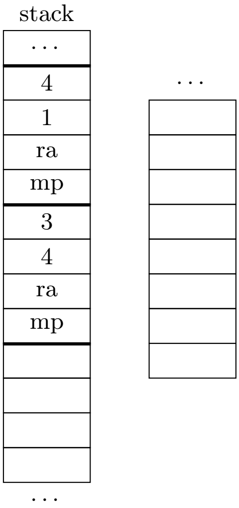
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.

Not necessary in LLVM.

Deforestation

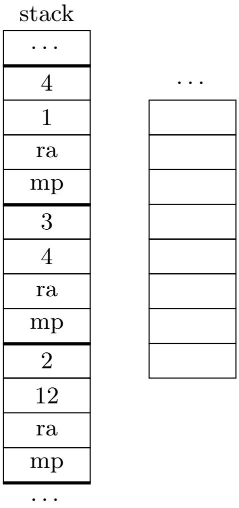
Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.

Easy for direct recursion, tricky for general TCE.

Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

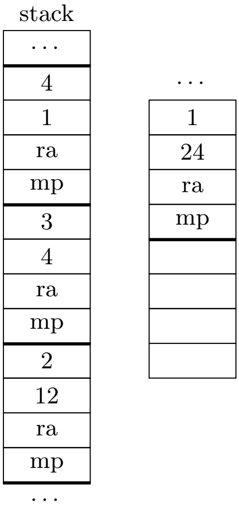
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

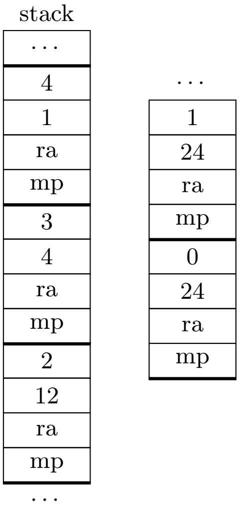
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

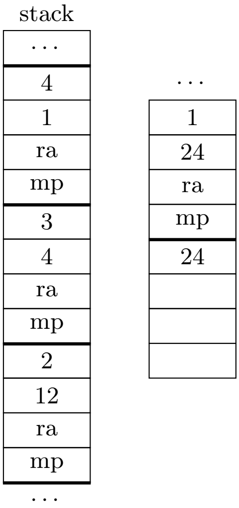
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.

Not necessary in LLVM.

Deforestation

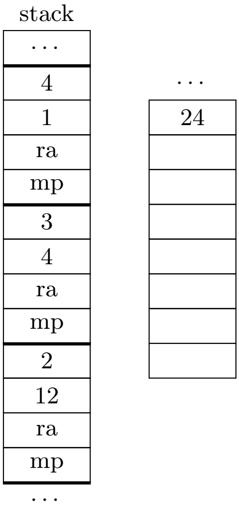
Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.

Easy for direct recursion, tricky for general TCE.

Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

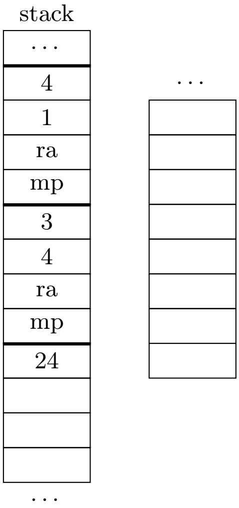
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

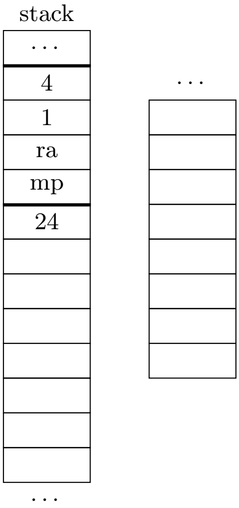
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

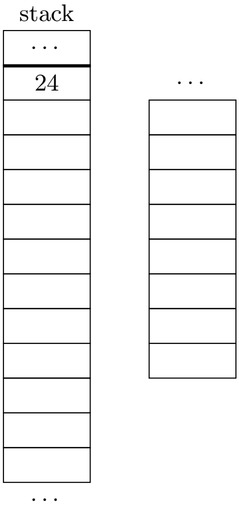
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

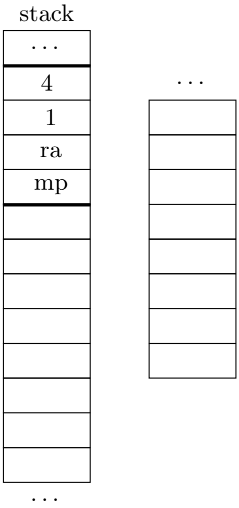
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

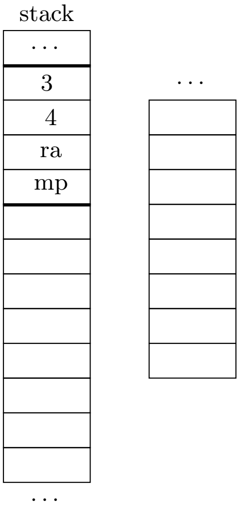
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.

Not necessary in LLVM.

Deforestation

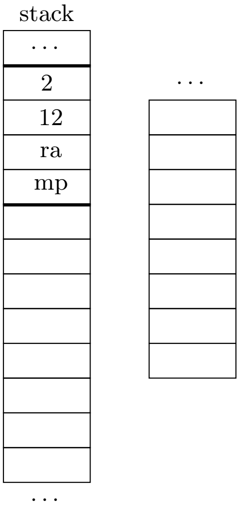
Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.

Easy for direct recursion, tricky for general TCE.

Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

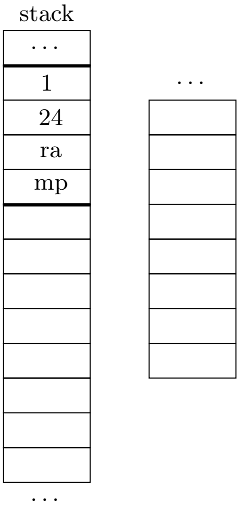
Not very interesting on SSM.
Not necessary in LLVM.

Deforestation

Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.
Easy for direct recursion, tricky for general TCE.
Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.

Not necessary in LLVM.

Deforestation

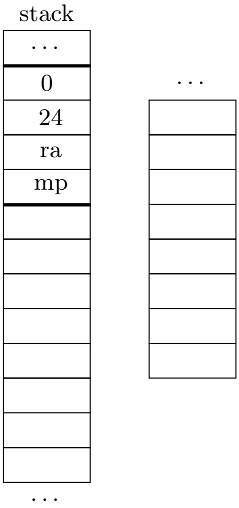
Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.

Easy for direct recursion, tricky for general TCE.

Not necessary in LLVM



Optimisations

Reorder basic blocks

Last lecture

Register allocation

Not very interesting on SSM.

Not necessary in LLVM.

Deforestation

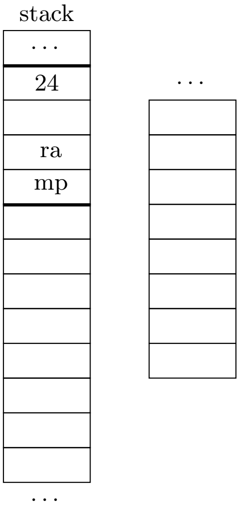
Optimize intermediate lists

Tail call elimination

Reuse stack space for tail calls.

Easy for direct recursion, tricky for general TCE.

Not necessary in LLVM



Code generator related

Separate compilation

Linker

Code generator related

Separate compilation

Linker

- Compile separate files
- Linker connects them, resolves addresses
- Header files?
- Only parse function signatures?

Code generator related

Separate compilation

Linker

- Compile separate files
- Linker connects them, resolves addresses
- Header files?
- Only parse function signatures?

Real target machine

Later

Meta extensions

Language Server Protocol

Plugin for an editor, refactor, etc.

Meta extensions

Language Server Protocol

Plugin for an editor, refactor, etc.

Formal semantics

Create and verify?

Higher order functions

Higher order functions

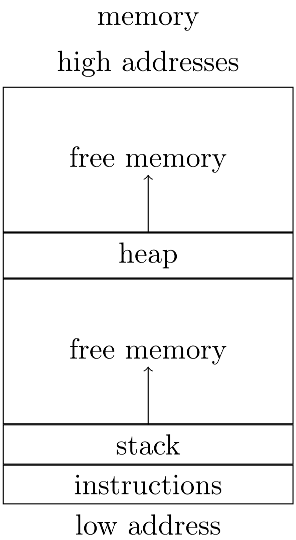
```
map (f : a -> b, l : [a]) : [b] {
  if (isEmpty(l)) {
    return [];
  } else {
    return f(l.hd) : map(f, l.tl);
  }
}

twice (f : a -> a, a : a) : a {
  return f(f(a));
}
```

Higher order functions

```
map (f : a -> b, l : [a]) : [b] {
  if (isEmpty(l)) {
    return [];
  } else {
    return f(l.hd) : map(f, l.tl);
  }
}

twice (f : a -> a, a : a) : a {
  return f(f(a));
}
```



How to store this?

- Function objects
- Thunks
- Closures

How to store this?

- Function objects

C

```
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));

int cmp (const void *l, const void *r) {
  return *(int *)l < *(int *)r ? -1 : 1;
}

int arr[] = {3, 1, 4, 1, 5};
qsort (arr, sizeof(arr)/sizeof(int), sizeof(int),
     &cmp);
```

How to store this?

- Function objects

SPL

```
sort(l : [a], cmp : a -> a -> Int) : [a] { ...  }
cmp (l : Int, r : Int) :: Int Int -> Int) {
  if (l < r) { return -1;
  } else     { return 1; }
}
main () {
  print(sort(3 : 1 : 4 : 1 : 5 : [], cmp));
}
```

How to store this?

- Function objects
- Function pointer in C

SPL

```
sort(l : [a], cmp : a -> a -> Int) : [a] { ...  }
cmp (l : Int, r : Int) :: Int Int -> Int) {
  if (l < r) { return -1;
  } else     { return 1; }
}
main () {
  print(sort(3 : 1 : 4 : 1 : 5 : [], cmp));
}
```

How to store this?

- Function objects
- Function pointer in C
- Function as argument

SPL

```
sort(l : [a], cmp : a -> a -> Int) : [a] { ...  }
cmp (l : Int, r : Int) :: Int Int -> Int) {
  if (l < r) { return -1;
  } else     { return 1; }
}
main () {
  print(sort(3 : 1 : 4 : 1 : 5 : [], cmp));
}
```

How to store this?

- Function objects
- Function pointer in C
- Function as argument
- Straightforward in SSM

SPL

```
sort(l : [a], cmp : a -> a -> Int) : [a] { ...  }
cmp (l : Int, r : Int) :: Int Int -> Int) {
  if (l < r) { return -1;
  } else     { return 1; }
}
main () {
  print(sort(3 : 1 : 4 : 1 : 5 : [], cmp));
}
```

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?

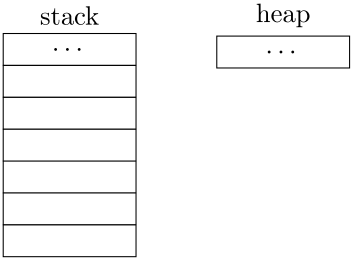
Haskell

```
ap :: (a -> b) -> a -> b
ap f a = f a

main = ap (ap (+) 4) 38
```

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?

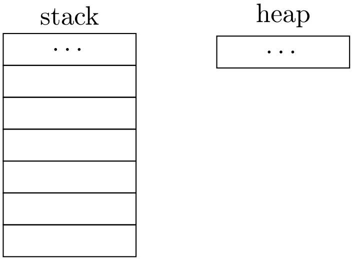


SPL

```
ap (f : a -> b, a : a) : b {
  return f(a);
}
main () {
  print(ap(ap(+, 4), 38));
}
```

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?



SPL

```
ap (f : a -> b, a : a) : b {
  return f(a);
}
main () {
  print(ap(ap(+, 4), 38));
}
```

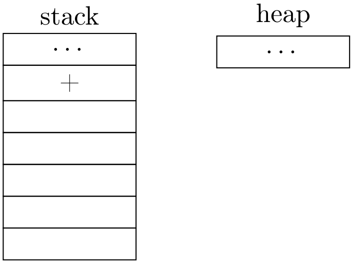
jsr versus bsr

```
SP_post = SP_pre
PC_post = M_pre[
    SP_pre]
M_post[SP_post] =
    PC_pre + 1
```

```
SP_post = SP_pre + 1
M_post[SP_post] = PC_pre + 2
PC_post = PC_pre +
    M_pre[PC_pre + 1] + 2
```

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?



SPL

```
ap (f : a -> b, a : a) : b {
  return f(a);
}
main () {
  print(ap(ap(+, 4), 38));
}
```

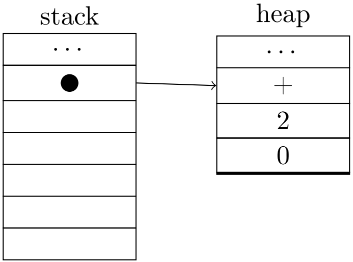
jsr versus bsr

```
SP_post = SP_pre
PC_post = M_pre[
    SP_pre]
M_post[SP_post] =
    PC_pre + 1
```

```
SP_post = SP_pre + 1
M_post[SP_post] = PC_pre + 2
PC_post = PC_pre +
    M_pre[PC_pre + 1] + 2
```

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?



SPL

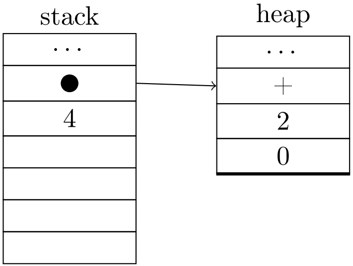
```
ap (f : a -> b, a : a) : b {
  return f(a);
}
main () {
  print(ap(ap(+, 4), 38));
}
```

jsr versus bsr

```
SP_post = SP_pre              SP_post = SP_pre + 1
PC_post = M_pre[              M_post[SP_post] = PC_pre + 2
    SP_pre]                   PC_post = PC_pre +
M_post[SP_post] =                 M_pre[PC_pre + 1] + 2
    PC_pre + 1
```

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?



SPL

```
ap (f : a -> b, a : a) : b {
  return f(a);
}
main () {
  print(ap(ap(+, 4), 38));
}
```

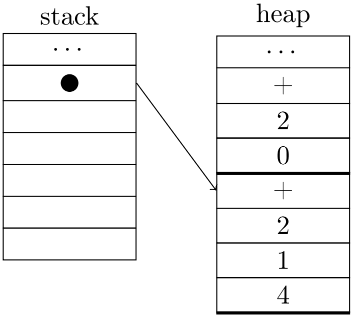
jsr versus bsr

```
SP_post = SP_pre
PC_post = M_pre[
    SP_pre]
M_post[SP_post] =
    PC_pre + 1
```

```
SP_post = SP_pre + 1
M_post[SP_post] = PC_pre + 2
PC_post = PC_pre +
    M_pre[PC_pre + 1] + 2
```

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?



SPL

```
ap (f : a -> b, a : a) : b {
  return f(a);
}
main () {
  print(ap(ap(+, 4), 38));
}
```

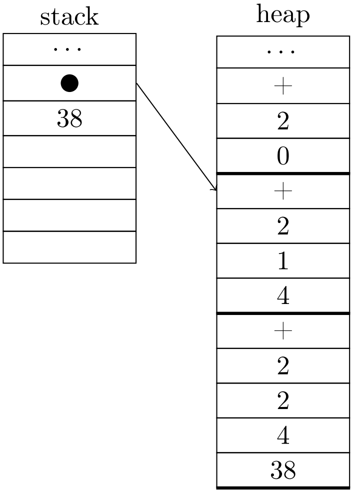
jsr versus bsr

```
SP_post = SP_pre
PC_post = M_pre[
    SP_pre]
M_post[SP_post] =
    PC_pre + 1
```

```
SP_post = SP_pre + 1
M_post[SP_post] = PC_pre + 2
PC_post = PC_pre +
    M_pre[PC_pre + 1] + 2
```

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?



SPL

```
ap (f : a -> b, a : a) : b {
  return f(a);
}
main () {
  print(ap(ap(+, 4), 38));
}
```

jsr versus bsr

```
SP_post = SP_pre
PC_post = M_pre[
    SP_pre]
M_post[SP_post] =
    PC_pre + 1
```

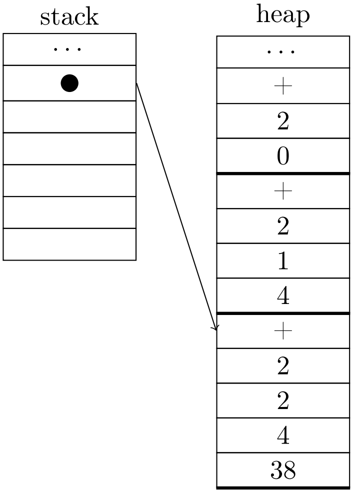
```
SP_post = SP_pre + 1
M_post[SP_post] = PC_pre + 2
PC_post = PC_pre +
    M_pre[PC_pre + 1] + 2
```

Thunks

- Thunks
- Curried arguments
- Partially applied function
- Remember **bsr** versus **jsr**?

SPL

```
ap (f : a -> b, a : a) : b {
  return f(a);
}
main () {
  print(ap(ap(+, 4), 38));
}
```



jsr versus bsr

```
SP_post = SP_pre
PC_post = M_pre[
    SP_pre]
M_post[SP_post] =
    PC_pre + 1
```

```
SP_post = SP_pre + 1
M_post[SP_post] = PC_pre + 2
PC_post = PC_pre +
    M_pre[PC_pre + 1] + 2
```

Closures

- Closure
- Environment
- Anonymous function
- Only thunks needed when lambda lifting

What is lambda lifting

Closures

- Closure
- Environment
- Anonymous function
- Only thunks needed when lambda lifting

What is lambda lifting

Closures

- Closure
- Environment
- Anonymous function
- Only thunks needed when lambda lifting

What is lambda lifting

```
plus38(x : Int) : Int -> Int {
  ret (y : Int) : Int {
    return y + x;
  }
  return ret;
}
```

Closures

- Closure
- Environment
- Anonymous function
- Only thunks needed when lambda lifting

What is lambda lifting

```
plus38(x : Int) : Int -> Int {
  ret (y : Int) : Int {
    return y + x;
  }
  return ret;
}
```

```
plus38_ret(x : Int, y : Int) : Int{
  return y + x;
}
plus38(x : Int) : Int -> Int {
  return plus38_ret(x);
}
```

Garbage collection

Storing stuff in the heap

Semantics: **sth**

Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
0	1	1	`0xd6`

Storing stuff in the heap

Semantics: **sth**

Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
0	1	1	`0xd6`

Description

Store into Heap. Pops 1 value from the stack and stores it into the heap. Pushes the heap address of that value on the stack.

Storing stuff in the heap

Semantics: **sth**

Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
0	1	1	`0xd6`

Description

Store into Heap. Pops 1 value from the stack and stores it into the heap. Pushes the heap address of that value on the stack.

Example

```
ldc 5
sth
```

Loading stuff from the heap

Semantics: **ldh/lda**

Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
1	1	1	`0x7c`

Loading stuff from the heap

Semantics: **ldh/lda**

Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
1	1	1	`0x7c`

Description

Load via Address. Dereferencing. Pushes the value pointed to by the value at the top of the stack. The pointer value is offset by a constant offset.

Loading stuff from the heap

Semantics: **ldh/lda**

Nr of inline Opnds	Nr of stack Opnds	Nr of stack Results	Instr code (hex)
1	1	1	`0x7c`

Description

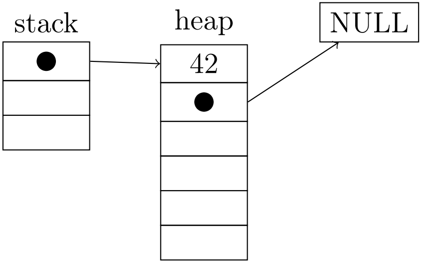
Load via Address. Dereferencing. Pushes the value pointed to by the value at the top of the stack. The pointer value is offset by a constant offset.

Example

```
ldc 5
sth
lda 0
```

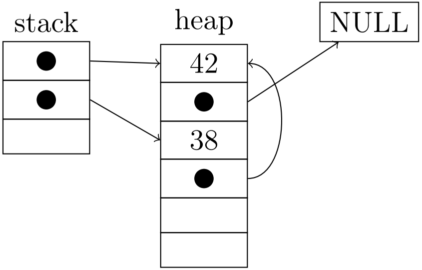
Example

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;
  return z
}
```



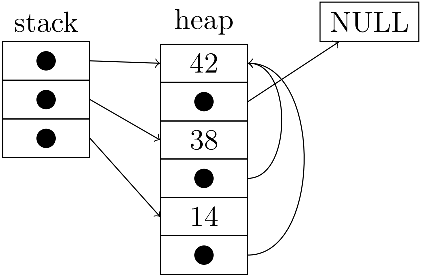
Example

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;
  return z
}
```



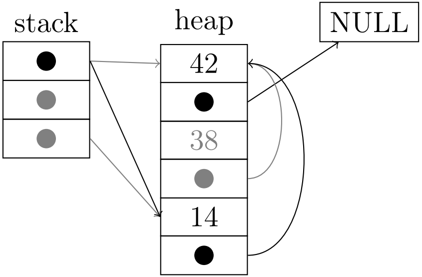
Example

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;
  return z
}
```



Example

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;
  return z
}
```



Garbage collection

Burden the programmer

`malloc`, `free`, **new**, **delete**

Garbage collection

Burden the programmer

`malloc`, `free`, **new**, **delete**

Tracing garbage collection

- GC on alloc
- Trace all pointers on the stack

Garbage collection

Burden the programmer

`malloc`, `free`, **new**, **delete**

Tracing garbage collection

- GC on alloc
- Trace all pointers on the stack

Reference counting

- GC on free
- Store reference count in meta
- On free, decrement
- On share, increment

Garbage collection

Burden the programmer

`malloc`, `free`, **new**, **delete**

Tracing garbage collection

- GC on alloc
- Trace all pointers on the stack

Reference counting

- GC on free
- Store reference count in meta
- On free, decrement
- On share, increment

What to do with free space

- Compacting

Garbage collection

Burden the programmer

`malloc`, `free`, **new**, **delete**

Tracing garbage collection

- GC on alloc
- Trace all pointers on the stack

Reference counting

- GC on free
- Store reference count in meta
- On free, decrement
- On share, increment

What to do with free space

- Compacting
  - Copying

Garbage collection

Burden the programmer

`malloc`, `free`, **new**, **delete**

Tracing garbage collection

- GC on alloc
- Trace all pointers on the stack

Reference counting

- GC on free
- Store reference count in meta
- On free, decrement
- On share, increment

What to do with free space

- Compacting
  - Copying
  - Compressing

Garbage collection

Burden the programmer

`malloc`, `free`, **new**, **delete**

Tracing garbage collection

- GC on alloc
- Trace all pointers on the stack

Reference counting

- GC on free
- Store reference count in meta
- On free, decrement
- On share, increment

What to do with free space

- Compacting
  - Copying
  - Compressing
- Free list

Tracing Garbage Collection

Intuition

- At some point we collect

Tracing Garbage Collection

Intuition

- At some point we collect
- Trace every pointer on the stack and mark

Tracing Garbage Collection

Intuition

- At some point we collect
- Trace every pointer on the stack and mark
- Traverse the heap and remove all non-marked data

Tracing Garbage Collection

Intuition

- At some point we collect
- Trace every pointer on the stack and mark
- Traverse the heap and remove all non-marked data
- Combined in some variants

Tracing Garbage Collection

Intuition

- At some point we collect
- Trace every pointer on the stack and mark
- Traverse the heap and remove all non-marked data
- Combined in some variants
- Suitable for all data

Tracing Garbage Collection

Intuition

- At some point we collect
- Trace every pointer on the stack and mark
- Traverse the heap and remove all non-marked data
- Combined in some variants
- Suitable for all data
- How to distinguish pointers and values?

Tracing Garbage Collection

Intuition

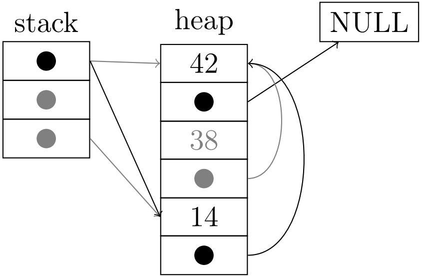
- At some point we collect
- Trace every pointer on the stack and mark
- Traverse the heap and remove all non-marked data
- Combined in some variants
- Suitable for all data
- How to distinguish pointers and values?

Tracing Garbage Collection

Intuition

- At some point we collect
- Trace every pointer on the stack and mark
- Traverse the heap and remove all non-marked data
- Combined in some variants
- Suitable for all data
- How to distinguish pointers and values?

Example



Reference Counting

Collect garbage on free

Intuition

- Administrate counts for all data

Reference Counting

Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment

Reference Counting

Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove

Reference Counting

Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

Reference Counting

Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

Reference Counting

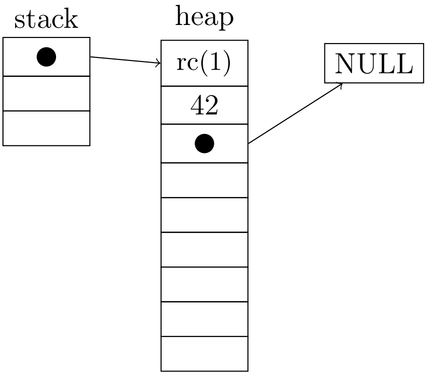
Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

```
f() : [Int] {
  var x = [42];     <-
  var y = 38 : x;
  var z = 14 : x;
  return z;
}
main () {
  print(f());
}
```

Example



Reference Counting

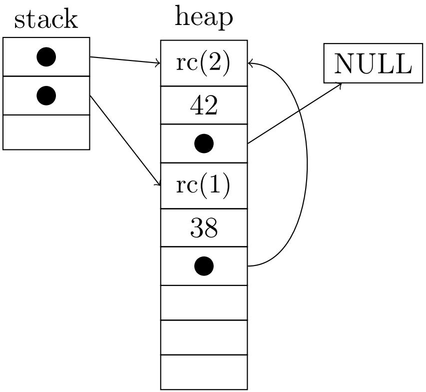
Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;  <-
  var z = 14 : x;
  return z;
}
main () {
  print(f());
}
```

Example



Reference Counting

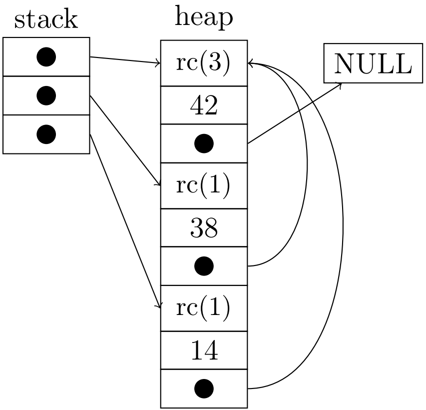
Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;   <-
  return z;
}
main () {
  print(f());
}
```

Example



Reference Counting

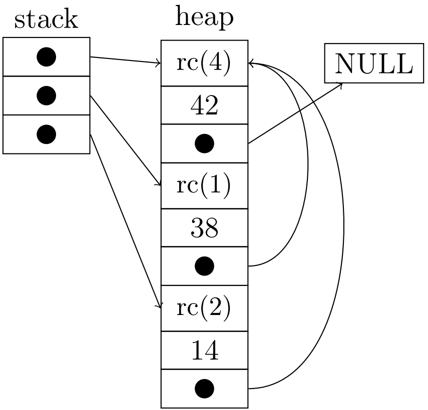
Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;
  return z;          <- share z
}
main () {
  print(f());
}
```

Example



Reference Counting

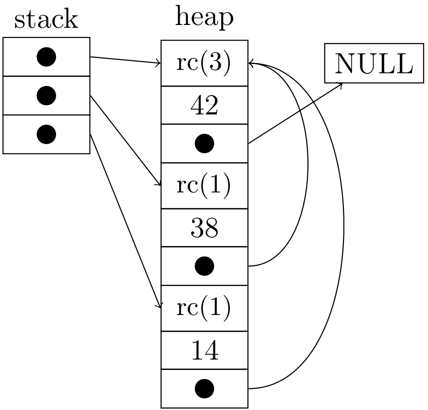
Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;
  return z;          <- decrement (x)
}
main () {
  print(f());
}
```

Example



Reference Counting

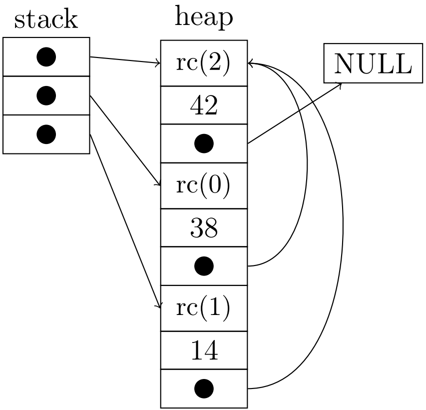
Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;
  return z;          <- decrement (y)
}
main () {
  print(f());
}
```

Example



Reference Counting

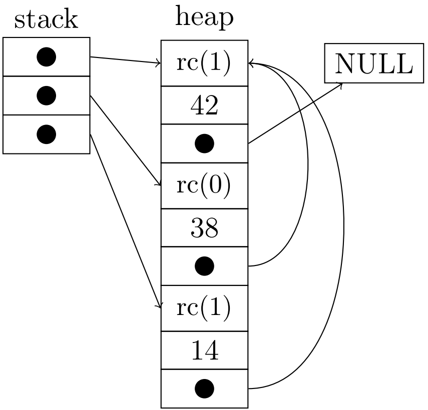
Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;
  return z;
}
main () {
  print(f());
}
```

Example



Reference Counting

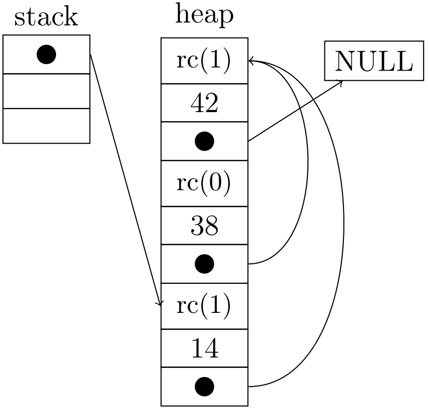
Collect garbage on free

Intuition

- Administrate counts for all data
- On free, decrement, on share increment
- If count becomes zero, remove
- Cannot handle cycles

```
f() : [Int] {
  var x = [42];
  var y = 38 : x;
  var z = 14 : x;
  return z;
}
main () {
  print(f());      <-
}
```

Example



Garbage collection in SPL

- Remember the start of the heap

Garbage collection in SPL

- Remember the start of the heap
- Choose Tracing or Reference counting

Garbage collection in SPL

- Remember the start of the heap
- Choose Tracing or **Reference counting**

Garbage collection in SPL

- Remember the start of the heap
- Choose Tracing or **Reference counting**
- Think of the metadata, what do you need

Garbage collection in SPL

- Remember the start of the heap
- Choose Tracing or **Reference counting**
- Think of the metadata, what do you need
- Think about when to increment/decrement the reference counter

Garbage collection in SPL

- Remember the start of the heap
- Choose Tracing or **Reference counting**
- Think of the metadata, what do you need
- Think about when to increment/decrement the reference counter
- Copying is more difficult but possible

Garbage collection in SPL

- Remember the start of the heap
- Choose Tracing or **Reference counting**
- Think of the metadata, what do you need
- Think about when to increment/decrement the reference counter
- Copying is more difficult but possible
- Compressing is easy since all objects are of the same size

Real architectures

Instruction sets

Instruction sets

Name	Word	Registers	Endianness
x86	32	8	Little
x86_64	64	16	Little
ARM/A32	32	15	Big*
ARM/A64	64	32	Big*
AVR	8	32	Little
RISC-V	8	32	Little

Instruction sets

Instruction sets

Name	Word	Registers	Endianness
x86	32	8	Little
x86_64	64	16	Little
ARM/A32	32	15	Big*
ARM/A64	64	32	Big*
AVR	8	32	Little
RISC-V	8	32	Little

Intermediate languages

- LLVM (SSA)

Instruction sets

Instruction sets

Name	Word	Registers	Endianness
x86	32	8	Little
x86_64	64	16	Little
ARM/A32	32	15	Big*
ARM/A64	64	32	Big*
AVR	8	32	Little
RISC-V	8	32	Little

Intermediate languages

- LLVM (SSA)
- Java bytecode

Instruction sets

Instruction sets

Name	Word	Registers	Endianness
x86	32	8	Little
x86_64	64	16	Little
ARM/A32	32	15	Big*
ARM/A64	64	32	Big*
AVR	8	32	Little
RISC-V	8	32	Little

Intermediate languages

- LLVM (SSA)
- Java bytecode
- ABC (slightly masochistic)

Instruction sets

Instruction sets

Name	Word	Registers	Endianness
x86	32	8	Little
x86_64	64	16	Little
ARM/A32	32	15	Big*
ARM/A64	64	32	Big*
AVR	8	32	Little
RISC-V	8	32	Little

Intermediate languages

- LLVM (SSA)
- Java bytecode
- ABC (slightly masochistic)
- C (slightly cheating)

Conclusion

Your project

Send in your extension proposal

Your project

Send in your extension proposal
Do not hesitate to discuss ideas