PURDUE
UNIVERSITY®

Functions - Arrays

Compilers: Principles And Practice

Tiark Rompf

Where Were We?

What did we learn in the last class?

Let's Add Functions - Type Checking

Yay, inference rules!

Let's Add Functions - Type Checking

Yay, inference rules!

```
              Env,a1:T1,...,an:Tn |- fb: T
------------------------------------------------------[FunDef]
  Env |- FunDef(f, a1...an, T, fb) : (T1...Tn) => T



  Env,f1:FT1,...,fn:FTn |- f1: FT1
               ...
  Env,f1:FT1,...,fn:FTn |- fn: FTn
   Env,f1:FT1,...,fn:FTn |- b : T
------------------------------------[LetRec]
    Env |- LetRec(f1...fn, b): T
```

Let's Add Functions - Type Checking

```
     Env |- f: (T1,...,Tn) => T
     Env |- a1: T1
     ...
     Env |- an: Tn
-------------------------------------[App]
  Env |- App(f, List(a1,...,an)): T
```

Interpreter

What does a function call mean?

Interpreter

What does a function call mean?

```
                  // LetRec(List(
def f(x: Int) = { //   FunDef("f", List(Arg("x", IntType)), ???,
  x + 1           //     Prim("+", Ref("x"), Lit(1))
};                //   )),
f(1)              //   App(Ref("f"), List(Lit(1)))
                  // )
```

What does a function call mean?

```
                      // LetRec(List(
def f(x: Int) = { //   FunDef("f", List(Arg("x", IntType)), ???,
  x + 1               //     Prim("+", Ref("x"), Lit(1))
};                    //   )),
f(1)                  //   App(Ref("f"), List(Lit(1)))
                      // )
```

```
def f(x: Int) = { g(x) + 1 };
def g(x: Int): Int = 2 * x;
f(1 + 1)
```

Interpreter

What does a function call mean?

```
                   // LetRec(List(
def f(x: Int) = { //   FunDef("f", List(Arg("x", IntType)), ???,
  x + 1            //     Prim("+", Ref("x"), Lit(1))
};                 //   )),
f(1)               //   App(Ref("f"), List(Lit(1)))
                   // )
```

```
def f(x: Int) = { g(x) + 1 };
def g(x: Int): Int = 2 * x;
f(1 + 1)
```

```
def f(x: Int): Int = { g(x) + 1 };
def g(x: Int): Int = if (x == 0) 0 else f(x-1);
f(1)
```

Interpreter

```
abstract class Val
case class Cst(x: Any) extends Val
case class Func(args: List[String], fbody: Exp, env: Env) extends Val

def eval(exp: Exp)(env: Env): Val = exp match {
  case LetRec(funs, body) =>
    val funcs = funs map { case f@FunDef(n, _, _, _) => (n, eval(f)(env)) }
    funcs foreach { case (_, f@Func(_, _, _)) => f.withVals(funcs) }
    eval(body)(env.withVals(funcs))
  case FunDef(name, args, rte, fbody) =>
    Func(args map { arg => arg.name }, fbody, env)
  case App(f, args) =>
    val eargs = args map { arg => eval(arg)(env) }
    val Func(fargs, fbody, fenv) = eval(f)(env)
    eval(fbody)(fenv.withVals(fargs zip eargs))
}
```

Compilation - Calling Conventions

One of the main concepts we have been using so far is **convention**.

Our compiler only generates code that uses registers ‘sp’ and above and which puts the result in ‘sp’.

Thus, we can keep intermediate results and know which memory locations are available at any given point.

Compilation - Calling Conventions

How should we call a function from any point in the program without losing data?

Example:

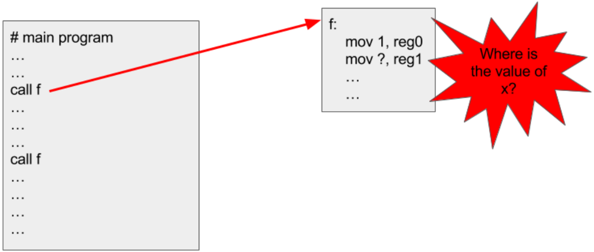
```
def f(x: Int) = 1 + x;

val y = f(1);
val z = f(2);
y + z
```

Compilation - Calling Conventions

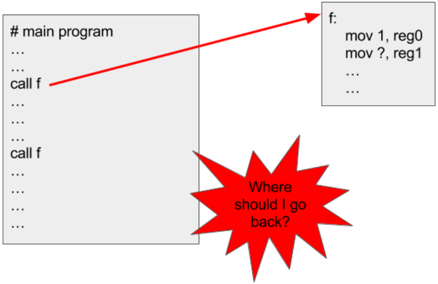
```
# main program
...
...
call f
...
...
...
call f
...
...
...
...
```

```
f:
    ...
    ...
    ...
    ...
```

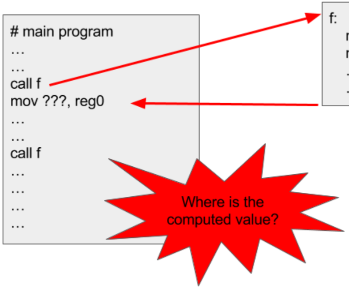
Compilation - Calling Conventions



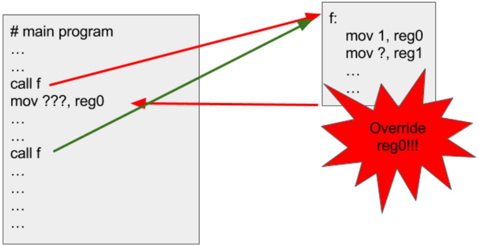
Compilation - Calling Conventions



Compilation - Calling Conventions



Compilation - Calling Conventions



Compilation - Calling Convention

- Argument passing: %rdi, %rsi, %rdx, %rcx, %r8, %r9

Compilation - Calling Convention

- Argument passing: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Return: use **call** and **ret**

Compilation - Calling Convention

- Argument passing: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Return: use **call** and **ret**
  - **call** pushes the instruction pointer on the stack, and jumps to the label

Compilation - Calling Convention

- Argument passing: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Return: use **call** and **ret**
  - **call** pushes the instruction pointer on the stack, and jumps to the label
  - **ret** pops the return address from the stack and jumps to it

Compilation - Calling Convention

- Argument passing: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Return: use **call** and **ret**
  - **call** pushes the instruction pointer on the stack, and jumps to the label
  - **ret** pops the return address from the stack and jumps to it
  - Corollary: we need to reset the stack before calling ret

Compilation - Calling Convention

- Argument passing: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Return: use **call** and **ret**
  - **call** pushes the instruction pointer on the stack, and jumps to the label
  - **ret** pops the return address from the stack and jumps to it
  - Corollary: we need to reset the stack before calling ret
- Return value will be saved in %rax

Compilation - Calling Convention

- Argument passing: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Return: use **call** and **ret**
  - **call** pushes the instruction pointer on the stack, and jumps to the label
  - **ret** pops the return address from the stack and jumps to it
  - Corollary: we need to reset the stack before calling ret
- Return value will be saved in %rax
- Before calling a function, all intermediate values will be saved on the stack

Compilation - Calling Convention

```
# main program
push %rbp
mov %rsp, %rbp
mov $1, %rdi
call f
mov %rax, %rdi
mov $2, %rsi
push %rdi
mov %rsi, %rdi
call f
pop %rdi
mov %rax, %rsi
mov %rdi, %rdx
mov %rsi, %rcx
add %rcx, %rdx
mov %rdx, %rsi
mov %rsi, %rdi
mov %rdi, %rax
mov %rbp, %rsp
pop %rbp
ret
```

```
f:
    push %rbp
    mov %rsp, %rbp
    mov $1, %rsi
    mov %rdi, %rdx
    add %rdx, %rsi
    mov %rsi, %rax
    mov %rbp, %rsp
    pop %rbp
    ret
```

Let’s Add Arrays

We are going to use Scala syntax, but we are not (yet) going to handle objects.

The array will behave more like a C array; the length will need to be remembered.

```
val arr = new Array[Int](4 + 5);
```

Let's Add Arrays - Syntax

Let's Add Arrays - Syntax

```
<type>   ::= <ident> | <type> '=>' <type>  // '=>' is right associative
           | '('[<type>[','<type>]*]')' '=>' <type>
<atom>   ::= <number> | <bool> | '()'
           | '('<simp>')'
           | <ident>
<tight>  ::= <atom>['('[<simp>[','<simp>]*]')']*['('<simp>')''=' <simp>]
           | '{'<exp>'}'
<uatom>  ::= [<op>]<tight> // Previously atom
<simp>   ::= ... // same as before
           | 'new' 'Array' '[' <type> ']' '('<simp>')'
<exp>    ::= ... // same as before
<arg>    ::= <ident>':'<type>
<prog>   ::=
   ['def'<ident>'('[<arg>[','<arg>]*]')'[':' <type>] '=' <simp>';']*
           <exp>
```

Let's Add Arrays - Syntax

Scala array read syntax:

```
arr(1)
```

Wait a minute! Is this a function application?

Let's Add Arrays - Syntax

Scala array read syntax:

```
arr(1)
```

Wait a minute! Is this a function application?

Array write syntax:

```
arr(1) = 5
```

Here, we can notice one major difference which lets us know this is an array update. . . “‘

Let's Add Arrays - Syntax

Scala array read syntax:

```
arr(1)
```

Wait a minute! Is this a function application?

Array write syntax:

```
arr(1) = 5
```

Here, we can notice one major difference which lets us know this is an array update. . . "'

The operations on arrays are primitive operations:

- "block-get"
- "block-set"

Let’s Add Arrays - AST

```
case class Prim(op: String, args: List[Exp]) extends Exp
```

Let's Add Arrays - AST

```scala
case class Prim(op: String, args: List[Exp]) extends Exp

case class ArrayDec(size: Exp, tp: Type) extends Exp
```

Let's Add Arrays - Semantic Analysis

```
case class ArrayType(tp: Type) extends Type
```

How do we know if an array is well-formed?

Let's Add Arrays - Semantic Analysis

```scala
case class ArrayType(tp: Type) extends Type
```

How do we know if an array is well-formed?

If `tp` is well-formed!

Let's Add Arrays - Semantic Analysis

An array type 'tp' conforms to type 'pt' if:

- 'pt' is an array type, and
- inner type 'tp' conforms to 'pt' inner type.

Let’s Add Arrays – Semantic Analysis

An array type ‘tp’ conforms to a type ‘pt’ if all of the following hold:

- ‘pt’ is a function type with one argument
- the function argument’s type conforms to IntType
- the inner type of ‘tp’ conforms to the return type of ‘pt’

Let's Add Arrays – Semantic Analysis

An array type 'tp' conforms to a type 'pt' if all of the following hold:

- 'pt' is a function type with one argument
- the function argument's type conforms to IntType
- the inner type of 'tp' conforms to the return type of 'pt'

In other words, Array[T] conforms to Int => T!

Let's Add Arrays - Semantic Analysis

Everyone's favorite: inference rules!

Let's Add Arrays - Semantic Analysis

Everyone's favorite: inference rules!

```
              Env |- size: Int
----------------------------------------[ArrayDec]
   Env |- ArrayDec(size, T): Array[T]



   Env |- arr: Array[T]      Env |- i: Int
---------------------------------------------[ArrayGet]
  Env |- Prim("block-get", List(arr, i)): T



 Env |- arr: Array[T]   Env |- i: Int   Env |- v: T
----------------------------------------------------[ArraySet]
  Env |- Prim("block-set", List(arr, i, v)): Unit
```

Let's Add Arrays - Interpreter

```scala
abstract class Val
case class Cst(x: Any) extends Val

def eval(exp: Exp)(env: Env): Val = exp match {
  case ArrayDec(size, _) =>
    val Cst(s: Int) = eval(size)(env) // Why is this safe?
    Cst(new Array[Any](s))
  case Prim("block-get", args) => ??? // left as an exercise for the reader
  case Prim("block-set", args) => ??? // left as an exercise for the reader
}
```

Let's Add Arrays - Compiler

Where do we want to store our arrays?

Let's Add Arrays - Compiler

Where do we want to store our arrays?

We will use the heap. The heap is permanent, i.e., not erased once a function call is over (unlike the stack and local variables).

Therefore the heap is used as persistent storage.

Let's Add Arrays - Compiler

At (compiled) program launch, the OS maps a memory space for our stack. Thus, we can `mov $4 -8(%rsp)`.

To have access to the heap, we call malloc in bootstrap.c and give the pointer to our main function as the first argument

Let's Add Arrays - Compiler

Where is this pointer going to be saved?

Let's Add Arrays - Compiler

Where is this pointer going to be saved?

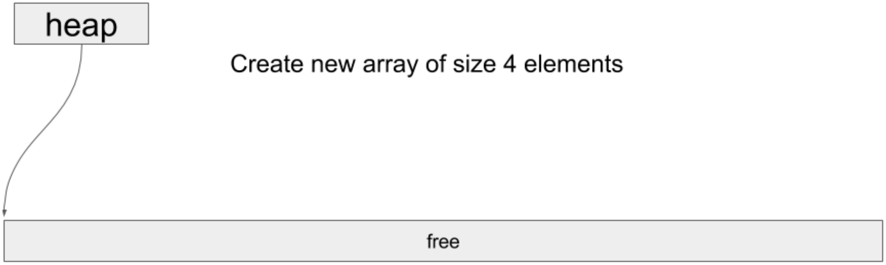
A global variable: heap. This address represents the first memory address that we are allowed to use.

So, how do we create an array (ArrayDec)?

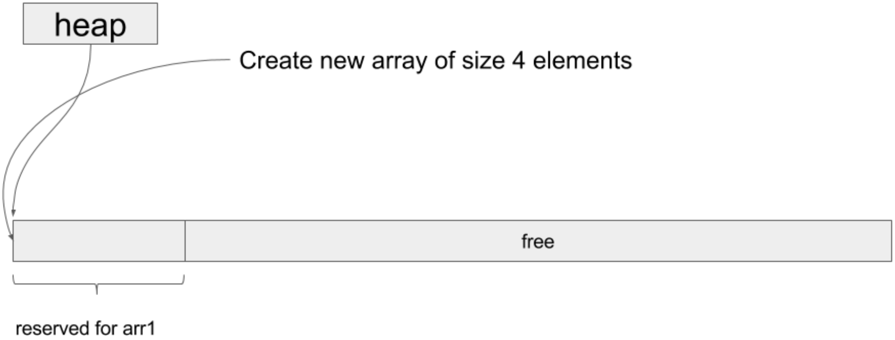
Subsequent array creations must not overlap!

Let’s Add Arrays - Compiler

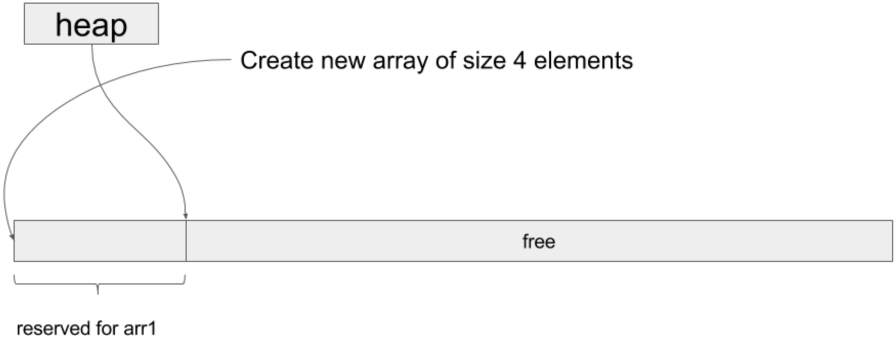
Let's Add Arrays - Compiler



Let's Add Arrays - Compiler



Let’s Add Arrays - Compiler



Let's Add Arrays - Compiler

We assume %rax contains the address of the array we want to access.

How to write to a memory location:

```
movq $1 (%rax)     // write one in the first element of the array
```

How to read from a memory location?

Let's Add Arrays - Compiler

We assume %rax contains the address of the array we want to access.

How to write to a memory location:

```
movq $1 (%rax)      // write one in the first element of the array
```

How to read from a memory location?

```
movq (%rax), %rax      // read the first element of the array
                       // and store it in %rax
```

Let's Add Arrays - Compiler

We assume %rax contains the address of the array we want to access.

How to write to a memory location:

```
movq $1 (%rax)     // write one in the first element of the array
```

How to read from a memory location?

```
movq (%rax), %rax     // read the first element of the array
                      // and store it in %rax
```

Shortcuts for arrays:

```
movq heap, %rax
movq $3, %rcx
movq (%rax, %rcx, 8), %rax  // read the 3rd (stored in %rcx)
                            // element of the array and store
                            // it in %rax
```

Where Are We?

Today, we learned the following:

- Type checking functions
- Intepreting functions
- Calling conventions
- Adding arrays

Where Are We?

Today, we learned the following:

- Type checking functions
- Intepreting functions
- Calling conventions
- Adding arrays

Questions?