Dynamic Memory, Objects, Closures, and More

Kinds of Memory in Compiled Programs

Program Data	Typical Machine Representation
intermediate values	registers, stack
local variables, parameters	registers, stack
return addresses of function calls	stack (+ 1 register)
global variables	data segment, pre-allocated
algebraic data type values	dynamic heap
objects	dynamic heap
closures (first class functions)	dynamic heap

Pre-allocated memory has fixed size at compile time
Stack can grow, but must shrink in the LIFO way

Heap is most general: allocate and deallocate in any order

- if we never de-allocate (as in the project), can use a stack separate from the stack for locals and returns
  $\rightsquigarrow$ out of memory unnecessarily

Memory as Array

Languages like C traditionally give full access to program memory through pointers that can be manipulated (and even write to stack!)
In C, the heap can be implemented as a library with **malloc** and **free**, and that uses operating system calls to obtain large blocks of available memory, then treats them as large arrays of bytes.

```
typedef struct node { // size 8 bytes
   int content; // offset 0
   struct node * next; // offset 4
} node_t;
head = malloc(sizeof(node_t)); // head = 8 bytes on heap
head -> content = 42; // RAM[head] = 42
second = malloc(sizeof(node_t)); // second = get 8bytes
head -> next = second; // RAM[head + 4] = second
```

Malloc and Free Using Free List

Need to know which memory is used and which is fresh.

Because allocation and de-allocation is in any order, memory array has interleaved regions of allocated and free memory.

Approach:

- allocated memory is responsibility of the program
- create a list of free blocks using only free memory!

What is free and unused memory for the application is a linked list data structure for the allocator

- list elements are variable length: size stored in each block
- allocation: find a sufficient block, split it, update the free list, return the split of part
- deallocation inserts the block into list, if possible merge with adjacent blocks

See also:

- Lectures of David August at This Link
- D. Knuth, The Art of Computer Programming, Vol. 1, "Dynamic Storage Allocation"

Lack of Memory Safety

Using pointers is flexible and easy to compile: emit memory access instructions and library calls to malloc and free.

- but it is not memory safe!

```
long* x = malloc(...);
*x = 9876543;
free(x);
// x is now dangling pointer
long* y = malloc(...);
*y = 1234567;
// y might use part of same memory as x
*x = 0;
// now *y may be changed and even corrupted
```

To ensure memory safety: cannot allow developer to use 'free' arbitrarily

- we want automated memory management

Automated Memory Management

Reference counting: maintain a field in each heap object that counts how many references to this object exist.

```
x.f = y
```

becomes:

```
x.f.count--;
if (x.f.count==0) deallocate(x.f)
y.count++;
x.f=y
```

Deallocation also decrements references and can recursively deallocate other objects.
This works as long as there are no cycles.
See: Automatic Reference Counting in Swift

Forms of compile time reference counting in Rust: Ownership, References and Borrowing

Garbage Collection

To automatically collect cyclic data structures and convenient functional programming with sharing data we use garbage collection (already introduced in LISP).
Periodically mark all objects reachable from global and local variables of all stack frames, free up the rest as garbage

Two main types of garbage collection algorithms:

- mark and sweep: mark all reachable objects and put them in a free list (good if there is little garbage, but suffers from fragmentation)
- copying collector: use twice the space, after marking copy all useful data into a separate region and put blocks next to each other

Generational collector: organize objects by generations, collect newly allocated objects more often, if they survive multiple collections, promote them to older generation.
Typically used in Java: generational parallel copying collector

Compiler Support for Garbage Collection

Collector needs to know:

- how to find roots in global variables, stack, registers (or ensure references are never only in registers)
- how to follow (non-weak) references through objects

For this, some amount of run-time type information is needed.
Generational GC may need to traverse all older generations to know what is alive in new generation. To speed this up, GC can use information that ensures that certaing groups of objects do not point to newer generation. To maintain that information, compiler may need to instrument all writes of object fields, with overhead similar to that of reference counting.

Dynamic Dispatch

Dynamic dispatch is key to object-oriented languages (and can be used to implement higher-order functions).

```
class Animal {
  def noise = "squeak!"
  def muchNoise = noise + noise
}
class Dog extends Animal {
  override def noise = "aw!"
}
d = new Dog
d.muchNoise

res0: String = aw!aw!
```

Compilation of muchNoise cannot make a direct call to method that returns "squeak!" but must invoke whatever method is most specific to the dynamic type of the object given by new declaration.

$\rightsquigarrow$ virtual method table

Dynamic Dispatch Implementation

```
type Animal = struct { vtable : FunPtrs[] }

def Animal_noise(this:Animal) = return "squeak!"
def Animal_muchNoise(this:Animal) =
  (this -> vtable)[0](this) +
  (this -> vtable)[0](this)

type Dog = struct { vtable : FunPtrs[] }

def Dog_noise(this:Dog) = return "aw!"

Animal_vtable[] = { Animal_noise, Animal_muchNoise }
Dog_vtable[] = { Dog_noise, Animal_muchNoise }

d = malloc(Dog)
d -> vtable = Dog_vtable
(d -> vtable)[1](d) // 1 is the index of muchNoise
```

Virtual methods calls have one extra indirection

First-Class Functions as Objects: Capturing Vals

```
val f = {
  val x = 42
  ((y:Int) => x + y) // Closure_1
}
f(20)
```

becomes:

```
abstract class Function[A-,B+] {
  def apply(x:A): B
}
class Closure_1(x:Int) extends Function {
  def apply(y: Int): Int = x + y
}
val f = {
   val x = 42
   new Closure_1(x)
}
f.apply(20)
```

Capturing Vars

```
val f = { // Block_2
  var x = 42
  ((y:Int) => x + y; x++) // Closure_2
}
f(20) + f(0)
```

becomes:

```
class Block_2_Vars { var x: Int = _ }
class Closure_2(block: Block_2_Vars) extends Function {
  def apply(y: Int): Int = { block.x + y; block.x++ }
}
val f = {
   val block2 = new Block_2_Vars
   block2.x = 42
   new Closure_2(block2)
}
f.apply(20) + f.apply(0)
```

Lazy Values

Lazy values can avoid/postpone computation:

```
lazy val wikipediaSize = computeSize(wikipedia)
lazy val worldPop = computePopulation(world)
wikipediaSize / 1024
```

Simple implementation (for real one, see CS-302)

```
class Lazy[A](computation: () => A) {
  var cached: A = _
  var defined: Boolean = false
  def force: A = {
    if (!defined) {
      cached = computation(())
      defined= true
    }
    cached
  }
}
val wikipediaSize = Lazy(() => computeSize(wikipedia))
val worldPop = Lazy(() => computePopulation(world))
```

Lazy Values as Default

Call by value breaks substitution principle, even for pure E,

```
{ val x = E; F}
```

may loop in some cases when replacing x

```
{ F[x:=E] }
```

would end up not touching E and thus terminate.

A more declarative approach is to say all val-s and parameters are lazy - approach taken by Haskell. But accessing lazy values is expensive (even after they are evaluated) One solution: **strictness analysis** that determines ahead of time that some parameter is **always** accessed, so it can be passed by value.

- dual to initialization analysis: function parameter is strict if all branches use it
- need to define it for higher-order functions

Compiling Logic Programming

In lazy evaluation we do not know if we will use a value, but in logic programming languages such as Prolog we do not even know which values are inputs and which ones are outputs.

Execution is often using Warren Abstract Machine (WAM) that supports backtracking and instructions for unification.

To make logic programs faster, there exist (type inference as well as) **mode analysis** that computes functional dependencies saying that certain values can be computed as function of others, and thus compiler can pre-generate functions that correspond to every direction of relation.

Further reading:

- Constraint-Based Mode Analysis of Mercury
- An overview of Ciao and its design philosophy

Code Specialization

By partially evaluating program at compile time, we can specialize its parts and generate more efficient code.

Such transformation can be done automatically or under user control using, for example, staged computation, macros, templates.

```
def fold(l: List[A], b: B, f : (A,B) => B): B = l match {
  case Nil => b
  case x::xs => f(x, fold(xs,b,f))
}
fold(l, 0, _ + _)
```

$\Downarrow$

```
def foldZeroPlus(l: List[A]): B = l match {
  case Nil => 0
  case x::xs => x + foldZeroPlus(xs) // no closure
}
foldZeroPlus(l)
```

Algebraic Transformations

Higher-order combinators such as map satisfy many laws that can be used for optimization, including parallel execution.

Typically these laws hold only when functions are pure

```
list.map(f).map(g) == list.map(x => g(f(x)))
```

Type systems and program analyses for purity are an active areas of research.

If a language has mutable objects and allows their sharing, it is particularly difficult to prove that a function behaves as pure: knowing if a modification is to auxiliary objects or externally observable objects requires reasoning about possible heap configurations (**shape analysis**, **alias analysis**).

RISC V Core: BOOM

This just presented over the weekend:
`https://www.youtube.com/watch?v=toc2GxL4RyA`
Points:

- open source cores as alternatives to propritary cores with potential back doors
- using domain-specific languages in Scala for hardware design results in much more compact description
- challenges in compilation to wires (for example, place and route that takes a day)

Synthesis of Functions of a Given Type

Type Judgments and Questions

In environment Γ, expression e has type T:

$$\Gamma \vdash e : T$$

After defining this relation inductively using type rules, we can ask different types of questions:

		given	task
type checking	$\Gamma \vdash e : T$	Γ, e, T	check if $\Gamma \vdash e : T$
type inference	$\Gamma \vdash e : ?$	Γ, e	find T s.t. $\Gamma \vdash e : T$
type inhabitation	$\Gamma \vdash ? : T$	Γ, T	find e s.t. $\Gamma \vdash e : T$

Inhabitation can be used for type-directed completion: in a given program context, can we construct the expression of a given type?

Also, type T can be viewed as a proposition, program e as a proof. Inhabitation asks if the proposition T has some proof e
(if it is a theorem, usually in some constructive logic).

Simply Typed Lambda Calculus (Church Style)

ee_1 means e applied to e_1; in Scala: $e(e_1)$

$\lambda x : T_1.e$ is anonymous function, in Scala: $(x : T_1) \Rightarrow e$

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{variable}$$

$$\frac{\Gamma \vdash e : T_1 \to T_2, \quad \Gamma \vdash e_1 : T_1}{\Gamma \vdash ee_1 : T_2} \quad \text{application}$$

$$\frac{\Gamma[x := T_1] \ \vdash e_2 : T_2}{\Gamma \vdash (\lambda x : T_1.\ e_2) : T_1 \to T_2} \quad \text{abstraction}$$

Type Checking is Easy and Types Unique

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T}$$ variable

$$\frac{\Gamma \vdash e : T_1 \to T_2, \quad \Gamma \vdash e_1 : T_1}{\Gamma \vdash e e_1 : T_2}$$ application

$$\frac{\Gamma[x := T_1] \ \vdash e_2 : T_2}{\Gamma \vdash (\lambda x : \mathbf{T_1}.\ e_2) : T_1 \to T_2}$$ abstraction

Which Cases of Inhabitation are Difficult?

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \qquad \text{variable}$$

$$\frac{\Gamma \vdash e : T_1 \to T_2, \quad \Gamma \vdash e_1 : T_1}{\Gamma \vdash e e_1 : T_2} \qquad \text{application}$$

$$\frac{\Gamma[x := T_1] \ \vdash e_2 : T_2}{\Gamma \vdash (\lambda x : T_1.\ e_2) : T_1 \to T_2} \qquad \text{abstraction}$$

Suppose that we are equally happy with any of the possible terms of a given type.

Which Cases of Inhabitation are Difficult?

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{variable}$$

$$\frac{\Gamma \vdash e : T_1 \to T_2, \quad \Gamma \vdash e_1 : T_1}{\Gamma \vdash e e_1 : T_2} \quad \text{application}$$

$$\frac{\Gamma[x := T_1] \ \vdash e_2 : T_2}{\Gamma \vdash (\lambda x : T_1.\ e_2) : T_1 \to T_2} \quad \text{abstraction}$$

Suppose that we are equally happy with any of the possible terms of a given type.

Challenge: in application rule, T_1 can be arbitrarily complex type; we would need to guess it.

Approaches to Solve the Difficulty

For more complex type systems, the problem is undecidable

- we can choose to restrict our search to expressions e of some bounded size
- the problem becomes decidable if type checking is decidable: can try all terms up to given size

In our simple case of simply typed lambda calculus: there is a **terminating algorithm** that solves type inhabitation problem (in deterministic singly exponential time and polynomial space), without requiring any bound on the size of e.

Key Idea: Long Normal Form of Lambda Terms

Reductions preserve types of terms. Apply them to get normal form.

Beta reduction: $(\lambda x.e_1)e_2 \longrightarrow \mathsf{subst}(e_1, x, e_2)$

If applied exhaustively ensures that left side of application is never a lambda.
This process can blow up, but terminates (“strong normalization”).

Result: applications have the form $f e_1 \ldots e_n$ where f is a variable.

Key Idea: Long Normal Form of Lambda Terms

Eta expansion: when e has type $T_1 \to T_2$ then: $e \longrightarrow (\lambda x : T_1 . e_1 x)$
Strategy: replace partially applied type variables by adding missing arguments.
Given

$$f : T_1 \to (T_2 \to \ldots (T_n \to T))$$

where T is not a function type, if $k < n$ then replace

$$f e_1 \ldots e_k$$

that is not of the form $f e_1 \ldots e_k e_{k+1}$ by

$$\lambda x_{k+1} . \lambda x_{k+2} . \ \ldots \lambda x_n . \ f e_1 \ldots e_k x_{k+1} \ldots x_n$$

As a result, application is always to a variable and applies all of its arguments that the type permits.

Type Rules for Long Normal Form Terms

full variable application $(n \geq 0)$, if T is non-function type:

$$\frac{(f, T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T) \in \Gamma, \quad \Gamma \vdash e_1 : T_1, \; \ldots \; \Gamma \vdash e_n : T_n}{\Gamma \vdash f e_1 \ldots e_n : T}$$

abstraction:

$$\frac{\Gamma \uplus \{(x_1, T_1), \ldots, (x_n, T_n)\} \; \vdash e : T}{\Gamma \vdash (\lambda x_1 : T_1. \; \ldots x_n : T_n.e) : T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T}$$

- if there is a term of given type, this calculus derives some equivalent term of this type
- applying rules backwards generates queries with types that appear somewhere in environment or the query (possibly inside the syntactically larger types)
- there is no need to explore query $\Gamma \vdash ? : T$ if a query $\Gamma' \vdash ? : T$ was already asked for $dom(\Gamma) \subseteq dom(\Gamma')$

There is a finite number of extensions of the original Γ, so the search can stop after a finite amount of steps.

Using Inhabitation to Generate Suggestions

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In ACM SIGPLAN PLDI, 2013. `https://doi.org/10.1145/2499370.2462192`

- optimize representation to ignore duplicate arguments of the same type to check inhabitation (succinct terms)
- extend the algorithm to enumerate terms of given type
- instead of ordering terms by size, assign cost for each variable (cost of term is sum of its variables)
- map synthesis of Scala and Java code to lambda calculus
- statistically estimate the cost of variables from code corpus