Topic 16: Issues in compiling functional and object-oriented languages

COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

Compiling functional and OO languages

General structure of compiler unchanged

Main challenges in functional languages:

- semantic analysis: parametric polymorphism
- code generation: higher-order functions

Main challenges in object-oriented languages:

- semantic analysis classes and inheritance, access restrictions (private/public,…)
- code generation: method dispatch

Also: garbage collection (Java, ML, Haskell, …)

Parametric polymorphism -- motivation

fun ilist_length (l: int_list) : nat := …

:

fun clist_length (l: char_list) : nat := …

fun slist_length (l: string_list) : nat := …

virtually identical definitions

α: type variable

fun list_length (l: α list) : nat :=
  case l with nil ➔ 0 | (h::t) ➔ 1 + list_length t

h: α   t: α list

Java

class List <T> {
  …
}

- benefits for programmer:
  - code reuse; flexible libraries
  - code clarity: same behavior/structure ➔ same code
  - modularity / information hiding

- benefits for compiler: no code duplication ➔ no duplicate analysis

Polymorphism – code generation strategies

- **monomorphization**: compiler identifies all possible instantiations, generates separate code for each version, and calls the appropriate version (type information at call sites)
  + conceptually simple – "core language" remains monomorphic
  + instantiations can use different representations, and be optimized more specifically
  - requires whole-program compilation (identify **all** instantiations); hence no separately compiled (polymorphic) libraries!
  - code duplication
- **monomorphization at JIT compilation**
  - not every compiler / language / application suitable for JIT
- **uniform representation** for all types ("boxed", ie one pointer indirection – even for scalar types like int, float)
  + avoids code duplication and JIT overhead
  - memory overhead; pointer indirection costly at runtime
- **"intensional types" / dynamic dispatch**: maintain runtime representations of types, use this to identify which code to invoke
  - memory overhead, runtime overhead

Polymorphism – type analysis

Intuitive interpretation of type variables: “for all”

- **explicit polymorphism:**
  - position of universal quantification syntactically explicit
  - in particular: non-top level quantification allowed

    (nat → (forall α, α list)) → nat), forall β, (β tree → forall α, (α x β)) → nat

  Very expressive! Only type checking!
- **implicit polymorphism*:**
  universal quantification only at top-level, hence syntactically redundant

  forall α, (α list → nat), forall α β, (β tree → (α x β)) → (α x β)

  Algorithmically more feasible (inference!), and sufficient for many application (➔ ML)

* formal distinction between types and type schemes…

Polymorphism – type substitution

<u>Substitution</u> $X[t/\alpha]$: instantiate a type variable α in X to t

$$(\alpha \times \beta \to \gamma \to \beta)[\,\mathbf{nat} / \alpha\,] = \mathbf{nat} \times \beta \to \gamma \to \beta$$

$$(\alpha \times \beta \to \gamma \to \beta)[\,\mathbf{nat} / \beta\,] = \alpha \times \mathbf{nat} \to \gamma \to \mathbf{nat}$$

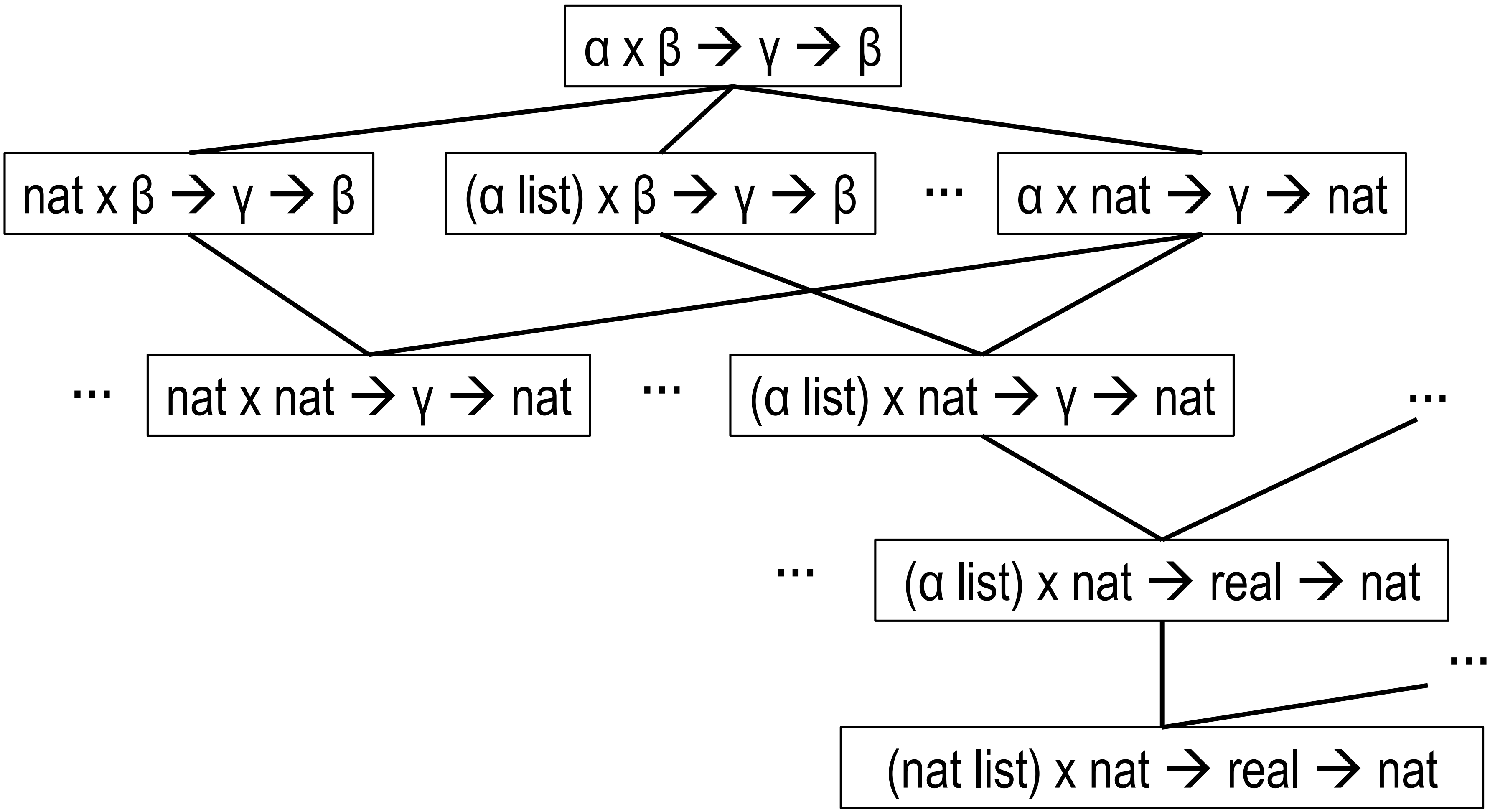
$$(\alpha \times \beta \to \gamma \to \beta)[\,(\delta\ \mathbf{list}) / \gamma\,] = \alpha \times \beta \to \delta\ \mathbf{list} \to \beta$$

$$(\alpha \times \beta \to \gamma \to \beta)[\,(\alpha\ \mathbf{list}) / \gamma\,] = \alpha \times \beta \to \delta\ \mathbf{list} \to \beta$$

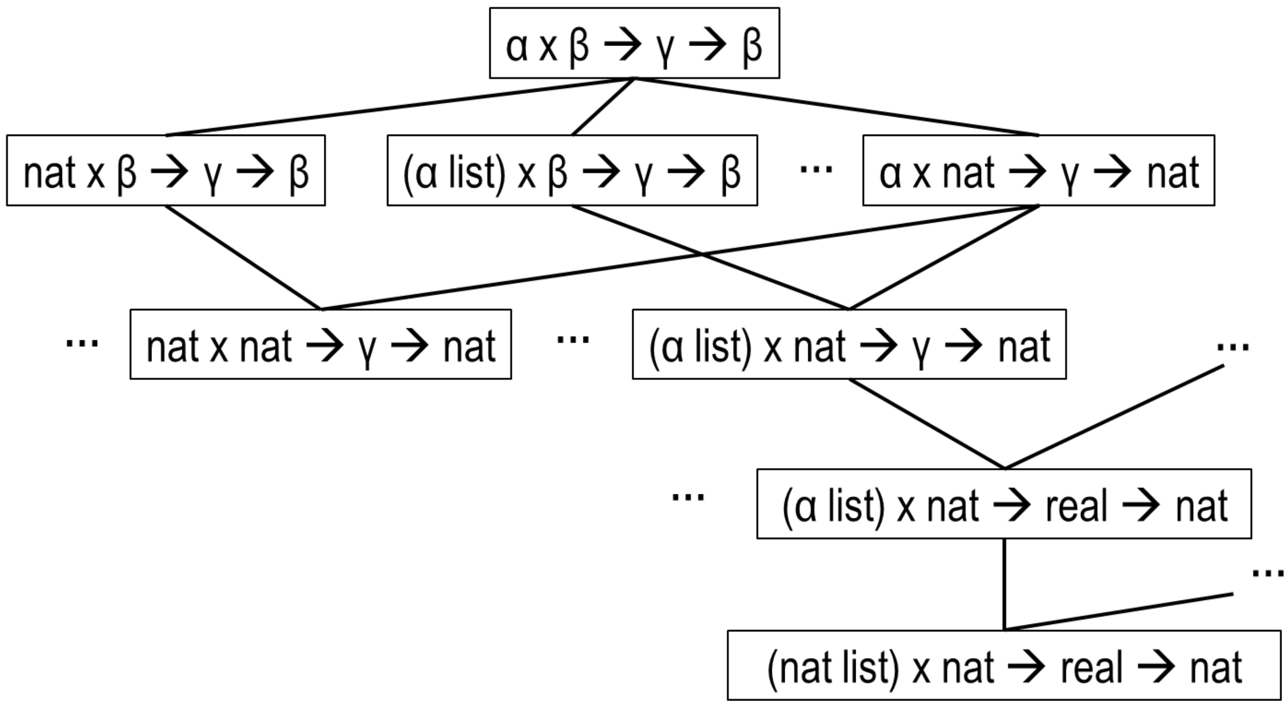
α is implicitly all-quantified here, too – substitution is "capture-avoiding", like α-renaming of term variables.

Polymorphism – type inference a la Hindley-Milner

ML's type system: types can be ordered by the "is-an-instantiation-of" relationship



Polymorphism – type inference a la Hindley-Milner



Thus: two types are either

- incompatible, eg. **int** vs **real**, **int** vs **int list**, **int** vs **α list**, or
- unifiable: there are are substitutions that make them “equal”:

  int vs **int** (substitutions: empty, empty)

  int vs **α** (substitutions: empty, **int** / α)

  int x **α list** vs **α** x **β** (substitutions: empty, [int / α, α list / β])

If they are unifiable, there is a (unique) most general type.

Polymorphism – type inference a la Hindley-Milner

Hindley-Milner type inference:

- recursively walk the code structure, as in lecture on type systems, and return most general type scheme
- when necessary (eg for matching type of a function): perform unification – report type error if not-unifiable

Algorithm **W** (cf function "infer" in slides on Types)

can now include TypeVars

substitution; map type variables to types

fun **W** (Σ: context) (e:expr): (Type x Subst) option = … (*next slide ..*)

assumptions; maps term variables to types

Auxiliary function: **Unify**: (Type x Type) -> Subst option

Polymorphism – type inference a la Hindley-Milner

```
fun W (Σ: context) (e:expr): (Type x Subst) option =
  case e of
   …
   | f a => case W Σ a of
              Some (T, τ) => case W (Στ) f of
                               Some (U, σ) => case Unify (Uσ, T -> β) of
                                                Some ω => Some (βω, τσω)
              … (*all other cases: None*)
   | … (*other cases of e*)
```

apply subst τ to types in Σ

apply subst τ to type U

fresh type variable

Details: Damas, Luis; Milner, Robin (1982): *Principal type-schemes for functional programs*; 9th Symposium on Principles of programming languages (POPL'82). ACM. pp. 207–212.

For full ML, inference is DEXPTIME-complete - but in practice: linear/polytime

Higher-order functions

Functional languages: arguments and return values can be functions.

```
type intfun = int → int

fun foo () : intfun =
      return (fun z => z + 5);

var f = foo ();
f 2;

fun apply42 (f:intfun): int = return (f 42);
var q = apply42 foo
```

return value is a function

function parameter is of functional type

function argument is a function

Also with polymorphism: **map** (f: α → β) → α list → β list

Higher-order functions

```
type intfun = int → int

fun foo () : intfun =
      return (fun z => z + 5);

var f = foo ();
f 2;
```

Q: where is the code for fun z => z + 5 located, i.e. what address should we jump to when calling f 2?

Higher-order functions

```
type intfun = int → int

fun foo () : intfun =
    return (fun z => z + 5);

var f = foo ();
f 2;
```

Q: where is the code for fun z => z + 5 located, i.e. what address should we jump to when calling f 2?

A: have compiler generate a fresh name, **bar**, and emit code for the function fun **bar** z => z + 5. Have **foo** return the address of / label **bar**.
Then use jump-register instruction (indirect jump) for call.

Higher-order functions

```
type intfun = int → int

fun foo () : intfun =
      return (fun z => z + 5);

var f = foo ();
f 2;
```

Q: where is the code for fun z => z + 5 located, i.e. what address should we jump to when calling f 2?

A: have compiler generate a fresh name, **bar**, and emit code for the function fun **bar** z => z + 5. Have **foo** return the address of / label **bar**. Then use jump-register instruction (indirect jump) for call.

```
fun apply42 (f:intfun): int = return (f 42);
var q = apply42 foo
```

Call to apply42 can pass address of **foo** as argument. Use jump-register for call f 42.

But what about this?

```
fun add (n:int) : intfun =
    let fun h (m:int) = n+m
    in h end

fun twice (f: intfun): intfun =
    let fun g(x:int) = f (f x)
    in g end

var addFive: intfun = add 5
var addTen : intfun = twice addFive
```

At runtime, calls **add** 5, **add** 42 should yield functions that behave like

h_5 (m:int) = 5+m

h_{42} (m:int) = 42+m.

But what about this?

```
fun add (n:int) : intfun =
    let fun h (m:int) = n+m
    in h end

fun twice (f: intfun): intfun =
    let fun g(x:int) = f (f x)
    in g end

var addFive: intfun = add 5
var addTen : intfun = twice addFive
```

At runtime, calls **add** 5, **add** 42 should yield functions that behave like
h_5 (m:int) = 5+m
h_{42} (m:int) = 42+m.
Each h_i outlives the stackframe of its static host, **add**, where h_i would usually look up n following the static link, -- but **add**'s frame is deallocated upon exit from **add**.

Similarly, **twice** addFive should yield
$g_{addFive}$(x:int) = addFive (addFive x)
but g_f needs to lookup f in stackframe of **twice**.

Combination of higher-order functions and nested function definitions conflicts with stack discipline of frame stack and with holding arguments and local variables in the stack frame.

Higher-order functions

Combination of higher-order functions and nested function definitions conflicts with stack discipline of frame stack and with holding arguments and local variables in the stack frame.

```
type intfun = int → int

fun add (n:int) : intfun =
      let fun h (n:int) (m:int) = n+m
      in h n end

fun twice (f: intfun): intfun =
      let fun g (f:intfun) (x:int) = f (f x)
      in g f end

var addFive: intfun = add 5
var addTen : intfun = twice addFive
```

parameter lifting

```
type intfun = int → int

fun h (n:int) (m:int) = n+m

fun add (n:int) : intfun = h n

fun g (f:intfun) (x:int) = f(f x)

fun twice (f: intfun): intfun = g f

var addFive: intfun = add 5
var addTen : intfun = twice addFive
```

parameter lifting + block raising = λ-lifting

Higher-order functions

Combination of higher-order functions and nested function definitions conflicts with stack discipline of frame stack and with holding arguments and local variables in the stack frame.

```
type intfun = int → int

fun add (n:int) : intfun =
      let fun h (n:int) (m:int) = n+m
      in h n end

fun twice (f: intfun): intfun =
      let fun g (f:intfun) (x:int) = f (f x)
      in g f end

var addFive: intfun = add 5
var addTen : intfun = twice addFive
```

parameter lifting

```
type intfun = int → int

fun h (n:int) (m:int) = n+m

fun add (n:int) : intfun = h n

fun g (f:intfun) (x:int) = f(f x)

fun twice (f: intfun): intfun = g f

var addFive: intfun = add 5
var addTen : intfun = twice addFive
```

parameter lifting + block raising = λ-lifting

Need to pair up code pointers with data for host-function's variables / parameters, ie construct representations of **h n** (like h 5, h 42) and of **g f** (like g addFive). These structures need to be allocated on the **heap**.

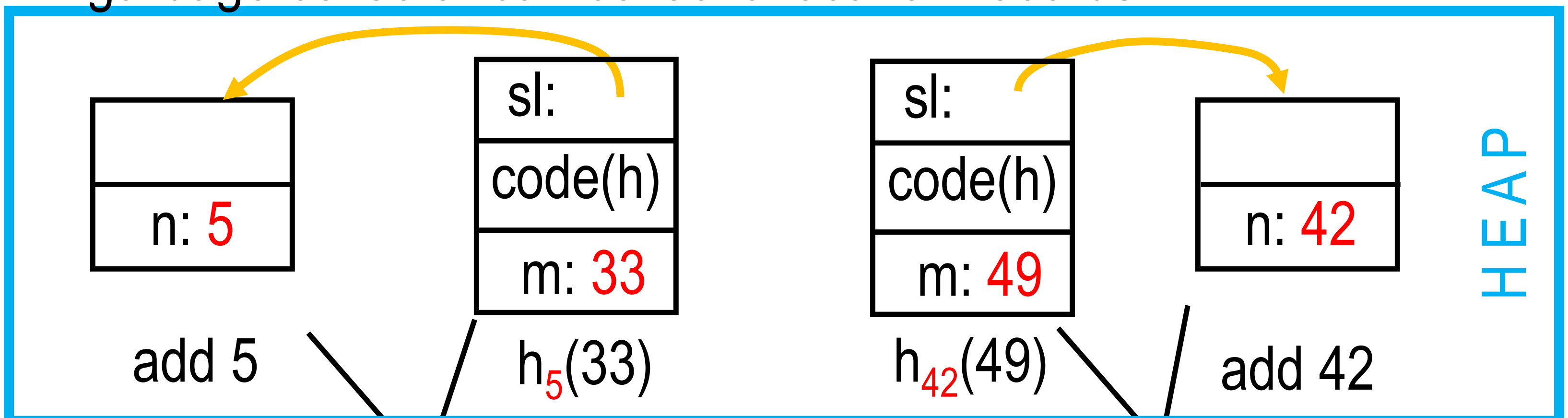
Closures

"code+data" pairs: representation of functions that have been provided with some of their arguments.

- "code": label/address of code to jump to
- "data": several representations possible

a) pointer to allocation record of host function's invocation: "static link"

- host function must still be heap-allocated to prevent stale pointers
- caller of closure creates activation record based on data held in closure, deposits additional arguments at known offsets and jumps to the code pointer provided in closure
- garbage collector can collect allocation records

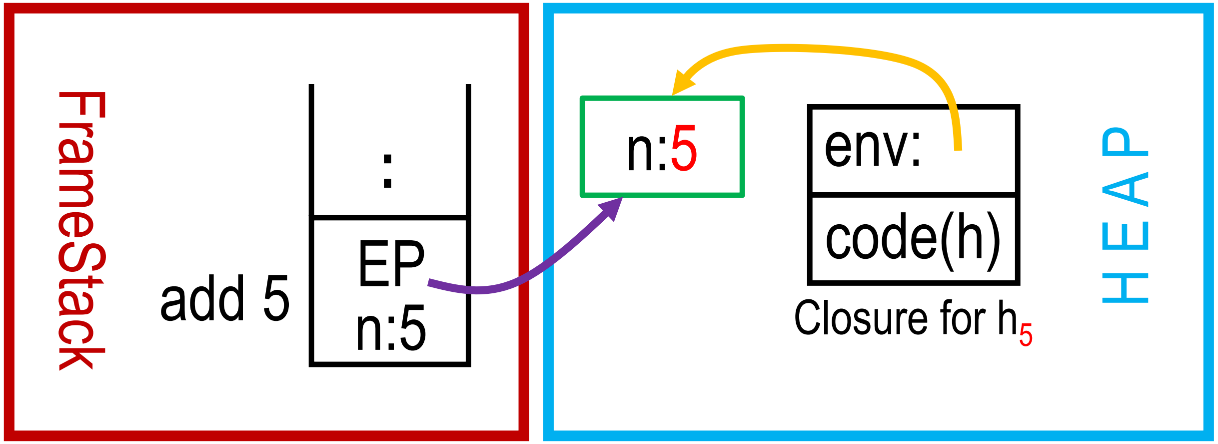


Activation records held in heap, linked by static links

Closures

b) pointer to a record (environment) in heap that holds the host function's escaping variables (ie exactly the variables the inner function might need)

- host function can be allocated on stack, receive its arguments as before, and hold non-escaping variables, spills, etc in stack frame
- New "local variable" EP points to environment
- host frame deallocated upon exit from host-function, but environment of escaping variables not deallocated (maybe later GC'ed)
- closure's data part points to environment

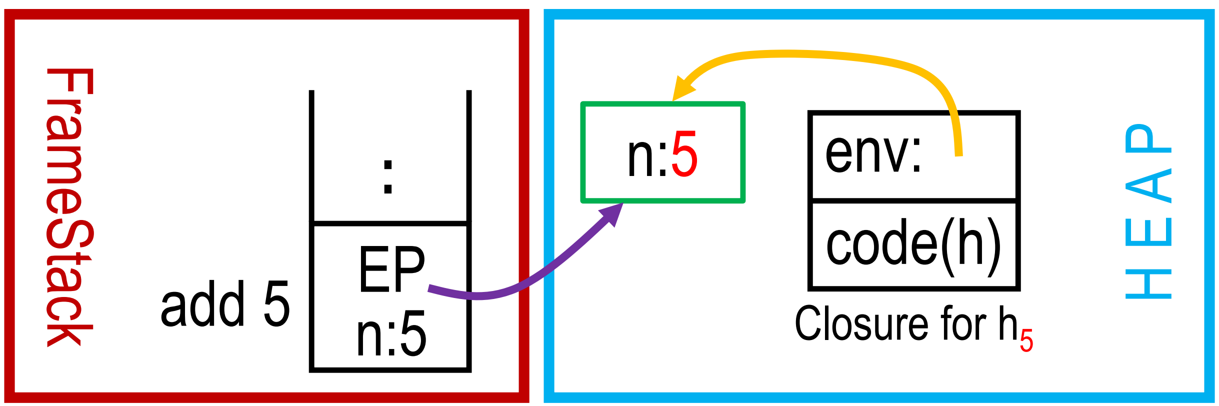


Allocation records for invocations to h_5 are held on frame stack, and have pointer to the closure.

Closures

b) pointer to a record (environment) in heap that holds the host function's escaping variables (ie exactly the variables the inner function might need)

- host function can be allocated on stack, receive its arguments as before, and hold non-escaping variables, spills, etc in stack frame
- New "local variable" EP points to environment
- host frame deallocated upon exit from host-function, but environment of escaping variables not deallocated (maybe later GC'ed)
- closure's data part points to environment



Allocation records for invocations to h_5 are held on frame stack, and have pointer to the closure.

Pitfall: need to prevent h from modifying n, so that repeated invocations $h_5(33)$, $h_5(22)$ don't interfere ➔ **no assignments to variables etc**

(Class-based) Object-oriented languages

static

Classes: enriched notion of types with support for

- record type containing first-order ("fields") and functional ("methods") components
- extension/inheritance/subclass mechanism
  - allows addition of data (fields) and functionality (methods)
  - allows modification of behavior: overriding of methods (often, types of parameters and result cannot be modified)
  - transitive (→ class hierarchy), with top element OBJECT etc
- self/this: name to refer to data component in methods; can often be considered an (implicit) additional method parameter
- initialization/creation method for class instances ("objects")
- limiting visibility/inheritance of fields/methods: private/public/final

dynamic

Objects: runtime structures arising from instantiating classes

- record on heap containing values for all fields
- invocation of methods: dispatch based on dynamic class, with pointer to data field passed as argument of "self/this"

Object-oriented languages: type checking

```
class B: extends A {
    A super; // often implicit
    int f1; // maybe with explicit explicit initialization
    B b; // fields may be (pointers to) objects of class we're defining
    C c; // fields may be (pointers to) object of other classes, too

    int foo (A a, D d) {…}
```

Tasks:
- maintain class table (maps class names to classes/types, cf context)
- maintain inheritance relationship (check absence of cycles)
- check type constraints regarding overriding method definitions
- checking of method bodies:
  add entry for **self** to local typing context
- check adherence to **private**/**public**/**final** declarations

Class can refer to each other in cyclic fashion; split analysis into phases

Object representation (single inheritance)

Single inheritance: each class extends at most one other class.

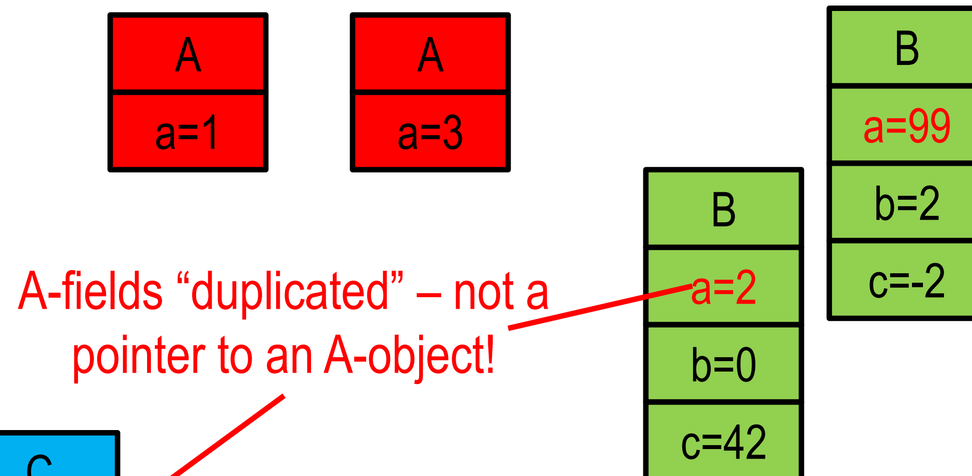
(typically: classes other than OBJECT extend **exactly** one class)

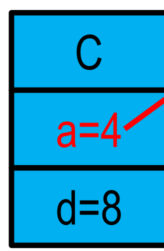
class A extends Object { int a }

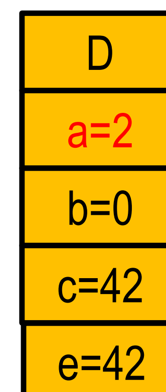
class B extends A { int b; int c }

class C extends A { int d }

class D extends B { int e }







Fields: objects of class C contain **first** the fields for objects of C’s superclass, A, then fields declared in C.

Avoids code duplication when implementing inherited methods: loads/stores to fields access same location, counted as offset from base of object

Static versus dynamic class of object

Typically, can assign an object of class C to a variable/field declared to be of type A, where A is a superclass of C.

class A extends Object { int a }

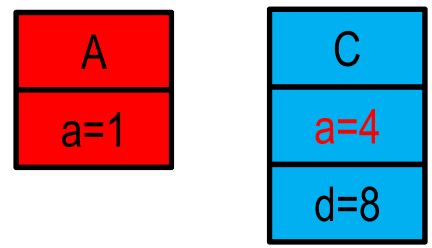
class B extends A { int b; int c }

class C extends A { int d }

class D extends B { int e }

var a_object : A := new C

Cf. subtyping



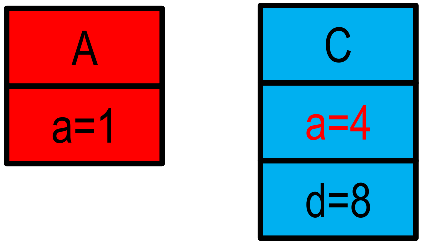
method m (x:A) : T = {…} in class X:
body of m well-typed w.r.t. x:A, so can only access x.a. Passing a larger C object is not harmful: additional fields ignored.

method k (…) : A = { … } in class Y:
k may return an object of any subclass of A – eg body of k can be **new** C - but client only knows that the returned object has a field a.

Method selection typically based on dynamic class

```
class A extends Object { int a;
    int f () = return (a +2) }
  :
class C extends A { int d;
    int f () = return d //overrides A.f() }

int m (x:A) {
  return x.f() // code generation: jump to A.f?
}
var c_obj := new C();
print m(c_obj); // should invoke C.f, not A.f()
```



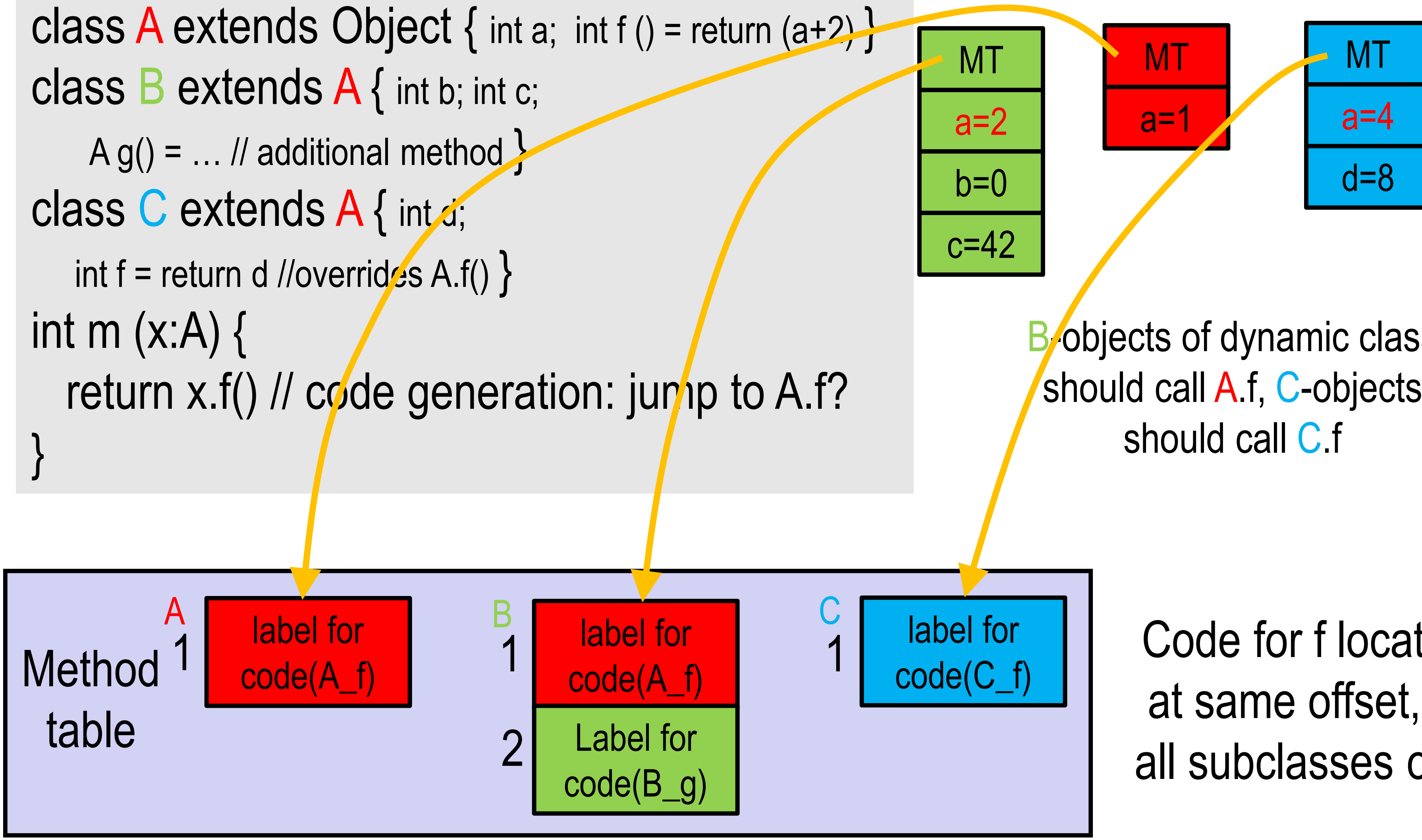
How to achieve this:

subclass of its static class

- object contains reference to its “dynamic class”
  - requires class/class names to be represented at runtime
- organize method dispatch table similar to fields (next slide)

Method dispatch based on dynamic class

```
class A extends Object { int a;  int f () = return (a+2) }
class B extends A { int b; int c;
    A g() = … // additional method }
class C extends A { int d;
    int f = return d //overrides A.f() }
int m (x:A) {
  return x.f() // code generation: jump to A.f?
}
```



B-objects of dynamic class should call A.f, C-objects should call C.f

Code for f located at same offset, in all subclasses of A

Now, the implementation of m follows it's A-argument's **link** to the method table, then knows where to find f.

Final exam

Saturday, May 14th, Friend 004,
7:30pm – 10:30pm

Closed book, laptop, iphone !

Don’t blame me…..

Final exam

Saturday, May 14th, Friend 004,
7:30pm – 10:30pm

Don't blame me.....

Closed book, laptop, iphone !

Cheat sheet: one A4 paper, double-sided.

Exam is cumulative: covers the **entire** semester

- lecture material incl today
- MCIML: except for last chapter and overly TIGER – specific implementation details
- HW 1 – HW 9, incl. basic ML programming