Compiler construction

Lecture 8: Functions

Thomas Sewell

Spring 2020

Chalmers University of Technology — Gothenburg University

Nested functions

A Nested Function

CHALMERS

Suppose we extended JAVALETTE with nested functions.

```
double hypSq(double a, double b) {
  double square(double d) {
    return d * d;
  }
  return square(a) + square(b);
}
```

Another example

To make nested functions useful we would like to have lexical scoping.

The inner functions are inside the outer blocks, so we would like to have access to the outer variables.

```
double sqrt(double s) {
  double newton(double y) {
    return (y + s / y) / 2;
  }
  double x = 0.0; int i = 0;
  while (i < 10) {
    x = newton(x);
    i++;
  }
  return x;
}
```

A Closely Related Problem

Here is a similar issue in Haskell:

```
f x = foldr (\i xs -> xs ++ [g i x]) [] [1 .. x]
```

The anonymous λ function is also a nested function. It can also be named explicitly:

```
f x = foldr f2 [] [1 .. x]
  where
    f2 i xs = xs ++ [g i x]
```

Again, the difficult part is the use of `x` from the outer scope.

This Lecture

CHALMERS

Today

- Native stack demo (attempt 2)
- Access links: imperative-style nested functions
- Closures and higher-order functional features

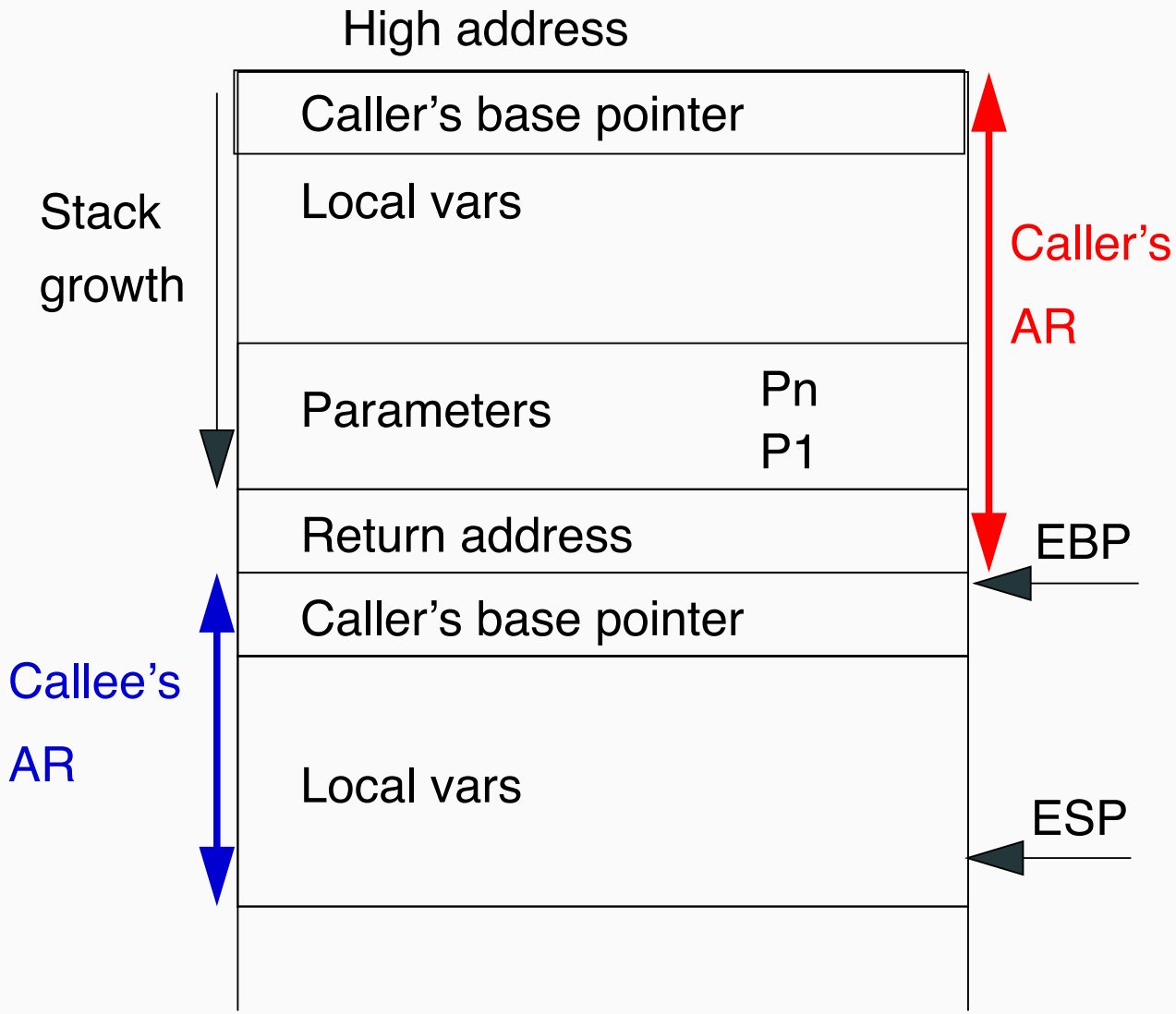
Demo Mk II: Popcount

CHALMERS

This is a slow implementation of the population-count/bit-count operation.

```
int
popcount (int n) {
  int i, j;
  if (n == 0) {
    return 0;
  }
  return popcount (n / 2) + (n % 2);
}
```

Reminder: x86 stack



Access Links

CHALMERS

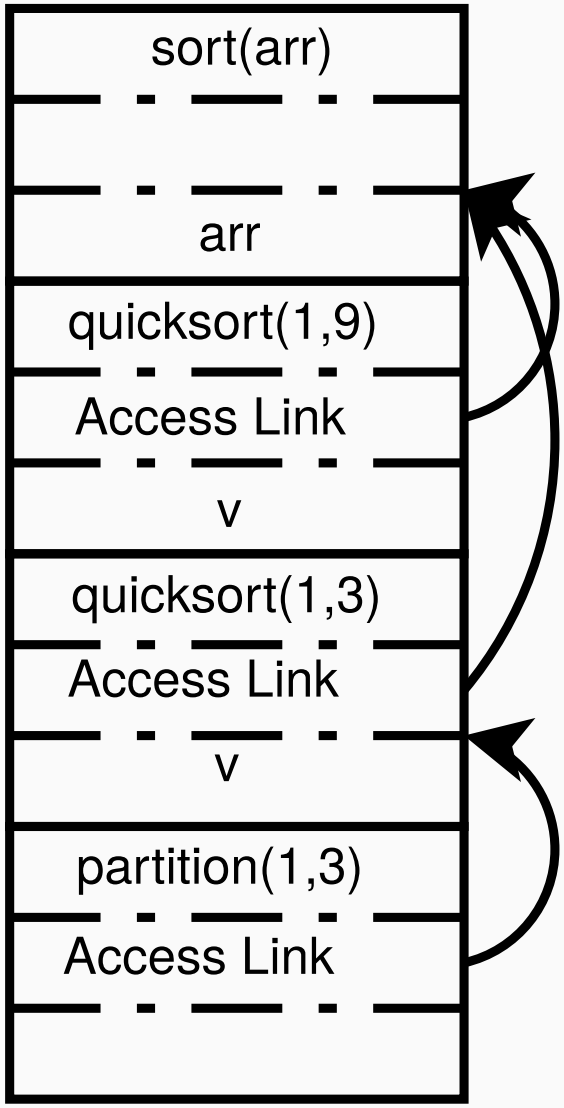
- Access Links: a mechanism to access variables defined in an enclosing procedure
- An access link is an extra field in a stack frame which points to the closest stack frame of the enclosing procedure

Access Links

Outline of a quicksort implementation:

```
void sort(int[] arr) {
  void quicksort(int m,int n) {
    v = ...
    void partition(int y,int z) {
      ... arr ... v ...
    }
    ... a ... v ... partition ... quicksort
  }
  ... quicksort ...
}
```

Example stack



The access links are pointers in the various stack frames that let us find the nested parent, `partition` → `quicksort`, `quicksort` → `sort`.

When accessing e.g. the variable `arr` in `partition` we need to go through the access link to `quicksort` and then to `sort`.

Note: without recursion, we could have a simpler solution.

Manipulating Access Links

When procedure `q` calls procedure `p` there are three cases to consider:

1. `p` has higher nesting depth than `q`

Then the depth of `p` must be exactly one larger than `q` and `p`'s access link must point to `q`.

2. `p` and `q` have the same nesting depth

The access link for `p` is the same as for `q`.

3. `p` has a lower nesting depth than `q`

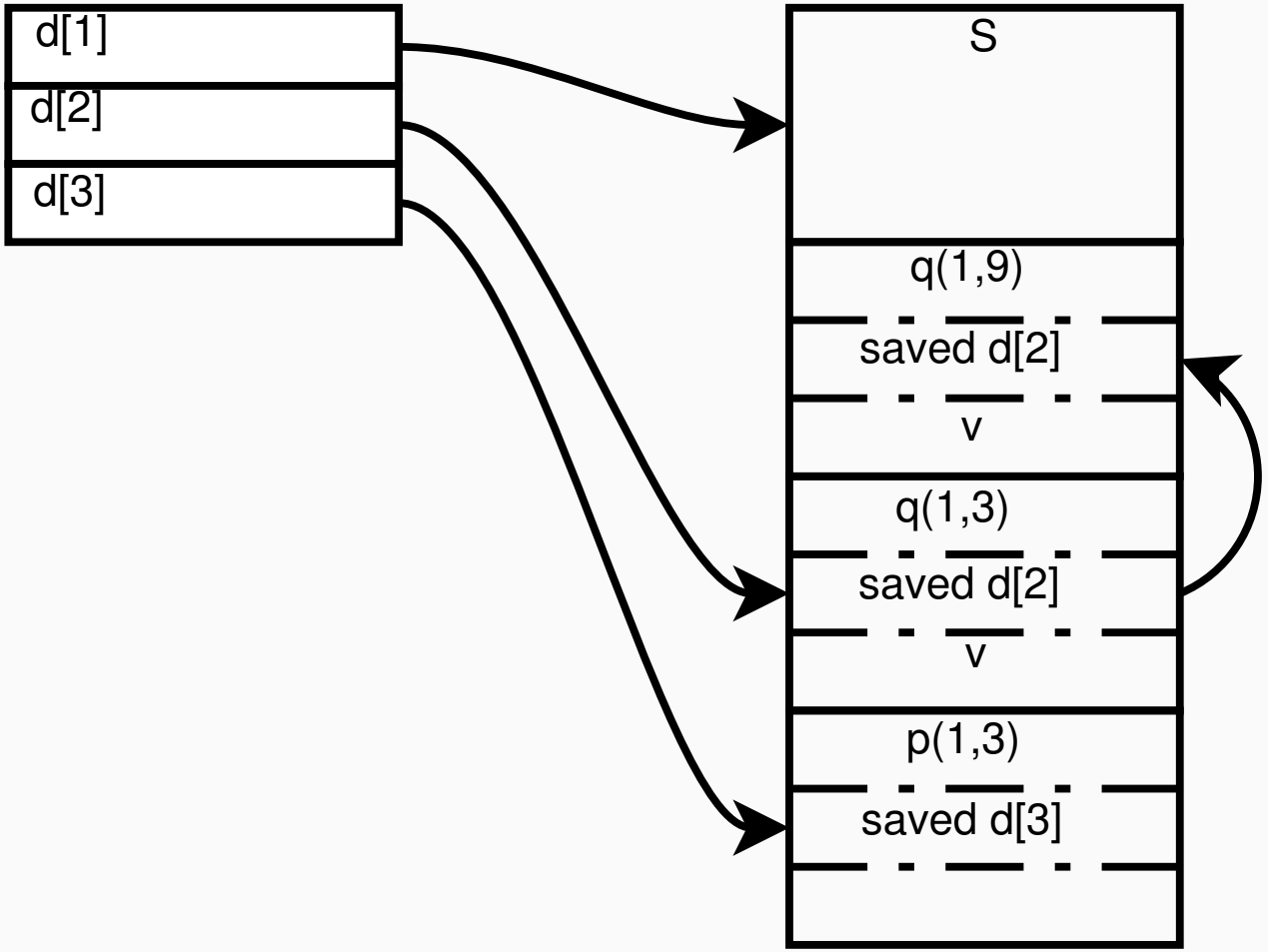
Let n_p be the nesting depth of `p` and n_q be the nesting depth of `q`. Furthermore, suppose that `p` is defined immediately within procedure `r`. The top activation record for `r` can be found by following $n_q - n_p + 1$ access links down the stack.

Displays

CHALMERS

- If the nesting depth is very large, then the link chains may be very long; traversing these links can be costly
- Displays were developed to speed up access
- A display is a stack, separate from the call stack, which maintains pointers to the most recent activation record of the different nesting depths
- The display grows and shrinks with the maximum nesting depth of the functions on the call stack
- The rules for updating the display are roughly the same as the rules for updating the access links

Displays



Lambda Lifting

CHALMERS

- Another way of implementing nested functions is by lifting them to the top level
- Free variables are handled by adding them as parameters to the lifted function

Lambda Lifting - example

CHALMERS

Original `sqrt`

```
double sqrt(double s) {
  double newton(double y) {
    return (y + s / y) / 2;
  }
  double x = 0.0;
  int i = 0;
  while (i < 10) {
    x = newton(x);
  }
  return x;
}
```

Lambda Lifting - example

CHALMERS

Lambda lifted `sqrt`

```
double newton(double y, double s) {
  return (y + s / y) / 2;
}

double sqrt(double s) {
  double x = 0.0;
  int i = 0;
  while (i < 10) {
    x = newton(x, s);
  }
  return x;
}
```

Call-by-reference

Consider lambda lifting the function below.

The local function `incc` modifies its free variable. In order to lift `incc` we have to pass the parameter `c` by reference.

```
void foo() {
  int c = 0;
  void incc() {
    c++;
  }
  incc();
  incc();
  printInt(c);
}
```

Call-by-reference

Consider lambda lifting the function below.

The local function `incc` modifies its free variable. In order to lift `incc` we have to pass the parameter `c` by reference.

```
void incc(int *c) {
  (*c)++;
}
void foo() {
  int c = 0;
  incc(&c);
  incc(&c);
  printInt(c);
}
```

Higher Order Functions

Higher Order Fuctions in JavaLette

Adding higher order functions to JavaLette we need a new form of types:

```
Type(Type, ..., Type)
```

Examples:

- `bool(int, int)`
  A function which takes two `int` arguments and returns a `bool`
- `void()`
  A function which takes no arguments and doesn't return anything

Higher Order Functions in JavaLette

CHALMERS

```
int main() {
  int(int) add(int n) {
    int h(int m) {
      return n + m;
    }
    return h;
  }

  int(int) addFive = add(5);
  printInt(addFive(15));
}
```

Higher Order Functions in JavaLette

CHALMERS

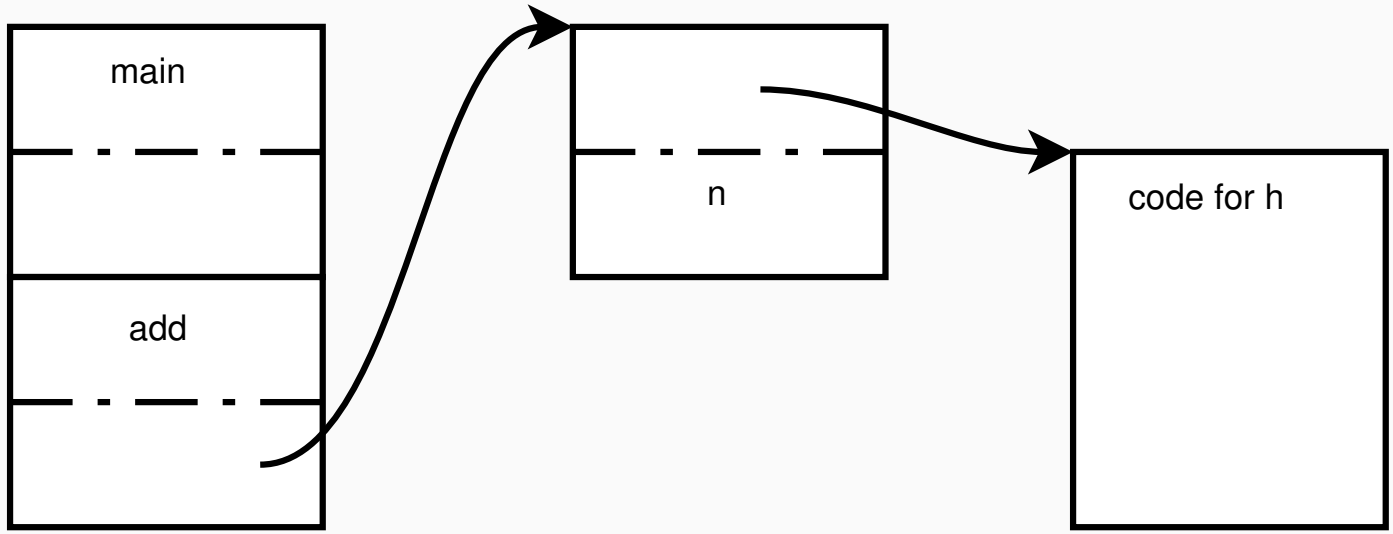
```
int main() {
  int(int) add(int n) { ... }
  int(int) addFive = add(5);

  int(int) twice(int(int) f) {
    int g(int x) {
      return f(f(x));
    }
    return g;
  }

  int(int) addTen = twice(addFive);
  printInt(twice(twice(addTen))(6));
}
```

Implementing higher order functions

There are several ways of implementing higher order functions:

- Access Links can be adapted to also deal with higher order functions
- Defunctionalization is a method to convert higher order functions to data structures; requires whole program compilation
- Closures are used to represent functions by a heap allocated record containing a code pointer and the free variables of the function
- Using closures is by far the most common implementation method

Closures



- The closure for `h` created by `add` contains a pointer to the code for `h` and the value for the variable `n`
- The closure is heap allocated
  - it is returned from `add` so must outlive it

Closures and mutable variables

What happens with the stack allocated variable `counter` once we exit the function `makeCounter`?

- Heap allocate part of the stack frame
- Forbid such programs (example: Java)

```
int() makeCounter(int start) {
  int counter = start;
  int inc() {
    counter++;
    return counter;
  }
  return inc;
}
```

Closures and mutable variables

Functional languages like Haskell and ML deal with the problem of closures and mutability as follows:

- Everything is immutable by default
- Mutation is introduced by references which always live on the heap

```
makeCounter = do
  r <- newIORef 0
  let inc = do
    n <- readIORef r
    writeIORef r (n+1)
    return n
  return inc
```

Anonymous nested functions

CHALMERS

Lambda expressions

- An increasingly popular language feature is to have anonymous nested functions, so called lambda expressions
- Compiling lambda expressions works the same way as nested functions with names

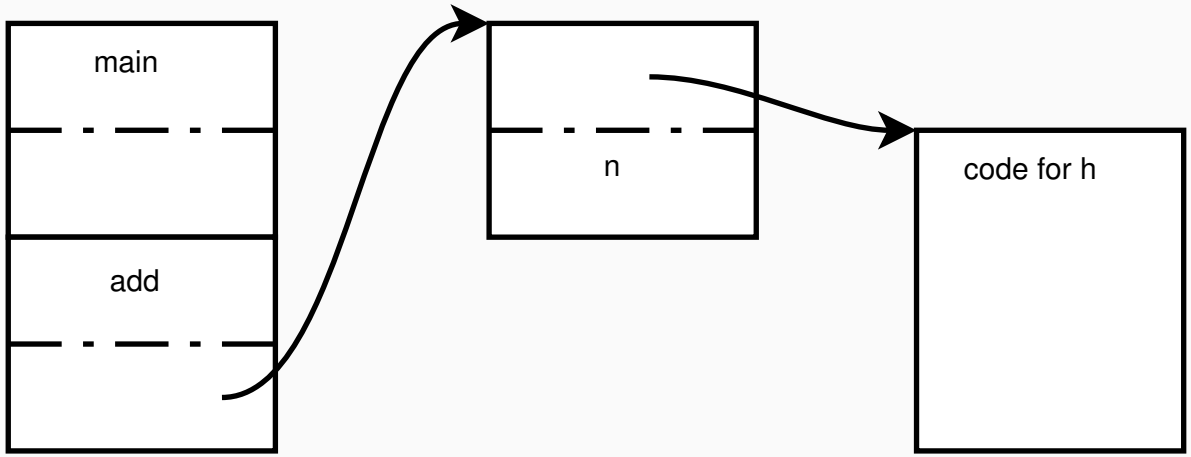
A note on terminology

One can often hear the phrase that a language “has closures”.

This is a somewhat unfortunate use of the word.

Closures are an implementation technique for the language feature called “higher order functions” or “first-class functions”.

Impact of Closures



- Calling a closure is a bit slower than calling a function pointer.
- Language design question: should we distinguish the types?
  - or have a special attribute?

Currying

Currying is a related language feature.

Currying can be expanded naively to lambda functions.

```
f x == (\y -> f x y)
```

However, calling the above lambda requires two function (closure) calls. Repeated currying may lead to inefficient chains of closure calls.

First optimisation: when constructing the above closure, check if `f` is itself a curry closure.

More difficult: try to adapt the final function body to suit the closure type it typically appears in.

Lazy evaluation

Question

- Is it possible to implement `if` as a function?

Question

- Is it possible to implement `if` as a function?
  - `if (x == null) {safe} {unsafe}`

Question

- Is it possible to implement `if` as a function?
  - `if (x == null) {safe} {unsafe}`
- We can fake it by using functions which take no arguments

```
void if(bool c, void() th) {
  if (c)
    th();
}
```

- We emulate lazy evaluation with this construct

Example - lazy lists

CHALMERS

```
typedef struct Node *lazylist;

struct Node {
  int elem;
  lazylist() next;
}

lazylist cons(int x, lazylist() xs) {
  list res  = new Node;
  res->elem = x;
  res->next = xs;
  return res;
}

int sum(lazylist xs) {
  if (xs == (lazylist)null)
    return 0;
  else
    return xs->elem + sum(xs->next());
}
```

Example - lazy lists

CHALMERS

```
int main() {
  printInt(sum(take(42, enumFrom(1))));
  return 0;
}

lazylist enumFrom(int n) {
  lazylist rec() { return enumFrom(n + 1); }
  return cons(n, rec);
}

lazylist take(int n, lazylist xs) {
  if (xs == (lazylist)null)
    return xs;
  else if (n < 1)
    return (lazylist)null;
  else {
    lazylist rec() { return take(n - 1, xs->next()); }
    return cons(xs->elem, rec);
  }
}
```

Thunks

CHALMERS

- Call-by-name is a calling convention where the arguments are not evaluated until needed
- Thunks are used to implement call-by-name
- Thunks are essentially functions which take no arguments
- They are typically implemented as closures

Lazy evaluation

- The difference between call-by-name and lazy evaluation is that once an argument is evaluated, it is not reevaluated if it is used twice
- In order to achieve laziness, once the value is computed we need to remember it. This can be done in two ways:
  - Overwrite the thunk with an indirection pointing to the value
  - Overwrite the thunk with the value directly, if the space allocated for the thunk is big enough to hold the value

A Note

CHALMERS

- Call-by-name and lazy evaluation is very handy as they allow the programmer to create new control structures
- Be careful with combining them with side-effects: it can yield very surprising results
- An impure language with lazy evaluation as default is a bad idea

Haskell

Haskell is a language that supports laziness.

- Laziness requires pure semantics.
- Pure semantics encouraged various developments, including monads.
- Laxiness itself is probably not a virtue.

Haskell

Haskell is a language that supports laziness.

- Laziness requires pure semantics.
- Pure semantics encouraged various developments, including monads.
- Laxiness itself is probably not a virtue.
    - it interacts poorly with parallelism

Farewell for now

CHALMERS