Compiler Construction

Chapter 5 – Code Generation (2)

Prof. Dr. Jörg Kreiker

joerg.kreiker@informatik.hs-fulda.de

Fachbereich Angewandte Informatik
Hochschule Fulda – University of Applied Sciences

June 8, 2017

Agenda

Code generation for

1. Expressions
2. (Sequence of) Statements
3. Conditionals
4. Loops
5. Arrays
6. Records
7. Pointer
8. Functions
9. Whole programs

Pointer computation means

1. Creation, i.e. referencing locations by pointers to them
2. Dereference, i.e. accessing values of locations through pointers to them

- Application of the address operator & yields a pointer to a variable, i.e. the address:

$$\boxed{\mathrm{code}_R\ \&e\ \rho = \mathrm{code}_L\ e\ \rho}$$

- Application of the dereference operator `*e` yields the content of the cell whose address is the r-value of `e`

$$\boxed{\mathrm{code}_L\ *e\ \rho = \mathrm{code}_R\ e\ \rho}$$

Example

- Let $\rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7\}$
- Generate code for $e \equiv$ `((pt -> b) -> a)` in environment ρ

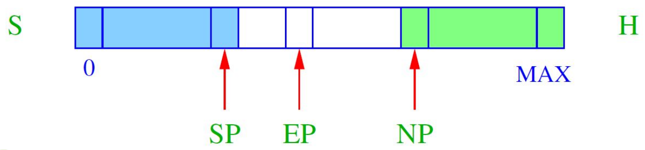
Types involved

```
struct t { int a[7]; struct t *b; };
int i,j;
struct t *pt;
```

The Heap

- Pointers allow access to anonymous, dynamically allocated data
- The lifetime of such data is not LIFO-based

⇒ We need a potentially arbitrarily large memory section H: the heap



NP New Pointer, points to lowest occupied memory cell

EP Extreme Pointer, points to the largest allowed value of the stack pointer within the current function frame

Heap and Stack Invariants

- Stack and heap must not overlap
- Overlap may occur whenever *SP* is increased
  $\rightarrow$ Stack Overflow
- Overlap may occur whenever *NP* is decreased
  $\rightarrow$ Out Of Memory
  - Can be captured by programmer, because malloc returns `NULL`
- *EP* simplifies overlap checks at function entry
- Checks at malloc still necessary

A call to `malloc` yields a pointer to a heap cell:

$$\text{code}_R\ \mathit{malloc}(e)\ \rho \quad = \quad \text{code}_R\ e\ \rho$$
$$\texttt{new}$$

where `new` replaces the topmost cell containing n (the number of cells to be allocated) by the updated NP (moved n cells lower)

Effect of `new`

```
if (NP - S[SP] <= EP)
    S[SP] = NULL;
else {
    NP = NP - S[SP];
    S[SP] = NP;
}
```

Issues

- Dynamically allocated storage needs to be freed
- Freed space may still be referenced by dangling pointers
- Fragmentation

$\Rightarrow$ Garbage Collection

Agenda

Code generation for

1. Expressions
2. (Sequence of) Statements
3. Conditionals
4. Loops
5. Arrays
6. Records
7. Pointer
8. Functions
9. Whole programs

Function definition consists of

- name
- formal parameters
- return type
- body

$$\boxed{\text{code}_R\ f\ \rho \quad = \quad \texttt{loadc}\ \_f}$$

- Function names must get an address just like other names

Instances

Hochschule Fulda
University of Applied Sciences

- During runtime, many instances of the same function may be active
- Called but not returned
- Formal parameters and local variables (function variables) of different instances must be distinguished

Storage of Function Variables

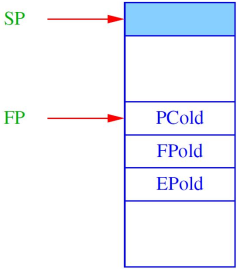
- Create a special section for each function call
- Manage these sections on the stack
- Each instance of a function gets its own private section on the stack

$\Rightarrow$ Stack Frames

Stack Frame Organization

Hochschule Fulda
University of Applied Sciences

- Frame Pointer *FP* points to the last management cell; all addresses relative to *FP*
- local variables: addresses $+1$, $+2$, ...
- formal parameters: addresses -3, -4, ... (below management cells)
- management cells: store old register values for function return



Address Environment

Hochschule Fulda
University of Applied Sciences

Some Variables

```
void f(int v, int w) {
    int a;
    if (a > 0) {
        int b;
    } else {
        int c;
    }
}
```

Environment

ν	$\rho(\nu)$
v	-3
w	-4
a	1
b	2
c	2

- Actual arguments are evaluated right-to-left
- For function $\tau\ f(\tau_1\ x_1, \ldots, \tau_k\ x_k)$

$$x_1 \mapsto -2- |\tau_1| \qquad x_i \mapsto -2- |\tau_1| - \ldots - |\tau_i|$$

f Caller $\quad g$ Callee

1.	Determine actual parameters		in f
2.	Save FP, EP	`mark`	in f
3.	Determine start address of g		in f
4.	Set new FP	`call`	in f
5.	Save PC and Jump to start of g	`call`	in f
6.	Set new EP	`enter`	in g
7.	Allocate local variables	`alloc`	in g

Function Return

f Caller $\qquad$ g Callee

0. Computation of return value
1. Store return value
2. Restore FP, EP, SP
3. Jump back to code of f, i.e. Restore PC

} `return`

4. Cleanup stack } `slide`

$$
\begin{array}{lll}
\mathrm{code}_R\ g(e_1, \ldots, e_n)\ \rho & = & \mathrm{code}_R\ e_{n-1}\ \rho \\
 & & \ldots \\
 & & \mathrm{code}_R\ e_1\ \rho \\
 & & \texttt{mark} \\
 & & \texttt{loadc}\ \rho(f) \\
 & & \texttt{call} \\
 & & \texttt{slide}\ (s-1)
\end{array}
$$

- Call-by-value: R-value of actual parameters
- $\rho(f)$ is the location, where the code for `f` starts in the instruction store

Consider the function definition

$$fd \equiv \tau\ f(specs)\{body\}$$

$$\begin{array}{lllll}
\text{code } fd\ \rho & = & & & \\
 & \texttt{_f:} & \texttt{enter q} & // & \text{set EP} \\
 & & \texttt{alloc k} & // & \text{init local variables} \\
 & & \text{code } body\ \rho_f & & \\
 & & \texttt{return} & // & \text{leave function}
\end{array}$$

$$\begin{array}{llll}
 & \texttt{q} & = & max + \texttt{k} \\
 & max & = & \text{maximal length of local stack} \\
\text{where} & \texttt{k} & = & \text{space for local variables} \\
 & \rho_f & = & \text{address environment for } \texttt{f} \\
 & & & \text{based on } \rho,\ specs\text{, and } body
\end{array}$$

Variables within Stack Frames

Hochschule Fulda
University of Applied Sciences

- Access to local variables or formal parameters happen relative to the current `FP`.
- Need to modify computation of L-values
- Let $\rho(x) = j$ then

$$\boxed{\mathrm{code}_L\ x\ \rho = \texttt{loadrc j}}$$

- `loadrc j` computes the sum of `FP` and `j`
  - `SP++;`
  - `S[SP] = FP + j;`

Agenda

Hochschule Fulda
University of Applied Sciences

Code generation for

1. Expressions
2. (Sequence of) Statements
3. Conditionals
4. Loops
5. Arrays
6. Records
7. Pointer
8. Functions
9. Whole programs

Whole Programs

- Before program execution
  - $\mathtt{SP} = -1$
  - $\mathtt{FP} = \mathtt{EP} = 0$
  - $\mathtt{PC} = 0$
  - $\mathtt{NP} = \mathrm{MAX}$
- Let `p` be the program consisting of function definitions $F_1, \ldots, F_n$, one of which is called `main`
- The code for `p` consists of
  - Code for F_i
  - Code for the call to `main`
  - The `halt` instruction

Code Generation Programs

$$\text{code } p\ \emptyset \quad = \quad$$

```
        enter 4
        alloc 1
        mark
        loadc _main
        call
        halt
_f1:    code F1 ∅
          ⋮
_fn:    code Fn ∅
```

- We assume, that `main` returns exactly one value
- This value lies on top of the stack after program execution