FP8-17: Episode 3, Feb 19, 2007

Environments (Symbol Tables)

- Type and Value Environments

Types and Type Analysis

- Typechecking expressions and variables
- Pointer Dereference
- Integer Type Promotion

Activation Records

- Parameter Passing, Return Value

Calling Conventions of TMS320C6xxx

- Register Conventions
- Parameter Passing, Return Value
- Caller/Callee Perspective
- Accessing Variables and Arguments

Environments

Handling several meanings of the same identifier

```
0 void f(int a, int b, int c) {
1    int j = a + b;
2    printf("%d", a+c);
3    {
4       const char * a = "hello";
5       printf("%s", a);
6       printf("%d", j);
7    }
8    printf("%d", b);
9 }
```

Two a vars: **a** hides **a** in the inner block.
Compilers use environments (symbol tables) to represent the semantic meaning of a given syntactic symbol at a given program point.

Value Environments

```
0 void f(int a, int b, int c) {
1    int j = a + b;
2    printf("%d", a+c);
3    {
4       const char * a = "hello";
5       printf("%s", a);
6       printf("%d", j);
7    }
8    printf("%d", b);
9 }
```

$\sigma_0 = [a \mapsto 1, b \mapsto 2, c \mapsto 3]$
$\sigma_1 = \sigma_0 \dagger [j \mapsto 3]$
$\sigma_2 = \sigma_3 = \sigma_1$
$\sigma_4 = \sigma_3 \dagger [a \mapsto \text{"hello"}]$
$\sigma_5 = \sigma_6 = \sigma_7 = \sigma_4$
$\sigma_8 = \sigma_9 = \sigma_2$

The $\dagger$ operator overrides previously defined value

Type Environments

```
0 void f(int a, int b, int c) {
1    int j = a + b;
2    printf("%d", a+c);
3    {
4       const char * a = "hello";
5       printf("%s", a);
6       printf("%d", j);
7    }
8    printf("%d", b);
9 }
```

$\sigma_0 = [a \mapsto \text{int}, b \mapsto \text{int}, c \mapsto \text{int}]$
$\sigma_1 = \sigma_0 \dagger [j \mapsto \text{int}];$
$\sigma_2 = \sigma_3 = \sigma_1$
$\sigma_4 = \sigma_3 \dagger [a \mapsto \text{const char *}]$
$\sigma_5 = \sigma_6 = \sigma_7 = \sigma_4$
$\sigma_8 = \sigma_9 = \sigma_2$

Multiple Symbol Tables

- Environments store info about identifiers
- Usually created on the fly, while traversing the abstract syntax tree.
  - Scope is entered: add local symbols
  - Scope is left: remove local symbols, restore the ones belonging to outer scope.
- Typically implemented using hash tables.
- More info in Appel, section 5.1.

Type Systems and Type Checking

- Type system provides an abstraction of program execution that ensures absence of some errors.
- Type checking is fast and can be done statically, at compile time, without executing the program.
- Typically type systems detect type mismatch errors like an attempt to multiply a number by a string constant, or calling a function with an unappropriate number of parameters.
- A type checker performs a traversal over an abstract syntax tree checking and inferring types of all expressions and objects (variables and functions in case of C).

Type Checking (II)

Type checking of expression $e_1 + e_2$:

- $t_1 \leftarrow$ type check e_1
- $t_2 \leftarrow$ type check e_2
- ensure that both types are int: $t_1 = t_2 = \text{int}$
- return int as the type of the entire expression.

Type Checking (III)

```
struct expty tycheckExp (S_table venv, S_table tenv, A_exp a) {
  switch (a->kind) {
    ⋮
    case A_opExp: {
      A_oper oper = a->u.op.oper;
      struct expty left = tycheckExp(venv,tenv,a->u.op.left);
      struct expty right = tycheckExp(venv,tenv,a->u.op.right);
      if (oper==A_plusOp) {
         if (left.ty->kind!=Ty_int)
            EM_error(a->u.op.left->pos, "integer required");
         if (right.ty->kind!=Ty_int)
            EM_error(a->u.op.right->pos, "integer required");
         return expTy(NULL,Ty_Int());
    ⋮
```

Appel, p. 117 top

Typechecking variables

- The typeckecker constructs a type environment, storing types for all declared variable.
- For each variable used the typechecker checks, whether it has been declared in the current environment.
- If it was not: an “undeclared variable” error is reported.
- If it was, the type is propagated further into the typechecking algorithm

Typechecking variables (II)

```
struct expty tycheckVar(S_table venv, S_table tenv, A_var v) {
   switch(v->kind) {
     case A_simpleVar: {
        E_enventry x = S_look(venv,v->u.simple);
        if (x && x->kind==E_varEntry)
           return expTy(NULL, actual_ty(x->u.var.ty));
        else {EM_error(v->pos, "undefined variable %s",
               return expTy(NULL,Ty_int());}
     }
     ⋮
```

source: Appel, p. 117 bottom

Typechecking pointer dereference

Corresponds to removing the pointer star from the type:

- Only pointers can be dereferenced, so first check whether dereferenced expression has a pointer type.
- If so, then return the type of the expression without the indirection.
- Otherwise report a type error.

Typechecking pointer dereference (II)

```
Type deref(Type ty) {
      if (isptr(ty))
         ty = ty->type;
      else
           error("type error: %s\n", "pointer expected");
      return isenum(ty) ? unqual(ty)->type : ty;
}
```

source: lcc CVS, types.c, rev.1.1 as of 20050410

Integer Type Promotion

An excerpt implementing integer type promotion:

```
Type promote(Type ty) {
      ty = unqual(ty);
      switch (ty->op) {
      case ENUM: return inttype;
      case INT:
        if (ty->size < inttype->size) return inttype;
        break;
      case UNSIGNED:
        if (ty->size < inttype->size) return inttype;
           if (ty->size < unsignedtype->size)
              return unsignedtype;
        break;
      case FLOAT:
        if (ty->size < doubletype->size)
           return doubletype;
      }
      return ty;
}
```

source: lcc CVS, types.c, rev.1.1 as of 20050410

Typecheckers: odds & ends

- A complete typechecker has also rules for: other operators, type casts, arrays, subscripting, field access, variable and function declarations, type declarations, address operator, ...
- See more in Appel, sections 5.2–5.4.
- Modern languages have sophisticated type systems (object-oriented, functional)
- Some are equipped with a type inference that allows omitting type annotations. Types are inferred automatically from the context

C type system: warning!

- **Beware:** The type system of C is very weak. It does not protect you from many typical programming errors. By means of type casts/pointer arithmetic you can interpret any value under any arbitrary type. This typically leads to unpredictable program behavior. Alas no compile time error is generated.
- Safer languages include: Java, C#, Standard ML,... so only use C when the application requires it.

Activation Records

Also known as Stack Frames

- In C functions have local variables
- Function invocations existing at the same time, each of them has an own copy of local variables

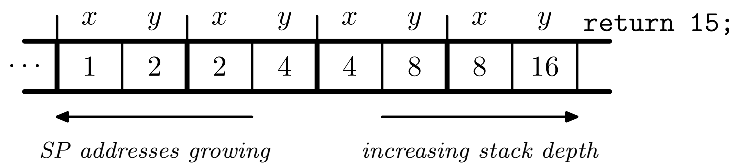
```
int f(int x) {
      int y = x + x;
      if (y < 10) return f(y);
      else return y-1; }
```

- New copies of y, x created at recursive calls to *f*
- They are destroyed when *f* returns
- A stack can be used to store local variables
- The same stack maintains return addresses
- Often parameters are also passed on the stack

Consider the following recursive function:

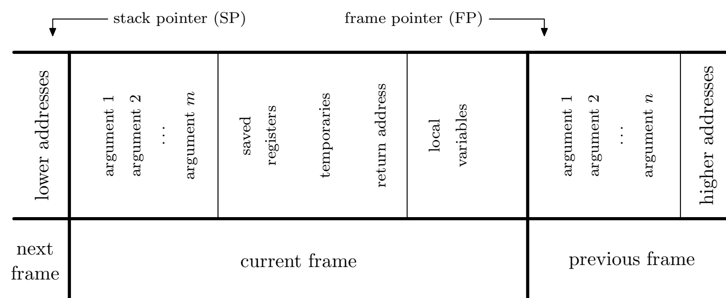
```
int f(int x) {
    int y = x + x;
    if (y < 10) return f(y);
    else return y-1;
}
```

The (abstract) stack during the evaluation of $f(1)$:



- *caller* — a function calling another function.
- *callee* — the function being called.
- In the example of our previous slide *f* was both the caller and the callee.
- This is usual for recursive functions.

- Often the stack grows from higher to lower addresses (pushing decreases the top pointer)
- For this reason the sides are now reversed:



source: Appel, p.128

- Stack pointer (SP) points to the first empty place in the stack
- Frame pointer (FP) points to the beginning of current stack frame.
- Current parameter values belong to the previous stack frame.
- Local variables can be addressed with negative offsets from FP
- Arguments can be addressed with positive offsets from FP.

- Modern architectures have many registers and as many variables, arguments, etc is allocated in the registers.
- Stack still remains the main resort in case of:
  - memory spilling (if there is not enough registers)
  - storing temporary register values (in order to prepare a stack frame for the next call)
  - some expressions take address of variables or arguments (one cannot take addresses of registers).
  - structures are passed.

Parameter Passing

- Nowadays parameters are passed in registers
- Faster than via stack, as only for non-leaf calls values need to be spilled to stack
- Even then, spilling is not always needed. Dead values do not have to be preserved. Register windows can be switched.
- Also values needed may already reside in necessary registers (due to interprocedural register allocation)
- If arguments need to be passed on the stack, then they are allocated by the caller and stored in the caller's stack frame

Return Value & Return Address

- Return value is most often left in a register
- After its body is executed, callee performs an indirect jump to the value hold in the return address register
- For non leaf calls the return address is stored on the stack

Register conventions TMS320C6xxx

A selection

- A4, A5: the first argument or/and the return value (A5 used for double, long, long long)
- Odd register contains the sign, the exponent, and the most significant part of the mantissa
- The even register contains the rest
- Remaining arguments (2–10) in similar manner: B4 (B5), A6 (A7), B6 (B7), A8 (A9), B8 (B9), A10 (A11), B10 (B11), A12 (A13), B12 (B13)
- A15: frame pointer (FP), B3: return address, B14: data pointer (DP), B15: stack pointer (SP), points to the next free location

Details: p.8-18, spru187

Parameter passing in TMS3206xxx

Examples:

```
int func1(int a,int b,int c);
A4        A4    B4    A6

int func2(int a,float b,int *c,struct A d,float e,int f,int g);
A4        A4    B4      A6     B6         A8      B8    A10

int func3(int a,double b,float c,long double d);
A4        A4    B5:B4    A6      B7:B6

struct A func4(int y);
A3             A4
```

Discuss this in exercise session.

source: p.8-20, spru187

Calling Conventions in TMS320C6xxx

Caller's tasks when calling the *callee*:

- Place arguments passed in registers (or stack)
- Arguments placed on the stack must be aligned to a value appropriate for their size
- Arguments not declared in prototypes whose size is less than the size of int passed as int
- Undeclared floats passed as double
- Structures passed as address
- It is up to the callee to make a local copy
- Save registers A0–A9 and B0–B9 on the stack if their values needed after the call
- Call the callee. Upon returning, reclaim stack space by increasing the stack pointer

Calling Conventions in TMS320C6xxx

Callee's tasks:

- Allocate space on the stack for local variables, temporaries, arguments to functions to be called (occurs once at the beginning)
- The frame pointer is used to read arguments from the stack and to handle register spilling
- If any arguments are placed on the stack or if the frame size exceeds 128K bytes:
  - Save the old FP (A15) on the stack
  - Set A15 to current SP (B15)
  - Allocate the frame (decrease SP by a constant)
  - Neither A15 nor B15 is decreased anywhere else
- Otherwise allocate the frame by subtracting a constant from B15

- If the callee makes any calls, the return address is saved on the stack
- Otherwise the address is left in the return register(B3) to be overwritten by the next call
- If the callee modifies any registers numbered A10 to A15 or B10 to B15, it must save them, either in other registers or on the stack.
- The callee can modify any other registers without saving them.

- If the callee expects a structure argument, it receives a pointer to the structure instead.
- If writes are made to the structure from within the callee, space for a local copy of the structure must be allocated on the stack and the local structure must be copied from the passed pointer to the structure.
- If no writes are made to the structure, it can be referenced in the callee indirectly through the pointer argument.
- The called function executes the code for the function.

- If the callee returns any integer, pointer, or float type, the return value is placed in A4.
- If the callee returns a double, long double, long, or long long type, the value is placed in A5:A4.
- If the callee returns a structure, the caller allocates space for the structure and passes the address of this space to the callee in A3.
- To return a structure, the callee copies the structure to the memory block pointed to by the extra argument (A3).

Finally

- Any register numbered A10 to A15 or B10 to B15 that was saved earlier is restored.
- If A15 was used as a frame pointer (FP), the old value of A15 is restored from the stack. The space allocated for the function is reclaimed by adding a constant to B15 (SP).
- The function returns by jumping to the value of the return register (B3) or the saved value of the return register.

Accessing Arguments and Local Variables on TMS320C6xxx

- Stack arguments and local nonregister variables are accessed indirectly through register A15 (FP) or through register B15 (SP)
- The stack grows toward smaller addresses, so the local and argument data for a function are accessed with a positive offset from FP or SP.
- Local variables, temporary storage, and the area reserved for stack arguments to functions called by this function are accessed with offsets smaller than the constant subtracted from FP or SP at the beginning of the function.

- Stack arguments passed to this function are accessed with offsets greater than or equal to the constant subtracted from register FP or SP at the beginning of the function.
- The compiler attempts to keep register arguments in their original registers if optimization is used or if they are defined with the register keyword.
- Otherwise, the arguments are copied to the stack to free those registers for further allocation.