Software Programmable DSP Platform Analysis

Episode 2, Tuesday 12 April 2005

Environments

Handling several meanings of the same identifier

- The same identifiers may have various meanings in the same program:

```
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 }
```

- There are two different variables a above. The blue one hides the value of the red one in the inner block.
- Interpreters and compilers use environments (also known as 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 }
```

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

The $\dagger$ operator overrides previously defined value (for example in σ_3).

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 }
```

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

Multiple Symbol Tables

- Similar environments can be constructed to store other information about identifiers.
- Symbol tables are usually created on the fly, while traversing the abstract syntax tree.
- When a scope is entered:
  - the information about new local symbols is added into the environment (imperative style) overwriting the previously stored information,
  - or a new environment is created wich contains the sum of the old table and a new table (functional style).
- When a scope is left
  - local changes in the environment are reversed (imperative style)
  - or an old symbol table is simply restored (functional style).
- Symbol tables are also used to rename all program variables so that no conflicts (hiding appear).
- This is usually achieved by assigning integer identifiers instead of symbolic names.
- Symbol tables are 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 entire 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 of expression $e_1 + e_2$ consists of:

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

```
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 typecheking algorithm

```
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.

```
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

In reality type checking for C is much more complicated than for the tiger language of Appel's. An excerpt implementing integer type promotion in lcc:

```
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

- A complete typechecker has many similar rules, also 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 object-oriented or functional type systems.
- Some are equipped with a type inference algorithm that allows omitting type annotations. Types are inferred automatically from the program text.
- **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
- Even if several invocations of a function exist at the same time, each of them enjoys a private copy of the variable

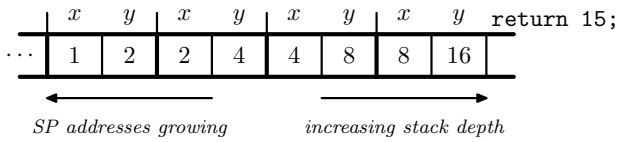
```
int f(int x) {
    int y = x + x;
    if (y < 10) return f(y);
    else return y-1;
}
```

- A new copy of y (and x) is created at each recursive call to *f*.
- This copy is destroyed as soon as *f* returns.
- Thus naturally a stack can be used to store local variables.
- Incidentally the same stack will maintain return addresses for each function call.
- Sometimes function parameters are also passed on the stack.

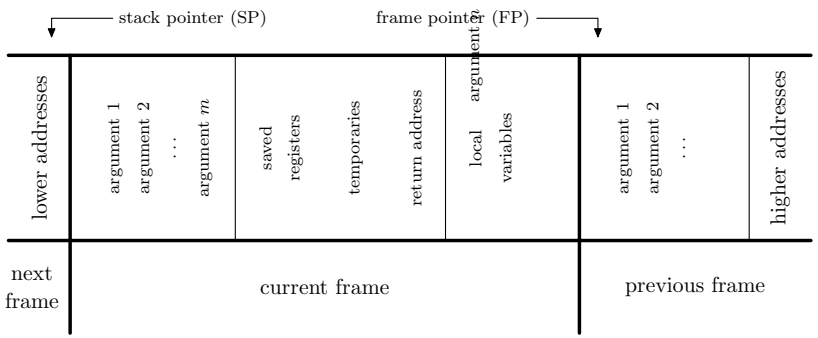
Activation Records (II)

```
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 for $x = 1$:



- *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.
- In reality the stack grows from higher addresses to lower addresses (so pushing on the stack decreases the address of the stack top).
- For this reason the sides are now reversed on the figure below:



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

- In modern architectures parameters are mostly passed in registers.
- This is faster than passing via stack, as only for non-leaf calls values need to be spilled to memory (stack).
- Even then, spilling is not only needed. Dead values do not have to be preserved. Some architectures provide switchable register windows.
- 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

- Similarly return value is most often left in the register.
- Only stored in the stack for non-leaf calls by the callee.
- After its body is executed, callee performs an indirect jump to the value hold in the return register.

Register conventions TMS320C6xxx (selection)

- A4, A5: the first argument or/and the return value (A5 used for double, long, long long)
- The odd register contains the sign bit, the exponent, and the most significant part of the mantissa.
- The even register contains the least significant part of the mantissa.
- 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
```

source: p.8-20, spru187

Calling Conventions in TMS320C6xxx [Caller]

Caller performs the following tasks when it calls the *callee*:

- Arguments passed are placed in registers or on the stack.
- Arguments placed on the stack must be aligned to a value appropriate for their size.
- An argument that is not declared in a prototype and whose size is less than the size of int is passed as an int.
- An argument that is a float is passed as double if it has no prototype declared.
- A structure argument is passed as the address of the structure.
- It is up to the callee to make a local copy.
- The caller must save registers A0 to A9 and B0 to B9 (and A16 to A31 and B16 to B31 if present), if their values are needed after the call by pushing the values onto the stack.
- The caller calls the callee.
- Upon returning, the caller reclaims any stack space needed for arguments by adding to the stack pointer.

Calling Conventions in TMS320C6xxx [Callee]

Callee performs the following tasks:

- Allocates space on the stack for local variables, temporaries, and arguments to functions that it may call.
- This allocation occurs once at the beginning of the function and may include the allocation of the frame pointer (FP).
- The frame pointer is used to read arguments from the stack and to handle register spilling instructions.
- If any arguments are placed on the stack or if the frame size exceeds 128K bytes, the frame pointer (A15) is allocated in the following manner:
  - The old A15 is saved on the stack.
  - The new frame pointer is set to the current SP (B15).
  - The frame is allocated by decrementing SP by a constant.
  - Neither A15 (FP) nor B15 (SP) is decremented anywhere else within this function.
- Otherwise the frame is allocated by subtracting a constant from register B15 (SP). Register B15 (SP) is not decremented anywhere else within this function.

Calling Conventions in TMS320C6xxx [Callee]
Continued

- If the callee makes any calls, the return address is saved on the stack.
- Otherwise leave the address 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.

Calling Conventions in TMS320C6xxx [Callee]
Continued

- 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 the 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).
- 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 in step 1 is reclaimed at the end of the function by adding a constant to register 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)
- Since the stack grows toward smaller addresses, 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.