Compiler Construction

Sibylle Schupp[1]

[1]Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

SoSe16

Lecture 9

TUHH STS Software Technology Systems

Outline

Imperative and object-oriented programming

Stateful programming

Paradigm-specific compilation issues

- Syntactic constructs: nothing paradigm-specific
- Semantic analysis
  - Polymorphism
- Code generation
  - Control flow
- Block-structured languages

Outline

Overloading

Overloading is a property of function and operator names. It means that a name can have different types in different contexts.

- Ex.: In most languages, arithmetic and comparison operators are overloaded.
  - `+` could refer to integer addition, floating-point addition, or string concatenation. It could be overloaded in the argument and return types.
  - `==` is also overloaded, but usually only in the argument type.
- Typing rules for overloaded identifiers contain type *variables* along with constraints on those type variables.
- The process of determining the type of an overloaded identifier is called *overload resolution*.

Example (overloading)

$$\frac{\Gamma \vdash a : t \qquad \Gamma \vdash b : t}{\Gamma \vdash a + b : t} \text{ IF T IS INT, DOUBLE, STRING}$$

The implementation of the inference rule exploits that the two operands must be of the same type:

```
infer (a+b):
    t := infer(a)
    error if t not int,double, string
    check (b,t)
    return t
```

Type conversion

Type conversion means that a type can be changed into another type. Most languages support some kind of type conversion. The target of a conversion must be defined.

- *Explicit* conversion is done by the user, at source-code level.
  - The source language must provide a special command or notation.
  - Ex. `(int)`, `static_cast<>`
- *Implicit* conversion is done by the compiler.
  - Ex. a `C` compiler implicitly converts `unsigned int` to `int`
- Conversions allowing typing to succeed that otherwise would fail.

Type conversion, internally

Mathematically, converting to and back results in the identity operation. At implementation level, the same does not hold true: internally, every type has its own length and binary representation.

- Ex. char vs. `uint`, int vs. double
- Type conversion is therefore also called *coercion*.
- For built-in types, the language specification defines the admissible conversions.
- A conversion is *safe* if it does not incur loss of information.
  - Converting from smaller to larger types is considered safe.
  - Despite popular belief that is not always true.
- A type checker holds internally a list of ordered types. Ex. for C (where $<$ means: "not smaller than")

$$\text{int} < \text{long} < \text{long long}$$

Example (conversion rule)

Provided the following inference rule

$$\frac{\Gamma \vdash a : t \qquad \Gamma \vdash b : u}{\Gamma \vdash a + b : max(t, u)} \text{ IF T,U IN INT, DOUBLE, STRING}$$

Further assume the order

$$\text{int} < \text{double} < \text{string}$$

Then, `5+''you''` can be typed (and has type `string`).

- Accordingly, it will evaluate to "5you."

In-class exercise

Can the following expression be typed?

In-class exercise (cont'd)

Outline

Control-flow statements

- Imperative constructs
  - Structured: selection (if-then-else) and repetition (for, while, repeat)
  - Unstructured: unconditional jump, conditional jump (goto)
- Object-oriented constructs
  - Iterators
  - Dynamic dispatch

Example (selection statement in LLVM)

http://llvm.org/docs/tutorial/LangImpl5.html#id6

```
declare double @foo()
declare double @bar()

define double @baz(double %x) {
entry:
  %ifcond = fcmp one double %x, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:       ; preds = %entry
  %calltmp = call double @foo()
  br label %ifcont

else:       ; preds = %entry
  %calltmp1 = call double @bar()
  br label %ifcont

ifcont:     ; preds = %else, %then
  %iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else ]
  ret double %iftmp
}
```

Issues in code generation

- SSA form, ϕ insertion
- Target labelling, forward references, 2-pass

```
  %3 = icmp sgt i32 %2, 1
  br i1 %3, label %4, label %7

; <label>:4
  %5 = load i32* %n, align 4
  %6 = sub nsw i32 %5, 1
  ...
  br label %10

; <label>:7
  %8 = load i32* %n, align 4
  %9 = add nsw i32 %8, 1
  ...
  br label %10

; <label>:10
```

Types

http://llvm.org/docs/ProgrammersManual.html

```
Value *IfExprAST::codegen() {...}
```

- Assumption: AST node `IfExprAST`
  - See grammar (otherwise: extend parser appropriately)
- Types (LLVM)
  - Evaluate test condition $\rightsquigarrow$ `Value`
  - Evaluate other expressions (each codegen() returns a value) $\rightsquigarrow$ `Value`, http://llvm.org/docs/doxygen/html/classllvm_1_1Value.html
  - Subclasses:
    - `BasicBlock`
    - `PHINode`
    - `Function` http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html

Code generation

http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html

Code generation

- Instructions $\rightsquigarrow$ `IRBuilder`
  - test $\rightsquigarrow$ member function `CreateFCmpONE`
  - branch $\rightsquigarrow$ member function `CreateCondBr`
  - unconditional branch $\rightsquigarrow$ member function `CreateBr`
- Recursive code generation
- Basic block generation:
  - Static method `BasicBlock::Create`
    http://llvm.org/docs/doxygen/html/classllvm_1_1BasicBlock.html

Composing

http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html

Composition (`IRBuilder<> Builder;`)

- Get enclosing function ⇝ `Builder.GetInsertBlock`
- Get insertion point within basic block
  - ⇝ `Builder.GetInsertPoint`
    http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilderBase.html
- Append new basic block ⇝
  `TheFunction->getBasicBlockList().push_back`

Kaleidoscope (1/4)

http://llvm.org/docs/tutorial/LangImpl5.html#id7

```
Function *TheFunction = Builder.GetInsertBlock()->getParent();

// Create blocks for the then and else cases.  Insert the 'then'
// block at the end of the function.
BasicBlock *ThenBB =
    BasicBlock::Create(getGlobalContext(), "then", TheFunction);
BasicBlock *ElseBB =
    BasicBlock::Create(getGlobalContext(), "else");
BasicBlock *MergeBB =
   BasicBlock::Create(getGlobalContext(), "ifcont");
Builder.CreateCondBr(CondV, ThenBB, ElseBB);
```

- Get the current function
- Create (empty) blocks, hook up “then” block
- Create conditional block, but do not yet insert it

Kaleidoscope (2/4)

```
// Emit then value.
Builder.SetInsertPoint(ThenBB);

Value *ThenV = Then->codegen();
if (!ThenV)
  return nullptr;

Builder.CreateBr(MergeBB);
// Codegen of 'Then' can change the current block,
// update ThenBB for the PHI.
ThenBB = Builder.GetInsertBlock();
```

- Set (end of) then-block as insertion point.
- Call codegen recursively, and insert generator statements.
- Generate break generator statement and insert it as last statement.
- Reset then-block as current block

Kaleidoscope (3/4)

```
// Emit else block.
TheFunction->getBasicBlockList().push_back(ElseBB);
Builder.SetInsertPoint(ElseBB);

Value *ElseV = Else->codegen();
if (!ElseV)
  return nullptr;

Builder.CreateBr(MergeBB);
// codegen of 'Else' can change the current block,
// update ElseBB for the PHI.
ElseBB = Builder.GetInsertBlock();
```

- Add new basic block, set insertion point
- Call codegen recursively, and insert generator statements.
- Generate break generator statement and insert it as last statement.
- Reset else-block as current block

Kaleidoscope (4/4)

```
  // Emit merge block.
  TheFunction->getBasicBlockList().push_back(MergeBB);
  Builder.SetInsertPoint(MergeBB);
  PHINode *PN =
     Builder.CreatePHI(Type::getDoubleTy(getGlobalContext()),
                       2, "iftmp");

  PN->addIncoming(ThenV, ThenBB);
  PN->addIncoming(ElseV, ElseBB);
  return PN;
}
```

- Add new basic block, set it as insertion point
- Create PHI node and connect it

Outline

STS Software Technology Systems TUHH

What is run-time support?

The actual execution of a program requires support by a run-time system.

Issues at run time

- How are data types represented in memory?
- How are variables allocated?
- How are procedures invoked?
- How are parameters passed?
- How can code be shared?

The code generator needs to adhere to the conventions by the run-time system and generate code accordingly, e.g., for allocation variables.

The design of a run-time system varies signifiantly among the different paradigms. We focus on block-structured languages.

Overview

For *block-structured* languages, memory is organized in different areas, at run time:

- Instructions
- Data objects
- Run-time stack

When information is statically known, parts of the organization can be done by the compiler. Even small portions of static information are considered:

- Value (of a constant)
- Size (of a type)
- Relative offset

Data representation (simple types)

The smallest addressable unit is a *byte*. Objects of simple types are directly described as a contiguous sequence of *n*-bytes

- Character types are 1 byte (Ascii); for Unicode longer (utf-8, utf-32)
- In the language C, there are several integer types: char, short, int, and long. The sizes are specified relative to another:
  - sizeof(short) $\leq$ sizeof(int) $\leq$ long
  - *char* must be at least 8 bits, *short, int* at least 16 bits, *long* at least 32 bits

A compiler typically *aligns* data objects, e.g., at addresses divisible by 4. Motivation:

- Efficiency
- Atomicity

Data representation (aggregate types)

Aggregate types typically occupy contiguous memory. The layout depends on the composing types, alignment, and order.

- Clear: the compile-time representation includes only data of fixed length (including pointers).
- The *offset* is a relative address, which denotes the difference between address of the object and the address of a sub-object.
- If sub-objects need to be aligned, *padding* is necessary: the bytes to the next boundaries go unused.
- If no padding is used, the representation is called *packed*.
- Order: In the language C, characters of a string and fields of a struct are laid out in the order in which they are declared.

Example (data layout of C++ classes)

http://llvm.org/docs/LangRef.html#data-layout

```
class myclass1 {
  double x;
  double y;
};

> clang -c dataLayout.cc -Xclang -fdump-record-layouts
*** Dumping AST Record Layout
   0 | class myclass1
   0 |   double x
   8 |   double y
     | [sizeof=16, dsize=16, align=8
     |  nvsize=16, nvalign=8]
```

Another example (C++ classes)

https://mentorembedded.github.io/cxx-abi/

```
class myclass1 {
  double x;
  double y;
};
```

```
class myclass2 {
public:
    int f(int p) {
        return n+p;
    }
private:
    int n, m;
    myclass1 m1;
    char c;
};
```

```
*** Dumping AST Record Layout
   0 | class myclass2
   0 |   int n
   4 |   int m
   8 |   class myclass1 m1
   8 |     double x
  16 |     double y
     |   [sizeof=16, dsize=16, align=8
     |    nvsize=16, nvalign=8]
  24 |   char c
     | [sizeof=32, dsize=25, align=8
     |  nvsize=25, nvalign=8]
```

In-class exercise (data layout)

In-class exercise (data layout)

Another example: arrays

Matrices can be implemented as arrays of arrays. Two options exists for their layout:

- Row-major (sequence of rows)
- Column-major (sequence of columns)

The address calculations depend on the layout. Let $M[n, m]$ be a $n \times m$ matrix in row-major form.

- The address of the element $M(i, j)$
  $\text{base} + (i - 1) \times m \times el + j \times el$
- Rewriting yields:
  $(\text{base} + (m - 1) \times el) + i \times el + j \times el$
- The first term can be computed at compile time.

Outline

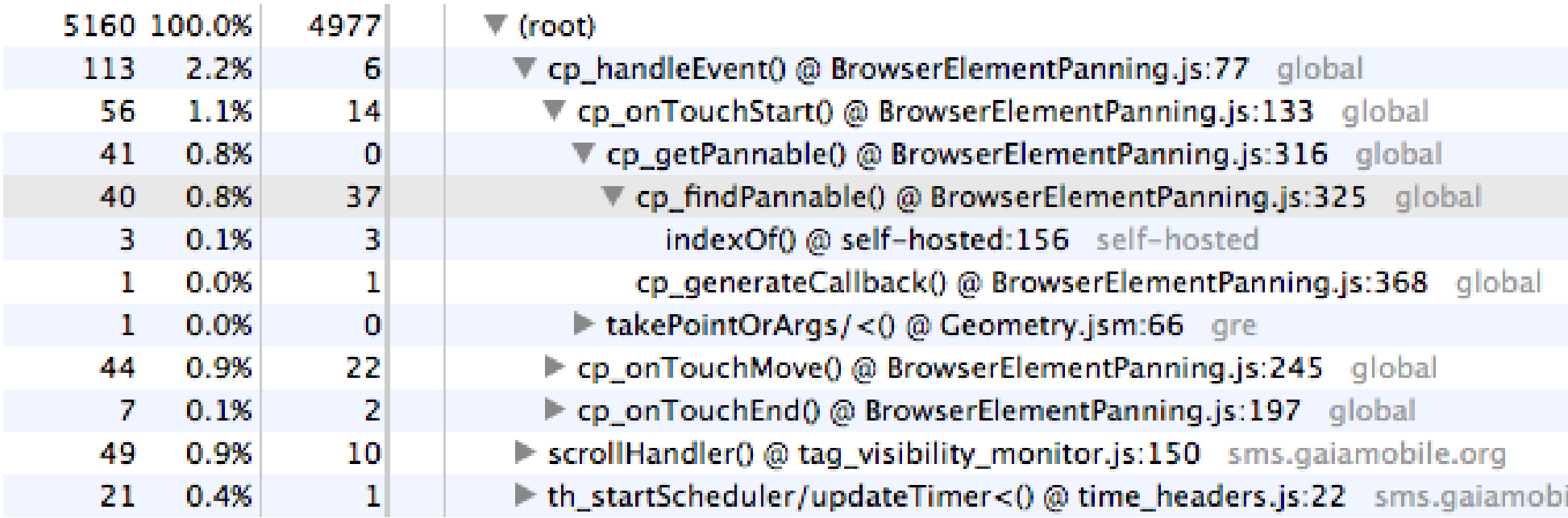
Activation of procedures

We distinguish between the definition of a procedure (static) and its *activation* (dynamic).

- The *lifetime* of an activation of a procedure comprises all steps for executing this procedure and all activations of its subroutines.
- In block-structured languages, two activations are either nested or disjoint.
- The control flow of a whole program can be represented by the *call stack (control stack)*:
  - Upon activation, a node is pushed onto a stack. When the routines exits, the node is popped.
  - For the top node, the sequence of elements on the stack represents the call sequence in the program.

Example (call stack)

https://mdn.mozillademos.org/files/7271/callstack.png



- Developers study the call stack sometimes during debugging.

STS Software Technology Systems TUHH

Recursive functions

A function is *recursive* if its activation contains an activation of itself.

- At source code level, the function might call itself directly or might be called by one of its callees (mutual recursion).

For the compilation of recursive functions, two issues arise:

- The size of the control stack is not statically known.
- Different activations of the same function need to be distinguished:
  - "Activation record" needed

Activation records

An *activation record* (frame) is a record that stores all information needed for a particular activation of a routine to run it, to interrupt it for a call to a subroutine, and to restart when the subroutine returns.

- Internal values (registers)
- Local objects
- Temporaries
- Parameters
- Return values
- Data links

The layout for all ARs of a routine is the same and the size of each entry is statically known. The size of a particular AR, however, varies.

Example (layout activation record)

http://www.x86-64.org/documentation/abi.pdf

Position	Contents	Frame
`8n+16(%rbp)`	memory argument eightbyte n	Previous
	...	
`16(%rbp)`	memory argument eightbyte 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	unspecified	
	...	
`0(%rsp)`	variable size	
`-128(%rsp)`	red zone	

3.2.2 The Stack Frame

In addition to registers, each function has a frame on the run-time stack. This stack grows downwards from high addresses. Figure 3.3 shows the stack organization.

Registers

Registers provide the fastest access to data. A compiler therefore tries to store as much information of the AR as possible in registers (and adjusts the AR design accordingly)

Standard candidates for registers

- Stack pointer
- Frame pointer
- Return address
- Parameters

x86-64

- %rax, %rbx, %rcx, %rdx, %rbp, %rsp, %rsi, %rdi
  + %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15
- First six parameters are passed in registers

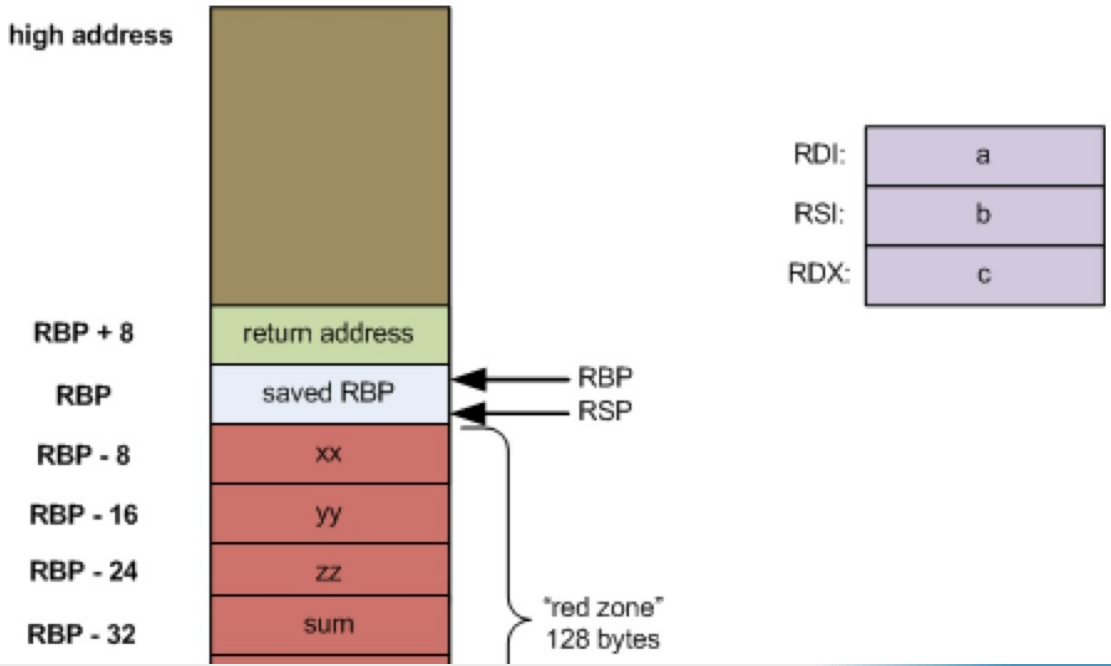
Example (layout activation record)

http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64

```
long utilfunc(long a, long b, long c)
{
    long xx = a + 2;
    long yy = b + 3;
    long zz = c + 4;
    long sum = xx + yy + zz;

    return xx * yy * zz + sum;
}
```

This is indeed a leaf function. Let's see how its stack frame looks when compiled with `gcc`:



The run-time stack

Block-structured languages organize the overall control flow of a program using the *run-time stack*.

- The run-time stack (RTS) is a control stack where nodes are activation records.
- When a subroutine is called, its activation record is pushed on the stack. When it returns, the AR is popped.

Procedure calls

In addition to the code of the procedure body, the compiler-generated code of a procedure contains:

- Precall (per invocation):
  - Prepares callee, evaluates parameters, determines return address and address of return value
- Prologue (per procedure)
  - Allocates AR, initializes local data, loads static data
- Epilogue (per procedure):
  - Stores return value, restore caller's AR, jumps to return address
- Postreturn (per invocation):
  - restores register, deallocates AR

Caller-callee conventions

- Procedure calls require flow of information between caller and callee.
- Communication by shared ARs
  - In the RTS, two ARs are next to each other. If the offsets are known, caller can access callee's return value.
  - Conventions vary among compilers, e.g., where to place parameters.
- Communication via registers
  - Parameter values: flow from caller to callee
  - Return value: flow from callee to caller
  - Who is responsible for stack pointer, frame pointer, ...
  - Conventions vary among compilers. Compromises are common: some *caller-saves* registers, some *callee-saves* registers.

Caller-callee collaboration (example)

One possible division of labor:
Caller:

- evaluates parameters
- stores status bits, parameters . . . in caller-save registers
- return value (on the stack): allocates and stores address in register

Callee:

- update base pointer, top pointer
- initializes local data
- executes procedure
- restores top pointer and other registers
- jumps to return address in caller's instructions

Scoping

In language design, two different scoping mechanisms exists for determining the corresponding declaration of an identifier

- Lexical scoping (aka static): examine the program text
- Dynamic scoping: traverse the run-time stack

Lexical scoping is often implemented via *access links*:

- An access link is a pointer to the lexical ancestor.
- It is computed as part of the compilation of a call: the compiler knows the lexical distance; if the offset is fixed and known, it can generate code for the access link.
- The access link is typically also stored in the AR.

Example (scoping)

```
$v = ''global''
sub f {
  print $v
}
sub g {                          sub h {
  my $v = ''my''                    local $v = ''local''
  print $v                          print $v
  &f                                &f
}                                }

# program
&f
&g
&h
```

- Assume "my" specifies static scoping, "local" dynamic scoping.
- Call sequence: f; g → f; h → f
  Output: global; my global; local local

Summary

- Overloading, coercion
- Control structures
- Data layout
- Activation record, run-time stack

References

- IPL, Ch. 4.6
- LLVM online resources