Cecko

Assignment 4 – Generating code for expressions

A more detailed introduction to LLVM IR

LLVM IR

- llvm::Value – Abstract class representing any "algorithm" providing a value
  - Held by llvm::Value * = cecko::CKIRValueObs
  - llvm::Constant
    - llvm::ConstantInt – integer constants computed during compilation
    - other immutable values like addresses of global variables, functions or basic blocks
      - Accessible by var_desc->get_ir() or function_desc->get_function_ir()
  - llvm::Argument
    - an argument of a function
      - your code will never access the argument directly through llvm::Argument
  - llvm::Instruction – anything computed at run-time
    - including llvm::AllocaInst - allocate an address for a local variable
      - Automatically generated by the cecko framework
- Instructions are created using llvm::IRBuilder
  - accessible as ctx->builder()->CreateXYZ(...)
  - the builder appends instructions at an **insertion point** into a **basic block**
    - it may also be in unbound state, unable to emit instructions
    - preset by the framework upon entering a function body (ctx->enter_function())
    - within assignment 4, we will not create other basic blocks
  - it is your responsibility to emit the final return instruction
    - ctx->builder()->CreateRet(value) or CreateRetVoid()
    - after explicit return, call ctx->builder()->ClearInsertionPoint()
    - at the end of the function body, test ctx->builder()->GetInsertBlock()
      - if not null, emit the implicit final return

LLVM IR

- LLVM IR is a Static-Single-Assignment (SSA) form of intermediate code
  - Each "variable" is assigned only once
    - This allows to use pointers to instructions instead of variables
    - There are no true variables in the IR
      - In the text dump, "variable" names are just symbolic names for the defining instructions
  - Cecko variables are represented by IR "Values" containing addresses
    - For global variables, it is a (load-time) Constant
    - For local variables, it is the output of an Alloca instruction
      - Allocas are inserted by the framework at the beginning of the function (not at the place of declaration)
      - For a function argument, the framework also generates a Store instruction to initialize it
      - local_var_desc->get_ir() is a pointer to the associated Alloca instruction

```
int a,b,c;
a=b+c;
```

      - translates to

```
; --- generated by the framework:
%a = alloca i32
%b = alloca i32
%c = alloca i32
; --- the following is your responsibility:
%t0 = load i32, i32* %b
%t1 = load i32, i32* %c
%t2 = add i32 %t0, %t1
store i32 %t2, i32* %c
```

LLVM IR

```
int a,b,c;
a=b+c;
```

- This IR...

```
%t0 = load i32, i32* %b
%t1 = load i32, i32* %c
%t2 = add i32 %t0, %t1
store i32 %t2, i32* %c
```

- ... is generated by the following framework and LLVM calls:
  - First, find what the identifiers mean:

```
CKNamedObs a_desc = ctx->find("a");
CKNamedObs b_desc = ctx->find("b");
CKNamedObs c_desc = ctx->find("c");
```

  - you have to determine the kind of these objects (non-variables are handled differently)
  - you have to check type compatibility within the operators
  - if all three are variables of type int, you will call:

```
CKIRTypeObs int_ir_type = ctx->get_int_type()->get_ir();
CKIRValueObs t0 = ctx->builder()->CreateLoad(int_ir_type,b_desc->get_ir(),"t");
CKIRValueObs t1 = ctx->builder()->CreateLoad(int_ir_type,c_desc->get_ir(),"t");
CKIRValueObs t2 = ctx->builder()->CreateAdd(t0,t1,"t");
ctx->builder()->CreateStore(t2,a_desc->get_ir());
```

  - LLVM automatically appends sequential numbers to the names of the instructions ("t")
- LLVM automatically determines the result type from the value arguments
  - except of Load, ZExt, Trunc, and few other instructions
  - it never performs implicit conversions

- In reality, this code must be separated into several semantic rules
  - a, b, c, +, =
  - You need a grammar attribute to represent a sub-expression

- You need a grammar attribute (a C++ type) to represent a sub-expression
    - This is the most complex problem in single-pass compilers
  - If you find an identifier in an expression...

```
CKNamedSafeObs desc = ctx->find(idf);
if (! desc) message(errors::UNDEF_IDF, loc, idf);
```

    - if you use SafeObs, you may call methods even if it is null

```
if (desc->is_var()) ...
```

    - if it is a variable, you don't know whether it will be read/written/...
        - variable is an L-value and it must be represented by its address

```
e.mode=Lvalue; e.address=desc->get_ir();
```

    - otherwise (enum constant or function), it does not have an address, only a value

```
e.mode=Rvalue; e.value=desc->get_ir();
```

    - in any case, you will also need the type and the const flag

```
e.type=desc->get_type();
e.is_const=desc->is_const();
```

  - There are also integer, character and string literals

  - You need at least two modes of expressions
    - **Lvalue** mode contains a CKIRValueObs representing the **address** of the expression
    - **Rvalue** mode contains a CKIRValueObs representing the **value** of the expression
        - This includes the case of constant expression – use CKTryGetConstantInt to check it
  - You will need a third mode to handle &&,||

- Reading of variables is done only after there is an operation on them
  - If you encounter an operator like e1+e2...
  - Perform some implicit conversions
    - array to pointer – this changes an Lvalue array into an Rvalue pointer (no IR instruction)
  - Check if the operation exists for the combination of types
  - Convert operands to Rvalue mode
    - by emitting a Load instruction
  - Perform other implicit conversions
    - _Bool/char to int – emit ZExt instruction
  - Emit the instruction for the operator
    - Beware of pointer arithmetics
  - The result will be an Rvalue
- Operator = emits the Store instruction
  - After checking compatibility and performing conversions
    - int to char – Trunc instruction
    - int/char to _Bool – ICmpNE instruction with ctx->get_int32/8_constant(0)
    - poiner to _Bool – IsNotNull instruction
- There are also Lvalue-producing operators (e1[e2],*e1,e1.x,e1->x)
  - *e1 produces no instruction if e1 is Rvalue
  - &e1 never produces any instruction

Instructions needed for Cecko

Instruction	Asgn 4	Asgn 5	Note
GlobalString	STRLIT		produces llvm::Constant*
ConstInBoundsGEP2_32	Array to ptr		use two 0 indexes
ICmpNE	char/int to _Bool	!=	in _Bool= and conditions
IsNotNull	ptr to _Bool		in _Bool= and conditions
ZExt	_Bool/char to char/int		in most operators
Trunc	int to char		in char=
Add,Sub,Mul,SDiv,SRem	int+int,-,*,/,%		
GEP	ptr+int,ptr-int,ptr[int]		
Neg	-int,ptr-int		
PtrDiff	ptr-ptr		
StructGEP	str.name,ptr->name		use get_idx()
ExtractValue	f().name		non-L-value before .name
Load	L-value to R-value		
Store	=		
Ret,RetVoid	return		incl. implicit
Call	Function call		incl. void

Instructions needed for Cecko

Instruction	Asgn 4	Asgn 5	Note
ICmpEQ		==	
ICmpSLT,SLE,SGT,SGE		int<int,<=,>,>=	
ICmpULT,ULE,UGT,UGE		ptr<ptr,<=,>,>=	
Not		!	
CondBr		if,while,for,&&,||	
Br		else,while,for	