Lecture 10

CMSC 350: COMPILER DESIGN
TAGGED DATATYPES
C-style Enumerations / ML-style datatypes

• In C:
  ```c
  enum Day {sun, mon, tue, wed, thu, fri, sat} today;
  ```

• In Haskell:
  ```haskell
  data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  ```

• Associate an integer tag with each case: `sun = 0, mon = 1, …`
  – C lets programmers choose the tags

• Haskell datatypes can also carry data:
  ```haskell
  data Foo = Bar Int | Baz Int Foo
  ```

• Representation: a `Foo` value is a pointer to a pair: (tag, data)
• Example: `tag(Bar) = 0, tag(Baz) = 1`
  ```haskell
  [let f = Bar 3] = [let g = Baz 4 f] =
  ```

```
Eisenberg  CMSC 350: Compiler Design

```
Switch Compilation

• Consider the C statement:

```c
switch (e) {
    case sun: s1; break;
    case mon: s2; break;
    ...
    case sat: s3; break;
}
```

• How to compile this?
  – What happens if some of the break statements are omitted? (Control falls
    through to the next branch.)
Cascading if's and Jumps

\[
\begin{array}{l}
\begin{array}{l}
\text{[switch(e) \{case tag\_1: s\_1; case tag\_2: s\_2; \ldots\}] =}
\end{array}
\end{array}
\begin{array}{l}
\begin{array}{l}
\%\text{tag} = [e];
\text{br label } %\text{l1}
\end{array}
\end{array}
\begin{array}{l}
\begin{array}{l}
\text{l1: } \%\text{cmp1} = \text{icmp eq } \%\text{tag}, \%\text{tag\_1}
\text{br } \%\text{cmp1 label } %\text{b1, label } %\text{l2}
\end{array}
\end{array}
\begin{array}{l}
\begin{array}{l}
\%\text{b1: } [s\_1]
\text{br label } %\text{b2}
\end{array}
\end{array}
\begin{array}{l}
\begin{array}{l}
\%\text{l2: } \%\text{cmp2} = \text{icmp eq } \%\text{tag}, \%\text{tag\_2}
\text{br } \%\text{cmp2 label } %\text{b2, label } %\text{l3}
\end{array}
\end{array}
\begin{array}{l}
\begin{array}{l}
\%\text{b2: } [s\_2]
\text{br label } %\text{b3}
\end{array}
\end{array}
\begin{array}{l}
\begin{array}{l}
\text{...}
\%\text{lN: } \%\text{cmpN} = \text{icmp eq } \%\text{tag}, \%\text{tag\_N}
\text{br } \%\text{cmpN label } %\text{bN, label } %\text{merge}
\end{array}
\end{array}
\begin{array}{l}
\begin{array}{l}
\%\text{bN: } [s\_N]
\text{br label } %\text{merge}
\end{array}
\end{array}

\text{merge:}
\end{array}
\]

- Each \$tag\_1...\$tag\_N is just a constant int tag value.

- Note: [[break;]] (within the switch branches) is:
  \text{br } %\text{merge}
Alternatives for Switch Compilation

- Nested if-then-else works OK in practice if # of branches is small
  - (e.g. < 16 or so).
- For more branches, use better data structures to organize the jumps:
  - Create a table of pairs \((v1, \text{branch\_label})\) and loop through
  - Or, do binary search rather than linear search
  - Or, use a hash table rather than binary search

- One common case: the tags are dense in some range \([\text{min}...\text{max}]\)
  - Let \(N = \text{max} – \text{min}\)
  - Create a branch table \(\text{Branches}[N]\) where \(\text{Branches}[i] = \text{branch\_label}\) for tag \(i\).
  - Compute \(\text{tag} = \lfloor e \rfloor\) and then do an \textit{indirect jump}: \(\text{J } \text{Branches}[\text{tag}]\)
- Common to use heuristics to combine these techniques.
Haskell-style Pattern Matching

- Haskell-style match statements are like C’s switch statements except:
  - Patterns can bind variables
  - Patterns can nest

- Compilation strategy:
  - “Flatten” nested patterns into matches against one constructor at a time.
  - Compile the match against the tags of the datatype as for C-style switches.
  - Code for each branch additionally must copy data from \([e]\) to the variables bound in the patterns.

- There are many opportunities for optimization, many papers about “pattern-match compilation”
  - Many of these transformations can be done at the AST level
DATATYPES IN THE LLVM IR
Structured Data in LLVM

• LLVM’s IR is uses types to describe the structure of data.

```
t ::= 
  void
  i1 | i8 | i16  N-bit integers
  [<#elts> x t] arrays
  fty function types
  {t_1, t_2, ..., t_n} structures
  t* pointers
  %Tident named (identified) type

fty ::= Function Types
  t (t_1, .., t_n) return, argument types
```

• `<#elts>` is an integer constant >= 0
• Structure types can be named at the top level:

```
%T1 = type {t_1, t_2, ..., t_n}
```

  – Such structure types can be recursive
Example LL Types

- An array of 350 integers: \([ 350 \times \text{i16} ]\)

- A two-dimensional array of integers: \([ 3 \times [ 4 \times \text{i16} ] ]\)

- Structure for representing arrays with their length:
  \[
  \{ \text{i16} , [0 \times \text{i16}] \}
  \]
  - There is no array-bounds check; the static type information is only used for calculating pointer offsets.

- C-style linked lists (declared at the top level):
  \%
  Node = type \{ i16, \%Node* \}

- Structs from the C program shown earlier:
  \%
  Rect = type \{ \%Point, \%Point, \%Point, \%Point \}
  \%
  Point = type \{ i16, i16 \}
• LLVM provides the `getelementptr` instruction to compute pointer values
  – Given a pointer and a “path” through the structured data pointed to by that pointer, `getelementptr` computes an address
  – It is a “type indexed” operation, since the sizes/computations involved depend on the type

```llvm
insn ::= ...
| getelementptr t, t* %val, t1 idx1, t2 idx2 ,...
```

• Example: access the x component of the first point of a rectangle:

```llvm
%tmp1 = getelementptr %Rect, %Rect* %square, i32 0, i32 0
%tmp2 = getelementptr %Point, %Point* %tmp1, i32 0, i32 0
```
struct RT {
    int A;
    int B[10][20];
    int C;
}
struct ST {
    struct RT X;
    int Y;
    struct RT Z;
}
int *foo(struct ST *s) {
    return &s[1].Z.B[5][13];
}

%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
entry:
%arrayidx = getelementptr %ST, %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
ret i32* %arrayidx
}

Final answer: ADDR + size_t(%ST) + size_t(%RT) + size_t(i32) + size_t(i32) + 5*20*size_t(i32) + 13*size_t(i32)
getelementptr

- GEP *never* dereferences the address it’s calculating:
  - GEP only produces pointers by doing arithmetic
  - It doesn’t actually traverse the links of a datastructure

- To index into a deeply nested structure, need to “follow the pointer” by loading from the computed pointer
  - See list1.ll from HW3
Compiling Datastructures via LLVM

1. Translate high level language types into an LLVM representation type.
   - For some languages (e.g. C) this process is straight forward
     • The translation simply uses platform-specific alignment and padding
   - For other languages, (e.g. OO languages) there might be a fairly complex elaboration.

2. Translate accesses of the data into getelementptr operations
Bitcast

• What if the LLVM IR’s type system isn’t expressive enough?
  – e.g. if the source language has subtyping, perhaps due to inheritance
  – e.g. if the source language has polymorphic/generic types

• LLVM IR provides a bitcast instruction
  – This is a form of (potentially) unsafe cast. Misuse can cause serious bugs
    (segmentation faults, or silent memory corruption)

```llvm
%rect2 = type { i64, i64 }          ; two-field record
%rect3 = type { i64, i64, i64 }     ; three-field record

define @foo() {
  %1 = alloca %rect3     ; allocate a three-field record
  %2 = bitcast %rect3* %1 to %rect2*    ; safe cast
  %3 = getelementptr %rect2* %2, i32 0, i32 1  ; allowed
  ...
}
```