

# CIS 341: Compilers

## Lecture 36

# The Plan

- Today:
  - Project 7 Q&A
  - Miscellaneous topics: Parametric polymorphism? Exceptions?
- Project 7: Register Allocation for OAT
  - Due: Friday, December 5<sup>th</sup> at 11:59 pm
- Announcements:
  - Final Exam: Wednesday, December 10<sup>th</sup> noon – 2:00pm CHEM 514

# Polymorphism

- **Polymorphism** (*poly* = many, *morph* = shape)
  - Code is polymorphic if it can be used to operate on values of more than one type.
- Different kinds of polymorphism:
  - *Subtype polymorphism*: (Java, C#, OAT) code can work on any class that implements a given interface.
  - *Ad-hoc polymorphism*: overloading (using the same name for different functions)
  - *Parametric polymorphism*: templates in C++, Parametric types in OCaml, Haskell, generics in Java
- Typechecking?
- Implementation?
- Interaction with other language features?

# Parametric Polymorphism

- Consider implementing a generic (bubble) sorting function that works for arrays holding values of any type T:

```
sort(T[] a) {  
    for(int i = 0; i<length(a); i=i+1) {  
        for(int j = length(a); j > i; j=j-1) {  
            if(a[j-1] > a[j]) {  
                swap(a[j-1], a[j])  
            }  
        }  
    }  
}
```

- What is wrong with the above?

# Parametric Polymorphism

- How to type check the use of the > operation?

```
sort[Type T](T[] a) {  
    for(int i = 0; i<length(a); i=i+1) {  
        for(int j = length(a); j > i; j=j-1) {  
            if(a[j-1] > a[j]) {  
                swap(a[j-1], a[j])  
            }  
        }  
    }  
}
```

- Templates: create an instance of the sort code for each use (e.g. sort[int] or sort[float]) and typecheck the body separately.

# Parametric Polymorphism

- Using higher-order functions (as in OCaml)

```
sort[Type T](T[] a, gt:T->T->bool) {
  for(int i = 0; i<length(a); i=i+1) {
    for(int j = length(a); j > i; j=j-1) {
      if(gt(a[j-1], a[j])) {
        swap(a[j-1], a[j])
      }
    }
  }
}
```

- Option 2: Pass in a parameter that can calculate the “greater than” function on the type T.
  - The above code doesn’t depend on *any* properties of T – it’s truly parametric in the type T.
  - No need for code duplication

# Typechecking?

- Add *type variables* and rules that express when types are well formed.

$$\boxed{\text{TVAR}} \frac{\alpha : \text{Type} \in E}{E \vdash \alpha : \text{Type}}$$

- Useful to add parameterized types (like array).

$$\boxed{\text{TARRAY}} \frac{E \vdash T : \text{Type}}{E \vdash T[] : \text{Type}}$$

- This gives rise to polymorphic types (e.g. this type of sort):  
 $\forall \alpha. \alpha[] \rightarrow (\alpha \rightarrow \alpha \rightarrow \text{bool}) \rightarrow \text{unit}$   
In Ocaml: `'a array -> ('a -> 'a -> bool) -> unit`

# Implementation?

- One option is to duplicate code as in C++ templates.
  - Code bloat
  - Type errors in library code when instantiated!
  - Libraries must be shipped in source form
  - Recompile of the ‘sort’ function when the code for ‘>’ changes.
- ML, Haskell, Java generics:
  - Parametric types work uniformly over the data.
  - But: data representation must be uniform (i.e. all values passed to a parametric polymorphic function must take up the same space)
  - Option 1: (ML,Haskell) Automatic *boxing/unboxing* of values.
    - boxing = representing a value by using a pointer to the value
  - Option 2: (Java) restrict polymorphism only to “reference types”

# Exceptions?

- Exceptions are alternate return paths from a function.
  - normal return path + several exceptional return paths
- Many different exception models
  - different designs impact efficiency
- Consider Java/C++-style exceptions:
  - **throw E** terminates exceptionally with exception E
  - Control propagates lexically within the current function to the nearest enclosing **try...catch** block containing the throw
  - If not caught within the function, the exception propagates dynamically upward in the call chain.
- How to implement efficiently?

# Implementing Exceptions

- Desired properties (ideally):
- Exceptions are for unusual situations and should not slow down the common case.
  - No performance cost when functions return normally
  - Little cost when executing try...catch whenever an exception is not thrown.
  - Cost of throwing/catching an exception could be more expensive than a normal termination.

# Local vs. Dynamic Throws

- Sometimes the compiler can identify statically the appropriate exception handler (i.e. when it's in the same procedure).
  - Rewrite “throw” into “goto”
- Example:  
try { if (b) throw new E(); else ... } catch (E e) {...handler...}  
→  
if (b) { e = newE(); goto L1; } else ...; goto L2  
L1: {...handler...}  
L2:
- Dynamic throws:
  - Need to find the closest try..catch

## Option 1: Extra return value

- Return an extra (hidden) boolean value from every function indicating whether the function returned normally or not:
  - throw  $e \rightarrow$  return (false,  $e$ )
  - return  $e \rightarrow$  return (true,  $e$ )
  - $a = f(b,c) \rightarrow$  (normal,  $t$ ) =  $f(b,c)$ ; if (!normal) goto handel\_341 else  $a = t$
- Every function call requires extra return, extra checks.
- No cost for try...catch unless exception is thrown
- Can be expressed as source-to-source translations

## Option 2: setjmp/longjmp

- The operation `setjmp(buf)` saves all registers (including ESP and EIP!) into a buffer and returns 0 in EAX
- Calling `longjmp(buf)` restores all registers from the buffer; places 1 into the return register
  - This makes `setjmp` “return again” with a different flag.
- Implementation: stack of state buffers, with ‘`current_catch`’ being the top one:
- `throw e` → `exc = e; longjmp(current_catch)`
- `try S catch C` → `push_catch();`  
`if (setjmp(current_catch) == 0) S else C;`  
`pop_catch();`

# Setjmp/Longjmp Summary

- **Advantages:**
  - No cost as long as try/catch & throw are unused
  - works even without declared exceptions (no static information is needed)
- **Disadvantages:**
  - try/catch is slow even if no exception is thrown
  - May need to walk up through several longjumps until the right try... catch is found (i.e. the handler may need to reraise the exception)

## Option 3: Continuations

- When we return from a function (either normally or exceptionally) we want to jump to the right “continuation”
  - Continuation is “the rest of the program”
- Abstractly, a continuation is a function that does not return.
  - Setjmp creates a continuation
  - Longjmp calls it
- Whole program transformation can thread the appropriate ‘return’ continuation and ‘exceptional’ continuations through the program...