Compilers

INF-400

Burak Arslan
ext-inf400@burakarslan.com

Galatasaray Üniversitesi

Lecture XI
2024-12-19

Course website

burakarslan.com/inf400

Course website

Runtime Environments (cont'd)

News

Note that:

1. This is the last lecture
2. Next week: **Mock Final**

Code Generation

Two goals:

1. Correctness
2. Speed

Most complications in code generation come from trying to be fast as well as correct

Code Generation

Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order
2. When a procedure is called, control eventually returns to the point immediately after the call

Do these assumptions always hold?

Activations

- An **invocation** of procedure P is an activation of P
- The lifetime of an activation of P is:
  - All the steps to execute P
  - Including all the steps in procedures P calls

Activations

Lifetimes of Variables

The **lifetime** of a variable `x` is the portion of execution in which `x` is defined

- Lifetime is a dynamic (run-time) concept
- Scope is a static concept

Activations

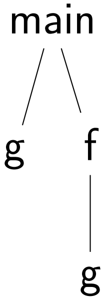
Activation Trees

- Assumption (2) requires that when `P` calls `Q`, then `Q` returns before `P` does
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

Activations

Activation Trees: An Example

```
func g() : Integer64 { return 1 };
func f() : Integer64 { return g(); };
func main(): Integer64 { g(); return f(); };
```



Activations

Activation Trees

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

Activations

Activation Records

- The information needed to manage one procedure activation is called an activation record (AR) or stack frame or just "frame".
- If procedure `F` calls `G`, then `G`'s activation record contains a mix of info about `F` and `G`.

Activations

Activation Records

WASM already contains an function stack implementation:

- We won't need to deal with managing the function call stack

Code Generation for Object-Oriented Programming Languages

Code Generation for OOPL

Is kiraz an OOPL?

Three pillars of Object Oriented Programming are:

1. Encapsulation
2. Inheritance
3. Polymorphism

Does kiraz support all three?

Code Generation for OOPL

Object Layout

- **OO Slogan**: If `B` is a subclass of `A`, then an object of class `B` can be used wherever an object of class `A` is expected
- This means that code in class `A` works unmodified for an object of class `B`

Code Generation for OOPL

Object Layout

Two issues:

- How are objects represented in memory?
- How is dynamic dispatch implemented?

Code Generation for OOPL

Object Layout

```
class A {
    a: Integer64;
    d: Integer64;
    func f(): Integer64 {
        a = a + d; return r;
    };
};
```

```
class B : A {
    b: Integer64;
    func f(): Integer64 {
        return a; };
    func g(): Integer64 {
        a = a + b; return a; };
};
```

```
class C : A {
   c: Integer64;
   func h(): Integer64 {
      a = a + c; return a;
   };
};
```

Code Generation for OOPL

Object Layout

Attributes `a` and `d` are inherited by classes `B` and `C`

- All methods in all classes refer to `a`
- For the methods of `A` to work correctly in `A`, `B`, and `C` objects, attribute `a` must be in the same "place" in each object

Code Generation for OOPL

Object Layout

Just like `structs` in C, The dot operator statement

```
foo.attribute
```

translates to an index into a `foo` struct at an offset corresponding to `attribute`

Code Generation for OOPL

Object Layout

Observation: Given a layout for class `A`, a layout for subclass `B` can be defined by extending the layout of `A` with additional slots for the additional attributes of `B`
Leaves the layout of A unchanged (B is an extension)

Code Generation for OOPL

Object Layout

Question: Given that each `Integer64` in kiraz needs 1 `i32` and 1 `i64` in memory,
how many bytes does each clas A, B and C take, given 64bit alignment ignoring all additional class metadata?

Code Generation for OOPL

Dynamic Dispatch

`e.g()`

- `g` refers to method in `B` if type of `e` is `B`

`e.f()`

- `f` refers to method in `A` if type of `e` is `A` or `C` (inherited in the case of `C`)
- `f` refers to method in `B` if type of `e` is `B`

Code Generation for OOPL

Dispatch Tables

Every class has a fixed set of methods (including inherited methods)

A dispatch table indexes these methods:

- An array of method entry points
- A method `f` lives at a fixed offset in the dispatch table for a class and all of its subclasses

Code Generation for OOPL

Dispatch Tables

The dispatch pointer in an object of class `X` points to the dispatch table for class `X`

- Every method `f` of class `X` is assigned an offset `Of` in the dispatch table at compile time

Code Generation for OOPL

Dispatch Tables

This is called a `vtable` in C++

- Each class with at least one virtual method has a `vtable pointer`
- There is one vtable per **class** (**not** instance!)
- Virtual functions are called by first looking up the actual function pointer in the vtable