

Motivation for the Duck Language

Geoffrey Irving

Basic Problem

“Compilers get scared very easily.”

- Kevin Bowers

Why do compilers get scared?

- Each optimization pass has three parts:
 1. Is this transform worth it (HARD)?
 2. Is this transform safe (HARD)?
 3. Do the transform (EASY).
- Difficulty: (1) >> (2) >> (3)
- Maybe we should stop designing languages that assume smart compilers.

Is it worth it?

- Current compilers use complicated heuristics to decide whether to optimize.
- E.g., inline if
 - The function is small enough
 - It isn't recursive
 - We haven't inlined too much already
- Heuristics are **fragile**.

Make the programmer decide

- The programmer knows where to optimize, so add one more line of code:

```
fast_function = <optimize>(slow_function)
```

- <optimize> can be very aggressive:
 - inline_all_function_calls
 - unroll_all_constant_loops
 - partially_specialize_everything

Is it safe?

- Imperative programming makes optimization hard

```
x = *p + *p;
```

```
*q = z;
```

```
y = *p + *p; // same as x?
```

- Need alias analysis to tell whether `*p` and `*q` might be the same.

Make the programmer tell us

- Everything purely functional by default.
- Aliasing doesn't matter without assignment
- When we need mutability, we can read aliasing information off the types:

```
p :: Mutable Int a    // a != b
```

```
q :: Mutable Int b    // so no aliasing
```

- No need for `-fstrict-aliasing` fragility.

What optimization would look like:

- Each optimization pass is:
 1. Is this transform safe (TRIVIAL)?
 2. Do the transform (EASY).
- If step (1) fails, **bail**.
- No more, “I wonder if that optimization happened” confusion.

Optimizations are library functions

- Language has primitives to
 - Extract source (IR) from a closure
 - Turn source (IR) into a closure
 - Compile a closure to machine code
 - Jump to C calling convention function
- Inlining, etc. built on top of these

Optimizations can't fix data structures

- We want to
 1. Describe data structures at a low level
 2. Operate on them at a high level
- Solution: overloading and templates
- Safer to use if you can trust inlining

Fitting onto heterogenous hardware

- Requires a few more library functions:
 - Eliminate all dynamic allocation, or **bail**.
 - Translate source into C.
 - Invoke tensilica/gc compiler.
- Also requires lots of imperative expressiveness (hard, but I'm optimistic)

Language details

- Implemented:
 - Strongly typed and type inferred
 - Functions can be overloaded arbitrarily
 - Similar to C++ overloaded templates, but not horrifically ugly.
 - Closures and higher order functions
 - Haskell-like syntax
- Not implemented: the previous slides

Type inference

- Don't try to be smart
- Run the interpreter with types only
- Cache type judgements for functions
- Use fixed point iteration to infer recursive functions
- Undecidable: $O(\text{number of types seen})$
- Use iteration cutoffs to avoid infinite loops

Next steps

- Finish type system and syntax
- Use LLVM for machine code translation
- Expand primitive data structures
 - float point, arrays, packed structs, etc.
- Expand support for imperative code
- Write some optimization passes
- Pick a benchmark and make it fast

Practical issues

- Two people spending free time only
 - Dylan Simon, me
 - Forces us to stay simple
- LLVM lacks built-in garbage collection
 - More work for us
- Lots of straightforward but time consuming things to do.

Hello world!

```
import prelude
```

```
mapM :: (a -> IO b) -> List a -> IO (List b)
```

```
mapM f l = case l of
```

```
  [] -> returnIO []
```

```
  x:l ->
```

```
    f x >>= \y ->
```

```
    mapM f l >>= \l ->
```

```
    returnIO (y:l)
```

```
put :: List Chr -> IO ()
```

```
put s = mapM put s >> returnIO ()
```

```
main = put ['H','e','l','l','o',' ',' ',  
           'w','o','r','l','d','!','\n']
```