

Compilation by Transformation

COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University
Semester 2, 2009

Syntactic Sugar

$\text{let } x = e_1 \text{ in } e_2 \stackrel{\text{def}}{=} (\lambda x. e_2) e_1$
 $\text{let } f \bar{x} = e_1 \text{ in } e_2 \stackrel{\text{def}}{=} \text{let } f = \lambda \bar{x}. e_1 \text{ in } e_2$
 $\text{letrec } x = e_1 \text{ in } e_2 \stackrel{\text{def}}{=} \text{let } x = (\mathbf{fix} (\lambda x. e_1)) \text{ in } e_2$
 $\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \stackrel{\text{def}}{=} \text{case } e_1 \text{ of } \{ \text{True} \rightarrow e_2; \text{False} \rightarrow e_3 \}$

- We use **fix** instead of Y because the former is easier assign a type to, more on this later.

A useful untyped core language

Expressions

e = x (value variable)
| $\lambda x. e$ (function abstraction)
| $e_1 e_2$ (function application)
| $\mathbf{fix} e_1$ (fix points)
| $\mathbf{case} e_1 \text{ of } \{\overline{alt}\}$ (case expression)
| C (data constructor)
| p (primitive value)

Evaluation

$(\lambda x. e_1) e_2 \longrightarrow e_1[x := e_2]$
 $\mathbf{fix} (\lambda x. e_1) \longrightarrow e_1[x := \mathbf{fix} (\lambda x. e_1)]$
 $\mathbf{case} C_n \bar{e} \text{ of } \{\dots C_n \bar{x} \rightarrow e' \dots\} \longrightarrow e'[\bar{x} := \bar{e}]$

Compilation of *reverse*

$reverse [] = []$
 $reverse (x : xs) = reverse xs ++ [x]$

$main = reverse [1, 2, 3]$

Wrap top-level bindings in letrec

```
letrec reverse [] = []  
       reverse (x : xs) = reverse xs ++ [x]  
in     reverse [1, 2, 3]
```

Desugar pattern matching

```
letrec reverse xx  
       = case xx of {  
           Nil      → Nil ;  
           Cons x xs → reverse xs ++ Cons x Nil ;  
       }  
in     reverse (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Desugar list expressions

```
letrec reverse Nil = Nil  
       reverse (Cons x xs) = reverse xs ++ Cons x Nil  
in     reverse (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Desugar infix expressions

```
letrec reverse xx  
       = case xx of {  
           Nil      → Nil ;  
           Cons x xs → append (reverse xs) (Cons x Nil) ;  
       }  
in     reverse (Cons 1 (Cons 2 (Cons 3 Nil)))
```

→ Conversion to C

```
Obj* Nil    (void);
Obj* Cons  (Obj*, Obj*);
Obj* append (Obj*, Obj*);

Obj* reverse (Obj* xx) {
  switch (TAG(xx)) {
    case TagNil: return Nil();
    case TagCons: return append(reverse(ARG(xx,2)),Cons(ARG(xx,1),Nil()));
  }
}

int main(int argc, char** argv) {
  printf("%d",
    UNBOX(reverse(Cons(BOX(1), Cons(BOX(2), Cons(BOX(3), Nil())))))));
}
```

Desugar letrec

```
let   reverse = fix ( $\lambda$ reverse.  $\lambda$ xx.
      case xx of {
          Nil       $\rightarrow$  Nil ;
          Cons x xs  $\rightarrow$  append (reverse xs) (Cons x Nil) ;
      })
in   reverse (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Desugar function binding syntax

```
letrec reverse =  $\lambda$ xx.
      case xx of {
          Nil       $\rightarrow$  Nil ;
          Cons x xs  $\rightarrow$  append (reverse xs) (Cons x Nil) ;
      }
in   reverse (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Desugar let

```
( $\lambda$ reverse. reverse (Cons 1 (Cons 2 (Cons 3 Nil))))
(fix ( $\lambda$ reverse.  $\lambda$ xx.
      case xx of {
          Nil       $\rightarrow$  Nil ;
          Cons x xs  $\rightarrow$  append (reverse xs) (Cons x Nil) ;
      })))
```

β -reduction

```
(fix (λreverse . λxx.  
  case xx of {  
    Nil      → Nil ;  
    Cons x xs → append (reverse xs) (Cons x Nil) ;  
  }) (Cons 1 (Cons 2 (Cons 3 Nil)))
```

β -reduction

```
case (Cons 1 (Cons 2 (Cons 3 Nil))) of {  
  Nil      → Nil ;  
  Cons x xs → append ((fix (λreverse ....)) xs) (Cons x Nil) ;  
}
```

fix-reduction

```
(λxx. case xx of {  
  Nil      → Nil ;  
  Cons x xs → append ((fix (λreverse ....)) xs) (Cons x Nil) ;  
})  
(Cons 1 (Cons 2 (Cons 3 Nil)))
```

case-match

```
append ((fix (λreverse ....)) (Cons 2 (Cons 3 Nil)))  
(Cons 1 Nil)
```

Expand definition of *append*

```
(fix (λappend....))  
  ((fix (λreverse....)) (Cons 2 (Cons 3 Nil)))  
  (Cons 1 Nil)
```

- etc, etc, etc ...
- Evaluation can be modeled as a sequence of reductions to normal form.
- In a compiler, we typically stop reducing when the program is in a state that makes it easy (and useful) to translate into another language.
- That other language might be C, Java Bytecodes, x86 assembly ...
- The downstream system handles the subsequent reduction.

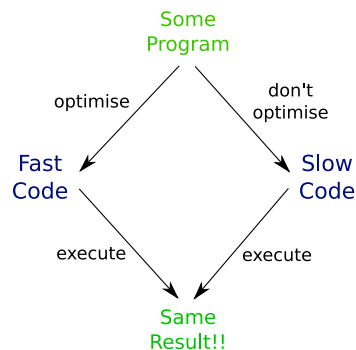
Optimisation

- The best optimisations are those that *always* make the program better.
- Next best are those that make *some* better, and don't make *any* worse.
- .. then those that make *most* programs better, and only a *few* worse.
- .. after that, we're probably wasting our time.

- Inlining and let-floating transformations are the great enablers. They move definitions into their use sites, enabling further optimisation.

Optimisation

- Compiler optimisations are reductions that make subsequent ones need less time or space to complete.
- Optimisations should be correctness preserving, that is, they should obey the diamond property.
- We don't take "time or space needed to evaluate" as part of the result.



Example transform: case-of-case

```
case (case x of { pat11 → e11; pat12 → e12 }) of {  
  pat21 → e21;  
  pat22 → e22  
}  
  
→coc case x of {  
  pat11 → case e11 of {pat21 → e21; pat22 → e22};  
  pat12 → case e12 of {pat21 → e21; pat22 → e22};  
}
```

- The outer **case** inspects the result of the inner one.
- **case-of-case** risks duplicating code, but the result doesn't require more reductions than the original expression.

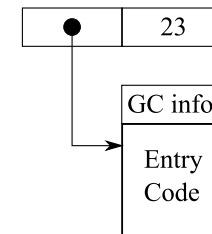
By your powers combined...

$\text{if not } x \text{ then } e_1 \text{ else } e_2$
 $\xrightarrow{\text{def}}$ $\text{case not } x \text{ of } \{ \text{True} \rightarrow e_1 ; \text{False} \rightarrow e_2 \}$
 $\xrightarrow{\text{inline}}$ $\text{case } (\lambda y. \text{case } y \text{ of } \{ \text{True} \rightarrow \text{False} ; \text{False} \rightarrow \text{True} \}) x \text{ of } \{$
 $\text{True} \rightarrow e_1 ;$
 $\text{False} \rightarrow e_2$
 $\}$
 $\xrightarrow{\beta}$ $\text{case } (\text{case } x \text{ of } \{ \text{True} \rightarrow \text{False} ; \text{False} \rightarrow \text{True} \}) \text{ of } \{$
 $\text{True} \rightarrow e_1 ;$
 $\text{False} \rightarrow e_2$
 $\}$

Boxed vs Unboxed Integers

- Polymorphic languages typically used a boxed data format, where all runtime objects have a standard layout.
- In GHC, the first word of every object is a pointer to the code which evaluates it, as well as the info table describing the object's layout to the garbage collector (GC).
- Java has a similar system, compare `Integer` vs `int`.
- This supports GC, but we can't perform arithmetic directly on boxed objects.

`sizeof(Int) = 8 bytes`



$\text{case } (\text{case } x \text{ of } \{ \text{True} \rightarrow \text{False} ; \text{False} \rightarrow \text{True} \}) \text{ of } \{$
 $\text{True} \rightarrow e_1 ;$
 $\text{False} \rightarrow e_2$
 $\}$
 $\xrightarrow{\text{coc}}$ $\text{case } x \text{ of } \{$
 $\text{True} \rightarrow \text{case False of } \{ \text{True} \rightarrow e_1 ; \text{False} \rightarrow e_2 \} ;$
 $\text{False} \rightarrow \text{case True of } \{ \text{True} \rightarrow e_1 ; \text{False} \rightarrow e_2 \}$
 $\}$
 $\xrightarrow{\text{case}}$ $\text{case } x \text{ of } \{$
 $\text{True} \rightarrow e_2 ;$
 $\text{False} \rightarrow e_1$
 $\}$
 \rightarrow **WIN!**



image: Tim Chevalier

Defining boxed integers in terms of unboxed ones

data $Int = I_{\#} Int_{\#}$

0 $\stackrel{\text{def}}{=} I_{\#} 0_{\#}$

1 $\stackrel{\text{def}}{=} I_{\#} 1_{\#}$

2 $\stackrel{\text{def}}{=} I_{\#} 2_{\#}$

...

$x + y = \text{case } x \text{ of } \{ I_{\#} x' \rightarrow \text{case } y \text{ of } \{ I_{\#} y' \rightarrow I_{\#} (x' +_{\#} y') \} \}$

- $2_{\#}$ is an unboxed integer.
- $Int_{\#}$ is the type of unboxed integers.
- $+_{\#}$ is the “real” unboxed addition operator.

case x of

$I_{\#} x' \rightarrow \text{case } y$ of

$I_{\#} y' \rightarrow \text{case (case } z$ of

$I_{\#} z' \rightarrow I_{\#} (y' *_{\#} z'))$ of

$I_{\#} w' \rightarrow I_{\#} (x' +_{\#} w')$

$\xrightarrow{\text{coc}}$ **case** x of

$I_{\#} x' \rightarrow \text{case } y$ of

$I_{\#} y' \rightarrow \text{case } z$ of

$I_{\#} z' \rightarrow \text{case } (I_{\#} (y' *_{\#} z'))$ of

$I_{\#} w' \rightarrow I_{\#} (x' +_{\#} w')$

$x + y * z$

$\xrightarrow{\text{def}}$ **case** x of

$I_{\#} x' \rightarrow \text{case (case } y$ of

$I_{\#} y' \rightarrow \text{case } z$ of

$I_{\#} z' \rightarrow I_{\#} (y' *_{\#} z'))$ of

$I_{\#} w' \rightarrow I_{\#} (x' +_{\#} w')$

$\xrightarrow{\text{coc}}$ **case** x of

$I_{\#} x' \rightarrow \text{case } y$ of

$I_{\#} y' \rightarrow \text{case (case } z$ of

$I_{\#} z' \rightarrow I_{\#} (y' *_{\#} z'))$ of

$I_{\#} w' \rightarrow I_{\#} (x' +_{\#} w')$

case x of

$I_{\#} x' \rightarrow \text{case } y$ of

$I_{\#} y' \rightarrow \text{case } z$ of

$I_{\#} z' \rightarrow \text{case } (I_{\#} (y' *_{\#} z'))$ of

$I_{\#} w' \rightarrow I_{\#} (x' +_{\#} w')$

$\xrightarrow{\text{case}}$ **case** x of

$I_{\#} x' \rightarrow \text{case } y$ of

$I_{\#} y' \rightarrow \text{case } z$ of

$I_{\#} z' \rightarrow I_{\#} (x' +_{\#} (y' *_{\#} z'))$

Making it easier to read.. (maybe)

$\text{case } e_1 \text{ of } C x \rightarrow e_2 \stackrel{\text{def}}{=} \text{esac } C x \leftarrow e_1 \text{ fo } e_2$

case x **of**

$I_{\#} x' \rightarrow$ **case** y **of**

$I_{\#} y' \rightarrow$ **case** z **of**

$I_{\#} z' \rightarrow I_{\#} (x' +_{\#} (y' *_{\#} z'))$

$\xrightarrow{\text{def}}$ **esac** $I_{\#} x' \leftarrow x$ **fo** (unbox x)
esac $I_{\#} y' \leftarrow y$ **fo** (unbox y)
esac $I_{\#} z' \leftarrow z$ **fo** (unbox z)
 $I_{\#} (x' +_{\#} (y' *_{\#} z'))$ (compute result and re-box)

Demo: Seeing what the compiler is doing

- `ghc -c Main.hs -ddump-simpl`
- `ghc -O2 -c Main.hs -ddump-simpl`
- `ghc -O2 -c Main.hs -ddump-stg`
- `ghc -O2 -c Main.hs -ddump-opt-cmm`
- `ghc -O2 -c Main.hs -ddump-asm`