

Compilation by Transformation

COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University

Semester 2, 2009

A useful untyped core language

Expressions

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

Evaluation

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

Syntactic Sugar

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

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

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 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 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 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 function binding syntax

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

Desugar letrec

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

Desugar let

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

β -reduction

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

fix -reduction

$$\begin{aligned} & (\lambda xx. \mathbf{case} \ xx \ \mathbf{of} \ \{ \\ & \quad \quad \quad Nil \quad \quad \rightarrow Nil ; \\ & \quad \quad \quad Cons \ x \ xs \ \rightarrow \mathit{append} \ ((\mathbf{fix} \ (\lambda reverse \ \dots)) \ xs) \ (Cons \ x \ Nil) ; \\ & \quad \quad \quad \}) \\ & \quad (Cons \ 1 \ (Cons \ 2 \ (Cons \ 3 \ Nil))) \end{aligned}$$

β -reduction

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

case -match

*append ((fix (λreverse)) (Cons 2 (Cons 3 Nil)))
(Cons 1 Nil)*

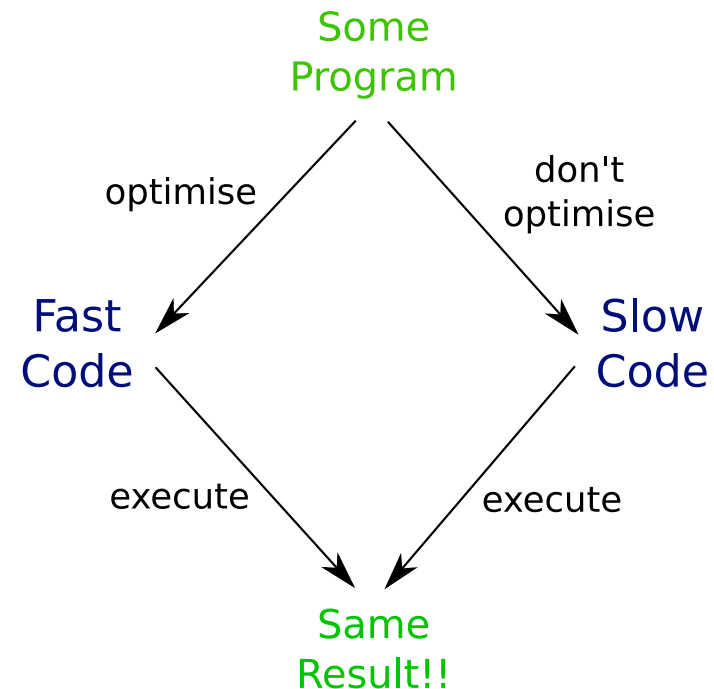
Expand definition of *append*

```
(fix ( $\lambda append$  ....))  
  ((fix ( $\lambda reverse$  ....)) (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

- 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.



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.

Example transform: case-of-case

$$\begin{array}{l} \text{case (case } x \text{ of } \{ pat_{11} \rightarrow e_{11}; pat_{12} \rightarrow e_{12} \}) \text{ of } \{ \\ \quad pat_{21} \rightarrow e_{21}; \\ \quad pat_{22} \rightarrow e_{22} \\ \} \\ \xrightarrow{\text{coc}} \text{case } x \text{ of } \{ \\ \quad pat_{11} \rightarrow \text{case } e_{11} \text{ of } \{ pat_{21} \rightarrow e_{21}; pat_{22} \rightarrow e_{22} \}; \\ \quad pat_{12} \rightarrow \text{case } e_{12} \text{ of } \{ pat_{21} \rightarrow e_{21}; pat_{22} \rightarrow e_{22} \} \\ \} \end{array}$$

- 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...

if *not* x **then** e_1 **else** e_2

$\xrightarrow{\text{def}}$ **case** *not* x **of** { $\text{True} \rightarrow e_1$; $\text{False} \rightarrow e_2$ }

$\xrightarrow{\text{inline}}$ **case** $(\lambda y. \text{case } y \text{ of } \{ \text{True} \rightarrow \text{False} ; \text{False} \rightarrow \text{True} \}) x$ **of** {
 $\text{True} \rightarrow e_1$;
 $\text{False} \rightarrow e_2$
}

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

$$\begin{array}{l}
\text{case } (\text{case } x \text{ of } \{ \text{True} \rightarrow \text{False}; \text{False} \rightarrow \text{True} \}) \text{ of } \{ \\
\quad \text{True} \rightarrow e_1 ; \\
\quad \text{False} \rightarrow e_2 \\
\} \\
\begin{array}{l}
\xrightarrow{\text{coc}} \\
\text{case } x \text{ of } \{ \\
\quad \text{True} \rightarrow \text{case } \text{False} \text{ of } \{ \text{True} \rightarrow e_1; \text{False} \rightarrow e_2 \} ; \\
\quad \text{False} \rightarrow \text{case } \text{True} \text{ of } \{ \text{True} \rightarrow e_1; \text{False} \rightarrow e_2 \} \\
\} \\
\begin{array}{l}
\xrightarrow{\text{case}} \\
\text{case } x \text{ of } \{ \\
\quad \text{True} \rightarrow e_2 ; \\
\quad \text{False} \rightarrow e_1 \\
\} \\
\longrightarrow \text{WIN!}
\end{array}
\end{array}$$

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`

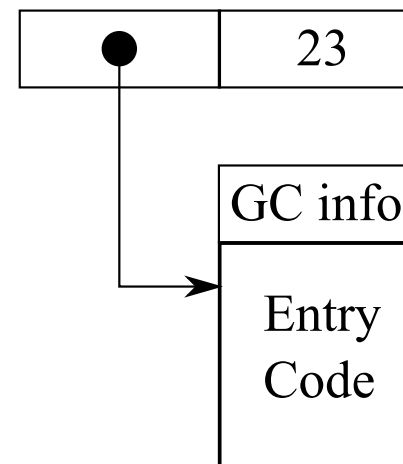




image: Tim Chevalier

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

...

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

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

$x + y * z$

$\xrightarrow{\text{def}}$ **case** x **of**
 $I_{\#} x' \rightarrow$ **case** (**case** y **of**
 $I_{\#} y' \rightarrow$ **case** z **of**
 $I_{\#} z' \rightarrow I_{\#} (y' *_{\#} z')$) **of**
 $I_{\#} w' \rightarrow I_{\#} (x' +_{\#} w')$)

$\xrightarrow{\text{coc}}$ **case** x **of**
 $I_{\#} x' \rightarrow$ **case** y **of**
 $I_{\#} y' \rightarrow$ **case** (**case** z **of**
 $I_{\#} z' \rightarrow I_{\#} (y' *_{\#} z')$) **of**
 $I_{\#} w' \rightarrow I_{\#} (x' +_{\#} w')$)

case x of

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

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

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

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

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

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

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

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

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

case x of

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

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

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

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

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

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

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

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

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`