Compiling functional languages

http://www.cse.chalmers.se/edu/year/2011/course/CompFun/

Lecture 7
Lazy evaluation

Johan Nordlander

Laziness

- Main principle: names stand for arbitrary <u>expressions</u>, not just their final <u>values</u>
- Consequence: an expression bound to a name must <u>not be evaluated until</u> the name is first referenced
- Optimization: an expression bound to a name must <u>not be re-evaluated after</u> the name is first referenced

Laziness

- Note: a lazy language
  - is fundamentally distinct from our strict core, with a different semantics
  - may however be implemented <u>by translation</u> into our strict core
  - may also reuse the syntax, type system, etc, of our strict core
- To emphasize distinction, we write a lazy e as $\underline{e}$

Translating laziness

- Delaying evaluation can be implemented by creating 0-arity closures (a.k.a. thunks)
- Must be done for
  - function arguments (bound to parameters)
  - constructor arguments (bound to var-patterns)
  - RHS of let-bindings
- Avoiding re-evaluation will require some form of memoization/mutation

Building/entering thunks

- Building a thunk:

  translate (e_0 e_1) = (translate e_0) (CL f ys)

  where

  ys = fvs(e_1)

  and f is a fresh top-level name defined as

  f = \x_{this} -> **case** x_{this} **of** CL _ ys -> translate e_1

- Similarly for RHS of top- and let-bindings

- Entering a thunk:

  translate x = **case** x **of** CL f -> f x

Avoiding re-evaluation

- After translation, a variable will denote a <u>thunk</u> (a closure expecting no arguments) on the heap
- Referencing a variable means entering its closure, which triggers the delayed evaluation
- After evaluation, the thunk should <u>mutate</u>:

(a) either into a thunk that just returns the value

(b) or into the value itself!

Thunk mutation (a)

- Concretely, a thunk CL f ys, with

  f = \x_{this} -> **case** x_{this} **of** CL _ ys -> e

  should mutate into CL f_{done} v when e has evaluated to v, and where

  f_{done} = \x_{this} -> **case** x_{this} **of** CL _ v -> v

  is a a common run-time system function
- Advantage: uniform code for referencing a variable before/after mutation
- Disadvantage: persistent overhead of a fun-call

Thunk mutation (b)

- Here, a thunk CL f ys, with

  f = \x$_{this}$ -> **case** x$_{this}$ **of** CL _ ys -> e,

  may mutate into K vs when e evaluates into v and v is a pointer to (a sufficiently small) K vs on the heap
- Advantage: no overhead of indirect jumps after evaluation
- Disadvantage: slightly more complicated variable referencing code:

  translate x = **case** x **of** CL f -> f x; _ -> x

Thunk mutation

- Scheme (a) ≈ push/enter (with no push!)
- Scheme (b) ≈ eval/apply (with no apply!)
- Both schemes can however coexist with an overall eval/apply arity-matching strategy
- Scheme (a) might also be necessary as a last resort, unless all thunks are made big enough to hold the largest possible heap value
- Moreover, scheme (a) allows update with unboxed non-pointer values

Thunk mutation

- Scheme (a) in concrete C code:

```
WORD f (WORD x_this) {
    WORD y1 = x_this[1]; ...; WORD ym = x_this[m];
    WORD v = <code for evaluating e>;
    x_this[0] = f_done;
    x_this[1] = v;
    return v;
}
```

- Scheme (b) is similar, but must resort to (a) if v points to a node bigger than m+1 words

Function thunks

- What if a thunk evaluates to a closure?

closureConvert (translate (x e_1 ... e_m)) =

```
case (case x of CL f -> f x) of CL f n
    | m == n  -> f x e1 ... em
    | m < n   -> CL pap_{n-m,m} (n-m) x e1 ... em
    | m > n   -> apply_{m-n} (f x e1 ... en) e_{n+1} ... em
```

- Observation: the ordinary closure-entry code works just as well for thunks (n = 0)! Equivalent:

```
case x of CL f n
    | m == n  -> f x e1 ... em
    | m < n   -> CL pap_{n-m,m} (n-m) x e1 ... em
    | m > n   -> apply_{m-n} (f x e1 ... en) e_{n+1} ... em
```

- Requires that thunks store a zero arity as param 2

Constructor thunks

- A common case with scheme (b):

  translate (h = \x -> **case** x **of** K_i xs_i -> e_i) =

  h = \x -> **case** (**case** x **of** CL f -> f x; _ -> x) **of**

  K_i xs_i -> translate e_i

- Might be optimized into

  h = \x -> **case** x **of** CL f -> h (f x); K_i xs_i -> translate e_i

- That is, the cost of checking evaluatedness can be hidden in the ordinary code for branching
- Requires that closures are also given a tagged representation, with a globally unique tag

Simple optimizations

- A very common pattern:

  translate (e_0 x) = (translate e_0) (CL f x)

  where

  ```
  f = \x_this -> case x_this of CL _ x ->
                     case x of CL f -> f x
  ```

- But a closure that just enters x is equal to x!
- Thus:

  translate (e_0 x) = (translate e_0) x

Simple optimizations

- Literals are already evaluated:

  translate (e_0 n) = (translate e_0) (CL f_{done} n)

- Variables known to be evaluated may be treated the same way (scheme (a)):

  let x = K es **in** ...

  translate (e_0 x) = (translate e_0) (CL f_{done} x)

- Or using scheme (b):

  let x = K es **in** ...

  translate (e_0 x) = (translate e_0) x

Exploiting strictness

- Consider a function `h = \x -> case x of Ki xsi -> ei` and a call `h (y 7)`
- After translation we would have

  `h = \x -> case x of CL f -> h (f x); Ki xsi -> ei`

  and the call would have become `h (CL f y)` where `f = \xthis -> case xthis of CL _ y -> y 7`
- Clearly the thunk given to `h` will be entered right away, so an equivalent call is simply `h (y 7)`
- A function like `h`, which can be called "by value" just as well as lazily, is characterized as strict

Strictness

- Formally, a function h is strict if h e diverges for all non-terminating e
- In other words, h either always diverges, or it needs to <u>inspect</u> the value of its argument:
  - branch according to its constructor tag, or
  - feed it to a primitive operator, or
  - apply it to other arguments
- (For higher arities, we say a function is strict/non-strict in argument 1, 2, ...)

Strictness analysis

- Despite many examples of obviously strict functions, the strictness property of functions in general is <u>undecidable</u>
- Still, even a coarse <u>approximation</u> to strictness is beneficial to the efficiency of lazy languages
- Many safe strictness analysis techniques exist (and every lazy language compiler implements one)
- The classic approach is based on <u>abstract interpretation</u> (example follows Wadler, 1987)

Abstract interpretation

- Reduce computations over big or infinite value domains to abstract computations over small and <u>finite abstract domains</u>
- Iterate abstract function behavior until <u>fixpoint</u>
- Choose the abstract domains so that they reveal intersting program properties
- For strictness analysis, let the abstract domains capture varying degrees of <u>definedness</u>

Strictness analysis abstract interpretation

- Let the abstract domain of integers be
  $\top$ – any concrete integer value
  $\perp$ - the undefined (non-terminating) integer
- Let the abstract domain of integer lists be
  $\top_\in$ - a finite list with no undefined elements
  $\perp_\in$ - a finite list with some undefined elements
  ∞ - an infinite list (with an undefined tail)
  $\perp$ - the fully undefined list
- <u>Order</u> elements as depicted!

Strictness analysis abstract interpretation

- Let $x^{\#}$ be the abstraction of value x
- Some abstract values:

  $0^{\#} = \top \quad 1^{\#} = \top \quad 99999^{\#} = \top \quad \bot^{\#} = \bot$

  $[]^{\#} = \top_{\in} \quad (1{:}2{:}[])^{\#} = \top_{\in} \quad (1{:}\bot{:}[])^{\#} = \bot_{\in} \quad (1{:}\bot)^{\#} = \infty$
- Let $f^{\#}$ be the abstraction of function f
- Calculate abstract function tables using finite value enumeration, monotonicity, least upper / greatest lower bounds, fixpoint iteration, ...

Example

len = \us -> **case** us **of** [] -> 0; x:xs -> 0 + len xs

$x^\#$	$xs^\#$	$(x{:}xs)^\#$
$\top$	$\top_\in$	$\top_\in$
$\top$	$\bot_\in$	$\bot_\in$
$\top$	∞	∞
$\top$	$\bot$	∞
$\bot$	$\top_\in$	$\bot_\in$
$\bot$	$\bot_\in$	$\bot_\in$
$\bot$	∞	∞
$\bot$	$\bot$	∞

$us^\#$	(**case** us ...)$^\#$
$\top_\in$	$\top$
$\bot_\in$	$len^\#\ \top_\in$
∞	$len^\#\ \infty$
$\bot$	$\bot$

$len^\#_0$	
in	out
$\top_\in$	$\bot$
$\bot_\in$	$\bot$
∞	$\bot$
$\bot$	$\bot$

$len^\#_1$	
in	out
$\top_\in$	$\top$
$\bot_\in$	$\bot$
∞	$\bot$
$\bot$	$\bot$

$len^\#_2$	
in	out
$\top_\in$	$\top$
$\bot_\in$	$\top$
∞	$\bot$
$\bot$	$\bot$

$len^\#_3$	
in	out
$\top_\in$	$\top$
$\bot_\in$	$\top$
∞	$\bot$
$\bot$	$\bot$

fixpoint

Example

- Conclusions from the abstract interpretation:
  - len maps $\perp$ to $\perp$, so it's safe to evaluate its argument before the call
  - len maps ∞ to $\perp$, so it's also safe to evaluate <u>all tails</u> of the argument before the call
  - len maps $\perp_\in$ to $\top$, so it's <u>not</u> safe to evaluate any <u>elements</u> of the argument before the call

Summary

- Laziness is straightforward to implement, but efficiency relies heavily on optimizations
- Strictness analysis is particularly useful, classic technique is based on abstract interpretation
- Course summary:
  - Mapping a FL to C quite simple (modulo GC issues)
  - Challenge lies in exploiting source-to-source transformations (including type-based ones)
  - Hands-on experience is the only lasting value, complete your compiler projects!