Compiling functional languages

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

Lecture 6
Type-based optimizations

Johan Nordlander

Aim

- To give an introduction to the rich field of program optimization
- To show how the principles of formal type systems and type inference algorithms can provide structure to program transformations in general
- To explain some simple but effective type-based optimizations, together with a not so simple (and not so type-based!) one

Remark

- Optimization = "make optimal", taken literally
- Optimality of efficiency is a far-reaching goal...
- Common use: optimization = "improving efficiency"
- ... with an added "for most programs"
- That is: no optimality or improvement guarantees
- But correctness (preservation of meaning) is commonly assumed (and often formally proved)

Checking arities

- Recall the closure conversion algorithm:
  - Calling a known function = good code, arity-matching at compile-time
  - Calling a function variable = costly arity-inspecting loop at run-time
- Difference is that a variable reveals nothing about the arity of the hidden function
- But what if arities were part of types?

Function types

- The standard abstract type grammar:

  t ::= a | T | $t\ t$ | t -> t **data** Type = FunTy Type Type

- A <u>non-standard</u> type grammar:

  t ::= a | T | $t\ t$ | ts -> t **data** Type = FunTy [Type] Type

  parse t_1 -> t_2 -> t_3 as t_1,t_2 -> t_3

  parse t_1 -> (t_2 -> t_3) as t_1 -> (t_2 -> t_3)

  not equal

- Captures arities in function types
- Inflexibility may be compensated for by automatic insertion of <u>coercions</u>

An arity-sensitive type system

$$\frac{A \vdash e : t_1 \ldots t_n \texttt{->} t \qquad A \vdash e_i : t_i}{A \vdash e\ e_1 \ldots e_n : t}\,\text{App}$$

$$\frac{A, xs{:}ts \vdash e : t}{A \vdash \backslash xs \texttt{->} e : ts \texttt{->} t}\,\text{Abs}$$

$$\frac{x{:}\forall as.t \in A}{A \vdash x : [ts/as]t}\,\text{Var}$$

$$\frac{A \vdash e : t \qquad A, x{:}\sigma \vdash e' : t'}{A \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : t'}\,\text{Let}$$

where $\sigma = gen(t, A)$

An arity-matching type inference algorithm

$$\frac{\theta_1 A \vdash^{w} e : t \leadsto e' \qquad \theta_2(\theta_1 A) \vdash^{w} es : ts \leadsto es' \qquad \theta_3 \Vdash e'' : \theta_2 t < ts\text{->}a}{(\theta_3\theta_2\theta_1)A \vdash^{w} e\ es : \theta_3\, a \leadsto \underline{(e''\ e')}\ es'}\ \text{App}$$

where a is new

$$\frac{x{:}\forall as.qs\text{=>}t \in A}{[\,]A \vdash^{w} x : \theta t \leadsto x}\ \text{Var}$$

where θ=[bs/as] and bs are new

$$\frac{\theta(A, xs{:}as) \vdash^{w} e : t \leadsto e'}{\theta A \vdash^{w} \backslash xs\text{->}e : \theta as\text{->}t \leadsto \backslash xs\text{->}e'}\ \text{Abs}$$

where as are new

$$\frac{\theta_1 A \vdash^{w} e_1 : t \leadsto e_1' \qquad \theta_2(\theta_1 A, x{:}\sigma) \vdash^{w} e_2 : t' \leadsto e_2'}{(\theta_2\theta_1)A \vdash^{w} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t' \leadsto \textbf{let}\ x{=}e_1'\ \textbf{in}\ e_2'}\ \text{Let}$$

where σ = gen(t,$\theta_1 A$)

Arity-matching unification

Output (a substitution and a coercion)

General form: $\theta \Vdash e : t < t'$

Inputs (two types)

Main idea:

All function types can be <u>coerced</u> into each other by currying/uncurrying

All other types must be <u>unifiable</u>
(including function types nested inside datatype constructors)

Arity-matching unification

$$\frac{\theta_1 \Vdash \mathtt{e} : \mathtt{t} < \mathtt{ss'\text{->}s} \qquad\qquad \theta_2 \Vdash \mathtt{es} : \mathtt{ss} < \mathtt{ts}}{\theta_2\theta_1 \Vdash \mathtt{\backslash \underline{v}\text{->}\backslash(\underline{ys,ys'})\text{->}\ e\ (v\ es@ys)\ ys'} : \mathtt{\underline{ts\text{->}t}} < \mathtt{(\underline{ss,ss'})\text{->}s}}$$

$$\frac{\theta_1 \Vdash \mathtt{e} : \mathtt{ts'\text{->}t} < \mathtt{s} \qquad\qquad \theta_2 \Vdash \mathtt{es} : \mathtt{ss} < \mathtt{ts}}{\theta_2\theta_1 \Vdash \mathtt{\backslash \underline{v}\text{->}\backslash \underline{ys}\text{->}\ e\ (\backslash xs'\text{->}\ v\ (es@ys,xs'))} : \mathtt{\underline{(ts,ts')\text{->}t}} < \mathtt{\underline{ss}\text{->}s}}$$

$$\frac{\theta_1 \Vdash \mathtt{e} : \mathtt{t} < \mathtt{s} \qquad \theta_2 \Vdash \mathtt{es} : \mathtt{ss} < \mathtt{ts}}{\theta_2\theta_1 \Vdash \mathtt{\backslash \underline{v}\text{->}\backslash \underline{ys}\text{->}\ e\ (v\ es@ys)} : \mathtt{\underline{ts\text{->}t}} < \mathtt{\underline{ss}\text{->}s}} \qquad \frac{\mathtt{t} \overset{\theta}{\sim} \mathtt{s}}{\theta \Vdash \mathtt{\backslash x\text{->}x} : \mathtt{t} < \mathtt{s}}$$

($\mathtt{es@ys}$ = zipWith Apply1 $\mathtt{es\ ys}$)

Example

- Assume

A function of arity 1

```
map : (a -> b) -> [a] -> [b]
plus : Int,Int -> Int  =  \m n -> ...
```

- Then

$$\texttt{map plus xs} \ \leadsto\ \texttt{(e map) plus xs} \ =\ \texttt{map (\textbackslash y->\textbackslash x->plus y x) xs}$$

where

$$e \ =\ \backslash v \to \backslash y_1\ y_2 \to (\backslash x \to x)\ (v\ (e_1\ y_1)\ ((\backslash x \to x)\ y_2)) \ =\ \backslash v \to \backslash y_1\ y_2 \to v\ (e_1\ y_1)\ y_2$$

$$e_1 \ =\ \backslash v \to \backslash y \to (\backslash x \to x)\ (\backslash x \to v\ ((\backslash x \to x)\ y)\ x) \ =\ \backslash v \to \backslash y \to \backslash x \to v\ y\ x$$

- Because (ignoring the substitutions)

$$\dfrac{\Vdash\ \backslash x \to x : [b_2] < a_1 \qquad \dfrac{\Vdash\ e_2 : I \to I < b_2 \qquad \Vdash\ \backslash x \to x : b_1 < I}{\Vdash\ e_1 : I,I \to I < b_1 \to b_2} \qquad \Vdash\ \backslash x \to x : [I] < [b_1]}{\Vdash\ e : (b_1 \to b_2) \to [b_1] \to [b_2] < (I,I \to I) \to [I] \to a_1}$$

On arity-matching

- Generates seemingly complicated terms, but simple static evaluation is extremely beneficial
- Main benefit: completely removes the need for run-time arity tests and iterative loops at call sites
- Can be sligthly non-deterministic:
  \f x y -> (f x y, f x) : (a->b->c)->a->b->c or (a->(b->c))->a->b->c
  However, alternatives are mutually coercible
- Datatype constructor rigidity, [a->b->c] $\not<$ [a->(b->c)], can be loosened by generating fmap-like coercions
- Unavoidable rigidity: m (a->b->c) $\not<$ m (a->(b->c))

Region inference

- Advanced method replacing garbage collection by stack-like popping of memory regions (Talpin & Tofte, 1993)
- A type-based translation based on an <u>effect type system</u> (Lucassen & Gifford, 1988)
  - Basically HM; effects ≈ our contexts; but effects also annotate function types (subject to unification)
  - Can distinguish between the effect (memory read or write) of <u>creating</u> a function and <u>calling</u> it
- Unfortunately a rather complex technique...

Poor man's region inference

- Apply qualified types in non-standard fashion!
- Source language

  e ::= x | e es | \xs -> e | **let** x = e **in** e |
  fst e | snd e | (e,e)

- Target language

  Run e with some temporary regions

  e ::= ... | **letregion** ys **in** e | e^y

  Store e in region y

- Type language

  t ::= a | ts $->^t$ t | $(t,t)^t$

  σ ::= ∀as . qs $=>^t$ t

  Use type (variables) to track region demands

  Write effect

  q ::= Put t

Poor man's region inference

$$\frac{P|A \vdash e : ts \to^{r} t \leadsto e' \qquad P|A \vdash es : ts \leadsto es'}{P|A \vdash e\ es : t \leadsto e'\ es'}\text{App}$$

$$\frac{P|A,xs{:}ts \vdash e : t \leadsto e' \qquad P \Vdash y : \mathrm{Put}\ r}{P|A \vdash \backslash xs\text{->}e : ts \to^{r} t \leadsto (\backslash xs\text{->}e')^{y}}\text{Abs}$$

$$\frac{x{:}\forall as.\ qs \Rightarrow^{r} t \in A \qquad P \Vdash ys : [ts/as]qs,\ y : \mathrm{Put}\ r}{P|A \vdash x : [ts/as]t \leadsto (x\ ys)^{y}}\text{Var}$$

$$\frac{P,ys{:}qs|A,x{:}\sigma \vdash e_1 : t \leadsto e_1' \qquad P|A,x{:}\sigma \vdash e_2 : t' \leadsto e_2' \qquad P \Vdash y : \mathrm{Put}\ r}{P|A \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : t' \leadsto \mathbf{let}\ x = (\backslash ys\text{->}e_1')^{y}\ \mathbf{in}\ e_2'}\text{Let}$$

where $\sigma = \forall\ fv(qs,t)\backslash fv(A,r) \Rightarrow^{r} t$

Poor man's region inference

$$\frac{P|A \vdash e : (t,t')^{r} \leadsto e'}{P|A \vdash \mathsf{fst}\ e : t \leadsto \mathsf{fst}\ e'}\ \text{Fst}$$

$$\frac{P|A \vdash e : (t,t')^{r} \leadsto e'}{P|A \vdash \mathsf{snd}\ e : t' \leadsto \mathsf{snd}\ e'}\ \text{Snd}$$

$$\frac{P|A \vdash e_1 : t_1 \leadsto e_1' \quad P|A \vdash e_2 : t_2 \leadsto e_2' \quad P \Vdash y : \mathsf{Put}\ r}{P|A \vdash (e_1,e_2) : (t_1,t_2)^{r} \leadsto (e_1',e_2')^{y}}\ \text{Pair}$$

$$\frac{P,ys{:}qs|A \vdash e : t \leadsto e' \quad fv(qs) \cap fv(t,A,P) = \varnothing \quad t \text{ is first-order}}{P|A \vdash e : t \leadsto \textbf{letregion}\ ys\ \textbf{in}\ e'}\ \text{Region}$$

A poor man's region inference algorithm

- Straightforward from inference system and algorithm W for qualified types
- Substantially simpler than T&T (no effect variables or unification of effect sets)
- Depends on inference of limited polymorphic recursion (just like T&T). Can probably reuse their iterative approach too
- Separate problem: finding out maximum size of regions (shared with T&T)

Example

(from Talpin & Tofte 1993)

let fib = \x -> **if** x <= 1 **then** 1 **else** fib (x-2) + fib (x-1)
in fib 15

$\rightsquigarrow$

(using type scheme fib : $\forall$a,b . Put b => Int^{a} -> Int^{b})

$$\begin{aligned}
&\textbf{let}\ \text{fib} = (\backslash^{y4} \to (\backslash x \to \textbf{if}\ x \le 1\ \textbf{then}\ 1^{y4}\ \textbf{else} \\
&\qquad \underbrace{(\underbrace{(\text{fib}^{\ y5})^{y7}\ (\underbrace{x-2^{y9})^{y8}}_{y9}}_{y7,y8} + \underbrace{(\text{fib}^{\ y6})^{y10}\ (\underbrace{x-1^{y12})^{y11}}_{y12}}_{y10,y11})^{y4}}_{y5,y6})^{y3})^{y2} \\
&\textbf{in}\ \underbrace{(\text{fib}^{\ y1})^{y14}\ 15^{y13}}_{y13,y14}
\end{aligned}$$

y2,y3

(region scopes only indicated graphically)

Limitation

Memory reads not tracked by types, correctness relies on read demands coinciding with evaluation. Only true for first-order data...

$$\dfrac{\dfrac{\dfrac{P|A \vdash T : B \qquad P|A \vdash F : B \qquad P \Vdash y_1{:}Put\ a_1}{P|A \vdash (T,F)^{y1} : (B,B)^{a1}} \qquad \dfrac{\dfrac{P|A,y{:}B \vdash x : (B,B)^{a1}}{P|A,y{:}B \vdash fst\ x : B} \qquad P \Vdash y_2{:}Put\ a_2}{P|A \vdash \backslash y\text{->}fst\ x)^{y2} : B \text{ ->}^{a2}\ B}}{P|\ \vdash \mathbf{let}\ x = (T,F)^{y1}\ \mathbf{in}\ (\backslash y\text{->}fst\ x)^{y2} : B \text{ ->}^{a2}\ B} \qquad \{a_1\} \cap \{a_2\} = \emptyset}{\cancel{y_2{:}Put\ a_2 |\ \vdash \mathbf{letregion}\ y1\ \mathbf{in}\ \mathbf{let}\ x = (T,F)^{y1}\ \mathbf{in}\ (\backslash y\text{->}fst\ x)^{y2} : B \text{ ->}^{a2}\ B}}$$

$P = y_1{:}Put\ a_1,\ y_2{:}Put\ a_2$

$A = x{:}(B,B)^{a1}$

Not first-order!

Consequence: Regions only used while <u>creating</u> a function must still be kept until function can no longer be called

Static evaluation

- Foolish not to transform away at compile-time:

  $[\![$ 1 + 2 $]\!]$ = 3

  $[\![$ (\x->x+2) y $]\!]$ = y+2

  $[\![$ **case** K_1 y **of** K_1 x -> e_1; K_2 -> e_2 $]\!]$ = [y/x]e_1

- Less clear-cut, but potentially beneficial:

  $[\![$ **let** x = K y z **in** e $]\!]$ = [K y z/x] e

  $[\![$ **let** x = K y z **in** ... **case** x **of** ... $]\!]$ = ...

  $[\![$ **let** x = K y z **in** ... **case** x **of** ... (x,x,x,x,x,x) $]\!]$ = ...

Static evaluation

- Function inlining: saves overhead; size increase?

  ⟦ abs x ⟧ = **if** x < 0 **then** negate x **else** x

  ⟦ f . g ⟧ = \x -> f (g x)

- Recursive function specialization: termination?

  ⟦ map (\x->x+1) xs ⟧ =

  case xs **of** [] -> []; y:ys -> y+1 : ⟦ map (\x->x+1) ys ⟧

- Might be desirable:

  ⟦ map (\x->x+1) xs ⟧ =

  let f = \xs -> **case** xs **of** []->[]; y:ys -> y+1 : f ys **in** f xs

Supercompilation

- A generic technique for evaluation at compile-time
  - application of a lambda-abstraction
  - case of a constructor expression
  - lookup of a known variable
  - primitive applied to literals
- Key features:
  - systematic application to nested sub-expressions
  - recursive function specialization with termination
  - generation of equivalences from branching conditions
- Recent: controlled speed/size trade-off

Supercompilation

Selected rules

application or case context

$$\llbracket\ R\langle \textbf{case}\ K_j\ es\ \textbf{of}\ K_i\ xs_i \to e_i \rangle\ \rrbracket^A = \llbracket\ R\langle \textbf{let}\ xs_j = es\ \textbf{in}\ e_j \rangle\ \rrbracket^A$$

$$\llbracket\ R\langle \textbf{case}\ x\ \textbf{of}\ K_i\ xs_i \to e_i \rangle\ \rrbracket^A = \textbf{case}\ x\ \textbf{of}\ K_i\ xs_i \to \llbracket\ [K_i\ xs_i/x]R\langle e_i \rangle\ \rrbracket^A$$

$$\llbracket\ R\langle (\backslash xs \to e)\ es \rangle\ \rrbracket^A = \llbracket\ R\langle \textbf{let}\ xs = es\ \textbf{in}\ e \rangle\ \rrbracket^A$$

$$\llbracket\ R\langle \backslash xs \to e \rangle\ \rrbracket^A = R\langle\ \backslash xs \to \llbracket\ e\ \rrbracket^A\ \rangle$$

$$\llbracket\ R\langle \textbf{let}\ x = e\ \textbf{in}\ e' \rangle\ \rrbracket^A = \llbracket\ R\langle [e/x]\ e' \rangle\ \rrbracket^A \qquad \text{if } e' \text{ linear (\& strict) in } x$$

$$= \textbf{let}\ x = \llbracket e \rrbracket^A\ \textbf{in}\ \llbracket R\langle e' \rangle \rrbracket^A \qquad \text{otherwise}$$

alpha-equivalence

$$\llbracket\ R\langle f\ es \rangle\ \rrbracket^A = g\ xs \qquad \text{if } \exists g.\ A(g) \approx \backslash xs \to R\langle f\ es \rangle$$

$$= R\langle f\ \llbracket\ es\ \rrbracket^A \rangle \qquad \text{if } \exists g.\ A(g) \trianglelefteq \backslash xs \to R\langle f\ es \rangle$$

homeomorphic embedding

$$= \textbf{let}\ g = \backslash xs \to \llbracket R\langle e_f \rangle \rrbracket^{A'}\ \textbf{in}\ g\ xs \qquad \text{otherwise}$$

where $xs = fv(R, es)$, $f = e_f$ in top-decl, g new, $A' = A, g = \backslash xs \to R\langle f\ es \rangle$

System F

- Our core language

```
e ::= x | e e | \x -> e | let x = e in e
t ::= a | T | t t | t -> t
σ ::= ∀a . σ | t
```

Explicit signatures

- System F

```
e ::= x | e e | \x:t -> e | let x:t = e in e |
      e {t} | /\a -> e
t ::= a | T | t t | t -> t | ∀a . t
```

Type application and abstraction

No type/scheme distinction

- Expressive, but type-reconstruction is undecidable

System F

$$\frac{A \vdash e : t' \to t \qquad A \vdash e' : t'}{A \vdash e\ e' : t}\ \text{App}$$

$$\frac{A, x{:}t' \vdash e : t}{A \vdash \backslash x{:}t' \to e : t' \to t}\ \text{Abs}$$

$$\frac{x{:}t \in A}{A \vdash x : t}\ \text{Var}$$

$$\frac{A \vdash e : t \qquad A, x{:}t \vdash e' : t'}{A \vdash \textbf{let}\ x{:}t = e\ \textbf{in}\ e' : t'}\ \text{Let}$$

Encodeable as (\x:t -> e') e

$$\frac{A \vdash e : \forall a.t}{A \vdash e\ \{t'\} : [t'/a]t}\ \text{Inst}$$

$$\frac{A \vdash e : t \qquad a \notin fv(A)}{A \vdash /\backslash a.e : \forall a.t}\ \text{Gen}$$

System F as a core

- Allows easy computation of types for all subexpressions (no unification/substitution threading)
- Needed if
  - target language is typed (no casts)
  - info on ptr/non-ptr distionction required by GC
  - polymorphic code must be duplicated for (some) non-ptr instances
- Good for trapping bugs in transformation passes!

Relation to System F

$$\frac{A \vdash e_1 : t' \to t \leadsto e_1' \qquad A \vdash e_2 : t' \leadsto e_2'}{A \vdash e_1\ e_2 : t \leadsto e_1'\ e_2'}\ \text{App}$$

$$\frac{A, x{:}t' \vdash e : t \leadsto e'}{A \vdash \backslash x \to e : t' \to t \leadsto \backslash x{:}\underline{t'} \to e'}\ \text{Abs}$$

$$\frac{x{:}\forall as.t \in A}{A \vdash x : [ts/as]t \leadsto \underline{x\ \{ts\}}}\ \text{Var}$$

$$\frac{A \vdash e_1 : t \leadsto e_1' \qquad A, x{:}\sigma \vdash e_2 : t' \leadsto e_2'}{A \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t' \leadsto \textbf{let}\ x{:}\underline{\sigma} = \underline{/\backslash as \to}\ e_1'\ \textbf{in}\ e_2'}\ \text{Let}$$

where $\sigma\ =\ gen(t, A) = \forall as\ .\ t$

Translation into System F

$$\frac{\theta_1 A \vdash^{W} e_1 : t \rightsquigarrow e_1' \qquad \theta_2(\theta_1 A) \vdash^{W} e_2 : t' \rightsquigarrow e_2' \qquad \theta_2 t \overset{\theta_3}{\sim} t' \text{->} a}{(\theta_3\theta_2\theta_1)A \vdash^{W} e_1\, e_2 : \theta_3\, a \rightsquigarrow \theta_3(\theta_2 e_1'\; e_2')}\ \text{App}$$

where a is new

$$\frac{x{:}\forall as.t \in A}{[\,]A \vdash^{W} x : [bs/as]t \rightsquigarrow \underline{x\,\{bs\}}}\ \text{Var}$$

where bs are new

$$\frac{\theta(A, x{:}a) \vdash^{W} e : t \rightsquigarrow e'}{\theta A \vdash^{W} \backslash x\text{->}e : \theta a\text{->}t \rightsquigarrow \backslash x{:}\theta a\text{->}e'}\ \text{Abs}$$

where a is new

$$\frac{\theta_1 A \vdash^{W} e_1 : t \rightsquigarrow e_1' \qquad \theta_2(\theta_1 A, x{:}\sigma) \vdash^{W} e_2 : t' \rightsquigarrow e_2'}{(\theta_2\theta_1)A \vdash^{W} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t' \rightsquigarrow \textbf{let}\ x{:}\theta_2\sigma\ = \underline{/\backslash as}\ \text{->}\ \theta_2 e_1'\ \textbf{in}\ e_2'}\ \text{Let}$$

where $\sigma = gen(t, \theta_1 A) = \forall as\ .\ t$

Summary

- Many optimizing transformations rely on non-local information about identifiers, and abstraction over the transformation state from which functions are invoked
- Non-standard types and/or predicate contexts can help structuring such problems as term-transforming type-inference algorithms
- Compile-time reductions (incl. function inlining) can be systematically formulated in terms of supercompilation
- Type-preservation of such transformations can easily be checked by using System F as the core language