Once Upon a Polymorphic Type

UNIVERSITY OF CAMBRIDGE

Keith Wansbrough

Computer Laboratory
University of Cambridge

`kw217@cl.cam.ac.uk`
`http://www.cl.cam.ac.uk/users/kw217/`

Simon Peyton Jones

Microsoft Research
Cambridge

20 January, 1999

Why usage analysis?

UNIVERSITY OF CAMBRIDGE

Problem:

- Lazy evaluation (call-by-need) is useful but slow

Solutions:

- *Strictness analysis:* convert call-by-need to call-by-*value*
- *Usage analysis:* convert call-by-need to call-by-*name*

Lazy evaluation

$$\begin{array}{ll} \textsf{let} & x = 3 + 4 \\ & y = 5 + 6 \\ \textsf{in} & x + x + y \end{array}$$

Heap, before and after:

$$\begin{array}{|l|}\hline x : \boxed{3+4} \\ y : \boxed{5+6} \\ \hline \end{array} \quad \leadsto \quad \begin{array}{|l|}\hline x : \boxed{\;\;7} \\ y : \boxed{11} \\ \hline \end{array}$$

unnecessary update

Unnecessary updates mean excess memory traffic. ✘

The goal

UNIVERSITY OF CAMBRIDGE

Identify variables and subexpressions that will be evaluated *at most once.*

Usage analysis enables other optimisations too

UNIVERSITY OF CAMBRIDGE

Inlining:

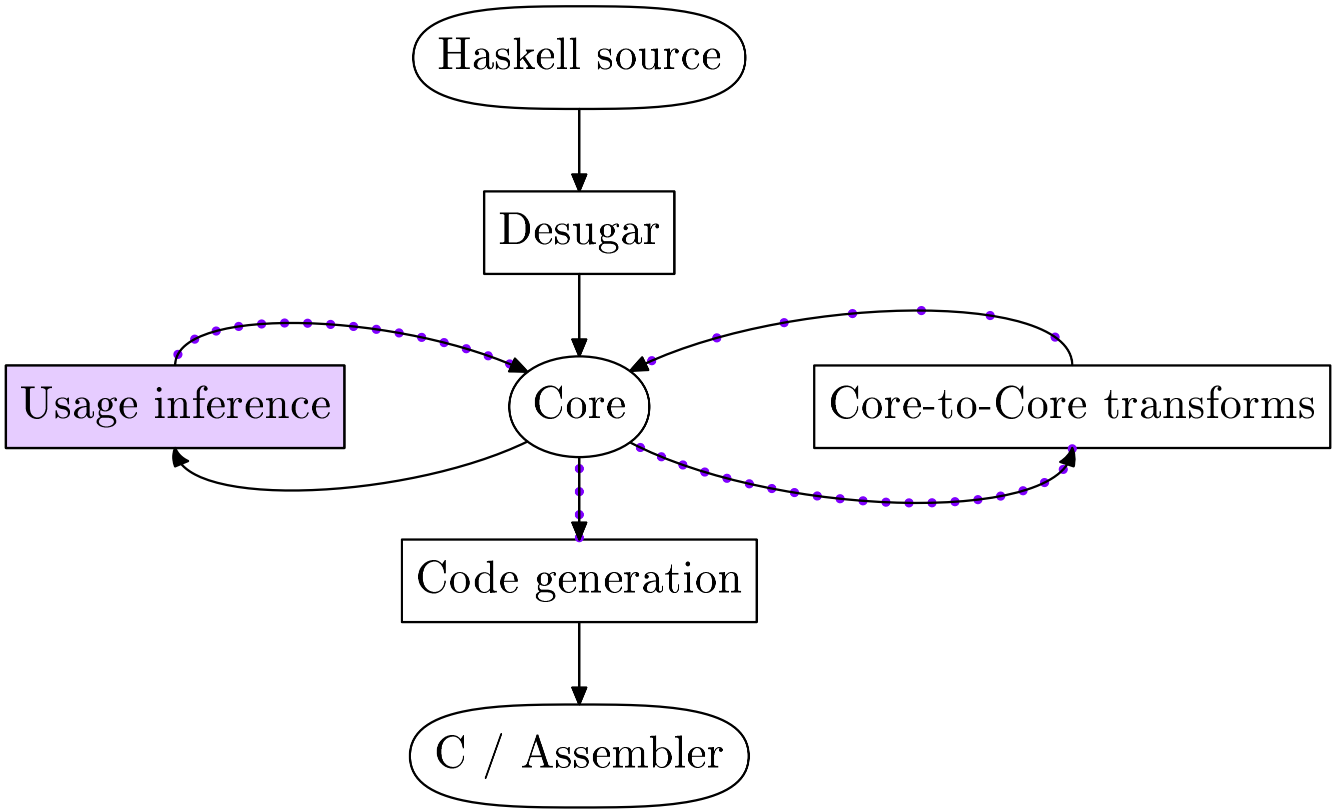
$$\begin{array}{ll} \mathsf{let} & x = e \\ \mathsf{in} & \lambda y\,.\,\mathsf{case}\ x\ \mathsf{of} \\ & \quad\quad \ldots \to \ldots \end{array} \quad\leadsto\quad \begin{array}{l} \lambda y\,.\,\mathsf{case}\ e\ \mathsf{of} \\ \quad\quad \ldots \to \ldots \end{array}$$

Here x occurs in none of the case alternatives. We avoid constructing a thunk for x entirely, by inlining e.

Always valid, but serious slow-down if lambda applied more than once; *'work-safe'* if lambda applied (*used*) at most once.

Several other optimisations can similarly benefit from usage information.

Plan of attack



Usage inference provides additional information at Core level to guide optimising transformations and code generation.

How do we do it?

UNIVERSITY OF CAMBRIDGE

It seems that we should simply be able to count syntactic occurrences. But this is not enough.

$$
\begin{array}{lll}
\mathsf{let} & y = 1 + 2 & \\
\mathsf{in} & \mathsf{let} & f = \lambda x\,.\,x + y \\
 & \mathsf{in} & f\;3 + f\;4
\end{array}
$$

Here y appears once in its scope. But it is used *twice*, once for each call to f.

The usage of y depends on the usage of f.

Types

We represent usage information in the *types* of expressions:

$$
\begin{array}{rcl}
42 & : & \mathsf{Int}^{\omega} \\[2ex]
\lambda x : \mathsf{Int}^{1} \,.\, x & : & (\mathsf{Int}^{1} \to \mathsf{Int}^{1})^{\omega} \\[2ex]
\lambda x : \mathsf{Int}^{1} \,.\, \lambda y : \mathsf{Int}^{1} \,.\, x + y & : & (\mathsf{Int}^{1} \to (\mathsf{Int}^{1} \to \mathsf{Int}^{\omega})^{1})^{\omega}
\end{array}
$$

$$
\begin{array}{lllcl}
\mathsf{let} & x : \mathsf{Int}^{\omega} & = 3 + 4 & & \\
 & y : \mathsf{Int}^{1} & = 5 + 6 & & \\
\mathsf{in} & x + x + y & & : & \mathsf{Int}^{\omega}
\end{array}
$$

Type syntax

UNIVERSITY OF CAMBRIDGE

Types (unannotated)	τ	$::=$	$T\ \overline{\tau_k}$
		$\mid$	$\sigma_1 \to \sigma_2$
		$\mid$	$\forall \alpha\ .\ \tau$
		$\mid$	α
Types (annotated)	σ	$::=$	τ^u
Usages	u	$::=$	$1 \mid \omega$

for example, $\Big((List\ \mathsf{Int})^{\,1} \to \mathsf{Int}^{\omega}\Big)^{\,\omega}$.

Type rules

Type judgements are of the form

$$\Gamma \vdash e : \sigma$$

context expression type

For example, the rule for addition:

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}^{u_1} \qquad \Gamma \vdash e_2 : \mathsf{Int}^{u_2}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}^{u_3}} \ (\vdash\text{-}\textsc{PrimOp})$$

Type rules for *UsageSP* – 1: Functions

UNIVERSITY OF CAMBRIDGE

$$\frac{\begin{array}{ll} \Gamma, x : \sigma_1 \vdash e : \sigma_2 & \\ occur(x, e) > 1 \Rightarrow |\sigma_1| = \omega & \text{(multiple occurrence)} \\ occur(y, e) > 0 \Rightarrow |\Gamma(y)| \geq u \quad \text{for all } y \in \Gamma & \text{(free variables)} \end{array}}{\Gamma \vdash \lambda x : \sigma_1 \,.\, e : (\sigma_1 \to \sigma_2)^u} \;(\vdash\text{-}\textsc{Abs})$$

$occur(\cdot, \cdot)$ is defined *syntactically*.

$$\begin{array}{lll} \text{let} & y : \mathsf{Int}^\omega = 1 + 2 & \\ \text{in} & \text{let} \quad f : (\mathsf{Int}^1 \to \mathsf{Int}^1)^\omega = \lambda x : \mathsf{Int}^1 \,.\, x + y & \\ & \text{in} \quad f\ 3 + f\ 4 & : \quad \mathsf{Int}^\omega \end{array}$$

Here $occur(x, x + y) = 1$, $occur(y, x + y) = 1$, $occur(f, f\ 3 + f\ 4) = 2$.

Three design decisions

UNIVERSITY OF
CAMBRIDGE

- Type polymorphism:
  - Should type variables be annotated or unannotated? What is the usage of a type abstraction?
- Algebraic data structures:
  - How should constructor applications be typed?
- The poisoning problem:
  - How can we avoid equating usages of all arguments to a common function?

Design decision 1: Type polymorphism

- Range of type variables:
  - Should type arguments be annotated or unannotated?

$$f : (\forall\alpha \,.\, \alpha \to (\alpha \to (\alpha, \alpha, \alpha)^1)^1)^\omega$$

$$\text{or} \quad f : (\forall\alpha \,.\, \alpha^1 \to (\alpha^\omega \to (\alpha, \alpha, \alpha)^1)^1)^\omega \quad ?$$

- Type of type abstractions:
  - Given $\alpha \vdash e : \tau^u$, what is $\vdash \Lambda\alpha \,.\, e : ?$

Type rules for *UsageSP* – 3: Type polymorphism

UNIVERSITY OF CAMBRIDGE

Type abstractions and applications are treated as 'transparent' for the purposes of usage annotation, since operationally they have no significance. These rules simply lift and lower the usage annotation.

$$\frac{\Gamma, \alpha \vdash e : \tau^u}{\Gamma \vdash \Lambda\alpha\ .\ e : (\forall\alpha\ .\ \tau)^u}\ (\vdash\text{-}\textsc{TyAbs})$$

$$\frac{\Gamma \vdash e : (\forall\alpha\ .\ \tau_2)^u}{\Gamma \vdash e\ \tau_1 : (\tau_2[\alpha := \tau_1])^u}\ (\vdash\text{-}\textsc{TyApp})$$

Design decision 2: Data structures

UNIVERSITY OF CAMBRIDGE

Our language features user-defined algebraic data types such as

$$\begin{aligned} \texttt{data}\ \mathit{Tree}\ \alpha &= \mathit{Branch}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha) \\ &\mid \mathit{Leaf}\ \alpha \end{aligned}$$

How should these be typed?

Data structures: First attempt

If we treat data constructors as normal functions, what usages should we place on the arguments and result?

$$Branch : \forall\alpha\ .\ (Tree\ \alpha)^{?} \to (\alpha^{?} \to ((Tree\ \alpha)^{?} \to (Tree\ \alpha)^{?})^{?})^{?}$$

To make $Branch$ universally applicable, we must set $? = \omega$. ✘ ... inaccurate.

The usages $?$ really depend on *how the constructed data is used*, not on the constructor itself.

The tantalising opportunity

$$
\begin{array}{ll}
\mathsf{let} & f \;:\; (\mathsf{Int}^{\omega} \to (\mathsf{Int}^{\omega} \to (\mathsf{Int}, \mathsf{Int})^{1})^{\omega})^{\omega} \\
 & f = \lambda x : \mathsf{Int}^{\omega} \;.\; \lambda y : \mathsf{Int}^{\omega} \;.\; \begin{array}[t]{ll} \mathsf{let} & p = \ldots \\ & q = \ldots \\ \mathsf{in} & (p, q) \end{array} \\
\mathsf{in} & \mathsf{case}\ f\ x\ y\ \mathsf{of} \\
 & \quad (p, q) \to p + q
\end{array}
$$

Each component of the pair returned by f is used only once. Hence p and q need not be updated on evaluation.

How can we propagate this information from the usage site (the case expression) to the construction site in f?

Data structure usage propagation – 1

We propagate usage information through the type of the constructed data. There are a number of alternatives here.

$$\text{data } \mathit{Pair}\ \alpha\ \beta = (,)\ \alpha\ \beta \qquad \text{data } \mathit{Tree}\ \alpha = \mathit{Branch}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha) \mid \mathit{Leaf}\ \alpha$$

1. Give usage annotations for each constructor argument explicitly in the type:

$$(,)\ 1\ 1\ \mathsf{Int}\ \mathsf{Int}\ 3\ 4 \qquad \mathit{Branch}\ \omega\ 1\ \omega\ 1\ \mathsf{Int}\ t_1\ 3\ t_2 \qquad \text{(typical application)}$$

$$((,)\ \mathsf{Int}^1\ \mathsf{Int}^1)^u \qquad (\mathit{Branch}\ (\mathit{Tree}\ \omega\ 1\ \omega\ 1\ \mathsf{Int})^\omega\ \mathsf{Int}^1\ (\mathit{Tree}\ \omega\ 1\ \omega\ 1\ \mathsf{Int})^\omega)^u \mid (\mathit{Leaf}\ \mathsf{Int}^1)^u \qquad \text{(effective type)}$$

This is the most general approach, but it is expensive.

Data structure usage propagation – 2

2. Attach usage annotations to each type argument [BS96]:

$$(,)\ \mathsf{Int}^1\ \mathsf{Int}^1\ 3\ 4 \qquad\qquad \mathit{Branch}\ \mathsf{Int}^1\ t_1\ 3\ t_2$$

$$((,)\ \mathsf{Int}^1\ \mathsf{Int}^1)^u \qquad\qquad (\mathit{Branch}\ (\mathit{Tree}\ \mathsf{Int}^1)^?\ \mathsf{Int}^1\ (\mathit{Tree}\ \mathsf{Int}^1)^?)^u$$

$$\mid (\mathit{Leaf}\ \mathsf{Int}^1)^u$$

3. Assume all constructor arguments will be used more than once.

$$(,)\ \mathsf{Int}\ \mathsf{Int}\ 3\ 4 \qquad\qquad \mathit{Branch}\ \mathsf{Int}\ t_1\ 3\ t_2$$

$$((,)\ \mathsf{Int}^\omega\ \mathsf{Int}^\omega)^u \qquad\qquad (\mathit{Branch}\ (\mathit{Tree}\ \mathsf{Int})^\omega\ \mathsf{Int}^\omega\ (\mathit{Tree}\ \mathsf{Int})^\omega)^u$$

$$\mid (\mathit{Leaf}\ \mathsf{Int}^\omega)^u$$

Data structure usage propagation – 3

4. Identify usage annotations for each constructor argument with the overall usage of the constructed data.

$$(,)\ \mathsf{Int}\ \mathsf{Int}\ 3\ 4 \qquad\qquad Branch\ \mathsf{Int}\ t_1\ 3\ t_2$$

$$((,)\ \mathsf{Int}^u\ \mathsf{Int}^u)^u \qquad\qquad (Branch\ (Tree\ \mathsf{Int})^u\ \mathsf{Int}^u\ (Tree\ \mathsf{Int})^u)^u$$

$$\mid\ (Leaf\ \mathsf{Int}^u)^u$$

We choose this solution because it catches the common cases but costs relatively little in terms of implementation.

Type rules for *UsageSP* – 4: Data structures

$$\frac{\text{data } T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}} \qquad \Gamma \vdash e_j : \sigma'_{ij} \quad \sigma'_{ij} \preccurlyeq (\tau_{ij}[\overline{\alpha_k := \tau_k}])^u \quad \text{for all } j}{\Gamma \vdash C_i\ \overline{\tau_k}\ \overline{e_j} : (T\ \overline{\tau_k})^u} \ (\vdash\text{-}\textsc{Con})$$

$$\frac{\text{data } T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}} \qquad \Gamma \vdash e : (T\ \overline{\tau_k})^u \qquad \Gamma \vdash e_i : \sigma'_i \quad \sigma'_i \preccurlyeq (\overline{(\tau_{ij}[\overline{\alpha_k := \tau_k}])^u} \to \sigma)^1 \quad \text{for all } i}{\Gamma \vdash \text{case } e \text{ of } \overline{C_i \to e_i} : \sigma} \ (\vdash\text{-}\textsc{Case})$$

Inference algorithm

1. Annotate program:

$$
\begin{array}{lll}
(\mathsf{let} & y : \mathsf{Int}^{u_1} = 1 + 2 & \\
\;\mathsf{in} & \mathsf{let}\ f\ :\ (\mathsf{Int}^{u_2} \to \mathsf{Int}^{u_3})^{u_4} & \\
 & \qquad\quad = \lambda x : \mathsf{Int}^{u_5}\ .\ x + y & \\
 & \mathsf{in}\quad f\ 3 + f\ 4)^{u_6} &
\end{array}
$$

2. Collect constraints: $\rightsquigarrow \{u_4 = \omega, u_4 \leq u_1, u_5 \leq u_2\}$

3. Find optimal solution: $\rightsquigarrow \{u_1 \mapsto \omega, u_2 \mapsto 1, u_3 \mapsto 1,$
$u_4 \mapsto \omega, u_5 \mapsto 1, u_6 \mapsto 1\}$

- A solution always exists (simply set all $u_i = \omega$).
- We choose to maximise the number of 1 annotations, calling this the 'optimal' annotation.
- Complexity is *approx. linear* in the size of the (typed) program.

Soundness

UNIVERSITY OF CAMBRIDGE

Claim: Thunks marked 1 are used at most once.

Proof strategy:

1. Provide an operational semantics expressing sharing and thunks.
2. Ensure evaluation becomes 'stuck' if we use a thunk more than once.
3. Show well-typed programs never become stuck.

Typed operational semantics

UNIVERSITY OF CAMBRIDGE

We base our operational semantics on Launchbury's natural semantics for lazy evaluation. This semantics is *untyped*, since types are deleted at runtime (our language does not have a `typecase` construct).

However, for the purposes of the proof it is convenient to annotate this semantics with *types*.

Dealing with polymorphism is *not* straightforward. Since we allow evaluation under a type lambda, it is possible for a `let` binding to create a thunk with a free type variable; if this thunk is naïvely placed in the heap the type variable becomes unbound. The solution is given in the paper.

Related work

- Non-type-based usage analysis:
  - Goldberg; Marlow, Gill (GHC)
- Type-based usage analysis:
  - Linear types (Girard *et. al.*), affine types (Jacobs *et. al.*)
  - Clean (Barendsen *et. al.*): *uniqueness* analysis (dual to usage?); subsumption, data types, ML-polymorphism
  - Turner, Mossin, and Wadler: *Once Upon A Type*
  - extended by Mogensen: subsumption, data types
  - extended by Gustavsson: update markers

Conclusion

UNIVERSITY OF
CAMBRIDGE

- The problem: unnecessary updates.
- The solution: *UsageSP* . . .
  - a *type-based analysis*
  - for a *realistic language*
  - that is *efficiently computable*
  - and has been *proven sound.*

Future work

UNIVERSITY OF
CAMBRIDGE

- Complete the implementation of the analysis in the Glasgow Haskell Compiler.
- Investigate strictness, absence, uniqueness analyses in the same framework.
- Investigate optimisations enabled by the analysis, and prove ‘work-safety’ results for them.