UNIVERSITY OF CAMBRIDGE

Once Upon a Polymorphic Type

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?

Problem:

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

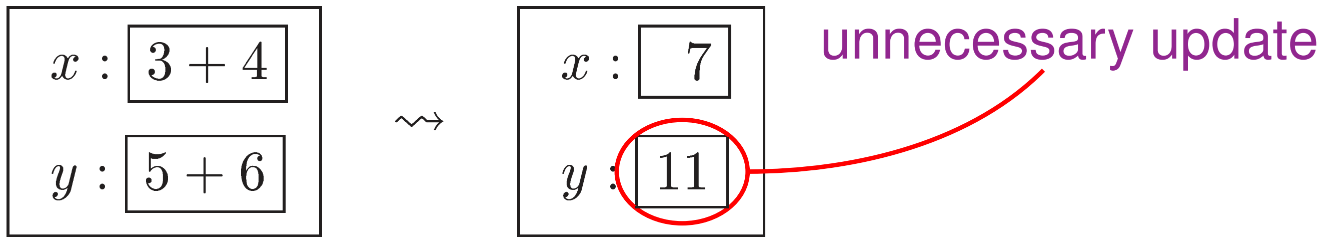
Solutions:

- *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:



Unnecessary updates mean excess memory traffic. ✘

The goal

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

Usage analysis enables other optimisations too

Inlining:

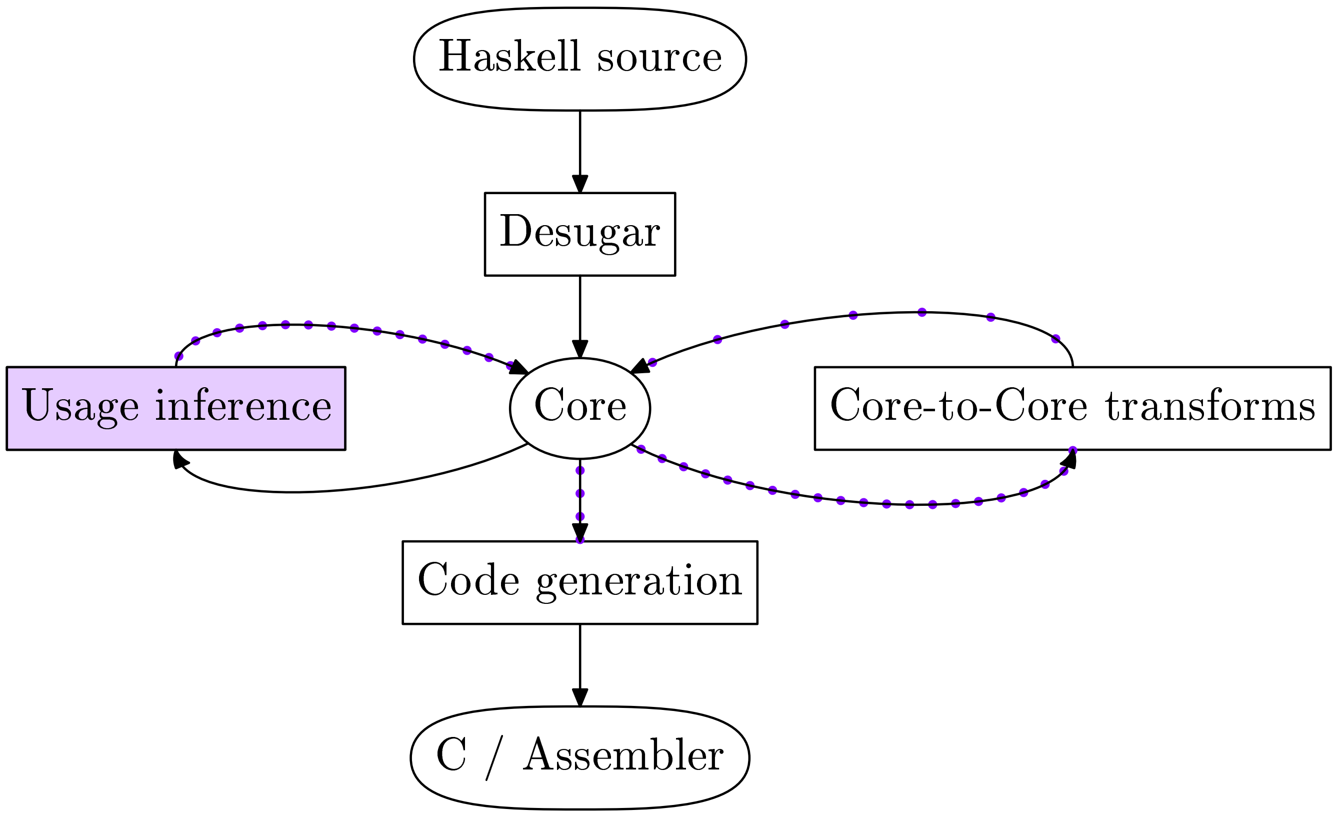
$$\begin{tikzcd}[column sep=small] \text{let} & x = \boxed{e} \arrow[d, bend left] & & \lambda y\ .\ \text{case}\ e\ \text{of} \\ \text{in} & \lambda y\ .\ \text{case}\ x\ \text{of} & \rightsquigarrow & \ldots \to \ldots \\ & \ldots \to \ldots & & \end{tikzcd}$$

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?

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

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

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:

$$42 \quad : \quad \mathsf{Int}^{\omega}$$

$$\lambda x : \mathsf{Int}^{1} \,.\, x \quad : \quad (\mathsf{Int}^{1} \to \mathsf{Int}^{1})^{\omega}$$

$$\lambda x : \mathsf{Int}^{1} \,.\, \lambda y : \mathsf{Int}^{1} \,.\, x + y \quad : \quad (\mathsf{Int}^{1} \to (\mathsf{Int}^{1} \to \mathsf{Int}^{\omega})^{1})^{\omega}$$

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

Type syntax

UNIVERSITY OF CAMBRIDGE

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

for example, $\left((\mathit{List}\ \mathsf{Int})^{\,1} \rightarrow \mathsf{Int}^{\omega} \right)^{\,\omega}$.

Type rules

Type judgements are of the form

$$\begin{tikzcd}[column sep=small] \Gamma & \vdash & e & : & \sigma \\ \text{context} \arrow[u] & & \text{expression} \arrow[u] & & \text{type} \arrow[u] \end{tikzcd}$$

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

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

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

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

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?

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

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

- 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

UNIVERSITY OF CAMBRIDGE

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

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

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

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

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.

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

$$\mid \mathit{Leaf}\ \alpha$$

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

UNIVERSITY OF CAMBRIDGE

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

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

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

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

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

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

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

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

Data structure usage propagation – 3

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

$(,)\ \mathsf{Int}\ \mathsf{Int}\ 3\ 4$

$((,)\ \mathsf{Int}^u\ \mathsf{Int}^u)^u$

$\mathit{Branch}\ \mathsf{Int}\ t_1\ 3\ t_2$

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

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

$$\text{data } T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}}$$

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

$$\text{data } T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}}$$

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

Design decision 3: The poisoning problem

$$\begin{array}{ll}
\textsf{let} & f \;:\; (\textsf{Int}^{?} \to \textsf{Int}^{1})^{\omega} \\
 & f = \lambda x : \textsf{Int}^{?} \;.\; x + 1 \\
 & a \;:\; \textsf{Int}^{\omega} \\
 & a = 2 + 3 \\
 & b \;:\; \textsf{Int}^{?} \\
 & b = 5 + 6 \\
\textsf{in} & a + (f\ a) + (f\ b)
\end{array}$$

$? = \omega$ ✘ ... bad for b

$? = 1$ ✘ ... bad for a

How can we make this work?

Solution A: Usage polymorphism

$$
\begin{array}{ll}
\mathsf{let} & f \;:\; \forall u\,.\,(\mathsf{Int}^u \to \mathsf{Int}^1)^\omega \\
 & f = \Lambda u\,.\,\lambda x : \mathsf{Int}^u\,.\,x + 1 \\[1ex]
 & a \;:\; \mathsf{Int}^\omega \\
 & a = 2 + 3 \\[1ex]
 & b \;:\; \mathsf{Int}^1 \\
 & b = 5 + 6 \\[1ex]
\mathsf{in} & a + (f\ \omega\ a) + (f\ 1\ b)
\end{array}
$$

Here f is polymorphic in the usage of its first argument. It is applied once at usage ω, and once at usage 1.

Solution B: Subsumption

$$
\begin{array}{ll}
\mathsf{let} & f \;:\; (\mathsf{Int}^{1} \to \mathsf{Int}^{1})^{\omega} \\
 & f = \lambda x : \mathsf{Int}^{1} \,.\, x + 1 \\[1ex]
 & a \;:\; \mathsf{Int}^{\omega} \\
 & a = 2 + 3 \\[1ex]
 & b \;:\; \mathsf{Int}^{1} \\
 & b = 5 + 6 \\[1ex]
\mathsf{in} & a + (f\ a) + (f\ b)
\end{array}
$$

Here f's type reflects its single use of its argument. An implicit subsumption rule allows the application of f (with argument type Int^{1}) to a (of type Int^{ω}): $\mathsf{Int}^{\omega} \preccurlyeq \mathsf{Int}^{1}$. This is safe when thunks are *self-updating*.

Type rules for *UsageSP* – 2: Subsumption

Consider the rule for applications:

$$\frac{\Gamma \vdash e_1 : (\sigma_1 \to \sigma_2)^u \qquad \Gamma \vdash e_2 : \sigma_1' \qquad \sigma_1' \preccurlyeq \sigma_1}{\Gamma \vdash e_1\; e_2 : \sigma_2} \; (\vdash\text{-}\textsc{App})$$

We define the subtyping relation $\sigma_1 \preccurlyeq \sigma_2$ by induction:

$$\begin{aligned} {\tau_1}^{u_1} \preccurlyeq {\tau_2}^{u_2} \quad &\Leftrightarrow \quad \tau_1 \preccurlyeq \tau_2 \text{ and } u_2 \leq u_1 \\ \sigma_1 \to \sigma_2 \preccurlyeq \sigma_3 \to \sigma_4 \quad &\Leftrightarrow \quad \sigma_3 \preccurlyeq \sigma_1 \text{ and } \sigma_2 \preccurlyeq \sigma_4 \end{aligned}$$

. . . and so on.

Subsumption *vs.* usage polymorphism – 1

We choose subsumption rather than usage polymorphism. In practice, usage polymorphism complicates the implementation and seems likely to bog the compiler down:

- Simple usage polymorphism is insufficient; *bounded* usage polymorphism is required.
- Two new productions, usage abstraction and application, must be added to the compiler's abstract syntax, along with corresponding support code.
- It is likely that *many* usage arguments will have to be passed at every function application.

Subsumption *vs.* usage polymorphism – 2

Rejecting usage polymorphism loses expressivity.

$$apply = \lambda f \,.\, \lambda x \,.\, f\ x$$

With usage polymorphism, it has the type: NB!

$$apply : (\forall u_1 \,.\, \forall u_2 \,.\, \forall \alpha \,.\, \forall \beta \,.\, (\alpha^{u_1} \to \beta^{u_2})^1 \to (\alpha^{u_1} \to \beta^{u_2})^1)^\omega$$

With subsumption, this dependency cannot be expressed, and we must choose a type such as the following:

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

Principal types *vs.* ‘universal’ types

UNIVERSITY OF CAMBRIDGE

We do not have principal types:

$$apply : (\forall\alpha\ .\ \forall\beta\ .\ (\alpha^{\omega} \to \beta^{\omega})^{1} \to (\alpha^{\omega} \to \beta^{\omega})^{1})^{\omega} \tag{1}$$

$$apply : (\forall\alpha\ .\ \forall\beta\ .\ (\alpha^{1} \to \beta^{1})^{1} \to (\alpha^{1} \to \beta^{1})^{1})^{\omega} \tag{2}$$

These are *incomparable*.

However, type (1) is ‘universal’, while type (2) is not: any well-(non-usage-)typed expression involving $apply$ can be (usage-)typed with (1), but not necessarily with (2).

Principal types could in principle be regained by introducing usage polymorphism.

Inference algorithm

1. Annotate program:

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

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

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.

Operational semantics for *UsageSP*

UNIVERSITY OF CAMBRIDGE

We base our operational semantics on Launchbury's natural semantics for lazy evaluation.

- A *heap* H is a set of named thunks $\{x : \sigma_1 \mapsto e_1, y : \sigma_2 \mapsto e_2, \ldots\}$.
- A *configuration* $\langle H \rangle\, e$ is an expression e, possibly with free variables bound in the heap H.

The judgement

$$\langle H_1 \rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\, v$$

signifies that the initial configuration $\langle H_1 \rangle\, e$ evaluates to the final configuration $\langle H_2 \rangle\, v$. ($\overline{\alpha_k}$ are type variables possibly free in e and v, to be explained later).

Thunk usage – 1

When a variable is evaluated, its thunk is retrieved from the heap, evaluated, and the result returned.

If the thunk is annotated ω, we update the thunk with the resulting value:

$$\frac{|\sigma| = \omega \qquad \langle H_1 \rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\, v}{\langle H_1, x : \sigma \mapsto e \rangle\, x \Downarrow_{\overline{\alpha_k}} \langle H_2, x : \sigma \mapsto \underbrace{v}_{\text{Updated value}} \rangle\, v}\ (\Downarrow\text{-}\textsc{Var-Many})$$

Thunk usage – 2

But if the thunk is annotated 1, we *remove the thunk from the heap entirely:*

$$\frac{|\sigma| = 1 \qquad \langle H_1 \rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\, v}{\langle H_1, x : \sigma \mapsto e \rangle\, x \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\, v}\ (\Downarrow\text{-}\textsc{Var-Once})$$

x removed!

Thus if we attempt to use the thunk again (*i.e.*, more than once), evaluation will become stuck.

This rule formalises what we mean by “using a thunk”.

Proof of soundness – 1

UNIVERSITY OF CAMBRIDGE

We must now prove that well-typed programs never become stuck. We prove this in two steps:

1. *Subject reduction:* well-typed configurations remain well-typed after one-step evaluation.

2. *Progress:* well-typed non-value configurations can always be further evaluated.

Proof of soundness – 2

For this, we must

- define ‘well-typed configuration’.
- be able to convert well-typed programs to well-typed configurations.

This suggests we should maintain types in our operational semantics.

But in reality, at run time types are no longer present (our language does not have a `typecase` construct). This has interesting consequences for the operational semantics.

Preservation of types – 1

For example (with and without types):

$$\begin{array}{l}
\text{let} \;\; x : \forall\alpha\,.\,\text{Pair}\,(\alpha \to \text{Int})\,(\alpha \to \text{Int}) \\
\qquad = \Lambda\alpha\,.\,\text{let} \;\; y : \alpha \to \text{Int} \\
\qquad\qquad\qquad = \lambda v : \alpha\,.\,1 \\
\qquad\qquad \text{in} \;\; \text{MkPair}\; y\; y \\
\text{in} \;\; (\mathit{fst}\;(x\;\text{Int}))\;3 + (\mathit{fst}\;(x\;\text{Bool}))\;\text{True}
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{l}
\text{let} \;\; x = \text{let} \;\; y = \lambda v\,.\,1 \\
\qquad\qquad \text{in} \;\; \text{MkPair}\; y\; y \\
\text{in} \;\; (\mathit{fst}\;x)\;3 + (\mathit{fst}\;x)\;\text{True}
\end{array}$$

After $(\mathit{fst}\;x)\;3$ is evaluated, the heap contains $\{x \mapsto \text{MkPair}\; y\; y,\; y \mapsto \lambda v\,.\,1\}$. Now the second use of x *shares the same* y.

A conventional semantics with $\Lambda\alpha\,.\,e$ a value does not permit this!

We would *lose sharing* that is in fact present. What can we do?

Preservation of types – 2

We must permit evaluation under type lambdas. 'Transparency' of type lambdas is reasonable since they do not appear at run time. This gives us the rules

$$\frac{\text{fresh } \alpha' \qquad \langle H_1 \rangle\, e[\alpha := \alpha'] \Downarrow_{\overline{\alpha_k}, \alpha'} \langle H_2 \rangle\, v}{\langle H_1 \rangle\, \Lambda\alpha\, .\, e \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\, \Lambda\alpha'\, .\, v}\ (\Downarrow\text{-}\textsc{TyAbs})$$

$$\frac{\langle H_1 \rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\, \Lambda\alpha\, .\, v}{\langle H_1 \rangle\, e\ \tau \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\, v[\alpha := \tau]}\ (\Downarrow\text{-}\textsc{TyApp})$$

Preservation of types – 3

But what type can we give to thunks such as y?

$$
\begin{array}{l}
\mathsf{let}\quad x \;:\; \forall\alpha\,.\,\mathsf{Pair}\;(\alpha\to\mathsf{Int})\;(\alpha\to\mathsf{Int})\\
\qquad\quad = \Lambda\alpha\,.\,\mathsf{let}\quad y \;:\; \alpha\to\mathsf{Int}\\
\qquad\qquad\qquad\qquad\qquad = \lambda v:\alpha\,.\,1\\
\qquad\qquad\quad\ \mathsf{in}\quad \mathsf{MkPair}\;y\;y\\
\mathsf{in}\quad \underbrace{\boxed{(\mathit{fst}\;(x\;\mathsf{Int}))\;3}}_{y\ \text{used at Int}} + \underbrace{\boxed{(\mathit{fst}\;(x\;\mathsf{Bool}))\;\mathsf{True}}}_{y\ \text{used at Bool}}
\end{array}
$$

We can't place it directly in the heap as $y : (\alpha \to \mathsf{Int}) \mapsto \lambda v : \alpha\,.\,1$, as α is then unbound. Nor can we instantiate it to Int, because it is later used at Bool.

Preservation of types – 4

The solution is to abstract out any type variables we are currently evaluating under before placing bindings in the heap, giving us the thunk $y : (\forall\alpha \,.\, \alpha \to \mathsf{Int}) \mapsto \Lambda\alpha \,.\, \lambda v : \alpha \,.\, 1$.

We could perform this as a source-to-source translation, giving:

$$
\begin{array}{ll}
\mathsf{let} & x : \forall\alpha \,.\, \mathsf{Pair}\ (\alpha \to \mathsf{Int})\ (\alpha \to \mathsf{Int}) \\
& \quad = \Lambda\alpha \,.\, \mathsf{let} \quad y : \forall\alpha \,.\, \alpha \to \mathsf{Int} \\
& \qquad\qquad\qquad\qquad = \Lambda\alpha \,.\, \lambda v : \alpha \,.\, 1 \\
& \qquad\qquad\quad\ \ \mathsf{in} \quad \mathsf{MkPair}\ (y\ \alpha)\ (y\ \alpha) \\
\mathsf{in} & (\mathit{fst}\ (x\ \mathsf{Int}))\ 3 + (\mathit{fst}\ (x\ \mathsf{Bool}))\ \mathsf{True}
\end{array}
$$

However, requiring this translation to be performed first is unnecessary.

Preservation of types – 5

Instead of a source-to-source translation, we simply translate each thunk as we place it in the heap.

$$\frac{\text{fresh } \overline{y_i} \qquad S = (\overline{x_j := y_j\ \overline{\alpha_k}}) \qquad \langle H_1, \overline{y_i : (\forall \overline{\alpha_k}\ .\ \tau_i)^{u_i} \mapsto \Lambda \overline{\alpha_k}\ .e_i[S]}\rangle\ e[S] \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\ v}{\langle H_1 \rangle \text{ letrec } \overline{x_i : \tau_i{}^{u_i} = e_i} \text{ in } e \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\ v}\ (\Downarrow\text{-}\textsc{LetRec})$$

Each thunk is abstracted over the vector of type variables under which we are currently evaluating, and all references to that thunk are passed the actual type variables currently in use.

Notice that this vector of type variables is provided by the $\overline{\alpha_k}$ annotation on $\cdot \Downarrow_{\overline{\alpha_k}} \cdot$. The $(\Downarrow\text{-}\textsc{TyAbs})$ rule adds the bound type variable here each time we go inside a type lambda.

Typing configurations

Now we can type configurations. The rule is almost the same as for letrec:

no $\overline{\alpha_k}$

$$\frac{\begin{array}{c} H = \overline{x_i : \sigma_i \mapsto e_i} \\ \Gamma, \overline{x_j : \sigma_j} \vdash e_i : \sigma_i' \qquad \sigma_i' \preccurlyeq \sigma_i \quad \text{for all } i \\ \Gamma, \overline{x_j : \sigma_j}, \overline{\alpha_k} \vdash e : \sigma \\ \mathit{occur}(x_i, e) + \sum_{j=1}^{n} \mathit{occur}(x_i, e_j) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i \end{array}}{\Gamma \,;\, \overline{\alpha_k} \vdash_{\mathit{Conf}} \langle H \rangle\, e : \sigma}$$

$(\vdash\text{-}\textsc{Conf})$

Here $\Gamma \,;\, \overline{\alpha_k} \vdash_{\mathit{Conf}} \langle H \rangle\, e : \sigma$ means that in context Γ, while evaluating under type lambdas $\overline{\alpha_k}$, the heap H and the expression e are well-typed and e has type σ. Free type variables are not permitted in the heap.

Syntactic soundness proof

Given this operational semantics, soundness is relatively straightforward to prove:

1. Convert $\cdot \Downarrow_{\cdot} \cdot$ to equivalent small-step reduction rules $\cdot \to_{\cdot} \cdot$.
2. Observe that a well-typed program $\vdash e : \sigma$ yields a well-typed initial configuration $; \vdash_{Conf} \langle\rangle\, e : \sigma$.
3. Demonstrate that for well-typed configurations, types are preserved by the reduction rules (*subject reduction*).
4. Demonstrate that well-typed non-value configurations can always be further reduced (*progress*).
5. Conclude evaluation never gets stuck, and therefore that *we use a 1-annotated thunk at most once.* $\square$

UNIVERSITY OF CAMBRIDGE

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.