Complete Laziness: a Natural Semantics

François-Régis Sinot

Universidade do Porto (DCC & LIACC)
Rua do Campo Alegre 1021–1051
4169–007 Porto, Portugal

Abstract

Lazy evaluation (or call-by-need) is widely used and well understood, partly thanks to a clear operational semantics given by Launchbury. However, modern non-strict functional languages do not use plain call-by-need evaluation: they also use optimisations like fully lazy λ-lifting or partial evaluation. To ease reasoning, it would be nice to have all these features in a uniform setting. In this paper, we generalise Launchbury's semantics in order to capture "complete laziness", as coined by Holst and Gomard in 1991, which is slightly more than fully lazy sharing, and closer to on-the-fly needed partial evaluation. This gives a clear, formal and implementation-independent operational semantics to completely lazy evaluation, in a natural (or big-step) style similar to Launchbury's. Surprisingly, this requires sharing not only terms, but also contexts, a property which was thought to characterise optimal reduction.

1 Introduction

Lazy evaluation (also known as call-by-need) is an evaluation strategy for functional languages providing some notion of sharing. The idea behind lazy evaluation is intuitive: a subterm should be evaluated only if it is needed, and if so, it should be evaluated only once. Since its introduction by Wadsworth [29], there have been several efforts, on one hand to improve its concrete implementation, e.g. [23,22], and on the other, to improve its abstract formalisation: big-step operational semantics of call-by-need have been given independently in [18] and [26]; small-step presentations based on contexts have been given in [2,21]. While all these works have their own merits, Launchbury's natural semantics [18] certainly gives one of the clearest accounts of the process of lazy evaluation.

Yet, lazy evaluation captures only the sharing of *values*. For example, evaluation of the term $(\lambda f.f I(f I))(\lambda w.(\underline{II})\,w)$ where $I = \lambda x.x$ will reduce the underlined redex II twice, because the subterm $\lambda w.(II)\,w$ will be shared, then copied as a whole when necessary, since it is already a value (the redex II is under a λ-abstraction). This is indeed what happens in standard implementations of call-by-need [23,22,13].

This is not usually considered as a problem, because this term can also be transformed into $(\lambda f.f I(f I))((\lambda z.(\lambda w.z\,w))(II))$ in which the redex II will be shared by a lazy interpreter, and evaluated only once, because it is no longer under a λ-

abstraction. This transformation is called *fully lazy λ-lifting* and is used at compile-time in compilers for non-strict languages [13,25,22].

Implementations allowing to share this kind of redexes are called *fully lazy*. Wadsworth was the first to define this notion: he noticed that the redex $I I$ should not be copied since no occurrence of the bound variable w occurs in it [29]. But still, the resulting redex $I\,w$ will be evaluated twice by a fully lazy implementation, while its evaluation could have been shared using partial evaluation [14]. In other words, there is a notion of laziness, beyond full laziness, with the same sharing power as partial evaluation. This notion has been coined "complete laziness" in [12] (and later "ultimate sharing" in [1]), but seems to be otherwise unstudied, and in particular lacks a suitable operational semantics.

Moreover, some recent works [27,28] are likely to implement completely lazy evaluation, which is left as an open problem in [12], but there is no hope of proving (or even stating) this formally without a proper operational semantics. This present work thus also goes one step further in this direction.

In this paper, we define a clear and implementation-independent operational semantics for completely lazy evaluation. It is a natural (or big-step) semantics, in a style similar to Launchbury's for lazy evaluation. This is both a formal and effective definition of completely lazy evaluation, and a step towards a better understanding of sharing in the λ-calculus.

2 Launchbury's Semantics

We first briefly review Launchbury's semantics for lazy evaluation, as we will follow the same approach for completely lazy evaluation in Section 3. It is defined on expressions of a λ-calculus enriched with recursive *lets*. As pointed out by Launchbury, *lets* are useful in the input language as they allow to build explicitly cyclic structures. Without them, this would be more difficult and some sharing could be lost. This is in particular one of Launchbury's criticisms against the semantics of [26]. *Lets* are also useful in the intermediate language, as they play an important role in the definition of the semantics.

Launchbury splits the presentation of the semantics in two distinct stages: a static transformation into simpler expression (called *normalisation*), and a dynamic semantics defined only on normalised expressions.

2.1 Normalising terms

First, every expression e is transformed into an expression $\hat{e}$ in which all bound variables are renamed to completely fresh variables. This amounts to performing enough α-conversions, so that expressions respect Barendregt's convention [4]. Expressions are then normalised to obey the following syntax, where arguments of applications are restricted to variables in order to share arguments with a *let* construct.

$$\begin{aligned} t, u &::= x \mid \lambda x.t \mid t\ x \mid \mathit{let}\ x_1 = u_1, \ldots, x_n = u_n\ \mathit{in}\ t \\ v, w &::= \lambda x.t \end{aligned}$$

Values $v, w, \ldots$ are not used in this section, but will allow us to characterise precisely the result of evaluation in Section 2.2. Launchbury's semantics is only defined on closed terms (or more precisely, on closed pairs of an environment and a term), and the dynamic rules of Section 2.2 preserve closedness (that is, if the left-hand side of the conclusion of a rule is closed, the left-hand side of all the premises of this rule are closed as well). That is why, in Launchbury's semantics, values are always λ-abstractions (and never of the form $x\, t_1 \ldots t_n$).

The restriction on application means that arguments are already explicitly named closures, ready to be shared. This normalisation stage thus already contributes to capture sharing. It is defined precisely by the following function $(\cdot)^*$ from standard, unconstrained λ-terms with recursive *lets* to terms t, u obeying the syntax above.

$$\begin{aligned}
x^* &= x \\
(\lambda x.t)^* &= \lambda x.t^* \\
(t\ u)^* &= \begin{cases} t^*\, u & \text{if } u \text{ is a variable,} \\ \mathit{let}\ x = u^*\ \mathit{in}\ t^*\, x & \text{otherwise } (x \text{ is a fresh variable}) \end{cases} \\
(\mathit{let}\ x_1 = u_1, \ldots, x_n = u_n\ \mathit{in}\ t)^* &= \mathit{let}\ x_1 = u_1^*, \ldots, x_n = u_n^*\ \mathit{in}\ t^*
\end{aligned}$$

2.2 Dynamic semantics

The semantics is not defined on terms alone; some data must be added to actually represent sharing. Launchbury's choice is to use heaps or environments (written Γ, Δ, Θ), which are defined as finite mappings from variables to terms (or equivalently as unordered association lists binding distinct variable names to terms).

Evaluation is only defined on closed pairs $\Gamma : t$, meaning that the free variables in t have to be bound in the environment Γ. Evaluation judgements are of the form $\Gamma : t \Downarrow_L \Delta : v$, to be read "the term t in the environment Γ reduces to the value v together with the new environment Δ", and are defined by the set of deduction rules in Figure 1.

The only rule where sharing is indeed captured is Var_L. To evaluate a variable x, the heap must contain a binding $x \mapsto t$, otherwise x has a direct dependency on itself and evaluation should fail. Then t is evaluated to a value v, in a heap where the binding for x has been removed, in order to avoid direct dependencies. The binding for x in the environment is then updated with the value v, in order to avoid a possible recomputation if x is needed several times, and the evaluation returns $\hat{v}$, i.e. v with fresh names for all its bound variables. It is the only rule where renaming occurs and this is sufficient to avoid all unwanted name capture [18]. An example is given in Figure 4 (A_n is defined in Section 4.2).

3 Modelling Complete Laziness

In lazy evaluation, only closed terms are shared; e.g., in $(\lambda f.f I(f I))(\lambda w.(I I)\, w)$, lazy evaluation will share $(\lambda w.(I I)\, w)$, but will reduce $I I$ twice. To obtain complete laziness (and reduce $I I$ only once), we need to share the body $(I I)\, w$ as well. In other words, to realise complete laziness, open terms need to be shared as well. More precisely, in an abstraction $\lambda x.t$, we do not want to share t as a whole, because, when x would be instantiated, the shared representation of t would be updated,

$$\frac{}{\Gamma : \lambda x.t \Downarrow_L \Gamma : \lambda x.t}\ \mathit{Lam}_L$$

$$\frac{\Gamma : t \Downarrow_L \Delta : \lambda y.t' \quad \Delta : t'\{y := x\} \Downarrow_L \Theta : v}{\Gamma : t\ x \Downarrow_L \Theta : v}\ \mathit{App}_L$$

$$\frac{\Gamma : t \Downarrow_L \Delta : v}{(\Gamma, x \mapsto t) : x \Downarrow_L (\Delta, x \mapsto v) : \hat{v}}\ \mathit{Var}_L$$

$$\frac{(\Gamma, x_1 \mapsto u_1, \ldots, x_n \mapsto u_n) : t \Downarrow_L \Delta : v}{\Gamma : \mathit{let}\ x_1 = u_1, \ldots, x_n = u_n\ \mathit{in}\ t \Downarrow_L \Delta : v}\ \mathit{Let}_L$$

Fig. 1. Launchbury's semantics

thus preventing x from being instantiated by another argument. In fact, we exactly want to share the part of t that does not depend on x. Specifically, if we write $t = C[x]$ where x does not appear in the context $C[\,]$ (possibly with several instances of the same hole), then $C[\,]$ is exactly what should be shared. In the example above, what should be shared is indeed $(II)\,[\,]$. The comparison with contexts is helpful to emphasise that the free variables are not part of what should be shared, but is otherwise misleading: there may be several occurrences of the same free variable (hence the notion of hole is not adequate), and normal capture-avoiding term-substitution should be used (instead of context-substitution [11]). It is really more adequate to say that we need to share open terms.

We thus need variables to represent open subterms. Since we may have to deal with several distinguished variables in these terms, it is just as simple to use the concept and notation of metavariables taken from Combinatory Reduction Systems [15,16]. We will thus write for instance $Z(x, y)$ (and we will call it a *metavariable*) for a variable representing an open term in which x and y denote the free variables. Just any term t can be substituted for $Z(x, y)$, but if x and y appear in t (perhaps even several times), then the rules will be able to treat them in a special way. It should also be noted that α-equivalence is extended in the obvious way, with for instance $\lambda x.Z(x) =_\alpha \lambda y.Z(y)$. There is no need for α-equivalence on metavariables.

We follow Launchbury's approach and present the semantics in two phases: a static transformation into simpler expression (again called *normalisation*), and a dynamic semantics for normalised expressions.

3.1 Normalising terms

The normalisation stage has two purposes. The first one is to avoid name capture, by renaming all (λ- and *let*-) bound variables to fresh variables. The second one is to name explicitly with a *let*-construct any subterm that may need to be shared. For lazy evaluation, it is enough to do this for arguments in applications. Here, for completely lazy evaluation, we also need to do this for bodies of abstractions. We will thus assume that expressions $t, u, \ldots$ belong to a new set, defined as follows, where we write $Z(\vec{x})$ for $Z(x_1, \dots, x_n)$. We also define *values* $v, w, \dots$ in this context, which will be used in Section 3.2 to characterise precisely the result of evaluation.

$$\begin{aligned} b &::= x \mid Z(\vec{x}) \\ t, u &::= b \mid \lambda x.b \mid t\ b \mid \mathit{let}\ b_1 = u_1, \dots, b_n = u_n\ \mathit{in}\ t \\ v, w &::= \lambda x.b \mid x\ b_1\ \dots\ b_n \end{aligned}$$

Similarly to Launchbury's semantics, the completely lazy semantics will also be defined only on closed terms. However, in the course of evaluation, we may have to evaluate an open term (this happens in the first premise of rule *MVar* in Figure 2, which will be explained later), and this evaluated open term will be used to update the binding of a metavariable. Therefore, it is important to allow open terms of the form $x\, b_1 \dots b_n$ as values, contrarily to Section 2.1. However, terms of the form $Z(\vec{x})\, b_1 \dots b_n$ are not values, for the same reason as $x\, t_1 \dots t_n$ was not a value in Section 2.1: because the completely lazy semantics is only defined on pairs environments/terms which are meta-closed, i.e. in which all metavariables are bound (either by *lets* or by the environment), and this property is preserved by the rules. The situation is really reminiscent of what happens in Combinatory Reduction Systems, where metavariables essentially play the same role as variables in first-order systems.

Standard λ-expressions with *lets* can be translated into this form by the following normalisation function, which takes an auxiliary list of variables as an extra argument (written as a subscript). The semantics is only defined on closed terms and this list should initially be empty. The normalisation function takes terms from an unconstrained λ-calculus with recursive *lets* and without metavariables to terms t, u obeying the syntax above.

$$\begin{aligned} (x)^*_{\vec{z}} &= x \\ (\lambda x.t)^*_{\vec{z}} &= \begin{cases} \lambda x.t & \text{if } t \text{ is a variable,} \\ \mathit{let}\ Z(\vec{z}, x) = (t)^*_{\vec{z},x}\ \mathit{in}\ \lambda y.Z(\vec{z}, y) & \text{otherwise} \end{cases} \\ (t\ u)^*_{\vec{z}} &= \begin{cases} (t)^*_{\vec{z}}\, u & \text{if } u \text{ is a variable,} \\ \mathit{let}\ Z(\vec{z}) = (u)^*_{\vec{z}}\ \mathit{in}\ (t)^*_{\vec{z}}\ Z(\vec{z}) & \text{otherwise} \end{cases} \\ (\mathit{let}\ x_1 = u_1, &\dots, x_n = u_n\ \mathit{in}\ t)^*_{\vec{z}} \\ &= \mathit{let}\ Z_1(\vec{z}) = (u_1)^*_{\vec{z}}, \dots, Z_n(\vec{z}) = (u_n)^*_{\vec{z}}, \\ &\qquad x_1 = Z_1(\vec{z}), \dots, x_n = Z_n(\vec{z})\ \mathit{in}\ (t)^*_{\vec{z}} \end{aligned}$$

All variable and metavariable names created by the function $(\cdot)^*$ are assumed to be fresh. The purpose of the auxiliary list $\vec{z}$ is to remember which variables are bound by outer λ's (and not by *let* constructs), because these are exactly the variables that could be instantiated by different terms in different copies. The normalisation function seems to introduce many indirections, but this is necessary in order to preserve sharing. For instance, in the case for *let* expressions, a new binding with a metavariable $Z_i(\vec{z})$ is introduced to share the evaluation of u_i when the variables $\vec{z}$ are free (that is, when it is considered as an open term), but it is still necessary to have a binding for x_i (which may appear in t or any u_j), in order to share the evaluation of u_i when the variables of $\vec{z}$ are bound to some expressions. When $\vec{z}$ is empty, nothing special happens, although we may want to simply write Z instead of $Z()$. It is not safe in general to replace such metavariables by normal variables. This is discussed on an example in Section 4.3. The procedure could be refined to save some indirections and minimise the number of variables bound by the new metavariables, however the present formulation suffices for our purpose.

3.2 Dynamic semantics

As in Launchbury's semantics, we use heaps to model sharing. Now heaps specify bindings from distinct variable or metavariable names to terms. Again, evaluation is only defined for meta-closed (see above) pairs $\Gamma : t$ in which all bound variables are distinct, and it is specified by the deduction rules in Figure 2. We observe that the result of evaluation is a pair $\Delta : v$ where v is a value (a term in weak head normal form, i.e. of the form $\lambda x.b$ or $x\, b_1 \dots b_n$).

The first four rules are exactly those of Launchbury's semantics (in fact of a variant already considered in [18, p. 8], which is equivalent). The *MVar* rule is called when it is needed to evaluate a shared, possibly open subterm. Completely lazy sharing is obtained here: t is evaluated, and $Z(\vec{x})$ is updated with the result. There would be a risk that t would be evaluated too much, if the variables in $\vec{x}$ were instantiated. This does not happen, because of the normalisation procedure, which ensures that variables bound by λ-abstractions are fresh and do not appear in *let*-bindings for metavariables (this property is preserved during evaluation, cf. Proposition 5.1). Then, the evaluation goes on with the right variable names, thanks to the substitution of x_1 by y_1, …, x_n by y_n, written $\{\vec{x} := \vec{y}\}$, in $\hat{v}$ (that is, v with all its bound variables renamed to fresh variables). However, $Z(\vec{x})$ should not be further updated, since the variables in $\vec{y}$ are likely to be bound in the environment. In this second phase, we keep the binding for $Z(\vec{x})$ in the environment so that this shared open term can be used with different instantiations of its free variables. The last two rules just deal with open terms in a natural way. The same rules would make sense in Launchbury's semantics to deal with open terms or constants.

Evaluation may fail in rule *MVar* only, if there is no binding for $Z(\vec{x})$ in the environment. This happens for example if $Z(\vec{x})$ has a direct dependency on itself. This allows us to detect some non-terminating programs (but of course not all of them). The same happens in Launchbury's semantics in the *Var* rule (here, for a variable, Var_1 or Var_2 is always applicable).

4 Examples

In this section, we illustrate the behaviour of the semantics in order to give some concrete evidence that it indeed captures completely lazy sharing. To make our point more concrete, we assume given additional rules for the evaluation of arithmetical expressions, as found in [18], for instance:

$$\frac{}{\Gamma : n \Downarrow \Gamma : n} \qquad \frac{\Gamma : t_1 \Downarrow \Delta : n_1 \qquad \Delta : t_2 \Downarrow \Theta : n_2}{\Gamma : t_1 + t_2 \Downarrow \Theta : n_1 + n_2}$$

$$\frac{}{\Gamma : \lambda x.b \Downarrow \Gamma : \lambda x.b}\;\mathit{Lam}$$

$$\frac{\Gamma : t \Downarrow \Delta : \lambda y.b' \quad (\Delta, y \mapsto b) : b' \Downarrow \Theta : v}{\Gamma : t\ b \Downarrow \Theta : v}\;\mathit{App}_1$$

$$\frac{\Gamma : t \Downarrow \Delta : v}{(\Gamma, x \mapsto t) : x \Downarrow (\Delta, x \mapsto v) : \hat{v}}\;\mathit{Var}_1$$

$$\frac{(\Gamma, b_1 \mapsto u_1, \ldots, b_n \mapsto u_n) : t \Downarrow \Delta : v}{\Gamma : \mathit{let}\ b_1 = u_1, \ldots, b_n = u_n\ \mathit{in}\ t \Downarrow \Delta : v}\;\mathit{Let}$$

$$\frac{\Gamma : t \Downarrow \Delta : v \quad (\Delta, Z(\vec{x}) \mapsto v) : \hat{v}\{\vec{x} := \vec{y}\} \Downarrow \Theta : w}{(\Gamma, Z(\vec{x}) \mapsto t) : Z(\vec{y}) \Downarrow \Theta : w}\;\mathit{MVar}$$

$$\frac{x \notin \Gamma}{\Gamma : x \Downarrow \Gamma : x}\;\mathit{Var}_2$$

$$\frac{\Gamma : t \Downarrow \Delta : x\ b_1 \ldots b_n}{\Gamma : t\ b \Downarrow \Delta : x\ b_1 \ldots b_n\ b}\;\mathit{App}_2$$

Fig. 2. Completely lazy semantics

We will also omit some inessential details, for instance some superfluous bindings, which could be avoided with a more clever normalisation procedure.

Following the tradition initiated by [18], we lay proofs out vertically, so as to stress the sequential nature of evaluation. If $\Gamma : t \Downarrow \Delta : v$, we write:

$\Gamma : t$

[*a sub-proof*

[*another sub-proof*

$\Delta : v$

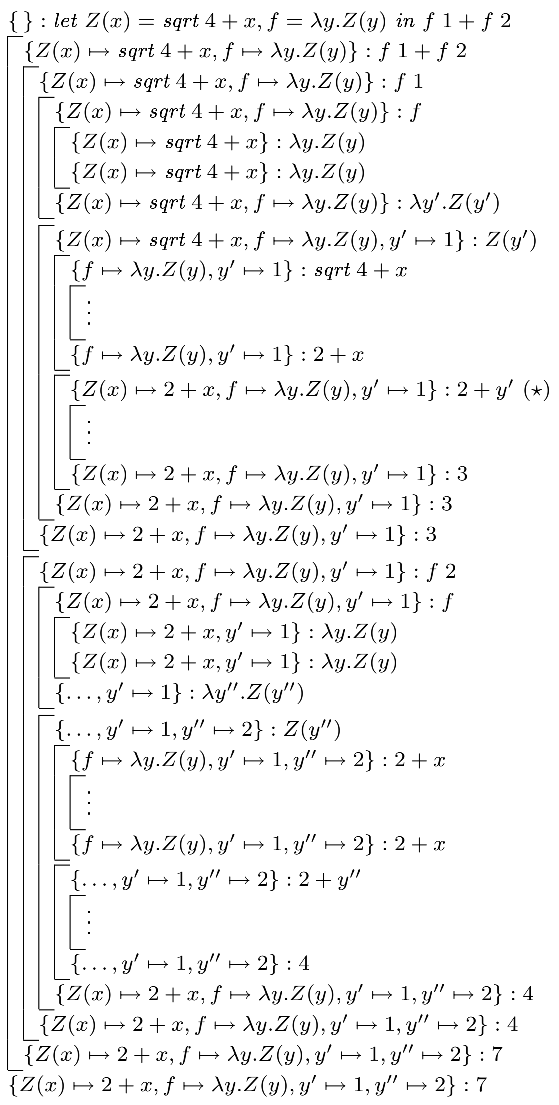
4.1 Simple examples

Let us begin with an example taken from [22, Chapter 15]:

$$\mathit{let}\ f = \lambda x.\mathit{sqrt}\, 4 + x\ \ \mathit{in}\ f\ 1 + f\ 2.$$

This first example illustrates the sharing of a constant expression inside a λ-abstraction, which would already be achieved by fully lazy λ-lifting, but not by standard lazy evaluation. For simplicity, let us omit some indirections and assume that it is normalised as:

$$\mathit{let}\ Z(x) = \mathit{sqrt}\, 4 + x, f = \lambda y.Z(y)\ \ \mathit{in}\ f\ 1 + f\ 2.$$


(a) Sharing of a constant subexpression

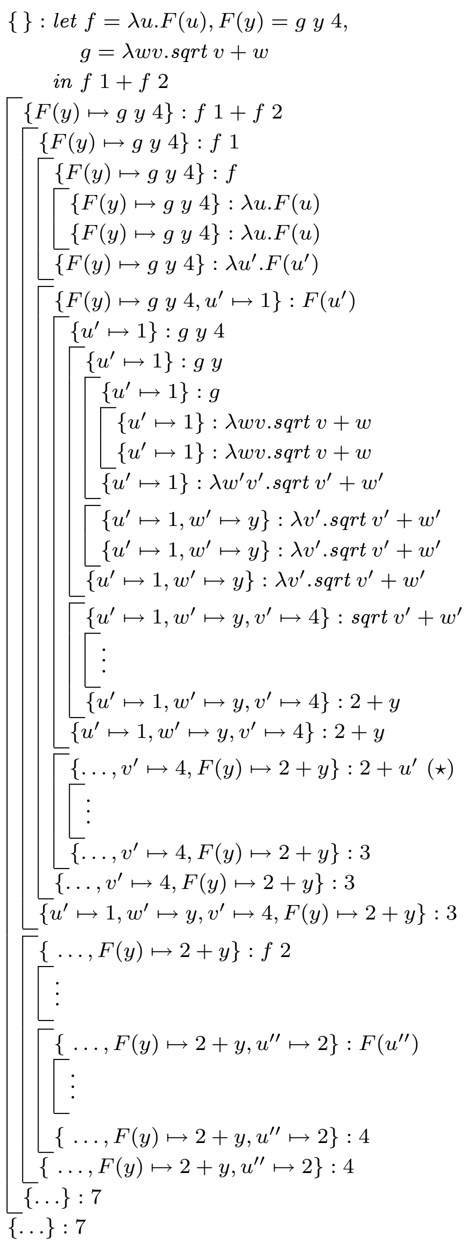

(b) Partial application

Fig. 3. Simple examples

The evaluation derivation of this example is sketched in Figure 3(a). We can observe in line $(\star)$ that $\mathit{sqrt}\,4$ is indeed evaluated only once, and that $Z(x)$ is indeed updated with $2 + x$ (in particular, we evaluate $\mathit{sqrt}\,4 + x$ first, rather than $\mathit{sqrt}\,4 + y'$).

However, such constant subexpressions may also be created dynamically, as in the following program, taken from [22, Chapter 15] as well (the translation is again simplified).

$$\left(\begin{array}{l} let\ f = \lambda y.g\ y\ 4, \\ \quad g = \lambda yx.sqrt\ x + y \\ \quad in\ f\ 1 + f\ 2 \end{array}\right)^{*} \quad = \quad \begin{array}{l} let\ f = \lambda u.F(u), F(y) = g\ y\ 4, \\ \quad g = \lambda wv.sqrt\ v + w \\ \quad in\ f\ 1 + f\ 2. \end{array}$$

During evaluation, the bindings for f and g will not be modified, since they are already bound to values, we thus omit them for conciseness: all environments implicitly contain $f \mapsto \lambda u.F(u)$ and/or $g \mapsto \lambda wv.sqrt\ v + w$. In this example again, shown in Figure 3(b), $sqrt\ 4$ is evaluated only once, even though this redex is only generated on the fly by a partial application. This can be seen from the fact that $F(y)$ is updated with $2+y$ in line $(\star)$. This example is further discussed in Section 6.

4.2 Efficiency

We can also give a striking example, adapted from [10,9,1], to demonstrate that completely lazy reduction can perform exponentially better than lazy evaluation. Consider the family of terms:

$$A_0 = \lambda x.I$$
$$A_n = (\lambda h.(\lambda w.w\,h\,(w\,w))\,A_{n-1}) \equiv let\ Z(h) = (\lambda w.w\,h\,(w\,w))\,A_{n-1}\ in\ \lambda h'.Z(h')$$

A_n has exactly one redex $(\lambda w.\ldots)\,A_{n-1}$, which is under a λ-abstraction, hence will not be shared by lazy evaluation. Consequently, evaluation of $A_n\,I$ using call-by-need requires a number of steps in $O(2^n)$ [10,9]. In Launchbury's semantics, this can be seen on the evaluation sketch in Figure 4 (only some significant steps are shown), where $T(n)$ denotes the number of steps necessary to evaluate $A_n\,x$ (this is indeed independent from x). Overall, $T(n) = O(2 \cdot T(n-1))$, hence $T(n) = O(2^n)$ with standard lazy evaluation. The point is that the A_i's are shared using $w, w', \ldots$, but no significant update will ever happen since they already are weak head normal forms (the redex is under an abstraction).

Now, with the completely lazy semantics, reduction will proceed as shown in Figure 5. The $T(n-1)$ first steps in this example are similar to the evaluation using call-by-need, except that not only $w, w', \ldots$ are updated, but also $Z(h), Z'(h), \ldots$ corresponding to the body under the outermost λ in $w, w', \ldots$ Then, in the second phase, almost no computation has to be performed since $Z'(h)$ is already bound to the identity (independently of h). Overall, $T(n) = O(T(n-1))$, hence $T(n) = O(n)$. Completely lazy evaluation of A_n is linear in n.

This example shows that, although some bookkeeping (indirections essentially) is added, completely lazy evaluation may be exponentially better than lazy evaluation, which is a very strong statement. As a matter of fact, the same improvement can be obtained by fully lazy λ-lifting on this example, but Section 6 will make clear that complete laziness has strictly more "sharing power" than full laziness. Note that all steps are taken into account: the bookkeeping due to the indirections is linear in n in this example. The exact details of implementation are fortunately not part of the semantics, but this means that however bad the implementation is, it will still perform better than any cutting-edge lazy interpreter on certain terms. In other

$\{\} : A_n\, I$

$\{h \mapsto I\} : (\lambda w.w\, h\, (w\, w))\, A_{n-1}$

$\{h \mapsto I, w \mapsto A_{n-1}\} : w\, h\, (w\, w)$

$\{h \mapsto I, w \mapsto A_{n-1}\} : w\, h$

$\{h \mapsto I, w \mapsto A_{n-1}\} : (\lambda w'.w'\, h\, (w'\, w'))\, A_{n-2}$

$\{h \mapsto I, w \mapsto A_{n-1}, w' \mapsto A_{n-2}\} : w'\, h\, (w'\, w')$

$\vdots$ $T(n-2)$ steps

$\{\ldots\} : I$

$\{\ldots\} : I$

$\{\ldots\} : I$

$T(n-1)$ steps

$\{h \mapsto I, w \mapsto A_{n-1}, w' \mapsto A_{n-2}, \ldots\} : w\, w$

$\vdots$ $T(n-1)$ steps

$\{\ldots\} : I$

$\{\ldots\} : I$

$\{\ldots\} : I$

$\{\ldots\} : I$

Fig. 4. Call-by-need evaluation of $A_n\, I$

$\{\} : A_n\, I$

$\{h' \mapsto I, Z(h) \mapsto (\lambda w.w\, h\, (w\, w))\, A_{n-1}\} : Z(h')$

$\{h' \mapsto I, w \mapsto A_{n-1}\} : w\, h\, (w\, w)$

$\{h' \mapsto I, w \mapsto A_{n-1}\} : w\, h$

$\{h' \mapsto I, w \mapsto A_{n-1}\} : w$

$\{h' \mapsto I, w \mapsto \lambda h.Z'(h), Z'(h) \mapsto (\lambda w'.w'\, h\, (w'\, w'))\, A_{n-2}\} : \lambda h''.Z'(h'')$

$\{h' \mapsto I, w \mapsto \lambda h.Z'(h), Z'(h) \mapsto (\lambda w'.w'\, h\, (w'\, w'))\, A_{n-2}\} : Z'(h)$

$\{h' \mapsto I, w \mapsto \lambda h.Z'(h), w' \mapsto A_{n-2}\} : w'\, h\, (w'\, w')$

$\vdots$ $T(n-2)$ steps

$\{h' \mapsto I, w \mapsto \lambda h.Z'(h), w' \mapsto \lambda h.Z''(h), \ldots Z''(h) \mapsto I\} : I$

$\{h' \mapsto I, w \mapsto \lambda h.Z'(h), w' \mapsto \lambda h.Z''(h), \ldots, Z''(h) \mapsto I, Z'(h) \mapsto I\} : I$

$\{h' \mapsto I, w \mapsto \lambda h.Z'(h), w' \mapsto \lambda h.Z''(h), \ldots, Z''(h) \mapsto I, Z'(h) \mapsto I\} : I$

$T(n-1)$ steps

$\{h' \mapsto I, w \mapsto \lambda h.Z'(h), w' \mapsto \lambda h.Z''(h), \ldots, Z''(h) \mapsto I, Z'(h) \mapsto I\} : w\, w$

$\{h' \mapsto I, w \mapsto \lambda h.Z'(h), w' \mapsto \lambda h.Z''(h), \ldots, Z''(h) \mapsto I, Z'(h) \mapsto I\} : w$

$\{h' \mapsto I, w \mapsto \lambda h.Z'(h), w' \mapsto \lambda h.Z''(h), \ldots, Z''(h) \mapsto I, Z'(h) \mapsto I\} : \lambda h''.Z'(h'')$

$\{h'' \mapsto w, h' \mapsto I, w \mapsto \lambda h.Z'(h), w' \mapsto \lambda h.Z''(h), \ldots, Z''(h) \mapsto I, Z'(h) \mapsto I\} : Z'(h'')$

$\{h'' \mapsto w, h' \mapsto I, w \mapsto \lambda h.Z'(h), w' \mapsto \lambda h.Z''(h), \ldots, Z''(h) \mapsto I, Z'(h) \mapsto I\} : I$

$\{\ldots\} : I$

$\{\ldots\} : I$

$\{\ldots\} : I$

$\{\ldots\} : I$

Fig. 5. Completely lazy evaluation of $A_n\, I$

$\{\} : \mathit{let}\ Z = x, x = Z\ \mathit{in}\ x$

$\{Z \mapsto x, x \mapsto Z\} : x$

$\{Z \mapsto x\} : Z$

$\{\} : x$

failure

Fig. 6. Recursion with a direct dependency

words, the amount of bookkeeping necessary for completely lazy reduction is not comparable to what is gained from the better sharing (on this example; this should also be studied in general). This contrasts with optimal reduction, where the cost of bookkeeping ruins the benefits of optimality [19].

This means that completely lazy evaluation, hence the semantics we are putting forward, should be considered as a promising basis for an implementation: it achieves much better sharing than call-by-need, yet does not fall into the well-known problems of optimal reduction, namely that it is complex to understand and implement, and that it is inefficient in practice.

4.3 Recursion

Finally, with respect to recursion, the situation is very similar to that in Launchbury's semantics. For instance, $\mathit{let}\; x = x \;\mathit{in}\; x$ is normalised to $\mathit{let}\; Z = x, x = Z \;\mathit{in}\; x$. Evaluation of this programs fails as shown in Figure 6. This illustrates why there is an extra indirection compared to the same program in Launchbury's framework: evaluation should not fail on a variable (because in completely lazy evaluation we need to perform reductions on open terms); it may only fail on a metavariable.

If we directly feed this example, without $(\cdot)^*$-translation, into the completely lazy semantics, we obtain: $\{\,\} : \mathit{let}\; x = x \;\mathit{in}\; x \Downarrow \{x \mapsto x\} : x$. In other words, we obtain a meaningless value, whereas the right behaviour is to fail. This illustrates that it is unsafe in general to replace metavariables (even without arguments) by normal variables. The converse is also unsafe: imagine we want to normalise the term $\mathit{let}\; x = \lambda y.x \;\mathit{in}\; x$ by replacing the *let*-bound variable x by a metavariable. The problem is that x appears both in a context where it is a closed term, and could be represented by Z, and in a context where it is potentially open, and should be represented by $Z(y)$. This is essentially why the normalisation procedure keeps a binding for x.

5 Properties

5.1 Well-formedness

The first important property to check is that the semantics is indeed well defined. Since it is defined only on terms of a particular form, as produced by the normalisation procedure of Section 3.1, we should check that the result of evaluation has the correct form as well. The property that arguments of applications and bodies of abstractions are variables or metavariables is clearly preserved, since we only ever substitute variables for variables. The naming property is also preserved, as we will now show.

Following [18], we say that $\Gamma : t$ is *distinctly named* if all bound variables and metavariables are distinct. There are three standard types of binding: by a *let* construct, by a λ-abstraction, by a top-level binding in the heap. However, there is a last type of binding here: if $Z(\vec{x}) \mapsto t$ is a binding (for Z) in Γ, we also consider that it is a binding for the variables in $\vec{x}$. In particular, it is crucial that these variables are distinct from other bound variables in rule *MVar*.

Proposition 5.1 *If $\Gamma : t \Downarrow \Delta : v$ and $\Gamma : t$ is distinctly named, then every heap/term pair in the evaluation proof tree is also distinctly named.*

Proof. In general, the rules preserve bound variables. The rules Var_1 and MVar copy a term, which may contain binders, but, one of the copies is renamed with fresh variables. $\square$

It is thus sufficient to perform α-conversion in rules Var_1 and MVar alone to keep all bound variables distinct. In the remainder of this paper, pairs $\Gamma : t$ are always assumed to be distinctly named.

5.2 Correctness

Now that we know that the semantics does nothing wrong syntactically, we should also prove that it does nothing wrong semantically; that is to say that evaluation preserves the denotational semantics of terms.

We define a readback function $(\cdot)^\circ$ from pairs $\Gamma : t$ to λ-terms (in fact, to potentially infinite λ-terms in case of cycles, but this is not really important) that removes the shared variables and metavariables. For every binding $Z(\vec{x}) \mapsto u$ or *let* $Z(\vec{x}) = u$, the readback substitutes every metavariable $Z(\vec{y})$ by $\hat{u}\{\vec{x} := \vec{y}\}$, and then removes the binding for $Z(\vec{x})$, and similarly for the bindings for variables. This is possible thanks to the distinct naming.

Lemma 5.2 (i) *If $\Gamma : t \Downarrow \Delta : v$, then $(\Gamma : t)^\circ \to_\beta^* (\Delta : v)^\circ$.*

(ii) *When a β-reduction is performed during evaluation, (a copy of) the corresponding redex after readback is the leftmost outermost.*

Proof sketch. The rule App_1 is the only one where β-reduction is performed. The other rules do not have any effect after readback, in the sense that, for every rule (except MVar), the readback of the left-hand side of each premise is exactly the readback of the left-hand side of the conclusion. For MVar, $((\Gamma, Z(\vec{x}) \mapsto t) : Z(\vec{y}))^\circ = ((\Gamma, Z(\vec{x}) \mapsto t) : t\{\vec{x} := \vec{y}\})^\circ$ and since the variables in $\vec{x}$ are not bound in the environment (thanks to Proposition 5.1), the redexes in the readback of the left-hand side of both premises of the MVar rule are already present in the left hand-side of its conclusion. Now for rule App_1, let us take the notations of Figure 2. The first premise of rule App_1 focuses on the left subterm of an application while outermost β-reduction is performed in the second one: $(\Delta : (\lambda y.b')\, b)^\circ \to_\beta ((\Delta, y \mapsto b) : b')^\circ$. $\square$

The previous lemma gives an idea of what happens during evaluation. In particular evaluation will always terminate on (even infinite) terms which have a weak head normal form (the strategy is normalising). However the semantics does not exactly coincide with call-by-name ($\Downarrow_{CBN}$): a redex shared in our semantics may correspond to two different β-redexes, one evaluated by call-by-name, and the other not (for example, under a λ). In more realistic functional languages with types and constants, *programs* are closed terms of base type (e.g. integers). The semantics coincide on these "basic observables":

Theorem 5.3 *If t is a program, then $\Gamma : t \Downarrow \Delta : v$ iff $(\Gamma : t)^\circ \Downarrow_{CBN} (\Delta : v)^\circ$.*

5.3 Sharing

Now that we know that the operational semantics given in Section 3 is correct with respect to its result, we should also give some evidence that it captures the sharing expected from complete laziness, which is defined in [12, Section 3.1] as follows:

Definition 5.4 An evaluation is completely lazy if all needed redexes are evaluated exactly once.

This sounds very much like optimal reduction, but it is weaker: optimal reduction also requires that *potential* redexes [20,17] are evaluated at most once. For instance, in the term $(\lambda x.x\,I)(\lambda y.\Delta\,(y\,I))$, the subterm $y\,I$ is not an actual redex, but it is a potential one since it may (and will) become an actual redex after substitution of y by I. We think that most of the conceptual and practical difficulty of optimal reduction comes from the requirement to share such subterms, which justifies the interest of complete laziness as a framework with as much sharing as possible, but excluding potential redexes. This is discussed further in Section 6.

Theorem 5.5 *Let r be a β-redex in t. Then in the derivation of $\Gamma : t \Downarrow \Delta : v$, r is reduced at most once.*

Proof sketch. The normalisation binds every non-trivial subexpression to a meta-variable. There is thus a subterm of t of the form *let* $Z(\vec{x}) = r^*$ *in* t'. If r is reduced, rules *Let* and *MVar* must have been used, and $Z(\vec{x})$ is indeed updated with a value where r has been fired. No occurrence of r thus remains in the expression, hence r cannot be reduced more than once. □

The proposed semantics thus captures completely lazy sharing, in a more direct and operational way than in [12], where complete laziness is formalised as a meta-interpreter implemented in a fully lazy language.

6 Related Work

In the λ-calculus, there is a tension between reduction of the leftmost outermost redex (which is the only normalising choice in general), and reduction of other redexes, which may endanger termination, but may also lead to shorter reduction paths. In this last family of strategies, reduction of the rightmost outermost redex (call-by-value) is the most traditional [24], but some have also studied the impact of performing certain reductions under λ-abstractions, for example [9,7]. The situation is nicely summed up in [8]:

> There is evidently a subtle interplay among the issues of efficiency, normalizability, and redex sharing. The quandary is then to find a way to edge closer to the brink of optimality without plunging into the abyss of non-normalizability.

This apparent tension can be resolved by sharing mechanisms: call-by-need resolves the tension between call-by-name and call-by-value by providing a way to share the evaluation of arguments. The framework we propose here generalises the approach, and resolves the tension between call-by-name and strategies which may reduce under λ's. We thus feel that this present work is a step forward in realising Field's programme.

There have been some works concerned with formal ways to express more than usually lazy reduction. One notable attempt is [2, Section 6], where a fully lazy calculus is given. This calculus can be viewed as a small step semantics for fully lazy evaluation, where reduction is restricted to some cleverly designed classes of contexts. However, the semantics performs on-the-fly λ-lifting, with one of the axioms involving costly conditions about maximal free expressions. It thus seems reasonable to say that this semantics is not effective, unless more details are given about how to implement these conditions at a reasonable cost. In contrast, our semantics does not perform any λ-lifting, it just has the right notion of sharing.

Fully lazy evaluation shares only the so-called maximal free expressions (MFE). This leads to cumbersome situations, as pointed out in [22, pages 398–400]. For instance, in the program

$$\mathit{let}\ g = \lambda yx.y + \mathit{sqrt}\,x, f = \lambda y.g\ y\ 4\ \mathit{in}\ (f\ 1) + (f\ 2),$$

the computation of $\mathit{sqrt}\ 4$ is performed twice, because $\mathit{sqrt}\ x$ is not an MFE of any λ-expression. This problem can be avoided with a different ordering of the parameters of g, but there are terms in which no ordering of the parameters is right. For instance, if the binding for g was in fact

$$g = \lambda xy.\mathit{sqrt}\,x + \mathit{sqrt}\,y,$$

then some sharing would be lost with any order of the arguments. We think that this should be taken as a hint that full laziness is a too syntactic notion to give it a reasonable semantics.

Our semantics indeed allows sharing expressions of this kind, as demonstrated by the second example of Section 4, and thus captures completely, rather than fully, lazy evaluation. In the case $g = \lambda xy.\mathit{sqrt}\ x + \mathit{sqrt}\ y$, our semantics would share a partial application indifferently on the first or the second argument of g.

We do believe that complete laziness is the rational way to capture the spirit of full laziness, abstracted away from syntactical consideration. Moreover, some implementations [27,28] are likely to follow our semantics more faithfully than fully lazy evaluation, because they do not use the very syntactic notion of fully lazy λ-lifting. In any case, the present work provides a formal tool to reason more precisely about fine issues concerning sharing, which was missing until now.

Another theme highly related to this present work is of course optimality theory, defined in [20] and implemented in [17]. In the introduction of [3], one may read:

> Lamping's breakthrough was a technique to share contexts, that is, to share terms with an unspecified part, say a hole. Each instance of a context may fill its holes in a distinct way.

This is of course true of optimal reduction, but what we learn here is that it is also true already for completely lazy reduction, which comes as a surprise. In other words, optimal reduction needs yet something more than the ability to share contexts. A simple example to show that the present semantics is not optimal is the term $(\lambda x.x\,I)(\lambda y.\Delta\,(y\,I))$ where $I = \lambda w.w$ and $\Delta = \lambda z.z\,z$. The semantics will perform the reduction $\Delta\,(y\,I) \to (y\,I)\,(y\,I)$, while the optimal choice is to share the potential redex $y\,I$ and reduce it only once when y is instantiated. It is beyond the scope of this paper to develop this issue further, yet this present work also paves the way to a better understanding of optimal reduction.

7 Conclusion

In this paper we have presented a natural semantics to model completely lazy evaluation. In contrast with Launchbury's work, this is not just a formalisation of a well-known and commonly implemented evaluation strategy. It is rather one of the first attempts to effectively define completely lazy evaluation.

The semantics is not meant to provide a direct specification for an abstract machine, but rather to be a general framework to reason about laziness and study various implementations. Since the framework is very simple compared to more concrete ones, it is also a good basis to study different extensions and properties, such as space behaviour (rules for garbage collection could be added in the same way as in [18]).

Besides a better understanding of the theoretical issues of sharing and efficiency in functional programming languages, this work aims at being used as a foundational basis for implementations. Of course, the legitimacy of (various degrees of) laziness has been decreasing along the years [19,5] and it may seem that our work is primarily of theoretical interest. We do not believe this.

First, laziness is not always useless and there are techniques to combine the advantages of strictness and laziness, such as static analyses [6] and optimistic evaluation [5]. There is no reason to think that these techniques cannot be adapted to our framework. Moreover, proof assistants, like Coq, are an emerging class of functional languages, where programs (proof terms) are built interactively, rather than written directly, and may have a very unusual and intricate shape, for which highly lazy strategies may be well-suited. We believe that the emergence of these new paradigms, with their specific problems, is the occasion to take a fresh look at the theory and practice of the implementation of programming languages.

References

[1] Amtoft, T., “Sharing of Computations,” Ph.D. thesis, University of Aarhus (1993).

[2] Ariola, Z. M. and M. Felleisen, *The call-by-need lambda calculus*, Journal of Functional Programming **7** (1997), pp. 265–301.

[3] Asperti, A. and S. Guerrini, “The Optimal Implementation of Functional Programming Languages,” Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[4] Barendregt, H. P., “The Lambda Calculus: Its Syntax and Semantics,” Studies in Logic and the Foundations of Mathematics **103**, North-Holland Publishing Company, 1984, second, revised edition.

[5] Ennals, R. and S. Peyton Jones, *Optimistic evaluation: an adaptive evaluation strategy for non-strict programs*, in: *International Conference on Functional Programming (ICFP'03)*, ACM Sigplan Notices **38, 9** (2003), pp. 287–298.

[6] Faxén, K.-F., *Cheap eagerness: speculative evaluation in a lazy functional language*, in: *International Conference on Functional Programming (ICFP'00)*, ACM Sigplan Notices **35, 9** (2000), pp. 150–161.

[7] Fernández, M., I. Mackie and F.-R. Sinot, *Closed reduction: Explicit substitutions without α-conversion*, Mathematical Structures in Computer Science **15** (2005), pp. 343–381.

[8] Field, J., *On laziness and optimality in lambda interpreters: Tools for specification and analysis*, in: *Principles of Programming Languages (POPL'90)* (1990), pp. 1–15.

[9] Fradet, P., *Compilation of head and strong reduction*, in: D. Sannella, editor, *European Symposium on Programming*, Lecture Notes in Computer Science **788** (1994), pp. 211–224.

[10] Frandsen, G. S. and C. Sturtivant, *What is an efficient implementation of the λ-calculus ?*, in: J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **523** (1991), pp. 289–312.

[11] Gunter, C. A., "Semantics of Programming Languages: Structures and Techniques," Foundations of Computing, MIT Press, 1992.

[12] Holst, C. K. and C. K. Gomard, *Partial evaluation is fuller laziness*, Sigplan Notices **26** (1991), pp. 223–233.

[13] Hughes, R. J. M., "The Design and Implementation of Programming Languages," Ph.D. thesis, Programming Research Group, Oxford University (1983).

[14] Jones, N. D., P. Sestoft and H. Søndergaard, *An experiment in partial evaluation: The generation of a compiler generator*, in: *Rewriting Techniques and Applications*, Lecture Notes in Computer Science **202** (1985), pp. 125–140.

[15] Klop, J. W., "Combinatory Reduction Systems," Mathematical centre tracts, Centre for Mathematics and Computer Science, Amsterdam (1980).

[16] Klop, J. W., V. van Oostrom and F. van Raamsdonk, *Combinatory reduction systems: introduction and survey*, Theoretical Computer Science **121** (1993), pp. 279–308.

[17] Lamping, J., *An algorithm for optimal lambda calculus reduction*, in: *Principles of Programming Languages (POPL'90)* (1990), pp. 16–30.

[18] Launchbury, J., *A natural semantics for lazy evaluation*, in: *Principles of Programming Languages (POPL'93)*, 1993, pp. 144–154.

[19] Lawall, J. L. and H. G. Mairson, *Optimality and inefficiency: What isn't a cost model of the lambda calculus?*, in: *International Conference on Functional Programming*, 1996, pp. 92–101.

[20] Lévy, J.-J., *Optimal reductions in the lambda calculus*, in: J. P. Hindley and J. R. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980 pp. 159–191.

[21] Maraist, J., M. Odersky and P. Wadler, *The call-by-need lambda calculus*, Journal of Functional Programming **8** (1998), pp. 275–317.

[22] Peyton Jones, S., "The Implementation of Functional Programming Languages," Prentice Hall International, 1987.

[23] Peyton Jones, S. and J. Salkild, *The spineless tagless G-machine*, in: *Functional Programming Languages and Computer Architecture (FPCA'89)* (1989), pp. 184–201.

[24] Plotkin, G., *Call-by-name, call-by-value, and the λ-calculus*, Theoretical Computer Science **1** (1975), pp. 125–159.

[25] Santos, A., "Compilation by Transformation in Non-Strict Functional Languages," Ph.D. thesis, Glasgow University, Department of Computing Science (1995).

[26] Seaman, J. and S. P. Iyer, *An operational semantics of sharing in lazy evaluation*, Science of Computer Programming **27** (1996), pp. 289–322.

[27] Shivers, O. and M. Wand, *Bottom-up beta-reduction: uplinks and lambda-DAGs*, in: *European Symposium on Programming (ESOP'05)*, Lecture Notes in Computer Science **3444** (2005), pp. 217–232.

[28] Sinot, F.-R., *Call-by-need in token-passing nets*, Mathematical Structures in Computer Science **16** (2006), pp. 639–666.

[29] Wadsworth, C. P., "Semantics and Pragmatics of the Lambda-Calculus," Ph.D. thesis, Oxford University (1971).