SOFSEM '92—Ždiar, Magura, Vysoké Tatry, Czechoslovakia, 22.11.–4.12.1992

Optimising compilation

Part II: lazy functional languages

Alan Mycroft and Arthur C. Norman

Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street
Cambridge CB2 3QG, England
E-mail: `Alan.Mycroft@cl.cam.ac.uk, acn1@cam.ac.uk`

Abstract: This lecture considers the optimisation of functional programming languages (particularly 'lazy' languages) based on 'strictness analysis'. Such optimisations alter evaluation order to allow more efficient translation into von Neumann architecture or to increase the parallelism in a sequentially specified system (such as that implied by lazy-semantics).

Keywords: Functional languages, lazy evaluation, eager evaluation, strictness analysis, abstract interpretation.

1 Functional Languages

The term 'functional' as applied to programming languages has been used to apply to a number of related but different models of computation. As used here it will refer to *purely* functional languages, characterised by a complete lack of side-effects and a very direct relationship with the λ-calculus. This rules out languages such as ML and Lisp which, although emphasising the status of functions in the construction of programming and containing well-understood pure subsets, include forms of assignment operator. An ultimately pure functional language would not need anything beyond the ability to describe functions and function application, but practical languages all model the applied λ-calculus and include a collection of constant data objects and functions.

While pure functional languages may not have assignment statements or traditional control structures such as "while" loops, their λ-calculus inheritance leads to an expectation that functions should be first class objects, capable of being passed as arguments and returned as values. The result is that such languages naturally support higher-order functions. In many cases the designers of functional programming languages have introduced pattern matching as a way of reducing the need for explicit conditional statements, with functions defined by gathering together a collection of partial definitions, as in

```
factorial 0 = 1
factorial n = n * factorial(n-1),  n > 0
factorial n = error,               n < 0
```

Research described here was supported by the authors' (UK) SERC grant GR/H14465 on "Strictness analysis, types and reduction machines".

but here we will suppose that such features have already been expanded out in terms of standard conditional operators. There is of course some opportunity for optimising the generation of good sequences of tests to detect which of a collection of patterns matches a particular function application, especially when richly structured datatypes are present, but those challenges will not be discussed here. Section 6 discusses the possible interpretations of `error` above.

2 Lazy Evaluation

One motivation for the study of pure functional programming languages is that the order of evaluation of parts of a complete program cannot lead to the program terminating with different results. However it is possible to have programs such that one evaluation strategy leads to a result while another does not terminate, consider

```
never_stop(x) = never_stop(x+1)
f(x, y) = x
g(x) = f(x, never_stop(x))
```

then one might reason that `f` ignores its second argument, and hence manage to evaluate `g(0)` to 0. Equally if one followed an evaluation strategy where all arguments to a function get processed before the function is called then the calculation of `g(0)` would loop in `never_stop`. The basic result from λ-calculus in this area is that there is available an evaluation process, *normal-order evaluation* that guarantees termination wherever possible. Normal-order evaluation works by selecting, as the next function application to expand, the leftmost outermost one available. In practical computing terms the direct implementation of normal-order evaluation can lead to gross inefficiency, because in expanding an outer function application an argument-expression can be duplicated, and may then have to be evaluated many times. A variant known as *lazy evaluation* avoids this problem and has become commonplace in computer languages. With lazy evaluation function applications are performed in the way that normal-order would suggest, save that whenever it would appear that some expression needs replication (because an argument is used several times in the body of a function) a shared data structure is used. Furthermore when the shared expression has to be evaluated its value updates this shared structure, and subsequent uses can just retrieve the value directly without need for further computation. Applicative-order evaluation follows that of more classical languages and reduces the argument(s) of an application before evaluating the body of the function.

Applicative, normal and lazy evaluation can be characterised by the observation that with applicative-order evaluation every argument expression gets evaluated exactly once. With normal-order arguments may not be evaluated at all, or they may be evaluated once, twice or any larger number of times. Lazy evaluation arranges that argument expressions are not evaluated unless their value is needed, and then they are evaluated at most once. It does, of course, depend on the fact that with a pure functional language if the same expression were to be evaluated over and over again the result would be identical each time.

Lazy evaluation seems to combine the pleasant theoretical idea of giving meaning to as many expressions as possible and the pragmatically attractive idea of only evaluating code when it can be determined that it is necessary. However, various implementations showed that the costs of suspending computations and later resuming them produced drastic loss of speed compared with applicative-order evaluation.

Inspection of realistic fragments of functional code shows that much of it does not really need the luxury of lazy evaluation. For instance any use of the factorial function shown earlier could perfectly happily and safely use applicative-order evaluation. In such circumstances a compiler for the functional language could produce object code very similar in quality to that which one would expect from a compiler for a simple language such as C. The key to detecting where applicative-order of evaluation can be used is known as *strictness analysis*. This literally means the process of analysing a function or program to determine its strictness. Although Mycroft [12] first considered such issues, the epithet 'strictness analysis' was added later. The next sections (in turn) define strictness, show how to calculate safe approximations to it and illustrate how to exploit it. The effect is that strictness optimisations can be seen to work by moving the 'suspend' and 'resume' operations on closures together where they can be peephole-optimised away completely.

3 Strictness and neededness

In this section attention will be restricted to programs without higher-order functions and where all data is restricted to being atomic. Thus list and tree data-structures will not be considered—it will be convenient to think in terms of programs where all data is purely numeric, such as that required by a type system with two levels such as

$$t ::= int \mid bool \qquad\qquad \phi ::= t_1 \times \cdots \times t_k \to t \quad (k \in \mathbb{N}).$$

In this case applicative-order evaluation corresponds to the familiar call-by-value, normal-order evaluation to call-by-name and lazy evaluation to call-by-need. Again call-by-need is a semantically equivalent, but more efficient, implementation of call-by-name.

Given a particular *operational* semantics, such as normal-order evaluation specified above, we say that a program function `f` *needs* its i^{th} argument if evaluating `f(`$e_1, \ldots, e_k$`)` always causes the evaluation of e_i. Barendregt et al. [2] discuss neededness for the λ-calculus. Singh [13] introduces a denotational analogue of neededness by likening strictness to differentiation. Obviously a function which needs its i^{th} argument can safely have that argument evaluated before the call thereby transforming call-by-need to call-by-value. However, neededness is not an easily inferable property of programs. Consider a contrived function (but typical in form):

```
plus(x,y) = if x=0 then y else plus(x-1,y+1).
```

Clearly `plus` always evaluates the first argument of any call. Unfortunately, the hypothesis that it needs (always evaluates) its second argument requires induction for positive `x` and fails to hold for negative `x`—the function loops. However, given a call `plus(`e_1, e_2`)` observe that no harm can result from pre-evaluating e_2 before the call; if e_1 is negative (and hence the computation fails to terminate anyway) then it is harmless to perform the (possibly) non-terminating evaluation of e_2 first as there is no observable way to determine where a program loops.[1] The key point to observe is that *if* `plus(`e_1, e_2`)` terminates then e_2 must have been evaluated.

This motivates a definition that a program function `f` is *strict* in its i^{th} argument if, whenever `f(`$e_1, \ldots, e_k$`)` returns, then e_i will have been evaluated. *I.e.*, in other words,

[1] Of course, the presence of a debugger would require optimisations to be either suppressed or their details communicated—just as in classical imperative languages.

if non-termination of e_i implies non-termination of `f(`$e_1, \ldots, e_k$`)`.[2] Thus `f(x) = f(x)` is strict in `x` but does not need `x`. This operational definition can be hard to reason about, especially at higher-order types or on lazy data structures, and strictness is normally expressed denotationally (see below).

Functional programming languages can be given a particularly natural *denotational* semantics. In a typed functional programming language each type t is associated with a *domain*, D^t. For simple types this is just the corresponding set of values augmented with a least element $\perp$ internalising non-termination; thus `int` is associated with $\mathbb{Z}_\perp$. The function type $t_1 \times \cdots \times t_k \to t$ is associated with $D^{t_1} \underline{\times} \cdots \underline{\times} D^{t_k} \underline{\to} D^t$, its least element being $\lambda(x_1, \ldots, x_k).\perp$. (Here $\underline{\times}$ represents cartesian product and $\underline{\to}$ the domain of (continuous) functions. Henceforth we drop the underlining.) For untyped languages we use a universal domain, such as one satisfying

$$U = \mathbb{Z} + (U \times U) + (U \to U).$$

Recall that a mathematical function $f : D \to E$ between domains is said to be *strict* if $f\perp = \perp$. This is easily extended to say that $f : D_1 \times \cdots \times D_k \to E$ is strict in its i^{th} argument if $(\forall x_1, \ldots, x_k)\ f(x_1, \ldots, x_{i-1}, \perp, x_{i+1}, \ldots, x_k) = \perp$.

Given this, we can now define that a program function `f` is strict in its i^{th} argument if its denotation is.

One might imagine that being able to calculate such *elementary* strictness properties suffices for optimisation purposes. However, it turns out that more information is needed internally—consider

```
f1(x,y,z) = if x=0 then y else z
f2(x,y,z) = if x=0 then y else 42
g1(x,y) = f1(x,y,-y)
g2(x,y) = f2(x,y,-y).
```

Even though `f1` and `f2` [3] have the same strictness in terms of properties so far described (both are strict in `x` only) we see that `g1` is strict in `x` and `y` whereas `g2` is just strict in `x`. The key observation is that `f1` is *jointly strict* in `y` and `z`, *i.e.* its semantics f satisfies $(\forall x)\ f(x, \perp, \perp) = \perp$, whereas `f2` is not.

Hence we are led to introduce a more complicated set of strictness properties in order that we can determine *compositionally* the elementary strictness properties above. Given a function f of k arguments and a subset $S \in \{1, \ldots k\}$, we have a property that f is strict on S (S-jointly strict) if

$$(\forall x_1, \ldots, x_k)\ ((\forall i \in S) x_i = \perp) \Longrightarrow f(x_1, \ldots, x_k) = \perp.$$

Note that f is strict on $S \subseteq T$, then f is strict on T. The effect of this is that the possible strictness properties of such a function are in 1–1 correspondence with the monotone boolean functions $\mathit{2}^k \to \mathit{2}$ where $\mathit{2} = \{0, 1\}$ ordered by $<$. (Each of the possible 2^k sets for S yields a value in $\mathit{2}$.)

[2] I am grateful for Nick Benton for the observation that *classical logic* implication in 'argument non-termination implies application non-termination' corresponds to strictness and that *causative* or *relevant* implication corresponds to neededness.

[3] Erratum: `f2` was incorrect in the original version of this paper.

3.1 Ideal-based strictness and types

This section can be skipped at first reading. Recall that a Scott-closed subset of a domain is one which is downwards closed and inherits limits. Types are often given meaning as ideals (non-empty Scott domains), see MacQueen, Plotkin and Sethi [11].

Note that, given $f : D_1 \times D_k \to D_0$, the strictness property

$$(\forall x_1, \ldots, x_k)\ f(x_1, \ldots, x_{i-1}, \bot, x_{i+1}, \ldots, x_k) = \bot$$

can be re-phrased as $f(D_1 \times D_{i-1} \times \{\bot\} \times D_{i+1} \times D_k) \subseteq \{\bot\}$ and note that $D_1 \times \cdots \times D_{i-1} \times \{\bot\} \times D_{i+1} \times \cdots \times D_k$ is an ideal of $D_1 \times \cdots \times D_k$ and $\{\bot\}$ an ideal of D_0. Given a mathematical function $f : D \to E$ we can thus generalise the idea of strictness property to be any property of the form $f(I) \subseteq J$ where I and J are respectively ideals of D and E respectively. Note that these can naturally be seen as 'threshold properties' generalising strictness presented above—if the input to a function is no more defined than δ, then the output is no more defined that ϵ.

For example, given a set of ideals we can re-express strictness as a collection of such ideal properties. In the above case, if we use the *uniform ideals* of [8] involving any of the 2^k mixtures of $\{\bot\}$ and the D_j above as arguments and either $\{\bot\}$ or D_0 as results, then sets of properties $f(I) \subseteq J$ exactly correspond to strictness functions in $\mathit{2}^k \to \mathit{2}$ above.

Kuo and Mishra [10] investigate further the connection between type inference and strictness after observing that ideals behave like sub-types. For example, writing Δ as a type whose semantics is $\{\bot\}$ we can determine that

$$\begin{array}{llll} + : & \mathit{int} \times \Delta \to \Delta & \mathit{cond} : & \mathit{bool} \times \Delta \times \Delta \to \Delta \\ + : & \Delta \times \mathit{int} \to \Delta & \mathit{cond} : & \Delta \times \mathit{int} \times \mathit{int} \to \Delta \end{array}$$

or, using an intersection type (interestingly normally used to express overloading), we can write

$$\begin{array}{rl} + : & (\mathit{int} \times \Delta \to \Delta) \cap (\Delta \times \mathit{int} \to \Delta) \\ \mathit{cond} : & (\mathit{bool} \times \Delta \times \Delta \to \Delta) \cap (\Delta \times \mathit{int} \times \mathit{int} \to \Delta). \end{array}$$

4 Calculating strictness—abstract interpretation

It is well known that no systematic analysis can guarantee to tell when a fragment of program is going to loop, hence we seek *safe* approximations. For instance, if `p(x)` is a tautologous predicate (such predicates cannot be uniformly determined) the code

```
f(x,y) = if p(x) then y else 42
```

is strict in `y` but this cannot be determined by the compiler. An inference algorithm for strictness is sound if, whenever the algorithm foretells strictness, the code is indeed strict.

A scheme that can provide good information about strictness (subject to the above limitation) is *abstract interpretation* [6]. In this we logically execute programs on non-standard data domains—these correspond to statements about strictness. Hence the understanding is of a standard and a non-standard interpretation of a given program. However, our presentation de-emphasises the abstract interpretation basis (just as a treatment of available expressions might).

Recalling the lattice $\mathit{2} = \{0, 1\}$ from section 3 and the need for safe approximation, the non-standard values used are the values 0 (indicating guaranteed non-termination) and 1

$$\begin{tikzcd}
& \lambda(x,y).1 & \\
& \lambda(x,y).x \vee y \arrow[u] & \\
\lambda(x,y).x \arrow[ur] & & \lambda(x,y).y \arrow[ul] \\
& \lambda(x,y).x \wedge y \arrow[ul] \arrow[ur] & \\
& \lambda(x,y).0 \arrow[u] &
\end{tikzcd}$$

Figure 1: $\mathit{2} \times \mathit{2} \to \mathit{2}$

(indicating possible termination). Allow x to range over $\mathit{2}$ and d over the standard domain of values, D, write $d \rhd x$ "d satisfies strictness property x" for $x = 0 \Longrightarrow d = \perp$ from the above intuition and extend it componentwise to tuples $\vec{d} \rhd \vec{x}$. Each built-in function symbol `f` of k arguments is associated with a function $f^\sharp : \mathit{2}^k \to \mathit{2}$ (section 3 justified such functions as strictness properties). These are induced from the standard meaning, f, by

$$\begin{array}{rcl} f^\sharp(x_1, \ldots, x_k) & = & 0 \quad \text{if } \vec{d} \rhd \vec{x} \implies f(d_1, \ldots, d_k) = \perp \\ & = & 1 \quad \text{otherwise.} \end{array}$$

Hence, we find that, for example, $\mathit{cond}^\sharp(x, y, z) = x \wedge (y \vee z)$ and $+^\sharp(x, y) = x \wedge y$ with (non-$\perp$) constants having value $1 \in \mathit{2}$.

Now, recalling the definition of `plus` above,

```
plus(x,y) = cond(x=0,y,plus(x+1,y-1)),
```

and substituting we obtain a recursive definition which then simplifies with boolean algebra:

$$\begin{array}{rcl} \mathit{plus}^\sharp(x, y) & = & \mathit{cond}^\sharp(=^\sharp(x, 1), y, \mathit{plus}^\sharp(+^\sharp(x, 1), -^\sharp(y, 1))) \\ & = & x \wedge (y \vee \mathit{plus}^\sharp(x, y)). \end{array}$$

This equation has two solutions *viz* $\mathit{plus}^\sharp(x, y) = x \wedge y$ and $\mathit{plus}^\sharp(x, y) = x$, but we seek the least solution (the former) corresponding to the least fixpoint corresponding to recursion in the standard semantics. Note the close similarity with 'dataflow equations' such as those for live variable analysis in imperative language compilers. The least solution can similarly be found by iteration starting with $\mathit{plus}^\sharp = \lambda(x, y).0$ and iterating to convergence (equality of monotone boolean functions can easily be tested if they are represented in a suitable normal form, *e.g.*, CNF). An implementation can speed up convergence by analysing strongly connected components in dependency order. Figure 1 shows the lattice $\mathit{2} \times \mathit{2} \to \mathit{2}$.

Now the question is what this solution tells us about `plus`. It turns out that the property

$$(\forall \vec{x})\ (\forall \vec{d} \rhd \vec{x})\ (f^\sharp(x_1, \ldots, x_k) = 0 \implies f(d_1, \ldots, d_k) = \perp),$$

which holds for built-in functions by definition, is preserved by composition and (least) fixpoint-taking and hence holds for user-defined functions. (By definition of $f^\sharp$ for built-in functions the $\Longrightarrow$ is invertible to $\Longleftrightarrow$, but this is not preserved by composition.)

The particular case we finally wish to exploit is that $f^\sharp(1,\ldots,1,0,1,\ldots,1) = 0$ is a safe test for the corresponding program function `f` being strict its i^{th} argument.

5 Strictness based optimisation

For the context of the above framework, or even its generalisation to the λ-calculus but not including lazy data structures, the only optimisation we have at our disposal is to alter call-by-need into call-by-value. As discussed above, it is possible to do this if (and only if) the function is strict in the corresponding argument. So, we optimise a user-defined function `f` to take its i^{th} argument by value (saving this fact so that all callers can be compiled to pass it by value) whenever $f^\sharp(1,\ldots,1,0,1,\ldots,1) = 0$ (where the 0 appears as the i^{th} argument).

On a parallel machine more optimisations are possible. In general, given that parallel machines have a significantly bounded number of processors, we only wish to spawn processes for sub-expressions which contribute towards the answer. This is a problem in that lazy evaluation is inherently sequential (at each point just one application is reducible). Now, if we know that `f` is strict in both of its arguments, a call `f(`e_1, e_2`)` can have e_1 and e_2 non-wastefully evaluated in parallel (or even have the body of `f` evaluated in parallel with these—although this requires more complex process synchronisation).

Note that strictness analysis and optimisation should be categorised as interprocedural in the classical imperative language classification—the strictness of a function depends on the strictness of functions it references. However, strictness optimisations presented in this section are *forward* in some sense in that we optimise a *caller* in terms of information of the *callee* (the called function). We can imagine more general optimisations such as optimising a function based on the calls which are made to it. This is important (but not sufficiently discussed) at higher types, for example, given `f(g,x) = g(x)`, we can optimise `x` to use call-by-value *only* if we know that every call of `f` will have a strict function for its first argument.

In general only such forwards optimisations are consistent with the idea of 'separately compiled module'—we compile a module into code and auxiliary information telling dependent modules how calls to functions in the former are to be made.

Burn [4] considers optimisations of lazy data structures based on a generalisation of strictness analysis to these.

6 Treatment of errors

Typed functional languages have little scope for run-time errors—really only calling a function defined by cases at a missing case (division by zero can be considered an archetype). Untyped languages generally have such erroneous cases for almost every built-in function; it is tiresome, for example, to check each first argument to `cond` is a boolean. If such errors are treated as fatal, *i.e.* uncatchable, then the worst possibility is that strictness optimisation alters order of evaluation so that a program which previously produced an error instead loops. We must, however, be very careful of introducing a constant, `error` say, which can be used in a comparison to test for error values. For example, in such circumstances, if we define

```
cond(true,y,z) = y
cond(false,y,z) = z
cond(x,y,z) = error       % otherwise
```

then `cond` is no longer jointly strict in `y` and `z`.

The moral is that treating errors as $\perp$, even if an implementation happens to be able to report them rather than looping, results in more functions being strictness-optimisable than treating errors as a non-$\perp$ value which can therefore legally be tested.

7 Projection- and PER-based strictness

Although the above ideal-based strictness is probably the best-known, Wadler and Hughes [14] introduce another form of strictness called projection-based strictness. Recently Hunt's thesis [9] showed how both forms could be embedded in a framework based on partial equivalence relations (PERs). We give a brief treatment of this work.

Suppose D is a domain. A *projection*, α, on D is a mapping in $D \to D$ which is idempotent ($\alpha \circ \alpha = \alpha$) and less that identity ($\alpha \sqsubseteq 1_{D \to D}$). Given projections α and β respectively on domains D and E, we say that a mathematical function $f : D \to E$ is *α-strict in a β-strict context* if

$$\beta \circ f \circ \alpha = \beta \circ f$$

or, in words, if destroying an α amount of information in the argument does not affect the result of f seen after a β information-destroying filter.

Projection-based strictness captures some properties which cannot be defined as ideal-based properties, for example the equation

$$(\lambda x.x) \circ f \circ (\lambda x.\perp) = (\lambda x.x) \circ f$$

is equivalent to $(\forall x \in D) f(x) = f(\perp)$, *i.e.* that f is constant. However, it is debatable whether constancy should be classified as a 'strictness' property.

Projection-based analysis was restricted to first-order functions and moreover required a rather clumsy 'lifting' operation to encompass ideal-based strictness.

PER-based analysis subsumes both projection-based strictness (extending it to higher-order) and also ideal-based strictness (without lifting). Given a domain D a PER p over D is a relation over D which is symmetric and transitive. Equivalently, writing $|p|$ for $\{d \in D \mid (d,d) \in p\}$, such a PER may be thought of as a subset $|p| \subseteq D$ together with an equivalence relation $p \cap (|p| \times |p|)$ on $|p|$. Given a function $f : D \to E$, we have a PER on D given by $\{(d,d') \mid f(d) = f(d')\}$. If $D = E$ and f is a projection then this process is injective. Similarly, an ideal (or even a set) $I \subseteq D$ can be uniquely encoded either by the square PER $I \times I$ or by the diagonal PER $\{(d,d) \mid d \in I\}$. See Hunt's thesis for more details.

References

[1] Abramsky, S. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, Oxford University Press, 1990.

[2] Barendregt, H.P., Kennaway, J.R., Klop, J.W., and Sleep, M.R. Needed Reduction and Spine Strategies for the Lambda Calculus, *Information and Control*, vol. 73, 1987.

[3] Burn, G. A relationship between abstract interpretation and projection analysis (extended abstract). Proc. 17th ACM symp. on Principles of Programming Languages, 1990.

[4] Burn, G.L. Lazy functional languages: interpretation and compilation. MIT Press, Cambridge, Mass., 1991.

[5] Burn, G.L., Hankin, C. and Abramsky, S. The theory and practice of strictness analysis for higher order functions. In Jones, N.D. and Ganzinger, H. (eds.), Programs as Data Objects: Proc. of a Workshop, Copenhagen, LNCS vol. 215, Springer-Verlag, 1985.

[6] Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proc. 4th ACM symp. on Principles of Programming Languages, 1977.

[7] Cousot, P. and Cousot, R. Inductive definitions, semantics and abstract interpretation. Proc. 19th ACM symp. on Principles of Programming Languages, 1992.

[8] Ernoult, C. and Mycroft A. Uniform ideals and strictness analysis. Proc. 18th ICALP, LNCS vol. 510, Springer-Verlag, 1991.

[9] Hunt, S. Abstract interpretation of functional languages: from theory to practice. PhD thesis, Department of Computing, Imperial College, London, 1991.

[10] Kuo, T.-M. and Mishra, P. Strictness analysis: a new perspective based on type inference. ACM-IFIP, Proc. of the functional programming and computer architecture conference, 1989.

[11] MacQueen, D., Plotkin, G.D. and Sethi, R. An ideal model for recursive polymorphic types. Proc. 11th ACM symp. on Principles of Programming Languages, 1984.

[12] Mycroft, A. Abstract interpretation and optimising transformations of applicative programs. Ph.D. thesis, Edinburgh University, 1981. Available as computer science report CST-15-81.

[13] Singh, S. Differentiating strictness. In Peyton-Jones, Hutton and Holst (eds.), Proc. Glasgow workshop on functional programming. Workshops in Computing series, Springer-Verlag, 1990.

[14] Wadler, P.L. and Hughes R.J.M. Projections for strictness analysis. In G. Kahn (ed.) Proc. of the functional programming and computer architecture conference, LNCS vol. 274, Springer-Verlag, 1987.

The above list contains important articles in addition to those referenced.