Abstract

Since the mid ‘80s, compiler writers for functional languages (especially lazy ones) have been writing papers about identifying and exploiting thunks and lambdas that are used only once. However it has proved difficult to achieve both power and simplicity in practice. We describe a new, modular analysis for a higher-order language, which is both simple and effective, and present measurements of its use in a full-scale, state of the art optimising compiler. The analysis finds many single-entry thunks and one-shot lambdas and enables a number of program optimisations.

Categories and Subject Descriptors  D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages — Program analysis, Operational semantics

General Terms  Languages, Theory, Analysis

Keywords  compilers, program optimisation, static analysis, functional programming languages, Haskell, lazy evaluation, thunks, cardinality analysis, types and effects, operational semantics

1. Introduction

Consider these definitions, written in a purely functional language like Haskell:

\[
\begin{align*}
\text{wurble}1, \text{wurble}2 & : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \\
\text{wurble}1 \ k & = \text{sum} \ (\text{map} \ k \ [1..10]) \\
\text{wurble}2 \ k & = 2 \star \ k \ 0 \\
\text{f1} & : [\text{Int}] \rightarrow \text{Int} \\
f1 \ xs & = \text{let} \ ys = \text{map} \ \text{costly} \ xs \\
& \quad \text{in} \ \text{wurble} \ (\text{n. sum} \ (\text{map} \ (+ \ n) \ ys))
\end{align*}
\]

Here we assume that \text{costly} is some function that is expensive to compute and \text{wurble} is either \text{wurble}1 or \text{wurble}2. If we replace \text{ys} by its definition, we could transform \text{f1} into \text{f2}:

\[
\text{f2} \ xs = \text{wurble} \ (\text{n. sum} \ (\text{map} \ (+ \ n) \ (\text{map} \ \text{costly} \ xs)))
\]

A compiler like GHC can now use short-cut deforestation to fuse the two maps into one, eliminating the intermediate list altogether, and offering a substantial performance win [Gill et al.1993].

Does this transformation make the program run faster or slower? It depends on \text{wurble}. For example, \text{wurble}1 calls its function argument ten times, so if \text{wurble} = \text{wurble}1, function \text{f2} would compute \text{costly} ten times for each element of \text{xs}; whereas \text{f1} would do so only once. On the other hand if \text{wurble} = \text{wurble}2, which calls its argument exactly once, then \text{f2} is just as efficient as \text{f1}, and \text{map/map} fusion can improve it further.

The reverse is also true. If the programmer writes \text{f2} in the first place, the full laziness transformation (Peyton Jones et al.1996) will float the sub-expression (\text{map} \ \text{costly} \ \text{xs}) out of the \text{n-expression}, so that it can be shared. That would be good for \text{wurble}1 but bad for \text{wurble}2.

What is needed is an analysis that can provide a sound approximation of how often a function is called – we refer to such an analysis as a cardinality analysis. An optimising compiler can then use the results of the analysis to guide its transformations. In this paper we provide just such an analysis:

- We characterise two different, useful forms of cardinality, namely (a) how often a function is called, and (b) how often a thunk is forced in a lazy language (Section[2]). Of these, the former is relevant under both call-by-need and call-by-value, while the latter is specific to call-by-need.
- We present a backwards analysis that can soundly and efficiently approximate both forms of cardinality for a non-strict, higher-order language (Section[3]). A significant innovation is our use of \text{call demands} to model the usage of a function; this makes the analysis both powerful and modular.
- We prove that our algorithm is sound; for example if it claims that a function is called at most once, then it really is (Section[4]). This proof is not at all straightforward, because it must take account of sharing — that is the whole point! So we cannot use standard denotational techniques, but instead must use an operational semantics that models sharing explicitly.
- We formalise a number of program optimisations enabled by the results of the cardinality analysis, prove them sound and, what is more important, improving (Section[5]).
- We have implemented our algorithm by extending the Glasgow Haskell Compiler (GHC), a state of the art optimising compiler for Haskell. Happily, the implementation builds directly on GHC’s current strictness and absence analyser, and is both simple and efficient (Section[6]).
- We measured how often the analysis finds called-once functions and used-once thunks (Section[7]), and how much this knowledge improved the performance of real programs (Sections[7.1,7.2]). The analysis proves quite effective in that many one-shot lambdas and single-entry thunks are detected (in the range 0-30%, depending on the program). Improvements in performance are modest but consistent (a few percent): programs already optimised by GHC are a challenging target!
We discuss related work in Section 8. Distinctive features of our work are (a) the notion of call demands, (b) a full implementation measured against a state of the art optimising compiler, and (c) the combination of simplicity with worthwhile performance improvements.

2. What is Cardinality Analysis?

Cardinality analysis answers three inter-related questions, in the setting of a non-strict, pure functional language like Haskell:

- How many times is a particular, syntactic lambda-expression called (Section 2.1)?
- Which components of a data structure are never evaluated; that is, are absent (Section 2.3)?
- How many times is a particular, syntactic thunk evaluated (Section 2.4)?

2.1 Call cardinality

We saw in the introduction an example where it is helpful to know when a function calls its argument at most once. A lambda that is called at most once is called a one-shot lambda, and they are extremely common in functional programming: for example a continuation.

Nor is that all. As we saw in the Introduction, inlining under a one-shot lambda (to transform f1 into f2) allows short-cut deforestation to fuse two otherwise-separate calls of map. But short-cut deforestation itself introduces many calls of the function build:

\[
\text{build} :: (\forall a. b \rightarrow b) \rightarrow (a \rightarrow a) \rightarrow [a] \rightarrow [b] \rightarrow [b]
\]

You can see that build calls its argument exactly once, and inlining ys in calls like \((\text{build } \langle \alpha \ldots \alpha \ldots \rangle)\) turns out to be crucial to making short-cut deforestation work in practice. Gill devotes a section of his thesis to elucidating this point (Gill 1996, Chapter 4.3). Gill lacked an analysis for one-shot lambdas, so his approach is necessarily visible; he taught GHC’s optimiser to behave specially for one-shot lambdas. We discuss related work in Section 8. Distinctive features of our work are (a) the notion of call demands, (b) a full implementation measured against a state of the art optimising compiler, and (c) the combination of simplicity with worthwhile performance improvements.

2.2 Currying

In a higher order language with curried functions, we need to be careful about the details. For example, consider:

\[
f3 a = \text{wurble} a (\langle x. \lambda t = \text{costly} x \in \alpha \rightarrow t \rangle y)
\]

\[
\text{wurble1} a g = g 2 a + g 3 a
\]

\[
\text{wurble2} a g = \text{sum} (\text{map} g \langle 1 \ldots 1000 \rangle)
\]

If \text{wurble} was \text{wurble1}, then in \text{f3} it would be best to inline \text{t} at its use site, thus:

\[
f4 a = \text{wurble1} a (\langle x.\alpha \rightarrow \text{costly} x + y \rangle)
\]

The transformed \text{f4} is much better than \text{f3}: it avoids allocating a thunk for \text{t}, and avoids allocating a function closure for the \text{y}. But if \text{f3} called \text{wurble2} instead, such a transformation would be disastrous. Why? Because \text{wurble2} applies its argument \text{g} to one argument \text{a}, and the function thus computed is applied to each of 1000 integers. In \text{f3} we will compute \text{costly} \text{a} once, but \text{f4} will compute it 1000 times, which is arbitrarily bad.

So our analysis of \text{wurble2} must be able to report ‘\text{wurble2}’s argument \text{g} is called\(^1\) once (applied to one argument), and the result is called many times’. We formalise this by giving a usage signature to \text{wurble}, like this:

\[
\text{wurble1} :: U \rightarrow C(C(U)) \rightarrow \bullet
\]

\[
\text{wurble2} :: U \rightarrow C(C(U)) \rightarrow \bullet
\]

The notation \(C(C(U))\) is a usage demand: it describes how a (function) value is used. The demand type \(U \rightarrow C(C(U)) \rightarrow \bullet\) describes how a function uses its arguments, therefore it gives a usage demand for each argument. (The ’\bullet’ has no significance; we are just used to seeing something after the final arrow!) Informally, the \(C(C(d))\) means “this argument is called once (applied to one argument), and the result is used with usage \(d\)”, whereas \(C(C(d))\) means “this argument may be called many times, with each result used with usage \(d\)”. The \(U\) means “is used in some unknown way (or even not used at all)”. Note that \text{wurble1}’s second argument usage is \(C(C(U))\), not \(C(C(C(U)))\); that is, in all cases the result of applying \text{g} to one argument is then called only once.

2.3 Absence

Consider this function:

\[
f x = \text{case } x \text{ of } (p, q) \rightarrow <\text{cbody}>
\]

A strictness analyser can see that \(f\) is strict in \(x\), and so can use call-by-value. Moreover, rather than allocate a pair that is passed to \(f\), which immediately takes it apart, GHC uses a worker/wrapper transformation to pass the pieces separately, thus:

\[
f x = \text{case } x \text{ of } (p, q) \rightarrow \text{fw} p q
\]

\[
f w p q = <\text{cbody}>
\]

Now \(f\) (the “wrapper”) is small, and can be inlined at \(f\)’s call sites, often eliminating the allocation of the pair; meanwhile \(\text{fw}\) (the “worker”) does the actual work. Strictness analysis, and the worker/wrapper transform to exploit its results, are hugely important to generating efficient code for lazy programs (Peyton Jones and Partain 1994, Peyton Jones and Santos 1998).

In general, \(f\)’s right-hand side often does not have a syntactically visible case expression. For example, what if \(f\) simply called another function \(g\) that was strict in \(x\)? Fortunately the worker/wrapper transform is easy to generalise. Suppose the right hand side of \(f\) was just \(<\text{cbody}\>). Then we would transform to:

\[
f x = \text{case } x \text{ of } (p, q) \rightarrow \text{fw} p q
\]

\[
f w p q = \text{let } x = (p, q) \text{ in } <\text{cbody}>
\]

Now we hope that the binding for \(x\) will cancel with case expressions in \(<\text{cbody}\)>, and indeed it usually proves to be so (Peyton Jones and Santos 1998).

But what if \(<\text{cbody}\) did not use \(q\) at all? Then it would be stupid to pass \(q\) to \(\text{fw}\). We would rather transform to:

\[
f x = \text{case } x \text{ of } (p, q) \rightarrow \text{fw} p q
\]

\[
f w p q = \text{let } x = (p, \text{error } "urk") \text{ in } <\text{cbody}>
\]

This turns out to be very important in practice. Programmers seldom write functions with wholly-unused arguments, but they frequently write functions that use only part of their argument, and ignoring this point leads to large numbers of unused arguments being passed around in the “optimised” program after the worker/wrapper transformation. Absence analysis has therefore been part of GHC since its earliest days (Peyton Jones and Partain 1994), but it has never been formalised. In the framework of this paper, we give \(f\) a usage signature like this:

\[
f :: U(U, A) \rightarrow \bullet
\]

The \(U(U, A)\) indicates that the argument is a product type; that is, a data type with just one constructor. The \(A\) (for “absent”) indicates that \(f\) discards the second component of the product.
2.4 Thunk cardinality

Consider these definitions:

\[ f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ f \ x \ c = \begin{cases} 1 & \text{if } x > 0 \\ c + 1 & \text{else} \end{cases} \]
\[ g \ y = f \ y \ (\text{costly } y) \]

Since \( f \) is not strict in \( c \), \( g \) must build a thunk for (costly \( y \)) to pass to \( f \). In call-by-name evaluation, thunks are memoised. That is, when a thunk is evaluated at run-time, it is overwritten with the value so that if it is evaluated a second time the already-computed value can be returned immediately. But in this case we can see that \( f \) never evaluates its second argument more than once, so the memoisation step is entirely wasted. We call these single-entry thunks.

Memoisation is not expensive, but it is certainly not free. Operationally, a pointer to the thunk must be pushed on the stack when evaluation starts, it must be black-holed to avoid space leaks [Jones [1992], and the update involves a memory write. If cardinality analysis can identify single-entry thunks, as well as one-shot lambdas, that would be a Good Thing. And so it can: we give \( f \) the usage signature:

\[ f :: \omega \ast \text{U} \rightarrow 1 \ast \text{U} \rightarrow \bullet \]

The \( \omega \ast \) modifier says that \( f \) may evaluate its first argument more than once, while the \( \ast \) says that it evaluates its second argument at most once.

2.5 Call vs evaluation

For functions, there is a difference between being evaluated once and called once, because of Haskell’s seq function. For example:

\[ f1 \ g = g \ \text{seq} \ 1 \quad -- \quad f1 :: 1 \ast \text{U} \rightarrow \bullet \]
\[ f2 \ g = g \ \text{seq} \ 2 \quad -- \quad f2 :: \omega \ast \text{C}^1(\text{U}) \rightarrow \bullet \]
\[ f3 \ g = g \ 3 \quad -- \quad f3 :: 1 \ast \text{C}^1(\text{U}) \rightarrow \bullet \]

The function seq evaluates its first argument (head-normal form) and returns its second argument. If its first argument is a function, the function is evaluated to a lambda, but not called. Notice that \( f2 \)’s usage type says that \( g \) is evaluated more than once, but applied only once. For example consider the call

\[ f \ (x, x + y) \]

How many times is \( y \) evaluated? For \( f \) equal to \( f1 \) the answer is zero; for \( f2 \) and \( f3 \) it is one.

3. Formalising Cardinality Analysis

We now present our analysis in detail. The syntax of the language we analyse is given in Figure 1. It is quite conventional: just lambda calculus with pairs and (non-recursive) let-expressions. Constants \( \kappa \) include literals and primitive functions over literals, as well as Haskell’s built-in seq. We use A-normal form [Sabry and Felleisen [1992] so that the issues concerning thunks show up only for let and not also for function arguments.

3.1 Usage demands

Our cardinality analysis is a backwards analysis over an abstract domain of usage demands. As with any such analysis, the abstract domain embodies a balance between the cost of the analysis and its precision. Our particular choices are expressed in the syntax of usage demands, given in Figure 1. A usage demand \( d \) is one of the following:

- \( \text{U}(d^1_1, d^1_2) \) applies to pairs. The pair itself is evaluated and its first component is used as described by \( d^1_1 \) and its second by \( d^1_2 \).

Expressions and values

\[ e ::= x | v | (e \ e) | \text{let } x = \text{e} \text{ in } e_2 | \text{case } e_1 \text{ of } (x_1, x_2) \rightarrow e_2 \]

Annotated expressions and values

\[ e ::= x | v | (e | e) | \text{let } x \equiv e \text{ in } e_2 | \text{case } e_1 \text{ of } (x_1, x_2) \rightarrow e_2 \]

Usage demands and multi-demands

\[ d ::= C^n(d) | U(d^1_1, d^1_2) | U \mid \text{HU} \]

Usage types

\[ \tau ::= \bullet | d^1 \rightarrow \tau \]

Usage type expansion

\[ d^1 \rightarrow \tau \leq d^1 \rightarrow \tau \]

Free-variable usage environments (fv-usage)

\[ \varphi ::= (x^d_1), \varphi \mid e \]

Auxiliary notation on environments

\[ \varphi(x) = d^1 \text{ when } (x^d_1) \in \varphi \]

Usage signatures and signature environments

\[ \rho ::= (k: \tau; \varphi) \quad k \in \mathbb{Z}_{>0} \]

\[ P ::= (x^e_1), P \mid e \]

Usage signatures and signature environments

\[ \text{transform}((k: \tau; \varphi), d) = (\tau; \varphi) \quad \text{if } d \subseteq C^1(\ldots k\text{-fold } \ldots C^1(\text{U})) \]

\[ (\omega \ast \tau; \omega \ast \varphi) \text{ otherwise} \]

Figure 1: Syntax of terms, values, usage types, and environments

- \( C^n(d) \) applies to functions. The function is called at most \( n \) times, and on each call the result is used as described by \( d \). Call demands are, to the best of our knowledge, new.
- \( U \), or “used”, indicating no information; the demand can use the value in an arbitrary way.
- \( HU \), or “head-used”, is a special case; it is the demand that \( \text{seq} \) places on its first argument: \( \text{seq} :: \text{HU} \rightarrow \text{U} \rightarrow \bullet \).

A usage demand \( d \) always uses the root of the value exactly once; it cannot express absence or multiple evaluation. That is done by \( d^1 \), which is either \( A \) (absent), or \( n \ast d \) indicating that the value is used at most \( n \) times in a way described by \( d \). In both \( C^n(d) \) and \( n \ast d \), the multiplicity \( n \) is either \( 1 \) or \( \omega \) (meaning “many”). Notice that a call demand \( C^n(d) \) has a \( d \) inside it, not a \( 1 \ast d \); if a function is called, its body is evaluated exactly once. This is different from pairs. For example, if we have

\[ \text{let } x = (e_1, e_2) \text{ in } \text{fst } x + \text{fst } x \]

then \( e_1 \) is evaluated twice.

Both \( U \) and \( HU \) come with some non-semantic equalities, denoted by \( \equiv \) in Figure 1 and necessary for the proof of well-typedness (Section 4). For example, \( U \) is equivalent to a pair demand whose components are used many times, or a many-call-demand where the result is used in an arbitrary way. Similarly, for pairs \( HU \) is equivalent to \( U(A, A) \), while for functions \( HU \) is equivalent to \( C^n(A) \), if we had such a thing. In the rest of the paper all definitions
\[ \mu(d^1) = m \]
\[ \mu(A) = 0 \quad \mu(n * d) = n \]
\[ d_1^* \land d_2^* = d_3^1 \quad d_1^1 \land d_2^1 = d_3^1 \]

- \[ A \land d^1 = d^1 \]
- \[ d^1 \land A = d^1 \]
- \[ n_1 * d_1 \land n_2 * d_2 = \omega * (d_1 \land d_2) \]
- \[ A \land d^1 = d^1 \]
- \[ n_1 * d_1 \land n_2 * d_2 = (n_1 \land n_2) * (d_1 \land d_2) \]

\[ d_1 \land d_2 = d_3 \quad d_1 \land d_2 = d_3 \]

\[ d \land U = U \]
\[ U \land d = U \]
\[ d \land \text{HU} = d \]
\[ \text{HU} \land d = d \]
\[ C^n(d_1) \land C^n(d_2) = C^n(d_1 \land d_2) \]
\[ U(d_1^1, d_2^1) \land U(d_1^1, d_2^1) = U(d_1^1, d_1^1, d_2^1, d_2^1) \]

- \[ \tau_1 \land \tau_2 = \tau_3 \]
- \[ (d_1^1 \land d_2^1 \land \tau_3) \rightarrow (\tau_1 \land \tau_2) \]
- \[ (\tau_1 \land x) \land (\tau_2 \land y) = (\tau_3 \land \tau_4) \]
- \[ (\tau_1 \land \tau_2 \land \tau_3) \land (\tau_2 \land x) = \tau_3 \land \tau_5 \]

\[ n * d^1 = d^1 \quad n * \tau_1 = \tau_2 \quad n * x = \varphi_2 \]

\[ \text{APP} \]

\[ P \vdash x \downarrow d \Rightarrow (\tau ; \varphi) \Rightarrow \varphi \]
\[ \text{APPB} \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow x \]
\[ \text{VARDN} \]

\[ x \not\in \text{dom}(P) \]
\[ P \vdash x \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow x \]
\[ \text{VARUP} \]

\[ x \not\in \text{dom}(P) \]
\[ P \vdash x \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow x \]
\[ \text{LAM} \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow \varphi \]
\[ \text{LAMU} \]

\[ P \vdash \lambda x, e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow \lambda \cdot \varphi \]
\[ \text{LAMH} \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow \varphi \]
\[ \text{PAIR} \]

\[ \text{PAIRH} \]

\[ P \vdash x \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow x \]
\[ \text{MULTI} \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow e \]
\[ \text{CASE} \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow e \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow e \]
\[ \text{ABS} \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow e \]
\[ \text{MULTI} \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow e \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow e \]

\[ P \vdash e \downarrow d \Rightarrow (\varphi \downarrow d) \Rightarrow e \]

Figure 2: Demands and demand operations

Figure 3: Algorithmic cardinality analysis specification, part 1.

and metatheory are modulo-\(\equiv\) (checking that all our definitions do respect \(\equiv\) is routine).

3.2 Usage analysis

The analysis itself is shown in Figures 3 and 4. The main judgement form is written thus

\[ P \vdash e \downarrow d \Rightarrow (\tau ; \varphi) \Rightarrow e' \]

which should be read thus: in signature environment \(P\), and under usage demand \(d\), the term \(e\) places demands \((\tau ; \varphi)\) on its components, and elaborates to an annotated term \(e'\). The syntax of each of these components is given in Figures 1 and their roles in the judgement are the following:
For example, consider the expression
e = \lambda x . \text{case } x \text{ of } (p, q) \rightarrow (p, f \text{ True})
Suppose we place demand \( C^1(U) \) on \( e \), so that \( e \) is called, just once. What demand does it then place on its arguments and free variables?
\[
e \vdash e \downarrow C^1(U) \Rightarrow (1 \circ U(\omega \times U), A) \rightarrow \bullet; \{ f \rightarrow 1 \circ C^1(U) \}
\]
That is, \( e \) will use its argument once, its argument’s first component (the \( A \) in the usage type). Moreover \( e \) will call \( f \) just once.

In short, we think of the analysis as describing a \textit{demand transformer}, transforming a demand on the result of \( e \) into demands on its arguments and free variables.

### 3.3 Pairs and case expressions

With these definitions in mind, we can look at some of the analysis rules in Figure 3. Rule \text{PAIR} explains how to analyse a pair under a demand \( U(d_1^1, d_2^1) \). We simply analyse the two components, under \( d_1^1 \) or \( d_2^1 \) respectively, and combine the results with “\( \odot \)”. The auxiliary judgement \( \psi \) (Figure 5) deals with the multiplicity of the argument demands \( d_1^1 \).

The “\( \circ \)” operator, pronounced “both”, is defined in Figure 2 and combines the free-variable usages \( \varphi_1 \) and \( \varphi_2 \). For the most part the definition is straightforward, but there is a very important wrinkle for call demands:

\[
C^{\circ 1}(d_1) \& C^{\circ 2}(d_2) = C^{\circ}(d_1 \uplus d_2)
\]

The “\( \uplus \)” part is easy, since \( n_1 \) and \( n_2 \) are both at least 1. But note the switch from \& to the least upper bound \( \uplus \). To see why, consider what demand this expression places on \( f \):

\[
f \downarrow 1 \downarrow 2 \downarrow f \downarrow 3 \downarrow 4
\]

Each call gives a usage demand for \( f \) of \( 1 \uplus C^1(C^1(U)) \), and if we use \& to combine that demand with itself we get \( \omega \uplus C^\circ(C^\circ(U)) \). The inner \( \downarrow 1 \) is a consequence of the switch to \& and, rightly expresses the fact that no partial application of \( f \) is called more than once.

The other rules for pairs \text{PAIRU}, \text{PAIRHU}, and case expressions \text{CASE} should now be readily comprehensible (\( \varphi \downarrow x.y \) stands for the removal of \( \{x, y\} \) from the domain of \( \varphi \)).

### 3.4 Lambda and application

Rule \text{LAM} for lambdas expects the incoming demand to be a call demand \( C^n(d_1) \). Then it analyses the body \( e \) with demand \( d_1 \) to give \( \langle \tau ; \varphi \rangle \). If \( n = 1 \) the lambda is called at most once, so we can return \( \langle \tau ; \varphi \rangle \); but if \( n = \omega \) the lambda may be called more than once, and each call will place a new demand on the free variables. The \( n \uplus \varphi \) operation on the bottom line accounts for this multiplicity, and is defined in Figure 2. Rule \text{LAMHU} handles an incoming demand of \( U \) by treating it just like \( C^0(U) \), while \text{LAMHU} deals with the head-used demand \( HU \), where the lambda is not even called so we do not need to analyse the body, and \( e \) is obtained from \( e \) by adding arbitrary annotations. Similarly the return type \( \tau \) can be any type, since the \( \lambda \)-abstraction is not going to be applied, but is only head-used. Dually, given an application \( (e \ y) \), rule \text{APP} analyses \( e \) with demand \( C^0(d_1) \), reflecting that \( e \) is here called once. This returns the demand \( (d_1^1 \rightarrow \tau_1 ; \varphi_1) \) on the context. Then we can analyse the argument under demand \( d_1^1 \), using \( \uplus^1 \), yielding \( \varphi_2 \); and combine \( \varphi_1 \) and \( \varphi_2 \). Rule \text{APPB} applies when analysing \( e_1 \) yields the less-informative usage type \( \bullet \).

### 3.5 Usage signatures

Suppose we have the term

\[
\text{let } f = \lambda x . \lambda y . x \text{ True in } f \ p \ q
\]

We would like to determine the correct demands on \( p \) and \( q \), namely \( L\circ C^0(U) \) and \( A \) respectively. The gold standard would be to analyse \( f \)’s right-hand side at every call site; that is, to behave as if \( f \) were inlined at each call site. But that is not very modular; with deeply nested function definitions, it can be exponentially expensive to analyse each function body afresh at each call site; and it does not work at all for recursive functions. Instead, we want to analyse \( f \), summarise its behaviour, and then use that summary at each call site. This summary is called \( f \)’s \textit{usage signature}. Remember that the main judgement describes how a term transforms a demand for the value into demands on its context. So a usage signature must be a (conservative approximation of this) demand transformer.

There are many ways in which one might approximate \( f \)’s demand transformer, but rule \text{LETDN} (Figure 4) uses a particularly simple one:

- Look at \( f \)’s right hand side \( \lambda y_1 \ldots \lambda y_k . \ e_1 \), where \( e_1 \) is not a lambda-expression.
- Analyse \( e_1 \) in demand \( U \), giving \( \langle \tau_1 ; \varphi_1 \rangle \).
- Record the triple \( (k ; \varphi(\psi) \rightarrow \tau_1 ; \varphi_1 ; \varphi) \) as \( f \)'s usage signature in the environment \( P \) when analysing the body of the \text{let}.

Now, at a call site of \( f \), rule \text{VARDN} calls \text{transform}(\rho, d)\) to use the recorded usage signature \( \rho \) to transform the demand \( d \) for this occurrence of \( f \).

What does \text{transform}(\langle k ; \tau ; \varphi \rangle, d)\) do? If the demand \( d \) on \( f \) is stronger than \( C^0(C^0(U)) \), where the call demands are nested \( k \) deep, we can safely unleash \( \langle \tau ; \varphi \rangle \) at the call site. If not, we simply treat the function as if it were called many times, by unleashing \( \omega \uplus \tau ; \omega \uplus \varphi \), multiplying both the demand type \( \tau \) and the usage environment \( \varphi \) (Figure 2). Rule \text{LETDNABS} handles the case when the variable is not used in the body.

### 3.6 Thunks

The \text{LETDN} rule unleashes (an approximation to) the demands of the right-hand side at each usage site. This is good if the right hand side is a lambda, but not good otherwise, for two reasons. Consider

\[
\text{let } x = y + 1 \text{ in } x + x
\]

How many times is \( y \) demanded? Just once! The thunk \( x \) is demanded twice, but \( x \)’s thunk is memoised, so the \( y+1 \) is evaluated only once. So it is wrong to unleash a demand on \( y \) at each of \( x \)’s occurrence sites. Contrast the situation where \( x \) is a function

\[
\text{let } x = \lambda v . y + v \text{ in } x \ 42 + x \ 239
\]

Here \( y \) really is demanded twice, and \text{LETDN} does that. Another reason that \text{LETDN} would be sub-optimal for thunks is shown here:
The body of the `let` places usage demand \( 1 + U(U, A) \) on \( x \), and if we analysed \( x \)'s right-hand side in that demand we would see that \( q \) was unused. So we get more information if we wait until we know the aggregated demand on \( x \), and use it to analyse its right-hand side.

This idea is embodied in the \( \text{LetUp} \) rule, used if \( \text{LetDN} \) does not apply (i.e., the right hand side is not a lambda). Rule \( \text{LetUp} \) first analyses the body \( e_2 \) to get the demand \( \varphi_2(x) \) on \( x \); then analyses the right-hand side \( e_1 \) using that demand. Notice that the multiplicity \( n \) of the demand that \( e_2 \) places on \( x \) is ignored; that is because the thunk is memoised. Otherwise the rule is quite straightforward. Rule \( \text{LetUPAbs} \) deals with the case when the bound variable is unused in the body.

### 3.7 Elaboration

How are we to take advantage of our analysis? We do so by elaborating the term during analysis, with annotations of two kinds (see the syntax of annotated expressions in Figure 1):

- \( \text{let} \)-bindings carry an annotation \( m \in 0, 1, \omega \), to indicate how often the \( \text{let} \) binding is evaluated.
- Lambdas \( \lambda^m x.e \) carry an annotation \( m \in 0, 1, \omega \), to indicate how often the lambda is called. \( 0 \) serves as an indicator that the lambda is not supposed to be called at all.

Figure 3 shows the elaborated terms after the “\( \rightsquigarrow \)”. The operational semantics (Section 4) gets stuck if we use a thunk or lambda more often than its claimed usage; and the optimising transformations (Section 5) are guided by the same annotations.

### 3.8 A more realistic language

The implementation handles all of Haskell, or rather the Core language to which Haskell is translated by GHC. In particular:

- Usage signatures for constants \( \kappa \) are predefined.
- All data types with a single constructor (i.e., simple products) are treated analogously to pairs in the analysis.
- Recursive data types with more than one constructor and, correspondingly, case expressions with more than one alternative (and hence also conditional statements) are supported. The analysis is more approximate for such types: only usage demands that apply to such types are \( U \) and \( \text{HU} \) not \( U(d_1, d_2) \).
- Furthermore, case expressions with multiple branches give rise to a least upper bound \( \sqcup \) combination of usage types, as usual.
- Recursive functions and \( \text{let} \)-bindings are handled, using the standard kind of fixpoint iteration over a finite-height domain.

### 4. Soundness of the Analysis

We establish the soundness of our analysis in a sequence of steps. Soundness means that if the analysis claims that, say, a lambda is one-shot, then that lambda is only called once; and similarly for single-entry thunks. We formalise this property as follows:

- We present an operational semantics, written \( \dashv \rightarrow \), for the annotated language that counts how many times thunks have been evaluated and \( \lambda \)-abstractions have been applied. The semantics simply gets stuck when these counters reach zero, which will happen only if the claims of the analysis are false (Section 4.1).
- Our goal is to prove that if an expression \( e \) is elaborated to \( b \) by the analysis, then \( e \) in the instrumented semantics behaves identically to \( b \) in a standard un-instrumented call-by-need semantics (Section 4.3). For reasons of space we omit the rules for the un-instrumented call-by-need semantics which are completely standard (Sestoft 1997), and are identical to the rules of Figure 5 if one simply ignores all the annotations and the multiplicity side-conditions. We refer to this semantics as \( \rightarrow \).
- We prove soundness by giving a type system for the annotated terms, and showing that for well-typed terms, the instrumented semantics \( \dashv \rightarrow \) simulates \( \rightarrow \), in a type-preserving way.

### 4.1 Counting operational semantics

We present a simple counting operational semantics for annotated terms in Figure 5. This is a standard semantics for call-by-need, except for the fact that multiplicity annotations decorate the terms, stacks, and heaps. The syntax for heaps, denoted with \( H \), contains two forms of bindings, one for expressions \( [x \mapsto e] \) and one for already evaluated expressions \( [x \mapsto v] \). The multiplicity \( m \in \{0, 1, \omega \} \) denotes how many more times are we allowed to de-reference this particular binding. The stacks, denoted with \( S \), are just lists of frames. The syntax for frames includes application frames \( \langle y, e \rangle \), which store a reference \( y \) to an argument, case-frames \( \langle (x, y) \rightarrow e \rangle \), which account for the execution of a `case`-branch, and update frames of the form \#(\langle x, m \rangle), which take care of updating the heap when the active expression reduces to a value. The first component of an update frame is a name of a variable to be updated, and the second one is its thunk cardinality.

Rule \( \text{ELET} \) allocates a new binding on the heap. The rules \( \text{EBETA} \) fires only if the cardinality annotation is non-zero; it de-references an \( \text{Exp}(e) \) binding and emits an update frame. Rules \( \text{EBETA}, \text{EAPP}, \text{EPAIR} \) and \( \text{EPRRED} \) are standard. Notice that \( \text{EBETA} \) also fires only if the \( \lambda \)'s multiplicity \( m \) is non-zero. Note that the analysis does not assign zero-annotations to lambdas, but we need them for the soundness result.

Rule \( \text{ELKPV} \) de-references a binding for an already-evaluated expression \( [x \mapsto v] \), and in a standard semantics would return \( v \) leaving the heap unaffected. In our counting semantics however,
Moreover, the value it is now the active expression), the value de-reference of $m$.

We would like to prove that if we analyse a term $\rho$ the annotation on a $\langle H; e_1 : S_1 \rangle \mapsto \langle H_1 ; e_1 ; S_1 \rangle$

because these rules conjure up functions $\varrho$ out of thin air, and have universally quantified premises (in $WFTRANS$), they do not constitute an algorithm. But for the very same reasons they are convenient to reason about in the metatheory, and that is the only reason we need them. In effect, Figure 6 constitutes an elaborate invariant for the operational semantics.

4.3 Soundness of the analysis

The first result is almost trivial.

Lemma 4.1 (Analysis produces well-typed terms). If $P \vdash e \downarrow d \Rightarrow (\tau ; \varphi) \sim e$ then $P \vdash e \downarrow d \Rightarrow (\tau ; \varphi)$.

We would next like to show that well-typed terms do not get stuck.

To present the main result we need some notation first.

Figure 5: A non-deterministic counting operational semantics. The guards for counting restrictions are highlighted by grey boxes.

4.2 Checking well-annotated terms

We would like to prove that if we analyse a term $e$, producing an annotated term $e_1$ then if $e$ executes for a number of steps in the standard semantics $\Rightarrow$, then execution of $e$ does not get stuck in the instrumented semantics $\Rightarrow$ of Figure 5. To do this we need to prove preservation and progress lemmas, showing that each step takes a well-annotated term to a well-annotated term, and that well-annotated terms do not get stuck.

Figure 5 says what it means to be “well-annotated”, using notation from Figures 1 and 2. The rules look very similar to the analysis rules of Figure 5 except that we check an annotated term, rather than producing one. For example, rule $TLM$ checks that the annotation on a $\lambda$-abstraction $\lambda m . x . e$ is at least as large as the call cardinality we press on this $\lambda$-abstraction $m$. As evaluation progresses the terms get more concrete, so the annotations may become more conservative than the checkers require, but that is fine.

A more substantial difference is that instead of holding concrete demand transformers $\varrho$ as the analysis does (Figure 5), the environment $P$ holds generalised demand transformers $\varrho$. A generalised demand transformer is simply a monotone function from a demand to a pair $(\tau ; \varphi)$ of a type and a usage environment (Figure 5). In the TLEDSN rule, we clairvoyantly choose such a function $\varrho$, which is sound for the RHS expression – denoted with $P \vdash_\varrho e_1 : \varrho$.

We still check that that $e_1$ can be type checked with some demand $d_1$ that comes from type-checking the body of the let ($\varrho_2(x)$). In rule TVAR we simply apply the transformer $\varrho$ to get a type and tv-usage environment.

Rule $WFTRANS$ imposes two conditions necessary for the soundness of the transformer. First, it has to be a monotone function on the demand argument. Second, it has to soundly approximate any type and usage environment that we can attribute to the expression. One can easily confirm that the intensional representation used in the analysis satisfies both properties for the $\lambda$-expressions bound with LETDSN.
The demand exercised on $x$ from the body of the `let`-binding will be $C^1(U) \land C^1(U) = C^2(U)$ and hence the value $v$ will be checked against this demand (using the `LETUP` rule), unleashing an environment $\varphi$. However, after substituting $v$ in the body (which is ultimately what call-by-need will do) we will have checked it against $C^1(U)$ and $C^1(U)$ independently, unleashing $\varphi_1$ and $\varphi_2$ in each call site. Lemma 4.4 ensures that reduction never increases the demand on the free variables of the environment, and hence safety is not compromised. It is precisely the proof of Lemma 4.4 that requires demand transformers to be monotone in the demand arguments, ensured by `WTRANS`.

**Theorem 4.5 (Safety of analysis).** If $\epsilon \vdash e \downarrow HU \Rightarrow (\tau ; \epsilon) \Rightarrow e_1$ and $(\epsilon ; e_1 ; \epsilon) \Rightarrow^k (H ; e_2 ; S)$, then there exist $H, e_2$ and $S$, such that $(\epsilon ; e_1 ; \epsilon) \Rightarrow^k (H ; e_2 ; S)$, $H^1 = H$, $S^1 = S$ and $e_2 = e_2$.

The proof is just a combination of Lemma 4.4 and Theorem 4.2.

## 5. Optimisations

We discuss next the two optimisations enabled by our analysis.

### 5.1 Optimised allocation for thunks

We show here that for 0-annotated bindings there is no need to allocate an entry in the heap, and for 1-annotated ones we don’t have to emit an update frame on the stack. Within the chosen operational model, this optimisation is of *dynamic* flavour so we express this by providing a new, *optimising* small-step machine for the annotated expressions. The new semantics is defined in Figure 7. We will show that programs that can be evaluated via the counting semantics (Figure 5) can be also evaluated via the optimised semantics in a smaller or equal number of steps.

The proof is a simulation proof, hence we define relations between heaps / optimised heaps, and stacks / optimised stacks that are preserved during evaluation.

---

**Figure 6:** Well-annotated terms

**Figure 7:** Auxiliary simulation relation (heaps and stacks)

---

**Lemma 4.4 (Value demand splitting).** If $P \vdash v \downarrow (d_1 \& d_2)$ then there exists a split $(v \downarrow (v_1, v_2))$ such that:

\[
P \vdash v_1 \downarrow d_1 \Rightarrow (\tau ; \varphi_1) \quad \text{and} \quad P \vdash v_2 \downarrow d_2 \Rightarrow (\tau_2 ; \varphi_2) \quad \text{and} \quad \tau_1 \subseteq \tau, \tau_2 \subseteq \tau \text{ and } \varphi_1 \& \varphi_2 \subseteq \varphi.
\]

**Why is Lemma 4.4 important?** Consider the following:

`let x = v in case x of (y, z) -> x * y`
Definition 5.1 (Auxiliary \(\alpha\)-relations). We write \(e_1 \approx e_2\) iff \(e_1\) and \(e_2\) differ only on the \(\lambda\)-annotations. \(H_1 \approx H_2\) and \(S_1 \approx S_2\) are defined in Figure 7.

For this optimisation the annotations on \(\lambda\)-abstractions play no role, hence we relate any expressions that differ on only those.

Figure 7 tells us that a heap \(H\) is related with an optimised heap \(H_{opt}\) with the relation \(H \approx H_{opt}\). As we have described, there are no bindings of the form \(\lambda^m x. e^j\) in the optimised heap. Moreover notice that there are no bindings of the form \(\lambda^m x. e\) in the optimised heap. It is easy to see why: every heap binding starts life as \(\lambda \overset{m}{\to} Exp(e)\). By the time \(Exp(e)\) has become a value \(Val(v)\), we have already used \(x\) once. Hence, if originally \(m = \omega\) then the value binding will also be \(\omega\) (in the optimised or unoptimised semantics). If it was \(m = 1\) then it can only be \(0\) in the un-optimised heap and non-existent in the optimised heap. If it was \(m = 0\) then no such bindings would have existed in the optimised heap anyway.

The relation between stacks is given with \(S \approx S_{opt}\). Rule SSM2 ensures that there are no frames \(\#(x, 0)\) in the optimised stack. In fact during evaluation it is easy to observe that there are not going to be any update frames \(\#(x, 0)\) in the original or optimised stack.

We can now state the optimisation theorem that follows.

Theorem 5.1 (Optimised semantics). If \((H_1; e_1; S_1) \approx (H_2; e_2; S_2)\) and \((H_1; e_1; S_1) \rightarrow (H_1'; e_1'; S_1')\) then there exists \(k \in \{0, 1\}\) s.t. \((H_2; e_2; S_2) \rightarrow^k (H_2'; e_2'; S_2')\) and \((H_1'; e_1'; S_1') \approx (H_2'; e_2'; S_2')\).

Notice that the counting semantics may not be able to take a transition at some point due to the wrong non-deterministic choice but in that case the statement of Theorem 5.1 holds trivially. Finally, we tie together Theorems 5.1 and 4.3 to get the following result.

Theorem 5.2 (Analysis is safe for optimised semantics). If \(\bullet \vdash e_1 \downarrow\) then \((H; e_1; e) \rightarrow (H; e_2; S)\) then \((H; e_1; e) \rightarrow (H; e_2; S)\) then \((H; e_1; e) \rightarrow (H; e_2; S)\) and there exist \(H_2\) and \(S_2\) s.t. \(H_2 \approx H\) and \(S_2 \approx S\). Theorem 5.2 says that if a program \(e_1\) evaluates in \(n\) steps to \(e_2\) in the reference semantics, then it also evaluates to the same \(e_2\) (modulo annotation) in the optimised semantics in \(n\) steps or fewer; and the heaps and stacks are consistent. Moreover, the theorem has informative content on infinite sequences. For example it says that for any point in the evaluation in the reference configuration we will have earlier reached a corresponding intermediate configuration in the optimised semantics with consistent heaps and stacks.

5.2 Let-in floating into one-shot lambdas

As discussed in Section 2 we are interested in the particular case of let-floating (Peyton Jones et al. 1996), moving the binder into the body of a lambda-expression. This transformation is trivially safe, given obvious syntactic side conditions (Moran and Sands 1999 §4.5), however, in general, it is not beneficial. Here we describe the conditions under which let-in floating makes things better in terms of the length of the program execution sequence.

We start by defining let-in floating in a form of syntactic rewriting:

Definition 5.2 (let-in floating for one-shot lambdas). Let \(\lambda^m x. e\) \(\rightarrow^\beta (\lambda^m x. \epsilon \in e)\) \(\rightarrow (\lambda^m x. \epsilon \in e)\) \(\epsilon \in e\)

for any \(m\), \(m_2\) and \(x \notin FV(e_2)\).

Next, we provide a number of definitions necessary to formulate the so called improvement result (Moran and Sands 1999). The improvement is formulated for closed, well-formed configurations. For a configuration \((H; e; S)\) to be closed, any free variables in \(H\), \(e\) and \(S\) must be contained in a union \(dom(H) \cup dom(S)\), where \(dom(H)\) is a set of variables bound by a heap \(H\), and \(dom(S)\) is a set of variables marked for update in a stack \(S\). A configuration is well-formed if \(dom(H)\) and \(dom(S)\) are disjoint.

Definition 5.3 (Convergence). For a closed configuration \((H; e; S)\), \(\langle H; e; S \rangle \uparrow^N \equiv \exists H', v, N. (\langle H; e; S \rangle \rightarrow^N \langle H'; v; e \rangle)\)

\(\langle H; e; S \rangle \uparrow^M \equiv \exists M. \langle H; e; S \rangle \uparrow^M \) and \(M \leq N\)

The following theorem shows that local let-in floating into the body of a one-shot lambda does not make the execution longer.

Theorem 5.3 (Let-in float improvement). For any \(H\) and \(S\), if \(\langle H; let z \equiv e_1 \in (let f \equiv \lambda^k x. e \in e_2) ; S \rangle \uparrow^N\) and \(z \notin FV(e_2)\), then \(\langle H; let f \equiv \lambda^k x. e \in e_2 ; S \rangle \uparrow^N\).

Even though Theorem 5.3 gives a termination-dependent result, its proof (Sergey et al. 2013) goes via a simulation argument, hence it is possible to state the theorem in a more general way without requiring termination.

We also expect that the improvement result extends to arbitrary program contexts, but have not carried out the exercise.

<table>
<thead>
<tr>
<th>(\langle H_0; e_0; S_0 \rangle \implies \langle H_1; e_1; S_1 \rangle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT-ELETA (\langle H; let x \equiv e_1 \in e_2 ; S \rangle) (\implies \langle H; e_2 ; S \rangle)</td>
</tr>
<tr>
<td>OPT-ELLETU (\langle H; let x \equiv e_1 \in e_2 ; S \rangle) (\implies \langle H[x \overset{n}{\to} Exp(e_1)] ; e_2 ; S \rangle) where (n \geq 1)</td>
</tr>
<tr>
<td>OPT-ELKEP (\langle H; x \overset{n}{\to} Exp(e) ; x ; S \rangle) (\implies \langle H ; e ; #(x, \omega) ; S \rangle)</td>
</tr>
<tr>
<td>OPT-ELKEPO (\langle H; x \overset{n}{\to} Exp(e) ; x ; S \rangle) (\implies \langle H ; e ; S \rangle)</td>
</tr>
<tr>
<td>OPT-ELKpV (\langle H; x \overset{n}{\to} Val(v) ; x ; S \rangle) (\implies \langle H [x \overset{n}{\to} Val(v)] ; v ; S \rangle)</td>
</tr>
<tr>
<td>OPT-EUPD (\langle H; v ; #(x, \omega) ; S \rangle) (\implies \langle H [x \overset{n}{\to} Val(v)] ; v ; S \rangle)</td>
</tr>
<tr>
<td>OPT-EBETA (\langle H; \lambda^m x. e ; (\bullet) y ; S \rangle) (\implies \langle H ; e[y/x] ; S \rangle)</td>
</tr>
<tr>
<td>OPT-EAPP (\langle H; e ; y ; S \rangle) (\implies \langle H ; e ; (\bullet) y ; S \rangle)</td>
</tr>
<tr>
<td>OPT-EPPAIR (\langle H; case e_0 of (x, y) \to e_1 ; S \rangle) (\implies \langle H ; e_1 ; (x, y) \to e_1 ; S \rangle)</td>
</tr>
<tr>
<td>OPT-EPRED (\langle H; (x_1, x_2) ; S \rangle \langle (y_1, y_2) \to e_1 ; S \rangle) (\implies \langle H ; e_1 ; [x_1, y_1, y_2, y_2] ; S \rangle)</td>
</tr>
</tbody>
</table>
6. Implementation

We have implemented the cardinality analyser by extending the demand analysis machinery of the Glasgow Haskell Compiler, available from its open-source repository. We briefly summarise some implementation specifics in this section.

6.1 Analysis

The implementation of the analysis was straightforward, because GHC’s existing strictness analyser is already cast as a backwards analysis, exactly like our new cardinality analysis. So the existing analyser worked unchanged; all that was required was to enrich the domains over which the analyser works. In total, the analyser increased from 900 lines of code to 1,140 lines, an extremely modest change.

We run the analysis twice, once in the middle of the optimisation pipeline, and once near the end. The purpose of the first run is to expose one-shot lambdas, which in turn enable a cascade of subsequent transformations (Section 6.3). The second analysis finds the single-entry thunks, which are exploited by the code generator. This second analysis is performed very late in the pipeline (a) so that it sees the result of all previous inlining and optimisation and (b) because the single-entry thunk information is not robust to certain other transformations (Section 6.4).

6.2 Absence

GHC exploits absence in the worker/wrapper split, as described in Section 2.3: absent arguments are not passed from the wrapper to the worker.

6.3 One-shot lambdas

As shown in Section 5.2, there is no run-time payoff for one-shot lambdas. Rather, the information enables some important compile-time transformations. Specifically, consider

\[
\text{let } x = \text{costly } v \text{ in } \ldots (\lambda y \ldots x \ldots) \ldots
\]

If the \(\lambda y\) is a one-shot lambda, the binding for \(x\) can be floated inside the lambda, without risk of duplicating the computation of costly. Once the binding for \(x\) is inside the \(\lambda y\), several other improvements may happen:

- It may be inlined at \(x\)’s use site, perhaps entirely eliminating the allocation of a thunk for \(x\).
- It may enable a rewrite rule (eg foldr/build fusion) to fire.
- It may allow two lambdas to be replaced by one. For example

\[
\begin{align*}
\text{let } y = 1 & \Rightarrow \text{let } x = y + 0 \text{ in } x \times x \\
\text{let } y = e & \Rightarrow \text{let } x = y \text{ in } x \times x \\
\text{Inline } x & \Rightarrow \text{let } y = e \text{ in } y \times y
\end{align*}
\]

This does not affect the formal results of the paper, but it is the reason that our second run of the cardinality analysis is immediately before code generation.

6.4 Single-entry thunks

The code that GHC compiles for a thunk begins with pushing an update frame on the stack, which includes a pointer to the thunk. Then the code for the thunk is executed. When evaluation is complete, the value is returned, and the update frame overwrites the thunk with an indirection to the values. It is easy to modify this mechanism to take advantage of single-entry thunks: we do not generate the push-update-frame code for single-entry thunks. There is a modest code size saving (fewer instructions generated) and a modest runtime saving (a few store instructions saved on thunk entry, and a few more when evaluation is complete).

Take care though! The single-entry property is not robust to program transformation. For example, common sub-expression elimination (CSE) can combine two single-entry thunks into one multiple-entry one, as can this sequence of transformations:

\[
\begin{align*}
\text{let } x = \text{costly } v \text{ in } \ldots (\lambda y \ldots x \ldots) \ldots
\end{align*}
\]

This does not affect the formal results of the paper, but it is the reason that our second run of the cardinality analysis is immediately before code generation.

7. Evaluation

To measure the accuracy of the analysis, we counted the proportion of (a) one-shot lambdas and (b) single-entry thunks. In both cases, these percentages are of the syntactically occurring lambdas or thunks respectively, measured over the code of the benchmark program only, not library code. Table 1 shows the results reported by our analysis for programs from the nofib benchmark suite. The numbers are quite encouraging. One-shot lambdas account for 0-30% of all lambdas, while single-entry thunks are 0-23% of all thunks.

The static (syntactic) frequency of single-entry thunks may be very different to their dynamic frequency in a program execution, so we instrumented GHC to measure the latter. (We did not measure the dynamic frequency of one-shot lambdas, because they confer no direct performance benefit.) The “RT Thnk” column of Table 1 gives the dynamic frequency of single-entry thunks in the same nofib programs. Note that these statistics include single-entry thunks from libraries, as well as the benchmark program code. The results vary widely. Most programs do not appear to use single-entry thunks much, while a few use many, up to 74% for cryptarithm2.

### Table 1. Analysis results for nofib: ratios of syntactic one-shot lambdas (Synt. \(\lambda^1\)), syntactic used-once thunks (Synt. Thnk\(^1\)) and runtime entries into single-entry thunks (RT Thnk\(^1\)).

<table>
<thead>
<tr>
<th>Program</th>
<th>Synt. (\lambda^1)</th>
<th>Synt. Thnk(^1)</th>
<th>RT Thnk(^1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>anna</td>
<td>4.0%</td>
<td>7.2%</td>
<td>2.9%</td>
</tr>
<tr>
<td>bact</td>
<td>5.0%</td>
<td>15.4%</td>
<td>1.5%</td>
</tr>
<tr>
<td>cacheprof</td>
<td>7.6%</td>
<td>11.9%</td>
<td>5.1%</td>
</tr>
<tr>
<td>calendar</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.2%</td>
</tr>
<tr>
<td>constraints</td>
<td>2.0%</td>
<td>3.2%</td>
<td>4.5%</td>
</tr>
<tr>
<td>cryptarithm2</td>
<td>0.6%</td>
<td>3.0%</td>
<td>74.0%</td>
</tr>
<tr>
<td>gcd</td>
<td>12.5%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>gen_regexp</td>
<td>5.6%</td>
<td>0.0%</td>
<td>0.2%</td>
</tr>
<tr>
<td>hpg</td>
<td>5.2%</td>
<td>0.0%</td>
<td>4.1%</td>
</tr>
<tr>
<td>integer</td>
<td>8.3%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>life</td>
<td>3.2%</td>
<td>0.0%</td>
<td>1.8%</td>
</tr>
<tr>
<td>mkhprog</td>
<td>27.4%</td>
<td>20.8%</td>
<td>5.8%</td>
</tr>
<tr>
<td>nucleic2</td>
<td>3.5%</td>
<td>3.1%</td>
<td>3.2%</td>
</tr>
<tr>
<td>partstof</td>
<td>5.8%</td>
<td>10.7%</td>
<td>0.1%</td>
</tr>
<tr>
<td>sphere</td>
<td>7.8%</td>
<td>6.2%</td>
<td>20.0%</td>
</tr>
</tbody>
</table>

... and 72 more programs

Arithmetic mean | 10.3% | 12.6% | 5.5%


[2] This claim is true in spirit, but in practice we substantially refactored the existing analyser when adding usage cardinalities.

[3] These statistics include single-entry thunks from libraries, as well as the benchmark program code.
any performance benefits directly; rather, they remove potential obstacles from other compile-time transformations. Single-entry thunks, on the other hand give an immediate performance benefit, by omitting the push-update-frame code, but it is a small one.

Table 2 summarises the effect of cardinality analysis when running the nofib suite. “Allocs” is the change in how much heap was allocated when the program is run and “Runtime” is a change in the actual program execution time.

In Section 4 we mentioned a hack, used by Gill in GHC, in which he hard-coded the call-cardinality information for three particular functions: build, foldr and runST. Our analysis renders this hack redundant, as now the same results can be soundly inferred. We therefore report two sets of results: relative to an un-hacked baseline, and relative to a hacked baseline. In both cases binary size of the (statically) linked binaries falls slightly but consistently (2.0% average), which is welcome. This may be due to less push-update-frame code being generated.

Considering allocation, the numbers relative to the non-hacked baseline are quite encouraging, but relative to the hacked compiler the improvements are modest: the hack was very effective! Otherwise, only one program, nucleic2 shows a significant (11%) reduction in allocation, which turned out to be because a thunk was floated inside a one-shot lambda and ended up never being allocated, exactly as advertised.

A shortcoming of nofib suite is that runtimes tend to be short and very noisy: even with the execution key slow only 18 programs from the suite run for longer than half second (with a maximum of 2.5 seconds for constraints). Among those long-runners the biggest performance improvement is 8.8% (for integer), with an average of 2.3%.

Table 2. Cardinality analysis-enabled optimisations for nofib

<table>
<thead>
<tr>
<th>Program</th>
<th>Allocs</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No hack</td>
<td>Hack</td>
</tr>
<tr>
<td>Anna</td>
<td>-2.1%</td>
<td>-0.2%</td>
</tr>
<tr>
<td>Bept</td>
<td>-2.2%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>Cacheprof</td>
<td>-7.9%</td>
<td>-0.6%</td>
</tr>
<tr>
<td>Calendar</td>
<td>-9.2%</td>
<td>+0.2%</td>
</tr>
<tr>
<td>Constraints</td>
<td>-0.9%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>Cryptarithm</td>
<td>-0.3%</td>
<td>-0.3%</td>
</tr>
<tr>
<td>Gcd</td>
<td>-15.5%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>Gen_regexps</td>
<td>-1.0%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>Hgp</td>
<td>-2.0%</td>
<td>-1.0%</td>
</tr>
<tr>
<td>Integer</td>
<td>-0.0%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>Life</td>
<td>-0.8%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Mkprogs</td>
<td>-11.9%</td>
<td>+0.1%</td>
</tr>
<tr>
<td>Nucleic2</td>
<td>-14.1%</td>
<td>-10.9%</td>
</tr>
<tr>
<td>Partstof</td>
<td>-95.5%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>Sphere</td>
<td>-1.5%</td>
<td>-1.5%</td>
</tr>
<tr>
<td>Min</td>
<td>-95.5%</td>
<td>-10.9%</td>
</tr>
<tr>
<td>Max</td>
<td>+3.5%</td>
<td>+0.5%</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>-6.0%</td>
<td>-0.3%</td>
</tr>
</tbody>
</table>

Table 3. Optimisation of the programs from Benchmarks Game

<table>
<thead>
<tr>
<th>Program</th>
<th>RT Thnk</th>
<th>No-Opt RT</th>
<th>RT Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary-trees</td>
<td>49.4%</td>
<td>66.83 s</td>
<td>-9.2%</td>
</tr>
<tr>
<td>Fannkuch-redu</td>
<td>0.0%</td>
<td>158.94 s</td>
<td>-3.7%</td>
</tr>
<tr>
<td>N-body</td>
<td>5.7%</td>
<td>38.41 s</td>
<td>-4.4%</td>
</tr>
<tr>
<td>Pidigits</td>
<td>8.8%</td>
<td>41.56 s</td>
<td>-0.3%</td>
</tr>
<tr>
<td>Spectral-norm</td>
<td>4.6%</td>
<td>17.83 s</td>
<td>-1.7%</td>
</tr>
</tbody>
</table>

Table 4. Analysis and optimisation results for hackage libraries

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>GHC Alloc Δ</th>
<th>GHC RT Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>No hack</td>
<td>Hack</td>
<td>No hack</td>
<td>Hack</td>
</tr>
<tr>
<td>Anna</td>
<td>5740</td>
<td>-1.6%</td>
<td>-1.5%</td>
</tr>
<tr>
<td>Cacheprof</td>
<td>1600</td>
<td>-1.7%</td>
<td>-0.4%</td>
</tr>
<tr>
<td>Fluid</td>
<td>1579</td>
<td>-1.9%</td>
<td>-1.9%</td>
</tr>
<tr>
<td>Gameb</td>
<td>1933</td>
<td>-0.5%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>Parser</td>
<td>2379</td>
<td>-0.7%</td>
<td>-0.2%</td>
</tr>
<tr>
<td>Veritas</td>
<td>4674</td>
<td>-1.4%</td>
<td>-0.3%</td>
</tr>
</tbody>
</table>

Table 5. Compilation with optimised GHC

For more realistic numbers, we measured the improvement in runtime, relative to the hacked compiler, for several programs from the Computer Language Benchmarks Game. The results are shown in Table 3. All programs were run with the official shootout settings (except spectral-norm, to which we gave a bigger input value of 7500) on a 2.7 GHz Intel Core i7 OS X machine with 8 Gb RAM. These are uncharacteristic Haskell programs, optimised to within an inch of their life by dedicated Haskell hackers. There is no easy meat to be had, and indeed the heap-allocation changes are so tiny (usually zero, and -0.2% at the most in the case of binary-trees) that we omit them from the table. However, we do get one joyful result: a solid speedup of 9.2% in binary-trees due to fewer thunk updates. As you can see, nearly half of its thunks entered at runtime are single-entry.

7.2 Real-world programs

To test our analysis and the cardinality-powered optimisations on some real-world programs, we chose four continuation-heavy libraries from the hackage repository: attoparsec, a fast parser combinator library, binary, a lazy binary serialisation library, bytestring, a space-efficient implementation of byte-vectors, and cassava, a CSV parsing and encoding library.

These libraries come with accompanying benchmark suites, which we ran both for the baseline compiler and the cardinality-powered one. Table 4 contains the ratios of syntactic one-shot lambdas and used-once thunks for the libraries, as well relative improvement in memory allocation for particular benchmarks. Since we were interested only in the absolute improvement against the state of

One can notice that the new compiler sometimes performs worse than the cardinality-unaware versions in a very few benchmarks in nofib. In a highly optimising compiler with many passes it is very hard to ensure that every “optimisation” always makes the program run faster, and, even if a pass does improve the program per se, to ensure that every subsequent pass will carry out all the optimisations that it did before the earlier improvement was implemented. The data show that we do not always succeed. We leave for the future some detailed forensic work to find out exactly why.
the art, we made our comparison with respect to the contemporary version of (hacked) baseline GHC. The encouraging results for attparsac are explained by its relatively high ratio of one-shot lambdas, which is typical for parser combinator libraries.

GHC itself is a very large Haskell program, written in a variety of styles, so we compiled it with and without cardinality-powered optimisations, and measured the allocation and runtime improvement when using the two variants to compile several programs. The results are shown in Table 5. As in the other cases, we get modest but consistent improvements.

8. Related Work

8.1 Abstract interpretation for usage and absence

The goal of the traditional usage/absence analyses is to figure out which parts of the programs are used, and which are not (Peyton Jones and Partain 1994). This question was first studied in the late 80's, when an elegant representation of the usage analysis in terms of projections (Hinze 1995) was given by Wadler and Hughes (Wadler and Hughes 1987). Their formulation allows one to define a backwards analysis — inferring the usage of arguments of a function from the usage of its result — an idea that we adopted wholesale. Our work has important differences, notably (a) call demands \( C^d(d) \), which appear to be entirely new; and (b) the ability to treat nested lambdas, which requires us to capture the usage of free variables in a usage signature. Moreover our formal underpinning is quite different to their (denotational) approach, because we fundamentally must model sharing.

8.2 Type-based approaches

The notion of “single-entry” thunks and “one-shot” lambdas is reminiscent of linear types (Girard 1995; Turner and Wadler 1999), a similarity that was noticed very early (Launchbury et al. 1993). Linear types per se are far too restrictive (see, for example, Wansbrough and Peyton Jones 1999) for details), but the idea of using a type system to express usage information inspired a series of “once upon a type” papers (Gustavsson 1998; Turner et al. 1999; Wansbrough 2002; Wansbrough and Peyton Jones 1999). Alas, a promising idea turned out to lead, step by step, into a deep swamp. Firstly, subtyping proved to be essential, so that a function that used its argument once could have a type like \( Int^1 \rightarrow Int \), but still be applied to an argument \( x \) that was used many times and had type \( Int^n \) (Wansbrough and Peyton Jones 1999). Then usage polymorphism proved essential to cope with currying: “[Using the monomorphic system] in the entirety of the standard libraries, just two thunks were annotated as used-once” (Wansbrough 2002). Gustavsson advocated bounded polymorphism to gain greater precision (Gustavsson and Svenningsson 2001), while Wansbrough extended usage polymorphism to data types, sometimes resulting in data types with many tens of usage parameters. The interaction of ordinary type polymorphism with all these usage-type features was far from straightforward. The inference algorithm for a polymorphic type system with bounds and subtyping is extremely complex. And so on. Burdened with these intellectual and implementation complexities, Wansbrough's heroic prototype in GHC (around 2,580 LOC of brand-new code; plus pervasive changes to thousands of lines of code elsewhere) turned out to be unsustainable, and never made it into the main trunk.

Our system sidesteps these difficulties entirely by treating the problem as a backwards analysis like strictness analysis, rather than as a type system. This is what gives the simplicity to our approach, but also prevents it from giving “rich” demand signatures to third- and higher-order functions: our usage types can account uniformly only for the first- and second-order functions, thanks to call demands. For example what type might we attribute to

\[
 f \times g = g x
\]

The usage of \( x \) depends on the particular \( g \) in the call, so usage polymorphism would be called for. This is indeed more expressive but it is also more complicated. We deliberately limit precision for very higher-order programs, to gain simplicity.

At some level abstract interpretation and type inference can be seen as different sides of the same coin, but there are some interesting differences. For example, our LETD\textsubscript{N} and LETU\textsubscript{R} rules are explicit about information flow; in the former, information flows from the definition of a function to its uses, while in the latter the flow is reversed. Type systems use unification variables to allow much richer information flow — but at the cost of generating constraints involving subtyping and bounds that are tricky to solve.

Another intriguing difference is in the handling of free variables:

\[
 \text{let } f = \lambda x. y + x \text{ in } \text{if } b \text{ then } f \text{ else } y
\]

How many times is the free variable \( y \) evaluated in this expression? Obviously just once, and LETD\textsubscript{N} discovers this, because we unleash the demand on \( y \) at \( f \)'s call site, and lub the two branches of the \( \text{if} \). But type systems behave like LETU\textsubscript{P}: compute the demand on \( f \) (namely, called once) and from that compute the demand on \( y \). Then combine the demand on \( y \) from the body of the \text{let} (used at most once), and from \( f \)'s right hand side (used at most once), yielding the result that \( y \) is used many times. We have lost the fact that the two uses come from different branches of the conditional. The fact that our usage signatures include the \( \varphi \) component makes them more expressive than mere types — unless we extend the type system yet further with an polymorphic effect system (Hage et al. 2007; Holdeman and Hage 2010). Moreover, the analysis approach deals very naturally with absence, and with product types such as pairs, which are ubiquitous. Type-based approaches do not do so well here.

In short, an analysis-based approach has proved much simpler intellectually than the type-based one, and far easier to implement. One might wonder if a clever type system might give better results in practice, but Wansbrough’s results (mostly zero change to allocation) were no more compelling than those we report. Our proof technique does however share much in common with Wansbrough and Gustavsson’s work, all three being based on an operational semantics with an explicit heap.

One other prominent type-based usage system is Clean’s uniqueness types (Barendsen and Smetsers 1996). Clean’s notion of uniqueness is, however, fundamentally different to ours. In Clean a unique-typed argument places a restriction on the \text{caller} (to pass the only copy of the value), whereas for us a single-entry argument is a promise by \text{callee} (to evaluate the argument at most once).

8.3 Other related work

Call demands, introduced in this paper, appear to be related to the notion of \textit{applicativeness}, employed in the recent work on relevance typing (Holdeman and Hage 2010). In particular, applicativeness means that an expression either “guaranteed to be applied to an argument” (S), or “may not be applied to an argument” (L). In this terminology \texttt{S} corresponds to a “strong” version of our demands \( C^d(d) \), which requires \( d \subseteq U \), and \texttt{L} is similar to our \texttt{U}. The \texttt{eq} like-evaluation of expressions corresponds to our demand \texttt{HU}. However, neither call- nor thunk-cardinality are captured by the concept of applicativeness.
Abstract counting or sharing analysis conservatively determines which parts of the program might be used by several components or accessed several times in the course of execution. Early work employed a forward abstract interpretation framework (Goldberg [1987] [Hudak [1986]). Since the forward abstract interpreter makes assumptions about arguments of a function it examines, the abstract interpretation can account for multiple combinations of those and may, therefore, be extremely expensive to compute.

Recent development on the systematic construction of abstract interpretation-based static analyses for higher-order programs, known as abstracted abstract machines (AAM), makes it straightforward to derive an analyser from an existing small-step operational semantics, rather than come up with an ad-hoc non-standard one (Van Horn and Might [2010]). This approach also greatly simplifies integration of the counting abstract domain to account for sharing (Might and Shivers [2005]). However, the abstract interpreters obtained this way are whole-program forward analysers, which makes them non-modular. It would be, however, an interesting future work to build a backwards analysis from AAM.

9. Conclusion

Cardinality analysis is simple to implement (it added 250 lines of code to a 140,000 line compiler), and it gives real improvements for serious programs, not just for toy benchmarks; for example, GHC itself (a very large Haskell program) runs 4% faster. In the context of a 20-year-old optimising compiler, a gain of this magnitude is a serious program, not just for toy benchmarks; for example, GHC itself (a very large Haskell program) runs 4% faster. In the context of a 20-year-old optimising compiler, a gain of this magnitude is a solid win.

Acknowledgements

We are grateful to Johan Tibell for the suggestion to use benchmark-accompanied haskage libraries and the cabal bench utility for the experiments in Section 7.2. We also thank the POPL'14 reviewers for their useful feedback.

References


