Modular, Higher-Order Cardinality Analysis in Theory and Practice

Ilya Sergey
IMDEA Software Institute
ilya.sergey@imdea.org

Dimitrios Vytiniotis Simon Peyton Jones
Microsoft Research
{dimitris,simonpj}@microsoft.com

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:

```
wurble1, wurble2 :: (Int -> Int) -> Int
wurble1 k = sum (map k [1..10])
wurble2 k = 2 * k 0

f1 :: [Int] -> Int
f1 xs = let ys = map costly xs
        in  wurble (\n. sum (map (+ n) ys))
```

Here we assume that `costly` is some function that is expensive to compute and `wurble` is either `wurble1` or `wurble2`. If we replace `ys` by its definition, we could transform `f1` into `f2`:

```
f2 xs = wurble (\n. sum (map (+ n) (map costly xs)))
```

A compiler like GHC can now use *short-cut deforestation* to fuse the two `map`s into one, eliminating the intermediate list altogether, and offering a substantial performance win (Gill et al. 1993).

POPL '14, January 22–24, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2535838.2535861

Does this transformation make the program run faster or slower? It depends on `wurble`! For example, `wurble1` calls its function argument ten times, so if `wurble` = `wurble1`, function `f2` would compute `costly` ten times for each element of `xs`; whereas `f1` would do so only once. On the other hand if `wurble` = `wurble2`, which calls its argument exactly once, then `f2` is just as efficient as `f1`, and `map`/`map` fusion can improve it further.

The reverse is also true. If the programmer writes `f2` in the first place, the *full laziness transformation* (Peyton Jones et al. 1996) will float the sub-expression `(map costly xs)` out of the `\n`-expression, so that it can be shared. That would be good for `wurble1` but bad for `wurble2`.

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 *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* is usually one-shot. So cardinality analysis can be a big win when optimising continuation-heavy programs.

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 of the function `build`:

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

You can see that `build` calls its argument exactly once, and inlining `ys` in calls like `(build (\cn. ...ys...))` 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 implementation (still extant in GHC today) relies on a gross hack: he taught GHC's optimiser to behave specially for `build` itself, and a couple of other functions. No user-defined function will have this good behaviour. Our analysis subsumes the hack, by providing an analysis that deduces the correct one-shot information for `build`, as well as many other functions.

2.2 Currying

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

```
f3 a = wurble a (\x.let t = costly x in \y. t+y)

wurble1 a g = g 2 a + g 3 a
wurble2 a g = sum (map (g a) [1..1000])
```

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

```
f4 a = wurble1 a (\x.\y. costly x + y)
```

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

So our analysis of `wurble2` must be able to report "`wurble2`'s argument `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 `wurble`, like this:

$$\begin{array}{lcl} \texttt{wurble1} & :: & U \to C^\omega(C^1(U)) \to \bullet \\ \texttt{wurble2} & :: & U \to C^1(C^\omega(U)) \to \bullet \end{array}$$

The notation $C^\omega(C^1(U))$ is a *usage demand*: it describes how a (function) value is used. The demand type $U \to C^\omega(C^1(U)) \to \bullet$ 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 $C^1(d)$ means "this argument is called *once* (applied to one argument), and the result is used with usage d", whereas $C^\omega(d)$ 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 `wurble1`'s second argument usage is $C^\omega(C^1(U))$, *not* $C^\omega(C^\omega(U))$; that is, in all cases the result of applying `g` to one argument is then called only once.

2.3 Absence

Consider this function

```
f x = case x of (p,q) -> <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 = case x of (p,q) -> fw p q
fw p q = <cbody>
```

Now `f` (the "wrapper") is small, and can be inlined at `f`'s call sites, often eliminating the allocation of the pair; meanwhile `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 `<fbody>`. Then we would transform to

```
f x = case x of (p,q) -> fw p q
fw p q = let x = (p,q) in <fbody>
```

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

But what if `<fbody>` did not use `q` at all? Then it would be stupid to pass `q` to `fw`. We would rather transform to:

```
f x = case x of (p,q) -> fw p
fw p = let x = (p, error "urk") in <fbody>
```

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:

$$\texttt{f} :: U(U, A) \to \bullet$$

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.

[1] We will always use "called" to mean "applied to one argument".

2.4 Thunk cardinality

Consider these definitions

```
f :: Int -> Int -> Int
f x c = if x > 0  then c + 1 else
        if x == 0 then 0     else c - 1

g y = f y (costly y)
```

Since `f` is not strict in `c`, `g` must build a thunk for `(costly y)` to pass to `f`. In call-by-need 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:

$$\texttt{f} :: \omega * U \to 1 * U \to \bullet$$

The "$\omega *$" modifier says that `f` may evaluate its first argument more than once, while the "$1 *$" 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 `seq` 1     -- f1  ::  1*U → •
f2 g = g `seq` g 2   -- f2  ::  ω*C¹(U) → •
f3 g = g 3           -- f3  ::  1*C¹(U) → •
```

The function `seq` evaluates its first argument (to 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 κ 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:

- $U(d_1^\dagger, d_2^\dagger)$ applies to pairs. The pair itself is evaluated and its first component is used as described by $d_1^\dagger$ and its second by $d_2^\dagger$.

Expressions and values

$$\begin{array}{rcl} e & ::= & x \mid v \mid e\ x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ & & \mid \texttt{case}\ e_1\ \texttt{of}\ (x_1, x_2) \to e_2 \\ v & ::= & \kappa \mid \lambda x\,.\,e \mid (x_1, x_2) \end{array}$$

Annotated expressions and values

$$\begin{array}{rcl} \texttt{e} & ::= & x \mid \texttt{v} \mid \texttt{e}\ x \mid \texttt{let}\ x \stackrel{m}{=} \texttt{e}_1\ \texttt{in}\ \texttt{e}_2 \\ & & \mid \texttt{case}\ \texttt{e}_1\ \texttt{of}\ (x_1, x_2) \to \texttt{e}_2 \\ \texttt{v} & ::= & \kappa \mid \lambda^m x\,.\,\texttt{e} \mid (x_1, x_2) \end{array}$$

Usage demands and multi-demands

$$\begin{array}{rcl} d & ::= & C^n(d) \mid U(d_1^\dagger, d_2^\dagger) \mid U \mid HU \\ d^\dagger & ::= & A \mid n * d \\ n & ::= & 1 \mid \omega \\ m & ::= & 0 \mid 1 \mid \omega \end{array}$$

Non-syntactic demand equalities

$$\begin{array}{rcl} C^\omega(U) & \equiv & U \\ U(\omega * U, \omega * U) & \equiv & U \\ U(A, A) & \equiv & HU \end{array}$$

Usage types

$$\tau ::= \bullet \mid d^\dagger \to \tau$$

Usage type expansion

$$\begin{array}{rcl} d^\dagger \to \tau & \preceq & d^\dagger \to \tau \\ \bullet & \preceq & \omega * U \to \bullet \end{array}$$

Free-variable usage environments (fv-usage)

$$\varphi ::= (x{:}d^\dagger), \varphi \mid \epsilon$$

Auxiliary notation on environments

$$\varphi(x) = \begin{cases} d^\dagger \text{ when } (x{:}d^\dagger) \in \varphi \\ A \text{ otherwise} \end{cases}$$

Usage signatures and signature environments

$$\begin{array}{rcl} \rho & ::= & \langle k\,;\tau\,;\varphi\rangle \quad k \in \mathbb{Z}_{>0} \\ P & ::= & (x{:}\rho), P \mid \epsilon \end{array}$$

$$\begin{array}{l} \mathit{transform}(\langle k\,;\tau\,;\varphi\rangle, d) \\ \quad = \langle \tau\,;\varphi\rangle \quad \text{if } d \sqsubseteq C^1(\ldots k\text{-fold} \ldots C^1(U)) \\ \quad = \langle \omega * \tau\,;\omega * \varphi\rangle \quad \text{otherwise} \end{array}$$

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 `seq` places on its first argument: $\texttt{seq} :: HU \to U \to \bullet$.

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

```
let x = (e1, e2) in fst x + fst x
```

then `e1` is evaluated twice.

Both U and HU come with some non-syntactic 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 $C^0(A)$, if we had such a thing. In the rest of the paper all definitions

$$\boxed{\mu(d^\dagger) = m}$$

$$\mu(A) = 0 \qquad \mu(n{*}d) = n$$

$$\boxed{d_1^\dagger \,\&\, d_2^\dagger = d_3^\dagger \qquad d_1^\dagger \sqcup d_2^\dagger = d_3^\dagger}$$

$$\begin{array}{rcl}
A \,\&\, d^\dagger & = & d^\dagger \\
d^\dagger \,\&\, A & = & d^\dagger \\
n_1{*}d_1 \,\&\, n_2{*}d_2 & = & \omega{*}(d_1 \,\&\, d_2) \\
\\
A \sqcup d^\dagger & = & d^\dagger \\
d^\dagger \sqcup A & = & d^\dagger \\
n_1{*}d_1 \sqcup n_2{*}d_2 & = & (n_1 \sqcup n_2){*}(d_1 \sqcup d_2)
\end{array}$$

$$\boxed{d_1 \,\&\, d_2 = d_3 \qquad d_1 \sqcup d_2 = d_3}$$

$$\begin{array}{rcl}
d \,\&\, U & = & U \\
U \,\&\, d & = & U \\
d \,\&\, HU & = & d \\
HU \,\&\, d & = & d \\
C^{n_1}(d_1) \,\&\, C^{n_2}(d_2) & = & C^{\omega}(d_1 \sqcup d_2) \\
U(d_1^\dagger, d_2^\dagger) \,\&\, U(d_3^\dagger, d_4^\dagger) & = & U(d_1^\dagger \,\&\, d_3^\dagger, d_2^\dagger \,\&\, d_4^\dagger) \\
\\
d \sqcup U & = & U \\
U \sqcup d & = & U \\
d \sqcup HU & = & d \\
HU \sqcup d & = & d \\
C^{n_1}(d_1) \sqcup C^{n_2}(d_2) & = & C^{n_1 \sqcup n_2}(d_1 \sqcup d_2) \\
U(d_1^\dagger, d_2^\dagger) \sqcup U(d_3^\dagger, d_4^\dagger) & = & U(d_1^\dagger \sqcup d_3^\dagger, d_2^\dagger \sqcup d_4^\dagger)
\end{array}$$

$$\boxed{\varphi_1 \,\&\, \varphi_2 = \varphi_3 \qquad \varphi_1 \sqcup \varphi_2 = \varphi_3}$$

$$\begin{array}{rcl}
\varphi_1 \,\&\, \varphi_2 & = & \{(x{:}d_1^\dagger \,\&\, d_2^\dagger) \mid \varphi_i(x) = d_i^\dagger\} \\
\varphi_1 \sqcup \varphi_2 & = & \{(x{:}d_1^\dagger \sqcup d_2^\dagger) \mid \varphi_i(x) = d_i^\dagger\}
\end{array}$$

$$\boxed{\tau_1 \sqcup \tau_2 = \tau_3}$$

$$\begin{array}{rcl}
(d_1^\dagger \to \tau_1) \sqcup (d_1^\dagger \to \tau_2) & = & (d_1^\dagger \sqcup d_2^\dagger) \to (\tau_1 \sqcup \tau_2) \\
\tau \sqcup \bullet & = & \bullet
\end{array}$$

$$\boxed{\langle \tau_1 \,;\, \varphi_1 \rangle \sqcup \langle \tau_2 \,;\, \varphi_2 \rangle = \langle \tau_3 \,;\, \varphi_3 \rangle}$$

$$\langle \tau_1 \,;\, \varphi_1 \rangle \sqcup \langle \tau_2 \,;\, \varphi_2 \rangle \quad = \quad \langle \tau_1 \sqcup \tau_2 \,;\, \varphi_1 \sqcup \varphi_2 \rangle$$

$$\boxed{n{*}d_1^\dagger = d_2^\dagger \qquad n{*}\tau_1 = \tau_2 \qquad n{*}\varphi_1 = \varphi_2}$$

$$\begin{array}{rcl}
1{*}d^\dagger & = & d^\dagger \\
\omega{*}d^\dagger & = & d^\dagger \,\&\, d^\dagger \\
n{*}\bullet & = & \bullet \\
n{*}(d^\dagger \to \tau) & = & (n{*}d^\dagger) \to (n{*}\tau) \\
n{*}\varphi & = & \{x : n{*}\varphi(x) \mid x \in dom(\varphi)\}
\end{array}$$

$$\boxed{n_1 \sqcup n_2 = n_3}$$

$$1 \sqcup 1 = 1 \quad \omega \sqcup n = \omega \quad n \sqcup \omega = \omega$$

$$\boxed{a \sqsubseteq b}$$

$$a \sqsubseteq b \quad \Leftrightarrow \quad (a \sqcup b) = b$$

Figure 2: Demands and demand operations

$$\boxed{P \vdash\!\blacktriangleright e \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle \rightsquigarrow \mathtt{e}}$$

$$\frac{(x : \rho) \in P \quad \langle \tau \,;\, \varphi \rangle = \mathit{transform}(\rho, d)}{P \vdash\!\blacktriangleright x \downarrow d \Rightarrow \langle \tau \,;\, \varphi \,\&\, (x{:}1{*}d) \rangle \rightsquigarrow x} \textsc{VarDn}$$

$$\frac{x \notin dom(P)}{P \vdash\!\blacktriangleright x \downarrow d \Rightarrow \langle \bullet \,;\, (x{:}1{*}d) \rangle \rightsquigarrow x} \textsc{VarUp}$$

$$\frac{P \vdash\!\blacktriangleright e \downarrow d_e \Rightarrow \langle \tau \,;\, \varphi \rangle \rightsquigarrow \mathtt{e}}{P \vdash\!\blacktriangleright \lambda x\,.\,e \downarrow C^n(d_e) \Rightarrow \langle \varphi(x) \to \tau \,;\, n{*}(\varphi \backslash_x) \rangle \rightsquigarrow \lambda^n x\,.\,\mathtt{e}} \textsc{Lam}$$

$$\frac{P \vdash\!\blacktriangleright \lambda x\,.\,e \downarrow C^\omega(U) \Rightarrow \langle \tau \,;\, \varphi \rangle \rightsquigarrow \mathtt{e}'}{P \vdash\!\blacktriangleright \lambda x\,.\,e \downarrow U \Rightarrow \langle \tau \,;\, \varphi \rangle \rightsquigarrow \mathtt{e}'} \textsc{LamU}$$

$$\frac{}{P \vdash\!\blacktriangleright \lambda x\,.\,e \downarrow HU \Rightarrow \langle \tau \,;\, \epsilon \rangle \rightsquigarrow \lambda^1 x\,.\,\mathtt{e}} \textsc{LamHU}$$

$$\frac{P \vdash\!\blacktriangleright e_1 \downarrow C^1(d) \Rightarrow \langle d_2^\dagger \to \tau_r \,;\, \varphi_1 \rangle \rightsquigarrow \mathtt{e}_1 \quad P \vdash\!\blacktriangleright^* y \downarrow d_2^\dagger \Rightarrow \varphi_2}{P \vdash\!\blacktriangleright e_1\ y \downarrow d \Rightarrow \langle \tau_r \,;\, \varphi_1 \,\&\, \varphi_2 \rangle \rightsquigarrow \mathtt{e}_1\ y} \textsc{AppA}$$

$$\frac{P \vdash\!\blacktriangleright e_1 \downarrow C^1(d) \Rightarrow \langle \bullet \,;\, \varphi_1 \rangle \rightsquigarrow \mathtt{e}_1 \quad P \vdash\!\blacktriangleright^* y \downarrow \omega{*}U \Rightarrow \varphi_2}{P \vdash\!\blacktriangleright e_1\ y \downarrow d \Rightarrow \langle \bullet \,;\, \varphi_1 \,\&\, \varphi_2 \rangle \rightsquigarrow \mathtt{e}_1\ y} \textsc{AppB}$$

$$\frac{P \vdash\!\blacktriangleright^* x_i \downarrow d_i^\dagger \Rightarrow \varphi_i \quad i \in 1,2}{P \vdash\!\blacktriangleright (x_1, x_2) \downarrow U(d_1^\dagger, d_2^\dagger) \Rightarrow \langle \bullet \,;\, \varphi_1 \,\&\, \varphi_2 \rangle \rightsquigarrow (x_1, x_2)} \textsc{Pair}$$

$$\frac{P \vdash\!\blacktriangleright (x_1, x_2) \downarrow U(\omega{*}U, \omega{*}U) \Rightarrow \langle \bullet \,;\, \varphi \rangle \rightsquigarrow \mathtt{e}}{P \vdash\!\blacktriangleright (x_1, x_2) \downarrow U \Rightarrow \langle \bullet \,;\, \varphi \rangle \rightsquigarrow \mathtt{e}} \textsc{PairU}$$

$$\frac{}{P \vdash\!\blacktriangleright (x_1, x_2) \downarrow HU \Rightarrow \langle \bullet \,;\, \epsilon \rangle \rightsquigarrow (x_1, x_2)} \textsc{PairHU}$$

$$\frac{\begin{array}{c} P \vdash\!\blacktriangleright e_r \downarrow d \Rightarrow \langle \tau \,;\, \varphi_r \rangle \rightsquigarrow \mathtt{e}_r \\ P \vdash\!\blacktriangleright e_s \downarrow U(\varphi_r(x), \varphi_r(y)) \Rightarrow \langle \_ \,;\, \varphi_s \rangle \rightsquigarrow \mathtt{e}_s \end{array}}{\begin{array}{l} P \vdash\!\blacktriangleright \mathtt{case}\ e_s\ \mathtt{of}\ (x,y) \to e_r \downarrow d \Rightarrow \langle \tau \,;\, \varphi_r \backslash_{x,y} \,\&\, \varphi_s \rangle \\ \qquad \rightsquigarrow \mathtt{case}\ \mathtt{e}_s\ \mathtt{of}\ (x,y) \to \mathtt{e}_r \end{array}} \textsc{Case}$$

$$\boxed{P \vdash\!\blacktriangleright^* x \downarrow d^\dagger \Rightarrow \varphi}$$

$$\frac{}{P \vdash\!\blacktriangleright^* x \downarrow A \Rightarrow \epsilon} \textsc{Abs} \qquad \frac{P \vdash\!\blacktriangleright x \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle \rightsquigarrow x}{P \vdash\!\blacktriangleright^* x \downarrow n{*}d \Rightarrow n{*}\varphi} \textsc{Multi}$$

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

$$P \vdash\!\blacktriangleright e \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle \rightsquigarrow \mathtt{e}'$$

which should be read thus: *in signature environment P, and under usage demand d, the term e places demands $\langle \tau \,;\, \varphi \rangle$ on its components, and elaborates to an annotated term $\mathtt{e}'$*. The syntax of each of these components is given in Figure 1, and their roles in the judgement are the following:

- The *signature environment* P maps some of free variables of e to their *usage signatures*, ρ (Section 3.5). Any free variables outside the domain of P have an uninformative signature.
- The *usage demand*, d, describes the degree to which e is evaluated, including how many times its sub-components are evaluated or called.
- Using P, the judgement transforms the incoming demand d into the demands $\langle \tau \; ; \; \varphi \rangle$ that e places on its *arguments* and *free variables* respectively:
  - The usage that e places on its argument is given by τ, which gives a demand $d^\dagger$ for each argument.
  - The usage that e places on its free variables is given by its free-variable usage (fv-usage), φ, which is simply a finite mapping from variables to usage demands.
- We will discuss the elaborated expressions $\mathtt{e}'$ in Section 3.7.

For example, consider the expression

$$e = \lambda x \; . \; \mathtt{case} \; x \; \mathtt{of} \; (p, q) \to (p, \, f \; \mathtt{True})$$

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?

$$\epsilon \mapsto e \downarrow C^1(U) \Rightarrow \langle 1 * U(\omega * U, A) \to \bullet \; ; \; \{f \mapsto 1 * C^1(U)\} \rangle$$

That is, e will use its argument once, its argument's first component perhaps many times, but will ignore its arguments second component (the A in the usage type). Moreover e will call f just once.

In short, we think of the analysis as describing a *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 PAIR explains how to analyse a pair under a demand $U(d_1^\dagger, d_2^\dagger)$. We simply analyse the two components, under $d_1^\dagger$ or $d_2^\dagger$ respectively, and combine the results with "&". The auxiliary judgement $\mapsto^*$ (Figure 3) deals with the multiplicity of the argument demands $d_i^\dagger$.

The "&" operator, pronounced "both", is defined in Figure 2, and combines the free-variable usages φ_1 and φ_2. For the most part the definition is straightforward, but there is a very important wrinkle for call demands:

$$C^{n_1}(d_1) \,\&\, C^{n_2}(d_2) = C^\omega(d_1 \sqcup d_2)$$

The "ω" part is easy, since n_1 and n_2 are both at least 1. But note the switch from & to the least upper bound $\sqcup$! To see why, consider what demand this expression places on `f`:

```
f 1 2 + f 3 4
```

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

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

3.4 Lambda and application

Rule LAM for lambdas expects the incoming demand to be a call demand $C^n(d_e)$. Then it analyses the body e with demand d_e 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 $n * \varphi$ operation on the bottom line accounts for this multiplicity, and is defined in Figure 2. Rule LAMU handles an incoming demand of U by treating it just like $C^\omega(U)$, while 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 τ can be any type, since the λ-abstraction is not going to be applied, but is only head-used. Dually, given an application $(e \;\; y)$, rule APPA analyses e with demand $C^1(d)$, reflecting that e is here called once. This returns the demand $\langle d_2^\dagger \to \tau_2 \; ; \; \varphi_1 \rangle$ on the context. Then we can analyse the argument under demand $d_2^\dagger$, using $\mapsto^*$, yielding φ_2; and combine φ_1 and φ_2. Rule APPB applies when analysing e_1 yields the less-informative usage type $\bullet$.

3.5 Usage signatures

Suppose we have the term

```
let f = \x.\y. x True  in  f p q
```

We would like to determine the correct demands on `p` and `q`, namely $1 * C^1(U)$ 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 *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 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 $\langle k \; ; \; \varphi(\overline{y}) \to \tau_1 \; ; \; \varphi_1 \backslash_{\overline{y}} \rangle$ as `f`'s usage signature in the environment P when analysing the body of the `let`.

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

What does $\mathit{transform}(\langle k \; ; \; \tau \; ; \; \varphi \rangle, d)$ do (Figure 1)? If the demand d on `f` is stronger than $C^1(\ldots C^1(U))$, 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 $\langle \omega * \tau \; ; \; \omega * \varphi \rangle$, multiplying both the demand type τ and the usage environment φ (Figure 2). Rule LETDNABS handles the case when the variable is not used in the body.

3.6 Thunks

The 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

```
let x = y + 1  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

```
let x = \v. y + v  in  x 42 + x 239
```

Here `y` really *is* demanded twice, and LETDN does that. Another reason that LETDN would be sub-optimal for thunks is shown here:

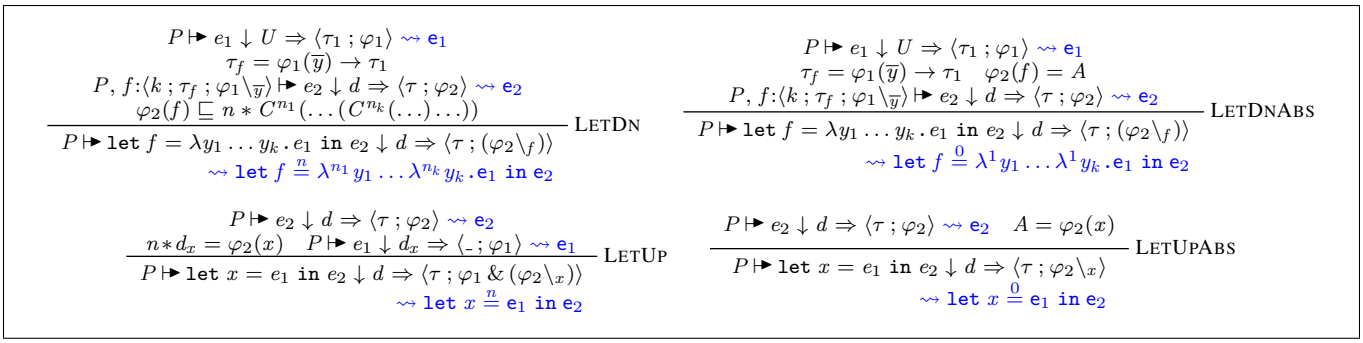

Figure 4: Algorithmic cardinality analysis specification, part 2 (`let`-rules).

```
let x = (p,q)  in  case x of (a,b) -> a
```

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 LETUP rule, used if LETDN does not apply (*i.e.*, the right hand side is not a lambda). Rule 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 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):

- `let`-bindings carry an annotation $m \in 0, 1, \omega$, to indicate how often the `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 language of Figure 1 is stripped to its bare essentials. Our implementation handles all of Haskell, or rather the Core language to which Haskell is translated by GHC. In particular:

- Usage signatures for constants κ 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 HU not $U(d_1^\dagger, d_2^\dagger)$. Furthermore, `case` expressions with multiple branches give rise to a least upper bound $\sqcup$ combination of usage types, as usual.
- Recursive functions and `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 $\hookrightarrow$, for the annotated language that counts how many times thunks have been evaluated and λ-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 `e` by the analysis, then `e` in the instrumented semantics behaves identically to e 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 $\longrightarrow$.
- We prove soundness by giving a type system for the *annotated terms*, and showing that for well-typed terms, the instrumented semantics $\hookrightarrow$ simulates $\longrightarrow$, 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 $[x \overset{m}{\mapsto} \mathtt{Exp(e)}]$ and one for already evaluated expressions $[x \overset{m}{\mapsto} \mathtt{Val(v)}]$. 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* $(\bullet\ y)$, which store a reference y to an argument, *case-frames* $((x, y) \to \mathtt{e})$, which account for the execution of a `case`-branch, and *update frames* of the form $\#(x, m)$, 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 ELET allocates a new binding on the heap. The rules EBETA fires only if the cardinality annotation is non-zero; it de-references an `Exp(e)` binding and emits an update frame. Rules EBETA, EAPP, EPAIR and EPRED are standard. Notice that EBETA also fires only if the λ'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 ELKPV de-references a binding for an already-evaluated expression $[x \overset{m}{\mapsto} \mathtt{Val(v)}]$, and in a standard semantics would return `v` leaving the heap unaffected. In our counting semantics however,

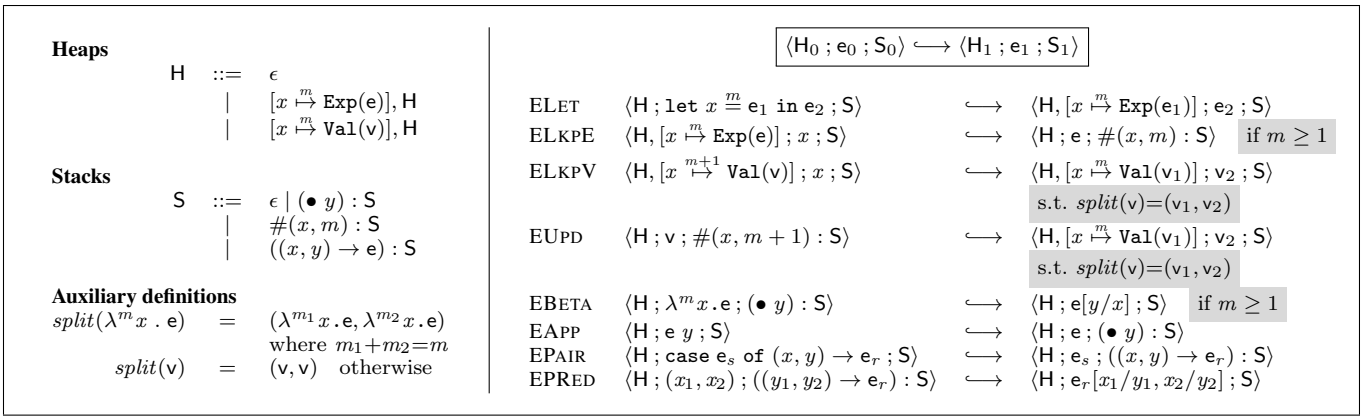

Figure 5: A non-deterministic counting operational semantics. The guards for counting restrictions are highlighted by grey boxes.

we need to account for two things. First, we decrease the multiplicity annotation on the binding (from $m+1$ to m in rule ELKPV). Moreover, the value v can in the future be used both directly (since it is now the active expression), *and* indirectly through a future de-reference of x. We express this by *non-deterministically splitting* the value v, returning two values v_1 and v_2 whose top-level λ-annotations sum up to the original (see *split* in Figure 5). Our proof needs only ensure that among the non-deterministic choices there *exists* a choice that simulates $\longrightarrow$. Rule EUPD is similar except that the heap gets updated by an update frame.

4.2 Checking well-annotated terms

We would like to prove that if we analyse a term e, producing an annotated term e, then if e executes for a number of steps in the standard semantics $\longrightarrow$, then execution of e does not get stuck in the instrumented semantics $\hookrightarrow$ 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 6 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 Figures 3-4, except that we check an annotated term, rather than producing one. For example, rule TLAM checks that the annotation on a λ-abstraction (m) is at least as large as the call cardinality we press on this λ-abstraction (n). As evaluation progresses the situation clarifies, so the annotations may become more conservative than the checker requires, but that is fine.

A more substantial difference is that instead of holding concrete demand transformers ρ as the analysis does (Figure 1), the environment P holds *generalised demand transformers* ϱ. A generalised demand transformer is simply a monotone function from a demand to a pair $\langle \tau \,;\, \varphi \rangle$ of a type and a usage environment (Figure 6). In the TLETDN rule, we clairvoyantly choose any such transformer ϱ, which is sound for the RHS expression – denoted with $P \Vdash \mathsf{e}_1 : \varrho$. 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` ($\varphi_2(x)$). In rule TVARDN we simply apply the transformer ϱ to get a type and fv-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 λ-expressions bound with LETDN.

Because these rules conjure up functions ϱ 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* $P \vdash\!\blacktriangleright e \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle \rightsquigarrow \mathsf{e}$ *then* $P \vdash \mathsf{e} \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle$.

We would next like to show that well-typed terms do not get stuck. To present the main result we need some notation first.

Definition 4.1 (Unannotated heaps and stacks and erasure)**.** We use H and S to refer to an un-instrumented heap and stack respectively. We use $\mathsf{e}^\natural = e$ to mean that the erasure of all annotations from e is e, and we define $\mathsf{S}^\natural = S$ and $\mathsf{H}^\natural = H$ analogously.

We can show that annotated terms run for at least as many steps as their erasures would run in the un-instrumented semantics:

Theorem 4.2 (Safety for annotated terms)**.** *If* $\epsilon \vdash \mathsf{e}_1 \downarrow HU \Rightarrow \langle \tau \,;\, \epsilon \rangle$ *and* $e_1 = \mathsf{e}_1^\natural$ *and* $\langle \epsilon \,;\, e_1 \,;\, \epsilon \rangle \longrightarrow^k \langle H \,;\, e_2 \,;\, S \rangle$ *then there exist* H, e_2 *and* S, *such that* $\langle \epsilon \,;\, \mathsf{e}_1 \,;\, \epsilon \rangle \hookrightarrow^k \langle \mathsf{H} \,;\, \mathsf{e}_2 \,;\, \mathsf{S} \rangle$, $\mathsf{H}^\natural = H$, $\mathsf{S}^\natural = S$ *and* $\mathsf{e}_2^\natural = e_2$.

Unsurprisingly, to prove this theorem we need to generalise the statement to talk about a single-step reduction of a configuration with arbitrary (but well-annotated) heap and stack. Hence we introduce *a well-annotated configuration relation*, denoted $\vdash \langle \mathsf{H} \,;\, \mathsf{e} \,;\, \mathsf{S} \rangle$, that extends the well-annotation invariant of Figure 6 to configurations. For reasons of space, we only give the statement of the theorem below, and defer the details of the well-annotation relation to the extended version of the paper (Sergey et al. 2013).

Lemma 4.3 (Single-step safety)**.** *Assume that* $\vdash \langle \mathsf{H}_1 \,;\, \mathsf{e}_1 \,;\, \mathsf{S}_1 \rangle$. *If* $\langle \mathsf{H}_1^\natural \,;\, \mathsf{e}_1^\natural \,;\, \mathsf{S}_1^\natural \rangle \longrightarrow \langle H_2 \,;\, e_2 \,;\, S_2 \rangle$ *in the un-instrumented semantics, then there exist* H_2, e_2 *and* S_2, *such that* $\langle \mathsf{H}_1 \,;\, \mathsf{e}_1 \,;\, \mathsf{S}_1 \rangle \hookrightarrow \langle \mathsf{H}_2 \,;\, \mathsf{e}_2 \,;\, \mathsf{S}_2 \rangle$, $\mathsf{H}_2^\natural = H_2$, $\mathsf{e}_2^\natural = e$ *and* $\mathsf{S}_2^\natural = S_2$, *and moreover* $\vdash \langle \mathsf{H}_2 \,;\, \mathsf{e}_2 \,;\, \mathsf{S}_2 \rangle$.

Notice that the counting semantics is non-deterministic, so Lemma 4.3 simply ensures that there *exists* a possible transition in the counting semantics that always results in a well-typed configuration. Lemma 4.3 crucially relies on yet another property, below.

$$P ::= \epsilon \mid P, (x{:}\varrho) \qquad \varrho \in d \mapsto \langle \tau \,;\, \varphi \rangle$$

$$\boxed{P \vdash \mathtt{e} \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle}$$

$$\frac{(x : \varrho) \in P \quad \langle \tau \,;\, \varphi \rangle = \varrho(d)}{P \vdash x \downarrow d \Rightarrow \langle \tau \,;\, \varphi \,\&\, (x{:}1{*}d) \rangle}\ \textsc{TVarDn}$$

$$\frac{x \notin dom(P)}{P \vdash x \downarrow d \Rightarrow \langle \bullet \,;\, (x{:}1{*}d) \rangle}\ \textsc{TVarUp}$$

$$\frac{d \sqsubseteq C^n(d_e) \quad m \geq n \quad P \vdash \mathtt{e} \downarrow d_e \Rightarrow \langle \tau \,;\, \varphi \rangle}{P \vdash \lambda^m x\,.\,\mathtt{e} \downarrow d \Rightarrow \langle \varphi(x) \to \tau \,;\, n{*}(\varphi \backslash_x) \rangle}\ \textsc{TLam}$$

$$\frac{d \sqsubseteq HU}{P \vdash \lambda^m x\,.\,\mathtt{e} \downarrow d \Rightarrow \langle \tau \,;\, \epsilon \rangle}\ \textsc{TLamHU}$$

$$\frac{P \vdash \mathtt{e}_1 \downarrow C^1(d) \Rightarrow \langle \tau_1 \,;\, \varphi_1 \rangle \quad \tau_1 \preceq d_2^\dagger \to \tau_r \quad P \vdash^* y \downarrow d_2^\dagger \Rightarrow \varphi_2}{P \vdash \mathtt{e}_1\ y \downarrow d \Rightarrow \langle \tau_r \,;\, \varphi_1 \,\&\, \varphi_2 \rangle}\ \textsc{TApp}$$

$$\frac{d \sqsubseteq U(d_1^\dagger, d_2^\dagger) \quad P \vdash^* x_1 \downarrow d_1^\dagger \Rightarrow \varphi_1 \quad P \vdash^* x_2 \downarrow d_2^\dagger \Rightarrow \varphi_2}{P \vdash (x_1, x_2) \downarrow d \Rightarrow \langle \bullet \,;\, \varphi_1 \,\&\, \varphi_2 \rangle}\ \textsc{TPair}$$

$$\frac{P \vdash \mathtt{e}_r \downarrow d \Rightarrow \langle \tau \,;\, \varphi_r \rangle \quad P \vdash \mathtt{e}_s \downarrow U(\varphi_r(x), \varphi_r(y)) \Rightarrow \langle \_ \,;\, \varphi_s \rangle}{P \vdash \mathtt{case}\ \mathtt{e}_s\ \mathtt{of}\ (x, y) \to \mathtt{e}_r \downarrow d \Rightarrow \langle \tau \,;\, \varphi_r \backslash_{x,y} \,\&\, \varphi_s \rangle}\ \textsc{TCase}$$

$$\frac{m \geq \mu(\varphi_2(x)) \quad \varphi_2(x) \sqsubseteq n * d_1 \quad P \vdash \mathtt{e}_1 \downarrow d_1 \Rightarrow \langle \tau_1 \,;\, \varphi_1 \rangle \quad P \vdash^\natural \mathtt{e}_1 : \varrho \quad P, (x{:}\varrho) \vdash \mathtt{e}_2 \downarrow d \Rightarrow \langle \tau \,;\, \varphi_2 \rangle}{P \vdash \mathtt{let}\ x \overset{m}{=} \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \downarrow d \Rightarrow \langle \tau \,;\, (\varphi_2 \backslash_x) \rangle}\ \textsc{TLetDn}$$

$$\frac{m \geq n \quad P \vdash \mathtt{e}_2 \downarrow d \Rightarrow \langle \tau \,;\, \varphi_2 \rangle \quad n{*}d_x = \varphi_2(x) \quad P \vdash \mathtt{e}_1 \downarrow d_x \Rightarrow \langle \_ \,;\, \varphi_1 \rangle}{P \vdash \mathtt{let}\ x \overset{m}{=} \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \downarrow d \Rightarrow \langle \tau \,;\, \varphi_1 \,\&\, (\varphi_2 \backslash_x) \rangle}\ \textsc{TLetUp}$$

$$\frac{P \vdash \mathtt{e}_2 \downarrow d \Rightarrow \langle \tau \,;\, \varphi_2 \rangle \quad A = \varphi_2(x)}{P \vdash \mathtt{let}\ x \overset{m}{=} \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \downarrow d \Rightarrow \langle \tau \,;\, \varphi_1 \,\&\, (\varphi_2 \backslash_x) \rangle}\ \textsc{TLetUpAbs}$$

$$\boxed{P \vdash^* x \downarrow d^\dagger \Rightarrow \varphi}$$

$$\frac{}{P \vdash^* x \downarrow A \Rightarrow \epsilon}\ \textsc{TAbs} \qquad \frac{P \vdash x \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle}{P \vdash^* x \downarrow n{*}d \Rightarrow n{*}\varphi}\ \textsc{TMulti}$$

$$\boxed{P \vdash^\natural \mathtt{e} : \varrho}$$

$$\frac{\forall d_1, d_2\,.\, d_1 \sqsubseteq d_2 \Longrightarrow \varrho(d_1) \sqsubseteq \varrho(d_2) \quad \forall d, \varphi, \tau\,.\,(P \vdash \mathtt{e} \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle) \Longrightarrow \langle \tau \,;\, \varphi \rangle \sqsubseteq \varrho(d)}{P \vdash^\natural \mathtt{e} : \varrho}\ \textsc{WfTrans}$$

Figure 6: Well-annotated terms

Lemma 4.4 (Value demand splitting)**.** *If* $P \vdash \mathtt{v} \downarrow (d_1 \,\&\, d_2) \Rightarrow \langle \tau \,;\, \varphi \rangle$ *then there exists a split* $split(\mathtt{v}) = (\mathtt{v}_1, \mathtt{v}_2)$ *such that:* $P \vdash \mathtt{v}_1 \downarrow d_1 \Rightarrow \langle \tau_1 \,;\, \varphi_1 \rangle$ *and* $P \vdash \mathtt{v}_2 \downarrow d_2 \Rightarrow \langle \tau_2 \,;\, \varphi_2 \rangle$ *and moreover* $\tau_1 \sqsubseteq \tau$, $\tau_2 \sqsubseteq \tau$ *and* $\varphi_1 \,\&\, \varphi_2 \sqsubseteq \varphi$.

Why is Lemma 4.4 important? Consider the following

```
let x = v  in  case x 3 of (y,z) -> x 4
```

$$\frac{}{\epsilon \propto \epsilon}\ \textsc{HSim1} \qquad \frac{\mathsf{H}_1 \propto \mathsf{H}_2}{\mathsf{H}_1, [x \overset{0}{\mapsto} \mathtt{Exp(e)}] \propto \mathsf{H}_2}\ \textsc{HSim2}$$

$$\frac{n \geq 1 \quad \mathsf{H}_1 \propto \mathsf{H}_2 \quad \mathtt{e}_1 \propto \mathtt{e}_2}{\mathsf{H}_1, [x \overset{n}{\mapsto} \mathtt{Exp}(\mathtt{e}_1)] \propto \mathsf{H}_2, [x \overset{n}{\mapsto} \mathtt{Exp}(\mathtt{e}_2)]}\ \textsc{HSim3}$$

$$\frac{\mathsf{H}_1 \propto \mathsf{H}_2}{\mathsf{H}_1, [x \overset{0}{\mapsto} \mathtt{Val(v)}] \propto \mathsf{H}_2}\ \textsc{HSim4}$$

$$\frac{\mathsf{H}_1 \propto \mathsf{H}_2 \quad \mathtt{v}_1 \propto \mathtt{v}_2}{\mathsf{H}_1, [x \overset{\omega}{\mapsto} \mathtt{Val}(\mathtt{v}_1)] \propto \mathsf{H}_2, [x \overset{\omega}{\mapsto} \mathtt{Val}(\mathtt{v}_2)]}\ \textsc{HSim5}$$

$$\frac{}{\epsilon \propto \epsilon}\ \textsc{SSim1} \qquad \frac{\mathsf{S}_1 \propto \mathsf{S}_2}{(\#(x, 1) : \mathsf{S}_1) \propto \mathsf{S}_2}\ \textsc{SSim2}$$

$$\frac{\mathsf{S}_1 \propto \mathsf{S}_2}{(\#(x, \omega) : \mathsf{S}_1) \propto (\#(x, \omega) : \mathsf{S}_2)}\ \textsc{SSim3}$$

$$\frac{\mathsf{S}_1 \propto \mathsf{S}_2}{(\bullet\ y) : \mathsf{S}_1 \propto (\bullet\ y) : \mathsf{S}_2}\ \textsc{SSim4}$$

$$\frac{\mathtt{e}_1 \propto \mathtt{e}_2 \quad \mathsf{S}_1 \propto \mathsf{S}_2}{((x, y) \to \mathtt{e}_1) : \mathsf{S}_1 \propto ((x, y) \to \mathtt{e}_2) : \mathsf{S}_2}\ \textsc{SSim5}$$

Figure 7: Auxiliary simulation relation (heaps and stacks)

The demand exercised on `x` from the body of the `let`-binding will be $C^1(U) \,\&\, C^1(U) = C^\omega(U)$ and hence the value `v` will be checked against this demand (using the LetUp rule), unleashing an environment φ. 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 φ_1 and φ_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 WfTrans.

Theorem 4.5 (Safety of analysis)**.** *If* $\epsilon \vdash\!\blacktriangleright e_1 \downarrow HU \Rightarrow \langle \tau \,;\, \epsilon \rangle \rightsquigarrow \mathsf{e}_1$ *and* $\langle \epsilon \,;\, e_1 \,;\, \epsilon \rangle \longrightarrow^k \langle H \,;\, e_2 \,;\, S \rangle$*, then there exist* H, e_2 *and* S*, such that* $\langle \epsilon \,;\, \mathsf{e}_1 \,;\, \epsilon \rangle \hookrightarrow^k \langle \mathsf{H} \,;\, \mathsf{e}_2 \,;\, \mathsf{S} \rangle$*,* $\mathsf{H}^\natural = H$*,* $\mathsf{S}^\natural = S$ *and* $\mathsf{e}_2^\natural = e_2$.

The proof is just a combination of Lemma 4.1 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 8. 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.

$$\boxed{\langle \mathsf{H}_0 \,;\, \mathsf{e}_0 \,;\, \mathsf{S}_0 \rangle \Longrightarrow \langle \mathsf{H}_1 \,;\, \mathsf{e}_1 \,;\, \mathsf{S}_1 \rangle}$$

OPT-ELETA	$\langle \mathsf{H} \,;\, \texttt{let}\ x \overset{0}{=} \mathsf{e}_1\ \texttt{in}\ \mathsf{e}_2 \,;\, \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H} \,;\, \mathsf{e}_2 \,;\, \mathsf{S} \rangle$
OPT-ELETU	$\langle \mathsf{H} \,;\, \texttt{let}\ x \overset{n}{=} \mathsf{e}_1\ \texttt{in}\ \mathsf{e}_2 \,;\, \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H}[x \overset{n}{\mapsto} \texttt{Exp}(\mathsf{e}_1)] \,;\, \mathsf{e}_2 \,;\, \mathsf{S} \rangle$ where $n \geq 1$
OPT-ELKPEM	$\langle \mathsf{H}, [x \overset{\omega}{\mapsto} \texttt{Exp}(\mathsf{e})] \,;\, x \,;\, \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H} \,;\, \mathsf{e} \,;\, \#(x,\omega) : \mathsf{S} \rangle$
OPT-ELKPEO	$\langle \mathsf{H}, [x \overset{1}{\mapsto} \texttt{Exp}(\mathsf{e})] \,;\, x \,;\, \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H} \,;\, \mathsf{e} \,;\, \mathsf{S} \rangle$
OPT-ELKPV	$\langle \mathsf{H}, [x \overset{\omega}{\mapsto} \texttt{Val}(\mathsf{v})] \,;\, x \,;\, \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H}, [x \overset{\omega}{\mapsto} \texttt{Val}(\mathsf{v})] \,;\, \mathsf{v} \,;\, \mathsf{S} \rangle$
OPT-EUPD	$\langle \mathsf{H} \,;\, \mathsf{v} \,;\, \#(x,\omega) : \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H}, [x \overset{\omega}{\mapsto} \texttt{Val}(\mathsf{v})] \,;\, \mathsf{v} \,;\, \mathsf{S} \rangle$
OPT-EBETA	$\langle \mathsf{H} \,;\, \lambda^m x.\mathsf{e} \,;\, (\bullet\ y) : \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H} \,;\, \mathsf{e}[y/x] \,;\, \mathsf{S} \rangle$
OPT-EAPP	$\langle \mathsf{H} \,;\, \mathsf{e}\ y \,;\, \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H} \,;\, \mathsf{e} \,;\, (\bullet\ y) : \mathsf{S} \rangle$
OPT-EPAIR	$\langle \mathsf{H} \,;\, \texttt{case}\ \mathsf{e}_s\ \texttt{of}\ (x,y) \to \mathsf{e}_r \,;\, \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H} \,;\, \mathsf{e}_s \,;\, ((x,y) \to \mathsf{e}_r) : \mathsf{S} \rangle$
OPT-EPRED	$\langle \mathsf{H} \,;\, (x_1,x_2) \,;\, ((y_1,y_2) \to \mathsf{e}_r) : \mathsf{S} \rangle$	$\Longrightarrow$	$\langle \mathsf{H} \,;\, \mathsf{e}_r[x_1/y_1, x_2/y_2] \,;\, \mathsf{S} \rangle$

Figure 8: Optimised counting semantics

Definition 5.1 (Auxiliary $\propto$-relations)**.** We write $\mathsf{e}_1 \propto \mathsf{e}_2$ iff e_1 and e_2 differ only on the λ-annotations. $\mathsf{H}_1 \propto \mathsf{H}_2$ and $\mathsf{S}_1 \propto \mathsf{S}_2$ are defined in Figure 7.

For this optimisation the annotations on λ-abstractions play no role, hence we relate *any* expressions that differ only on those.

Figure 7 tells us when a heap H is related with an *optimised* heap H_{opt} with the relation $\mathsf{H} \propto \mathsf{H}_{opt}$. As we have described, there are no $\overset{0}{\mapsto}$ bindings in the optimised heap. Moreover notice that there are no bindings of the form $[x \overset{1}{\mapsto} \texttt{Val}(v)]$ in either the optimised or unoptimised heap. It is easy to see why: every heap binding starts life as $[x \overset{m}{\mapsto} \texttt{Exp}(\mathsf{e})]$. By the time $\texttt{Exp}(\mathsf{e})$ has become a value $\texttt{Val}(\mathsf{v})$, we have already used x once. Hence, if originally $m = \omega$ then the value binding will also be ω (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 $\mathsf{S} \propto \mathsf{S}_{opt}$. Rule SSIM2 ensures that there are no frames $\#(x,1)$ 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 simulation theorem.

Theorem 5.1 (Optimised semantics)**.** *If* $\langle \mathsf{H}_1 ; \mathsf{e}_1 ; \mathsf{S}_1 \rangle \propto \langle \mathsf{H}_2 ; \mathsf{e}_2 ; \mathsf{S}_2 \rangle$ *and* $\langle \mathsf{H}_1 \,;\, \mathsf{e}_1 \,;\, \mathsf{S}_1 \rangle \longrightarrow \langle \mathsf{H}_1' \,;\, \mathsf{e}_1' \,;\, \mathsf{S}_1' \rangle$ *then there exists* $k \in \{0,1\}$ *s.t.* $\langle \mathsf{H}_2 \,;\, \mathsf{e}_2 \,;\, \mathsf{S}_2 \rangle \Longrightarrow^k \langle \mathsf{H}_2' \,;\, \mathsf{e}_2' \,;\, \mathsf{S}_2' \rangle$ *and* $\langle \mathsf{H}_1' \,;\, \mathsf{e}_1' \,;\, \mathsf{S}_1' \rangle \propto \langle \mathsf{H}_2' \,;\, \mathsf{e}_2' \,;\, \mathsf{S}_2' \rangle$.

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 4.5 to get the following result.

Theorem 5.2 (Analysis is safe for optimised semantics)**.** *If* $\vdash\!\blacktriangleright e_1 \downarrow HU \Rightarrow \langle \tau \,;\, \epsilon \rangle \rightsquigarrow \mathsf{e}_1$ *and* $\langle \epsilon \,;\, e_1 \,;\, \epsilon \rangle \longrightarrow^n \langle H \,;\, e_2 \,;\, S \rangle$ *then* $\langle \epsilon \,;\, \mathsf{e}_1 \,;\, \epsilon \rangle \Longrightarrow^m \langle \mathsf{H} \,;\, \mathsf{e}_2 \,;\, \mathsf{S} \rangle$ *s. t.* $\mathsf{e}_2^{\natural} = e_2$*,* $m \leq n$*, and there exist* H_2 *and* S_2 *s.t.* $\mathsf{H}_2^{\natural} = H$ *and* $\mathsf{S}_2^{\natural} = S$ *and* $\mathsf{H}_2 \propto \mathsf{H}$ *and* $\mathsf{S}_2 \propto \mathsf{S}$.

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 semantics, 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)**.**

$$\begin{array}{rl} & \texttt{let}\ z \overset{m_1}{=} \mathsf{e}_1\ \texttt{in}\ (\texttt{let}\ f \overset{m_2}{=} \lambda^1 x\ .\ \mathsf{e}\ \texttt{in}\ \mathsf{e}_2) \\ \Longrightarrow & \texttt{let}\ f \overset{m_2}{=} \lambda^1 x\ .\ (\texttt{let}\ z \overset{m_1}{=} \mathsf{e}_1\ \texttt{in}\ \mathsf{e})\ \texttt{in}\ \mathsf{e}_2, \end{array}$$

for any m_1, m_2 and $z \notin FV(\mathsf{e}_2)$.

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 $\langle \mathsf{H} \,;\, \mathsf{e} \,;\, \mathsf{S} \rangle$ to be *closed*, any free variables in H, e and S must be contained in a union $dom(\mathsf{H}) \cup dom(\mathsf{S})$, where $dom(\mathsf{H})$ is a set of variables bound by a heap H, and $dom(\mathsf{S})$ is a set of variables marked for update in a stack S. A configuration is *well-formed* if $dom(\mathsf{H})$ and $dom(\mathsf{S})$ are disjoint.

Definition 5.3 (Convergence)**.** For a closed configuration $\langle \mathsf{H} ; \mathsf{e} ; \mathsf{S} \rangle$,

$$\begin{array}{rcl} \langle \mathsf{H} \,;\, \mathsf{e} \,;\, \mathsf{S} \rangle \Downarrow^N & \overset{\text{def}}{=} & \exists \mathsf{H}', \mathsf{v}, N\ .\ \langle \mathsf{H} \,;\, \mathsf{e} \,;\, \mathsf{S} \rangle \longrightarrow^N \langle \mathsf{H}' \,;\, \mathsf{v} \,;\, \epsilon \rangle \\ \langle \mathsf{H} \,;\, \mathsf{e} \,;\, \mathsf{S} \rangle \Downarrow^{\leq N} & \overset{\text{def}}{=} & \exists M\ .\ \langle \mathsf{H} \,;\, \mathsf{e} \,;\, \mathsf{S} \rangle \Downarrow^M \text{ and } M \leq N \end{array}$$

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*

$$\langle \mathsf{H} \,;\, \texttt{let}\ z \overset{m}{=} \mathsf{e}_1\ \texttt{in}\ (\texttt{let}\ f \overset{m_1}{=} \lambda^1 x\ .\ \mathsf{e}\ \texttt{in}\ \mathsf{e}_2) \,;\, \mathsf{S} \rangle \Downarrow^N$$

and $z \notin FV(\mathsf{e}_2)$*, then*

$$\langle \mathsf{H} \,;\, \texttt{let}\ f \overset{m_1}{=} \lambda^1 x\ .\ (\texttt{let}\ z \overset{m}{=} \mathsf{e}_1\ \texttt{in}\ \mathsf{e})\ \texttt{in}\ \mathsf{e}_2 \,;\, \mathsf{S} \rangle \Downarrow^{\leq N}.$$

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.

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.[2] 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.[3] 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

$$\texttt{let}\, x = \texttt{costly}\, v\, \texttt{in} \ldots (\lambda y. \ldots x \ldots) \ldots$$

If the λy 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 λy, 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

$$\begin{array}{ll} & \texttt{f} = \lambda v.\, \texttt{let}\, x = \texttt{costly}\, v\, \texttt{in}\, \lambda y. \ldots x \ldots \\ \Longrightarrow & \texttt{f} = \lambda v. \lambda y. \ldots (\texttt{costly}\, v) \ldots \end{array}$$

  The latter produces one function with two arguments, rather than a curried function that returns a heap-allocated lambda (Marlow and Peyton Jones 2006).

6.4 Single-entry thunks

The code that GHC compiles for a thunk begins by 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 (Peyton Jones 1992). 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:

$$\begin{array}{rcll} & & \texttt{let}\, y \stackrel{1}{=} e\, \texttt{in let}\, x = y + 0\, \texttt{in}\, x * x & \\ \text{Identity of } + & \Longrightarrow & \texttt{let}\, y \stackrel{1}{=} e\, \texttt{in let}\, x = y\, \texttt{in}\, x * x & \\ \text{Inline } x & \Longrightarrow & \texttt{let}\, y \stackrel{1}{=} e\, \texttt{in}\, y * y & \text{Wrong!} \end{array}$$

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.

Program	Synt. λ^1	Synt. Thnk^1	RT Thnk^1
`anna`	4.0%	7.2%	2.9%
`bspt`	5.0%	15.4%	1.5%
`cacheprof`	7.6%	11.9%	5.1%
`calendar`	5.7%	0.0%	0.2%
`constraints`	2.0%	3.2%	4.5%
`cryptarithm2`	0.6%	3.0%	74.0%
`gcd`	12.5%	0.0%	0.0%
`gen_regexps`	5.6%	0.0%	0.2%
`hpg`	5.2%	0.0%	4.1%
`integer`	8.3%	0.0%	0.0%
`life`	3.2%	0.0%	1.8%
`mkhprog`	27.4%	20.8%	5.8%
`nucleic2`	3.5%	3.1%	3.2%
`partstof`	5.8%	10.7%	0.1%
`sphere`	7.8%	6.2%	20.0%
... and 72 more programs			
Arithmetic mean	10.3%	12.6%	5.5%

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

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 (Partain 1993). 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 Thunk" 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`.

7.1 Optimising `nofib` programs

In the end, of course, we seek improved runtimes, although the benefits are likely to be modest. One-shot lambdas do not confer

[2] `http://github.com/ghc/ghc`

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

Program	Allocs		Runtime	
	No hack	Hack	No hack	Hack
anna	-2.1%	-0.2%	+0.1%	-0.0%
bspt	-2.2%	-0.0%	-0.0%	+0.0%
cacheprof	-7.9%	-0.6%	-6.1%	-5.0%
calendar	-9.2%	+0.2%	-0.0%	-0.0%
constraints	-0.9%	-0.0%	-1.2%	-0.2%
cryptarithm2	-0.3%	-0.3%	-2.3%	-2.1%
gcd	-15.5%	-0.0%	-0.0%	+0.0%
gen_regexps	-1.0%	-0.1%	-0.0%	-0.0%
hpg	-2.0%	-1.0%	-0.1%	-0.0%
integer	-0.0%	-0.0%	-8.8%	-6.6%
life	-0.8%	-0.0%	-5.9%	-1.8%
mkhprog	-11.9%	+0.1%	-0.0%	-0.0%
nucleic2	-14.1%	-10.9%	+0.0%	+0.0%
partstof	-95.5%	-0.0%	-0.0%	-0.0%
sphere	-1.5%	-1.5%	+0.0%	-0.1%
... and 72 more programs				
Min	-95.5%	-10.9%	-28.2%	-12.1%
Max	+3.5%	+0.5%	+1.8%	+2.8%
Geometric mean	-6.0%	-0.3%	-2.2%	-1.4%

Table 2. Cardinality analysis-enabled optimisations for `nofib`

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 2.1 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.[4]

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%.

[4] 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.

Program	RT Thnk[1]	No-Opt RT	RT Δ
binary-trees	49.4%	66.83 s	-9.2%
fannkuch-redux	0.0%	158.94 s	-3.7%
n-body	5.7%	38.41 s	-4.4%
pidigits	8.8%	41.56 s	-0.3%
spectral-norm	4.6%	17.83 s	-1.7%

Table 3. Optimisation of the programs from Benchmarks Game

Library	λ^1	Thnk[1]	Benchmark	Alloc Δ
attoparsec	32.8%	19.3%	benchmarks	-7.1%
binary	16.8%	0.9%	bench	-0.2%
			builder	-0.3%
			get	-4.3%
bytestring	5.3%	4.3%	boundcheck	-0.5%
			all	-6.6%
cassava	26.4%	9.8%	benchmarks	-0.7%

Table 4. Analysis and optimisation results for `hackage` libraries

Program	LOC	GHC Alloc Δ		GHC RT Δ	
		No hack	Hack	No hack	Hack
anna	5740	-1.6%	-1.5%	-0.8%	-0.4%
cacheprof	1600	-1.7%	-0.4%	-2.3%	-1.8%
fluid	1579	-1.9%	-1.9%	-2.8%	-1.6%
gamteb	1933	-0.5%	-0.1%	-0.5%	-0.1%
parser	2379	-0.7%	-0.2%	-2.6%	-0.6%
veritas	4674	-1.4%	-0.3%	-4.5%	-4.1%

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.[5] 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:[6] `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

[5] `http://benchmarksgame.alioth.debian.org/`

[6] `http://hackage.haskell.org/`

the art, we made our comparison with respect to the contemporary version of (hacked) baseline GHC. The encouraging results for `attoparsec` 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 $C^n(d)$, 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, § 2.2) for details), but the idea of using a *type system* to express usage information inspired a series of "once upon a type" papers[7] (Gustavsson 1998; Turner et al. 1995; 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 $Int^1 \to Int$, but still be applied to an argument x that was used many times and had type Int^ω (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, 3.7). Gustavsson advocated *bounded* polymorphism to gain greater precision (Gustavsson and Sveningsson 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 x 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 LETDN and LETUP 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:

```
let f = \x. y + x  in  if b then f 1 else y
```

How many times is the free variable `y` evaluated in this expression? Obviously just once, and LETDN discovers this, because we unleash the demand on `y` at `f`'s call site, and lub the two branches of the `if`. But type systems behave like LETUP: 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 `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 φ 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; Holdermans 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; one program allocated 15% more, one 14% less (Wansbrough 2002)) 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 *caller* (to pass the only copy of the value), whereas for us a single-entry argument is a promise by *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 *applicativeness*, employed in the recent work on relevance typing (Holdermans 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 S corresponds to a "strong" version of our demands $C^\omega(d)$, which requires $d \sqsubset U$, and L is similar to our U. The `seq`-like evaluation of expressions corresponds to our demand HU. However, neither call- nor thunk-cardinality are captured by the concept of applicativeness.

[7] The title, as so often, is due to Wadler.

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 2006). 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 solid win.

Acknowledgements We are grateful to Johan Tibell for the suggestion to use benchmark-accompanied `hackage` 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

Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

Andy Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, Department of Computer Schence, 1996.

Andy Gill, John Launchbury, and Simon Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the Sixth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, 1993.

Jean-Yves Girard. Linear logic: its syntax and semantics. In *Proceedings of the workshop on Advances in linear logic*, pages 1–42. Cambridge University Press, 1995.

Benjamin Goldberg. Detecting sharing of partial applications in functional programs. In *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 408–425. Springer-Verlag, 1987.

Jörgen Gustavsson. A type based sharing analysis for update avoidance and optimisation. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 39–50. ACM, 1998.

Jörgen Gustavsson and Josef Sveningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Implementation of Functional Languages (IFL 2000), Selected Papers*, volume 2011 of *LNCS*, pages 140–157. Springer, 2001.

Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 235–246. ACM, 2007.

Ralf Hinze. *Projection-based strictness analysis - theoretical and practical aspects*. PhD thesis, Bonn University, 1995.

Stefan Holdermans and Jurriaan Hage. Making "stricternes" more relevant. In *PEPM'10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 121–130. ACM, 2010.

Paul Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 351–363. ACM, 1986.

Richard Jones. Tail recursion without space leaks. *J. Funct. Program.*, 2 (1):73–79, 1992.

John Launchbury, Andy Gill, John Hughes, Simon Marlow, Simon Peyton Jones, and Philip Wadler. Avoiding unnecessary updates. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 144–153. Springer, 1993.

Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16(4-5): 415–449, 2006.

Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 13–25. ACM, 2006.

Andrew Moran and David Sands. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *POPL'99: Proceedings of the 26th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 43–56. ACM, 1999.

Will Partain. The `nofib` benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 195–202. Springer, 1993.

Simon Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *J. Funct. Program.*, 2(2): 127–202, 1992.

Simon Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. In *Proceedings of the 1993 Glasgow Workshop on Functional Programming*, pages 201–220. Springer, 1994.

Simon Peyton Jones and André Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, 1998.

Simon Peyton Jones, Will Partain, and André Santos. Let-floating: Moving Bindings to Give Faster Programs. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 1–12. ACM, 1996.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 288–298. ACM, 1992.

Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. Modular, Higher-Order Cardinality Analysis in Theory and Practice. Extended version. Technical Report MSR-TR-2013-112, Microsoft Research, 2013. Available at `http://research.microsoft.com/apps/pubs/?id=204260`.

Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3): 231–264, 1997.

David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theor. Comput. Sci.*, 227(1-2):231–248, 1999.

David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 1–11. ACM, 1995.

David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, pages 51–62. ACM, 2010.

Philip Wadler and John Hughes. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 385–407. Springer-Verlag, 1987.

Keith Wansbrough. *Simple Polymorphic Usage Analysis*. PhD thesis, Computer Laboratory, University of Cambridge, 2002.

Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. In *POPL'99: Proceedings of the 26th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 15–28. ACM, 1999.