Modular, Higher-Order Cardinality Analysis in Theory and Practice

ILYA SERGEY
University College London, UK
(*e-mail:* i.sergey@ucl.ac.uk)

DIMITRIOS VYTINIOTIS
Microsoft Research, Cambridge, UK
(*e-mail:* dimitris@microsoft.com)

JOACHIM BREITNER
University of Pennsylvania, USA
(*e-mail:* joachim@cis.upenn.edu)

SIMON L. PEYTON JONES
Microsoft Research, Cambridge, UK
(*e-mail:* 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. In this paper we describe a new, modular analysis for a higher-order language, which is both simple and effective. We prove the analysis sound with respect to a standard call-by-need semantics, 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. This paper extends our preceding conference publication (Sergey *et al.* 2014) with proofs, expanded report on evaluation and a detailed examination of the factors causing the loss of precision in the analysis.

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)))
```

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

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 short-cut deforestation 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 define 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 equally important, improving in the sense of Moran & Sands (1999) (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 one-shot lambdas and single-entry 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 also measure how precise the analysis is, by comparing the static results with dynamic measurements using an instrumented runtime (Section 7.3), and explain the typical cases where the analysis as designed cannot be more precise.

Before this work, GHC conservatively assumed that every thunk could be entered more than once, and every lambda called more than once, thus losing useful opportuities for optimisation, as quantified in Section 7. We discuss other 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 due to enabled optimisations.

This is a longer version of a paper "Modular, Higher-Order Cardinality Analysis in Theory and Practice" by Sergey *et al.* (2014), containing proofs, an expanded report on evaluation, and detailed examination of the factors causing the loss of precision in the analysis.

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), a question that is complicated by currying in a higher-order language like Haskell (Section 2.2)?
- 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 fairly 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 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 (which was extant in GHC until recently) relied 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 = zowzy a (\x.let t = costly x in \y. t+y)

zowzy1 a g = g 2 a + g 3 a
zowzy2 a g = sum (map (g a) [1..1000])
```

If `zowzy` was `zowzy1`, then in `f3` it would be best to inline `t` at its use site, thus:

```
f4 a = zowzy1 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 `zowzy2` instead, such a transformation would be disastrous. Why? Because `zowzy2` 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 `zowzy2` must be able to report "`zowzy2`'s argument `g` is called[1] once, and the result is called many times". We formalise this by giving a *usage signature* to `zowzy`, like this:

$$\begin{array}{lcl} \texttt{zowzy1} & :: & U \to C^{\omega}(C^{1}(U)) \to \bullet \\ \texttt{zowzy2} & :: & U \to C^{1}(C^{\omega}(U)) \to \bullet \end{array}$$

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[2]. Informally, the $C^{1}(d)$ means "this argument is called *once*, 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 (which includes not being used at all)". Note that `zowzy1`'s second argument *precise* 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>
```

[1] We will always use "called" to mean "applied to one argument".
[2] The "•" has no significance; we are just used to seeing something after the final arrow!

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 & Partain 1994; Peyton Jones & 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 & 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 & Partain 1994), but it has never been formalised. In the framework of this paper, we give `f` from the last code fragment 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. The top-level U indicates that the overall argument has been used, and could have been omitted, but we keep it for the uniformity of the notation.

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*C^1(U) → •
f3 g = g 3           -- f3  ::  1*C^1(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? It depends on `f`, indeed. 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 Fig. 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 & Felleisen 1992) so that the issues concerning thunks show up only for `let` and not also for function arguments.

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

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

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

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

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)

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

Auxiliary notation on environments

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

Usage signatures and signature environments

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

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

Fig. 1: Syntax of terms, values, usage types, and usage environments.

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

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

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

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

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

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

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

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

Fig. 2: Demands and demand operations.

the analysis and its *precision*. Our particular choices are expressed in the syntax of usage demands, given in Fig. 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 $d_2^\dagger$.
- $C^n(d)$ 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: `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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3: Operations on demand types and usage environments, and generic partial order.

different for pairs; the demand $(d_1^\dagger, d_2^\dagger)$ must have $d^\dagger$ demands as the sub-components. For example, if we have

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

then `e1` is evaluated twice. So the usage demand for `x` is $\omega{*}U(\omega{*}U, A)$

Both U and HU come with some non-syntactic equalities, denoted by $\equiv$ in Fig. 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 and metatheory are modulo-$\equiv$ equivalence (checking that all our definitions respect $\equiv$ is routine and, hence, omitted).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 4: Algorithmic cardinality analysis specification, Part 1.

3.2 Usage analysis

The analysis itself is shown in Fig. 4 and 5. The main judgement form is written thus

$$P \blacktriangleright e \downarrow d \Rightarrow \langle \tau ; \varphi \rangle \rightsquigarrow \mathsf{e}'$$

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 e'.* The syntax of each of these components is given in Fig. 1, and their roles in the judgement are the following:

- The *signature environment* P maps some of the free variables of e to their *usage signatures*, ρ (Section 3.5). Any free variable outside the domain of P has 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 e' in Section 3.7.

For example, consider the expression

$$e = \lambda x \,.\, \texttt{case}\ x\ \texttt{of}\ (p, q) \rightarrow (p, f\ \texttt{True})$$

Suppose we place demand $C^1(U)$ on e, so that e is called, just once. What demand does e then place on its arguments and free variables?

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

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 Fig. 4. 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 $\blacktriangleright^*$ (Fig. 4) deals with the multiplicity of the argument demands $d_i^\dagger$.

The "&" operator, pronounced "both", is defined for demands in Fig. 2, and for demand types and usage environments in Fig. 3. It 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:

$$\frac{\begin{array}{c} P \vdash e_1 \downarrow U \Rightarrow \langle \tau_1 ; \varphi_1 \rangle \rightsquigarrow \mathtt{e}_1 \quad \tau_f = \varphi_1(\overline{y}) \to \tau_1 \\ P, f{:}\langle k ; \tau_f ; \varphi_1 \backslash_{\overline{y}} \rangle \vdash e_2 \downarrow d \Rightarrow \langle \tau ; \varphi_2 \rangle \rightsquigarrow \mathtt{e}_2 \\ \varphi_2(f) \sqsubseteq n * C^{n_1}(\ldots(C^{n_k}(\ldots)\ldots)) \end{array}}{\begin{array}{c} P \vdash \mathtt{let}\ f = \lambda\, y_1 \ldots y_k \,.\, e_1\ \mathtt{in}\ e_2 \downarrow d \Rightarrow \langle \tau ; (\varphi_2 \backslash_f) \rangle \\ \rightsquigarrow \mathtt{let}\ f \overset{n}{=} \lambda^{n_1} y_1 \ldots \lambda^{n_k} y_k \,.\, \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \end{array}} \ \textsc{LetDn}$$

$$\frac{\begin{array}{c} P \vdash e_1 \downarrow U \Rightarrow \langle \tau_1 ; \varphi_1 \rangle \rightsquigarrow \mathtt{e}_1 \quad \tau_f = \varphi_1(\overline{y}) \to \tau_1 \quad \varphi_2(f) = A \\ P, f{:}\langle k ; \tau_f ; \varphi_1 \backslash_{\overline{y}} \rangle \vdash e_2 \downarrow d \Rightarrow \langle \tau ; \varphi_2 \rangle \rightsquigarrow \mathtt{e}_2 \end{array}}{\begin{array}{c} P \vdash \mathtt{let}\ f = \lambda\, y_1 \ldots y_k \,.\, e_1\ \mathtt{in}\ e_2 \downarrow d \Rightarrow \langle \tau ; (\varphi_2 \backslash_f) \rangle \\ \rightsquigarrow \mathtt{let}\ f \overset{0}{=} \lambda^{1} y_1 \ldots \lambda^{1} y_k \,.\, \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \end{array}} \ \textsc{LetDnAbs}$$

$$\frac{\begin{array}{c} P \vdash e_2 \downarrow d \Rightarrow \langle \tau ; \varphi_2 \rangle \rightsquigarrow \mathtt{e}_2 \\ n * d_x = \varphi_2(x) \quad P \vdash e_1 \downarrow d_x \Rightarrow \langle \_ ; \varphi_1 \rangle \rightsquigarrow \mathtt{e}_1 \end{array}}{P \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \downarrow d \Rightarrow \langle \tau ; \varphi_1 \,\&\, (\varphi_2 \backslash_x) \rangle \rightsquigarrow \mathtt{let}\ x \overset{n}{=} \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2} \ \textsc{LetUp}$$

$$\frac{P \vdash e_2 \downarrow d \Rightarrow \langle \tau ; \varphi_2 \rangle \rightsquigarrow \mathtt{e}_2 \quad A = \varphi_2(x)}{P \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \downarrow d \Rightarrow \langle \tau ; \varphi_2 \backslash_x \rangle \rightsquigarrow \mathtt{let}\ x \overset{0}{=} \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2} \ \textsc{LetUpAbs}$$

Fig. 5: Algorithmic cardinality analysis specification, Part 2 (`let`-rules).

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

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. That is, one can think of the & operator as of adding two multi-demands, whereas $\sqcup$ is reminiscent to taking the maximum of two multi-demands.

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). In these rules, as well as in LamU, the pressed demands are treated modulo the syntactic equalities from Fig. 1 (*e.g.*, $HU \equiv U(A, A)$).

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 $\langle \varphi(x) \to \tau ; \varphi \backslash_x \rangle$; 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 Fig. 3. 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 (Fig. 5) 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 (Fig. 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 (Fig. 3), considering it to be the result of the $\mathit{transform}$. Rule LETDNABS handles the case when the variable is not used in the body, annotating the corresponding lambda with one-shot demands, in order to enable let-in floating, described in Section 5.2.

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:

```
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. Instead of removing the binding x from the elaborated program, we preserve the syntactic structure of the expressions, in order to simplify the proof of soundness of the analysis in Section 4.

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, as described by the grammar in Fig. 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. The symbol 0 serves as an indicator that the lambda is not supposed to be called at all.

Fig. 4 and 5 show 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 Fig. 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.

Heaps

$$\mathsf{H} \quad ::= \quad \varepsilon \mid [x \overset{m}{\mapsto} \mathtt{Exp}(\mathsf{e})], \mathsf{H} \mid [x \overset{m}{\mapsto} \mathtt{Val}(\mathsf{v})], \mathsf{H}$$

Stacks

$$\mathsf{S} \quad ::= \quad \varepsilon \mid (\bullet\, y) : \mathsf{S} \mid \#(x,m) : \mathsf{S} \mid ((x,y) \to \mathsf{e}) : \mathsf{S}$$

Auxiliary definitions

$$\begin{aligned} \mathit{split}(\lambda^m x\,.\,\mathsf{e}) &= (\lambda^{m_1} x\,.\,\mathsf{e}, \lambda^{m_2} x\,.\,\mathsf{e}) \text{ where } m_1 + m_2 = m \\ \mathit{split}(\mathsf{v}) &= (\mathsf{v}, \mathsf{v}) \quad \text{otherwise} \end{aligned}$$

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

$$\begin{array}{llll}
\textsc{ELet} & \langle \mathsf{H} ; \mathtt{let}\ x \overset{m}{=} \mathsf{e}_1\ \mathtt{in}\ \mathsf{e}_2 ; \mathsf{S} \rangle & \hookrightarrow & \langle \mathsf{H}, [x \overset{m}{\mapsto} \mathtt{Exp}(\mathsf{e}_1)] ; \mathsf{e}_2 ; \mathsf{S} \rangle \\
\textsc{ELkpE} & \langle \mathsf{H}, [x \overset{m}{\mapsto} \mathtt{Exp}(\mathsf{e})] ; x ; \mathsf{S} \rangle & \hookrightarrow & \langle \mathsf{H} ; \mathsf{e} ; \#(x,m) : \mathsf{S} \rangle \quad \text{if } m \geq 1 \\
\textsc{ELkpV} & \langle \mathsf{H}, [x \overset{m+1}{\mapsto} \mathtt{Val}(\mathsf{v})] ; x ; \mathsf{S} \rangle & \hookrightarrow & \langle \mathsf{H}, [x \overset{m}{\mapsto} \mathtt{Val}(\mathsf{v}_1)] ; \mathsf{v}_2 ; \mathsf{S} \rangle \\
 & & & \text{s.t. } \mathit{split}(\mathsf{v}) = (\mathsf{v}_1, \mathsf{v}_2) \\
\textsc{EUpd} & \langle \mathsf{H} ; \mathsf{v} ; \#(x, m+1) : \mathsf{S} \rangle & \hookrightarrow & \langle \mathsf{H}, [x \overset{m}{\mapsto} \mathtt{Val}(\mathsf{v}_1)] ; \mathsf{v}_2 ; \mathsf{S} \rangle \\
 & & & \text{s.t. } \mathit{split}(\mathsf{v}) = (\mathsf{v}_1, \mathsf{v}_2) \\
\textsc{EBeta} & \langle \mathsf{H} ; \lambda^m x\,.\,\mathsf{e} ; (\bullet\, y) : \mathsf{S} \rangle & \hookrightarrow & \langle \mathsf{H} ; \mathsf{e}[y/x] ; \mathsf{S} \rangle \quad \text{if } m \geq 1 \\
\textsc{EApp} & \langle \mathsf{H} ; \mathsf{e}\ y ; \mathsf{S} \rangle & \hookrightarrow & \langle \mathsf{H} ; \mathsf{e} ; (\bullet\, y) : \mathsf{S} \rangle \\
\textsc{EPair} & \langle \mathsf{H} ; \mathtt{case}\ \mathsf{e}_s\ \mathtt{of}\ (x,y) \to \mathsf{e}_r ; \mathsf{S} \rangle & \hookrightarrow & \langle \mathsf{H} ; \mathsf{e}_s ; ((x,y) \to \mathsf{e}_r) : \mathsf{S} \rangle \\
\textsc{EPRed} & \langle \mathsf{H} ; (x_1, x_2) ; ((y_1, y_2) \to \mathsf{e}_r) : \mathsf{S} \rangle & \hookrightarrow & \langle \mathsf{H} ; \mathsf{e}_r[x_1/y_1, x_2/y_2] ; \mathsf{S} \rangle
\end{array}$$

Fig. 6: Heaps, stacks and a non-deterministic counting operational semantics. The guards for counting restrictions are highlighted by grey boxes.

- 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: the 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 fixed-point iteration, with a conservative approximation in case of excessive iterations (Section 6.5).

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 *and then* an associated thunk

is accessed or lambda is invoked, 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 Fig. 6 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 Fig. 6. 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 $[x \overset{m}{\mapsto} \mathtt{Exp}(\mathsf{e})]$ and one for already evaluated expressions $[x \overset{m}{\mapsto} \mathtt{Val}(\mathsf{v})]$. 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* $((x, y) \to \mathsf{e})$, 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 rule EBETA fires only if the cardinality annotation is non-zero; it de-references an $\mathtt{Exp}(\mathsf{e})$ binding and emits an update frame. Rules EBETA, EAPP, EPAIR and EPRED are standard. 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 $[x \overset{m}{\mapsto} \mathtt{Val}(\mathsf{v})]$, and in a standard semantics would return v leaving the heap unaffected. In our counting semantics however, 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 Fig. 6). 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 Fig. 6. To do this we need to prove

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

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

$$\frac{(x:\rho)\in P \quad \langle \tau\,;\varphi\rangle = \rho(d)}{P \vdash x \downarrow d \Rightarrow \langle \tau\,;\varphi \,\&\, (x{:}1{*}d)\rangle}\ \textsc{TVarDn} \qquad \frac{x \notin \mathit{dom}(P)}{P \vdash x \downarrow d \Rightarrow \langle \bullet\,;(x{:}1{*}d)\rangle}\ \textsc{TVarUp}$$

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

$$\frac{d \sqsubseteq HU}{P \vdash \lambda^m x\,.\,\mathtt{e} \downarrow d \Rightarrow \langle \tau\,;\varepsilon\rangle}\ \textsc{TLamHU} \qquad \frac{\begin{array}{c} P \vdash \mathtt{e}_1 \downarrow C^1(d) \Rightarrow \langle \tau_1\,;\varphi_1\rangle \\ \tau_1 \preceq d_2^\dagger \to \tau_r \quad P \vdash^* y \downarrow d_2^\dagger \Rightarrow \varphi_2 \end{array}}{P \vdash \mathtt{e}_1\ y \downarrow d \Rightarrow \langle \tau_r\,;\varphi_1 \,\&\, \varphi_2\rangle}\ \textsc{TApp}$$

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

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

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

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

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

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

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

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

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

Fig. 7: Generalized demand transformers ρ, transformer environments P, and well-annotated terms with respect to a type τ and a usage environment φ.

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.

Fig. 7 says what it means to be "well-annotated", using notation from Fig. 1–3. The rules look very similar to the analysis rules of Fig. 4–5, 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 (Fig. 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 (Fig. 7). In the TLETDN rule, we make use of the auxiliary function μ (Fig. 2) and clairvoyantly choose any such transformer ρ, which is sound for the RHS expression – denoted with $P \Vdash^{\natural} \mathsf{e}_1 : \rho$. We still check 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, Fig. 7 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 $P \blacktriangleright e \downarrow d \Rightarrow \langle \tau ; \varphi \rangle \rightsquigarrow \mathsf{e}$ 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 4.1 (*Safety for annotated terms*)
If $\varepsilon \vdash \mathsf{e}_1 \downarrow HU \Rightarrow \langle \tau ; \varepsilon \rangle$ and $e_1 = \mathsf{e}_1^{\natural}$ and $\langle \varepsilon ; e_1 ; \varepsilon \rangle \longrightarrow^k \langle H ; e_2 ; S \rangle$ then there exist H, e_2 and S, such that $\langle \varepsilon ; \mathsf{e}_1 ; \varepsilon \rangle \hookrightarrow^k \langle \mathsf{H} ; \mathsf{e}_2 ; \mathsf{S} \rangle$, $\mathsf{H}^{\natural} = H$, $\mathsf{S}^{\natural} = S$ 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 Fig. 7 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 Appendix A.

Lemma 4.2 (*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 4.2 simply ensures that there *exists* a possible transition in the counting semantics that always results in a well-typed configuration. Lemma 4.2 crucially relies on yet another property, below.

Lemma 4.3 (*Value demand splitting*)
If $P \vdash \mathsf{v} \downarrow (d_1 \,\&\, d_2) \Rightarrow \langle \tau\,;\varphi\rangle$ then there exists a split $split(\mathsf{v}) = (\mathsf{v}_1, \mathsf{v}_2)$ such that: $P \vdash \mathsf{v}_1 \downarrow d_1 \Rightarrow \langle \tau_1\,;\varphi_1\rangle$ and $P \vdash \mathsf{v}_2 \downarrow d_2 \Rightarrow \langle \tau_2\,;\varphi_2\rangle$ and moreover $\tau_1 \sqsubseteq \tau$, $\tau_2 \sqsubseteq \tau$ and $\varphi_1 \,\&\, \varphi_2 \sqsubseteq \varphi$.

Why is Lemma 4.3 important? Consider the following

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

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 4.3 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 4.3 that requires demand transformers to be monotone in the demand arguments, ensured by WfTrans.

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

The proof is just a combination of Lemma 4.1 and Theorem 4.1.

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

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

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

Fig. 8: Optimised counting semantics.

semantics is defined in Fig. 8. We will show that programs that can be evaluated via the counting semantics (Fig. 6) 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.

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 Fig. 9.

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

Fig. 9 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 $[x \overset{1}{\mapsto} \mathtt{Val}(v)]$ in either the optimised or unoptimised heap. It is easy to see why: every heap binding starts life as $[x \overset{m}{\mapsto} \mathtt{Exp}(\mathsf{e})]$. By the time $\mathtt{Exp}(\mathsf{e})$ has become a value $\mathtt{Val}(\mathsf{v})$, 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 $k \in \{0,1\}$ such that $\langle \mathsf{H}_2\,;\mathsf{e}_2\,;\mathsf{S}_2\rangle \Longrightarrow^k \langle \mathsf{H}'_2\,;\mathsf{e}'_2\,;\mathsf{S}'_2\rangle$ and $\langle \mathsf{H}'_1\,;\mathsf{e}'_1\,;\mathsf{S}'_1\rangle \propto \langle \mathsf{H}'_2\,;\mathsf{e}'_2\,;\mathsf{S}'_2\rangle$.

Proof
The proof is by case analysis on the $\longrightarrow$ relation:

$$\frac{}{\varepsilon \propto \varepsilon}\ \textsc{HSim1} \qquad \frac{\mathsf{H}_1 \propto \mathsf{H}_2}{\mathsf{H}_1, [x \overset{0}{\mapsto} \mathtt{Exp}(\mathsf{e})] \propto \mathsf{H}_2}\ \textsc{HSim2} \qquad \frac{\mathsf{H}_1 \propto \mathsf{H}_2}{\mathsf{H}_1, [x \overset{0}{\mapsto} \mathtt{Val}(\mathsf{v})] \propto \mathsf{H}_2}\ \textsc{HSim3}$$

$$\frac{n \geq 1 \quad \mathsf{H}_1 \propto \mathsf{H}_2 \quad \mathsf{e}_1 \propto \mathsf{e}_2}{\mathsf{H}_1, [x \overset{n}{\mapsto} \mathtt{Exp}(\mathsf{e}_1)] \propto \mathsf{H}_2, [x \overset{n}{\mapsto} \mathtt{Exp}(\mathsf{e}_2)]}\ \textsc{HSim4}$$

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

$$\frac{}{\varepsilon \propto \varepsilon}\ \textsc{SSim1} \qquad \frac{\mathsf{S}_1 \propto \mathsf{S}_2}{(\#(x,1) : \mathsf{S}_1) \propto \mathsf{S}_2}\ \textsc{SSim2} \qquad \frac{\mathsf{S}_1 \propto \mathsf{S}_2}{(\bullet\, y) : \mathsf{S}_1 \propto (\bullet\, y) : \mathsf{S}_2}\ \textsc{SSim3}$$

$$\frac{\mathsf{S}_1 \propto \mathsf{S}_2}{(\#(x,\omega) : \mathsf{S}_1) \propto (\#(x,\omega) : \mathsf{S}_2)}\ \textsc{SSim4}$$

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

Fig. 9: Auxiliary simulation relation $\propto$ for heaps and stacks.

- Case ELET. We have two cases to consider. If $m \geq 1$ then it is obvious. If $m = 0$ then $\mathsf{H}_1' = \mathsf{H}_1, [x \overset{0}{\mapsto} \mathtt{Exp}(\mathsf{e}_1)]$ and $\mathsf{H}_2' = \mathsf{H}_2$ and $\mathsf{H}_1' \propto \mathsf{H}_2$ as required.
- Case ELKPE. In this case we have that:

$$\langle \mathsf{H}_1, [x \overset{m}{\mapsto} \mathtt{Exp}(\mathsf{e}_1)] \,;\, x \,;\, \mathsf{S}_1 \rangle \longrightarrow \langle \mathsf{H}_1 \,;\, \mathsf{e} \,;\, \#(x,m) : \mathsf{S}_1 \rangle$$

  given that $m \geq 1$. Then either OPT-ELKPEM or OPT-ELKPEO will fire:

  — If $m = \omega$ the result follows trivially.
  — If $m = 1$ then $\mathsf{S}_1' = \#(x,1) : \mathsf{S}_1$ and $\mathsf{S}_2' = \mathsf{S}_2$ and by rule SSIM2 we are done.
- Case ELKPV. By the side condition $m = m' + 1$ it can only be that $m = 1$ or $m = \omega$. By the heap invariant for H_1 and an easy induction it has to be that $m = \omega$. The corresponding rule that can fire in the optimised semantics is OPT-ELKPV and the result is trivial.
- Case EUPD. We have that:

$$\langle \mathsf{H}_1 \,;\, \mathsf{v} \,;\, \#(x,n) : \mathsf{S}_1 \rangle \longrightarrow \langle \mathsf{H}_1, [x \overset{m}{\mapsto} \mathtt{Val}(\mathsf{v}_1)] \,;\, \mathsf{v}_2 \,;\, \mathsf{S}_1 \rangle$$

  where $n = m + 1$ and $split(\mathsf{v}) = (\mathsf{v}_1, \mathsf{v}_2)$. Therefore, since $n = m + 1$, it has to be the case that $n = \omega$ or $n = 1$.

  — If $n = \omega$ then rule OPT-EUPD gives the result.
  — Let $n = 1$. Then, assume that $\langle \mathsf{H}_1 \,;\, \mathsf{v} \,;\, \#(x,n) : \mathsf{S}_1 \rangle \propto \langle \mathsf{H}_2 \,;\, \mathsf{v} \,;\, \mathsf{S}_2 \rangle$ which will happen if $\mathsf{S}_1 \propto \mathsf{S}_2$. However, in this case, $m = 0$, which means that it also must be the case that $\langle \mathsf{H}_1, [x \overset{m}{\mapsto} \mathtt{Val}(\mathsf{v}_1)] \,;\, \mathsf{v}_2 \,;\, \mathsf{S}_1 \rangle \propto \langle \mathsf{H}_2 \,;\, \mathsf{v}_2 \,;\, \mathsf{S}_2 \rangle$ so we are done in 0 steps (hence we have $\Longrightarrow^k$ and not just $\Longrightarrow$ in the statement of the theorem).

- Case EBETA follow directly from the rule OPT-EBETA.
- Case EAPP follows by OPT-EAPP.
- Cases EPRED and EPAIR follow directly from rules OPT-EPRED and OPT-EPAIR.

□

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 4.2 to get the following result.

Theorem 5.2 (*Analysis is safe for optimised semantics*)
If $\blacktriangleright e_1 \downarrow HU \Rightarrow \langle \tau ; \varepsilon \rangle \rightsquigarrow \mathsf{e}_1$ and $\langle \varepsilon ; e_1 ; \varepsilon \rangle \longrightarrow^n \langle H ; e_2 ; S \rangle$ then $\langle \varepsilon ; \mathsf{e}_1 ; \varepsilon \rangle \Longrightarrow^m \langle \mathsf{H} ; \mathsf{e}_2 ; \mathsf{S} \rangle$ s. t. $\mathsf{e}_2^\natural = e_2$, $m \leq n$, and there exist H_2 and S_2 such that $\mathsf{H}_2^\natural = H$ 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 no later have 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 & 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*)

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

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 & 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 $\langle \mathsf{H} ; \mathsf{e} ; \mathsf{S} \rangle$,

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

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

$$\langle \mathsf{H}\,;\mathtt{let}\ z \stackrel{m_1}{=} \mathsf{e}_1\ \mathtt{in}\,(\mathtt{let}\ f \stackrel{m_2}{=} \lambda^1 x\,.\,\mathsf{e}\ \mathtt{in}\ \mathsf{e}_2)\,;\mathsf{S}\rangle \Downarrow^N$$

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

$$\langle \mathsf{H}\,;\mathtt{let}\ f \stackrel{m_2}{=} \lambda^1 x\,.\,(\mathtt{let}\ z \stackrel{m_1}{=} \mathsf{e}_1\ \mathtt{in}\ \mathsf{e})\ \mathtt{in}\ \mathsf{e}_2\,;\mathsf{S}\rangle \Downarrow^{\leq N}.$$

Proof sketch: Let us refer to the first configuration as q and the second as q'. We say that two heaps, H_1 and H_2, are *related* ($\mathsf{H}_1 \simeq \mathsf{H}_2$) *iff* they are of the form

$$\begin{aligned}\mathsf{H}_1 &= \mathsf{H}_0, [z \stackrel{m}{\mapsto} \mathsf{e}_1], [f_1 \stackrel{m_1}{\mapsto} \lambda^{n_1} x\,.\,\mathsf{e}], \ldots, [f_k \stackrel{m_k}{\mapsto} \lambda^{n_k} x\,.\,\mathsf{e}]\\ \mathsf{H}_2 &= \mathsf{H}_0, [f_1 \stackrel{m_1}{\mapsto} \lambda^{n_1} x\,.\,\mathsf{e}_z], \ldots, [f_k \stackrel{m_k}{\mapsto} \lambda^{n_k} x\,.\,\mathsf{e}_z,]\end{aligned}$$

for some H_0 and k, where $\mathsf{e}_z = (\mathtt{let}\ z \stackrel{m}{=} \mathsf{e}_1\ \mathtt{in}\ \mathsf{e})$; and e, e_1 and z are from the statement of the theorem, and $\sum_{i=1}^{k} n_i = 1$.[3]

The proof goes in four stages.

1. It is the case that q evaluates in two steps to some $q_1 = \langle \mathsf{H}_1\,;\mathsf{e}_2\,;\mathsf{S}\rangle$ and q' evaluates in one step to some $q_2 = \langle \mathsf{H}_2\,;\mathsf{e}_2\,;\mathsf{S}\rangle$ such that $\mathsf{H}_1 \simeq \mathsf{H}_2$. Now we need to show that q_2 will make *at most one* step more than q_1 before they both terminate.
2. Taking ($\mathsf{H}_1 \simeq \mathsf{H}_2$) and the stacks and expressions *being the same for both configurations* as an invariant, we show that both configurations will make a step simultaneously, so the invariant is preserved until some f_k is in the configuration focus. Then we pass to the next stage.
3. If f_k is in the focus of both configurations, we consider the stack. If $\mathsf{S} = \varepsilon$ the case is done. (And so too if S contains a case alternative because both computations will be stuck.) If $\mathsf{S} = \#(x, n) : \mathsf{S}'$ then we update the heap in both branches in a $\simeq$-preserving way, so we are back to stage (2). If $\mathsf{S} = (\bullet\ y) : \mathsf{S}'$ then the "optimised" program makes *one additional step* to allocate z, and we pass to the last stage of the proof.
4. For the rest of the execution we can show that the programs will execute in lockstep with a simulation argument taking the invariant almost as in stage (2), but now with $\sum_{i=1}^{k} n_i = 0$ and z being allocated in the second heap too.

□

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

6 Implementation

We have implemented the cardinality analyser by extending the demand analysis machinery of the Glasgow Haskell Compiler (version 7.8 and later), available publicly from its open-source repository:

[3] The $\mathtt{Val}(\cdot)/\mathtt{Exp}(\cdot)$ distinction does not affect the core of the proof.

`http://git.haskell.org/ghc`

We elaborate on 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.[4] 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 only 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 & Peyton Jones 2006).

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

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 value (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:

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

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.

6.5 Handling of recursive functions

For our formal presentation we had the liberty to assume that `let`-expressions are *non-recursive*, in rule LETDN in Fig. 5. In reality, lets are recursive, and GHC has to deal with them. Ideally, we would like to find the least usage signature ρ so that

$$\frac{\begin{array}{c} P, f{:}\rho \mapsto e_1 \downarrow U \Rightarrow \langle \tau_1 \,;\, \varphi_1 \rangle \rightsquigarrow \mathtt{e}_1 \qquad \rho = \langle k \,;\, \varphi_1(\overline{y}) \to \tau_1 \,;\, \varphi_1 \backslash_{\overline{y}} \rangle \\ P, f{:}\rho \mapsto e_2 \downarrow d \Rightarrow \langle \tau \,;\, \varphi_2 \rangle \rightsquigarrow \mathtt{e}_2 \qquad \varphi_2(f) \sqsubseteq n * C^{n_1}(\ldots(C^{n_k}(\ldots)\ldots)) \end{array}}{P \mapsto \texttt{let}\ f = \lambda\, y_1 \ldots y_k \,.\, e_1\ \texttt{in}\ e_2 \downarrow d \Rightarrow \langle \tau \,;\, (\varphi_2 \backslash_f) \rangle \rightsquigarrow \texttt{let}\ f \stackrel{n}{=} \lambda^{n_1} y_1 \ldots \lambda^{n_k} y_k \,.\, \mathtt{e}_1\ \texttt{in}\ \mathtt{e}_2}$$

holds. But that is itself a recursive specification and hence non-executable.

Therefore, we employ a usual fixed-point iteration. We start with the most optimistic signature $\rho^0 = \langle k \,;\, A \to \cdots \to A \to \bullet \,;\, \varepsilon \rangle$ which claims that f uses neither any of its k arguments nor its free variables[5] and calculate

$$P, f{:}\rho^i \mapsto e_1 \downarrow U \Rightarrow \langle \tau_1 \,;\, \varphi_1 \rangle \rightsquigarrow \mathtt{e}_1^i \qquad \rho^{i+1} = \langle k \,;\, \varphi_1(\overline{y}) \to \tau_1 \,;\, \varphi_1 \backslash_{\overline{y}} \rangle \,.$$

If we have $\rho^i = \rho^{i+1}$ for some i, we found the desired fixed-point. We analyse the body

$$P, f{:}\rho^i \mapsto e_2 \downarrow d \Rightarrow \langle \tau \,;\, \varphi_2 \rangle \rightsquigarrow \mathtt{e}_2 \qquad \varphi_2(f) \sqsubseteq n * C^{n_1}(\ldots(C^{n_k}(\ldots)\ldots))$$

and obtain

$$P \mapsto \texttt{let}\ f = \lambda\, y_1 \ldots y_k \,.\, e_1\ \texttt{in}\ e_2 \downarrow d \Rightarrow \langle \tau \,;\, (\varphi_2 \backslash_f) \rangle \rightsquigarrow \texttt{let}\ f \stackrel{n}{=} \lambda^{n_1} y_1 \ldots \lambda^{n_k} y_k \,.\, \mathtt{e}_1^i\ \texttt{in}\ \mathtt{e}_2 \,.$$

Note that, unless the `let` is not actually recursive, e_2 will put a demand on both f and its other free variables. The strictness signature of f will (eventually) mention the free

[5] In the implementation, which is combined with GHC's strictness analysis, the initial signature is actually "hyperstrict", *i.e.*, that of a bottoming function.

variables of e_2, so the demands put on the free variables are necessary multiple-use, and no $1*\_$ annotation that is not hidden behind a $C^n(\_)$ demand will survive there, even when in fact there is only one use in the complete recursion. This is one cause of imprecision (Section 7.3).

Unfortunately, our domain (i.e., the cpo of usage signatures ρ) does not have finite height and therefore it is not guaranteed that this iteration terminates. If no fixed-point is found after a finite number of steps (currently 10), we abort the search. In order to obtain a sound result, we re-analyse e_1 one final time, this time with a most pessimistic signature ρ^∞. If the domain of triples had a top element, that would be a suitable choice, but such an element would have to mention all variables in its usage of free variables, which is not expressible. Instead, we use φ^{10}, the free-variable usage component of ρ^{10}, which mentions all free variables that are relevant to e_2, but possibly with a demand that is too good to be true, and adjust that pessimistically:

$$\rho^\infty = \langle k\,;\, U \to \cdots \to U \to \bullet\,;\{x{:}U \mid x \in dom(\varphi^{10})\}\rangle$$

This signature is larger than any analysis result that we expect for e_2 and hence a conservative assumption.

After analysing e_1 and e_2 using ρ^∞ as the signature for f, i.e.

$$\begin{array}{l} P, f{:}\rho^\infty \mapsto e_1 \downarrow U \Rightarrow \langle \tau_1\,;\varphi_1\rangle \rightsquigarrow \mathtt{e}_1 \\ P, f{:}\rho^\infty \mapsto e_2 \downarrow d \Rightarrow \langle \tau\,;\varphi_2\rangle \rightsquigarrow \mathtt{e}_2 \qquad \varphi_2(f) \sqsubseteq n * C^{n_1}(\ldots(C^{n_k}(\ldots)\ldots)), \end{array}$$

we obtain

$$P \mapsto \mathtt{let}\, f = \lambda\, y_1 \ldots y_k\,.\, e_1\ \mathtt{in}\ e_2 \downarrow d \Rightarrow \langle \tau\,;(\varphi_2 \backslash f)\rangle \rightsquigarrow \mathtt{let}\, f \stackrel{n}{=} \lambda^{n_1} y_1 \ldots \lambda^{n_k} y_k\,.\,\mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2\,.$$

6.6 Accelerating fixed-point computation

Running the analyser on nested recursive definitions can be expensive at compile-time. For instance, for two functions `f` and `g`, such that `g` is nested under `f`, the analyser must find a fixed-point for the inner function `g` at each iteration of the fixed-point computation for function `f`. To remedy this, we use the simple widening strategy from the literature (Henglein 1994), based on the observation that iterations of the fixed-point process for `f` generates a *monotonically increasing* sequence of usage signatures for `f`. Therefore, each time we begin the fixed-point process for `g`, the environment contains values that are no smaller (in the demand partial order) than the corresponding values the previous time we encountered `g`. It follows that the correct fixed-point for `g` will be greater than the correct fixed-point found on the previous iteration of `f`. Therefore *we can begin the fixed-point process for* `g` *not with the bottom value, but rather with the result of the previous analysis*. In the implementation, this result is conveniently available in the elaborated term $\mathtt{e}_1$.

We also improve it a bit more by *splitting* the environment component φ of a usage signature, separating variables with *multiple*-use demands from the other ones. The intuition is that multiple-use demands cannot be increased any further, and, therefore, do not contribute to the fixed-point computation.

Program	Syntactic 1S-λ	Syntactic 1U-Thunks	Runtime 1U-Thunks
`anna`	4.0%	7.2%	2.9%
`banner`	14.3%	20.0%	5.3%
`boyer2`	3.3%	20.0%	0.0%
`bspt`	5.0%	15.4%	1.5%
`cacheprof`	7.6%	11.9%	5.1%
`calendar`	5.7%	0.0%	0.2%
`circsim`	2.6%	4.0%	3.0%
`constraints`	2.0%	3.2%	4.5%
`cryptarithm1`	0.0%	0.0%	5.3%
`cryptarithm2`	0.6%	3.0%	74.0%
`cse`	4.2%	2.8%	1.8%
`eliza`	0.0%	0.0%	48.7%
`expert`	3.4%	4.3%	3.9%
`fem`	19.2%	17.6%	1.7%
`fft2`	6.6%	0.0%	0.4%
`fluid`	7.3%	4.6%	2.3%
`fulsom`	5.4%	7.3%	8.0%
`gamteb`	40.2%	22.0%	0.9%
`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%
`knights`	10.4%	23.4%	1.3%
`life`	3.2%	0.0%	1.8%
`lift`	2.1%	0.0%	1.1%
`listcopy`	11.5%	21.4%	1.8%
`mandel`	12.3%	4.2%	3.9%
`mkhprog`	27.4%	20.8%	5.8%
`nucleic2`	3.5%	3.1%	3.2%
`parser`	7.5%	24.7%	1.4%
`partstof`	5.8%	10.7%	0.1%
`puzzle`	16.5%	28.0%	68.9%
`reptile`	10.2%	13.8%	1.0%
`rewrite`	6.7%	6.0%	19.9%
`scc`	0.0%	0.0%	0.8%
`solid`	5.5%	2.4%	0.0%
`sphere`	7.8%	6.2%	20.0%
`typecheck`	3.9%	9.4%	0.9%
`wheel-sieve1`	10.5%	0.0%	0.0%
`x2n1`	0.0%	0.0%	0.1%
	... and 50 more programs		
Arithmetic mean	10.3%	12.6%	5.5%

Table 1: Analysis results for `nofib`: ratios of *syntactic* one-shot lambdas, *syntactic* single-entry thunks and *runtime* entries into single-entry thunks.

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). For the sake of presentation, in the table we show the most interesting programs with non-trivial contributions to the overall analysis statistics. The numbers are quite encouraging. One-shot lambdas account for 0-30% of all lambdas (with the arithmetic mean being 10.3%), while single-entry thunks are 0-23% of all thunks (with the arithmetic mean 12.6%).

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 "Runtime 1U-Thunks" 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`.

It is important to note that the results of the optimised execution, although related with the numbers of one-shot lambdas and single-entry thunks in the `nofib` programs *themselves*, are much likely caused by the analysis results and the subsequent optimisations for the standard libraries.

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 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. "Allocation" 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 the 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, but it's virtually impossible to say for sure: any change that affects inlining (which discovering one-shot-lambdas certainly does) has knock-on effects propagate down the long optimisation pipeline, with unpredictable consequences for code size.

Program	Allocation		Runtime	
	No hack	Hack	No hack	Hack
`anna`	-2.2%	-0.2%	+0.1%	+0.1%
`banner`	+3.5%	-0.1%	-0.0%	-0.0%
`boyer2`	-0.4%	-0.4%	+0.0%	-0.0%
`bspt`	-2.2%	-0.0%	-0.0%	+0.0%
`cacheprof`	-7.5%	-0.6%	-6.0%	-1.7%
`calendar`	-9.2%	+0.2%	-0.0%	-0.0%
`circsim`	-7.5%	-0.0%	-4.3%	-2.0%
`constraints`	-0.9%	-0.0%	-1.2%	-0.2%
`cryptarithm1`	-0.0%	-0.0%	+2.3%	+0.0%
`cryptarithm2`	-0.3%	-0.0%	-2.3%	-0.0%
`cse`	-4.6%	-0.0%	+0.0%	+0.0%
`eliza`	-2.2%	-0.1%	+0.0%	+0.0%
`expert`	-1.8%	-0.1%	-0.0%	-0.0%
`fem`	-2.2%	-0.0%	-0.0%	-0.0%
`fft2`	-34.8%	-0.0%	+0.0%	-0.0%
`fluid`	-3.4%	-0.0%	-0.0%	-0.0%
`fulsom`	-0.7%	-0.0%	-0.0%	+1.8%
`gamteb`	+3.1%	+0.5%	+0.0%	+0.0%
`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%
`knights`	-1.9%	-0.0%	+0.0%	+0.0%
`life`	-0.8%	-0.0%	-3.4%	+0.0%
`lift`	-1.9%	-0.0%	-0.0%	-0.0%
`listcopy`	+1.2%	-0.0%	+0.1%	+0.1%
`mandel`	-1.9%	-0.0%	+0.0%	+0.0%
`mkhprog`	-11.9%	+0.1%	-0.0%	-0.0%
`nucleic2`	-14.1%	-10.9%	+0.0%	+0.0%
`parser`	-0.2%	-0.2%	+0.0%	+0.0%
`partstof`	-95.5%	-0.0%	-0.0%	-0.0%
`puzzle`	-8.2%	-0.0%	+0.1%	+0.1%
`reptile`	-2.7%	-0.0%	-0.0%	-0.0%
`rewrite`	-6.6%	-0.0%	-0.0%	-0.0%
`scc`	-0.3%	-0.4%	-0.0%	-0.0%
`solid`	-0.6%	-0.0%	+0.0%	+0.0%
`sphere`	-1.5%	-1.5%	-0.0%	-0.1%
`typecheck`	-0.5%	-0.0%	+0.1%	-0.1%
`wheel-sieve1`	-18.7%	-0.0%	-4.0%	+0.7%
`x2n1`	-29.9%	-0.0%	-0.0%	-0.0%
... and 50 more programs				
Best improvement	95.5%	10.9%	8.8%	6.6%
Worst degradation	3.5%	0.5%	2.3%	2.6%
Geometric mean improvement	6.0%	0.3%	1.8%	1.0%

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

Program	Runtime 1U-Thunks	No-Opt Runtime	Runtime 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 the Computer Language Benchmark Game.

Considering *allocation*, the numbers relative to the un-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. One can notice, though, 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 (even comparing to the un-hacked baseline compiler).

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%. To produce more realistic average numbers for the whole `nofib` suite, we have re-run the suite several times. As a result, some short-running outliers have been averaged out, and overall runtime statistics for individual programs has slightly changed comparing to the conference version of this paper (Sergey *et al.* 2014).

For more realistic numbers, we measured the improvement in runtime, relative to the hacked compiler, for several programs from the Computer Language Benchmarks Game.[6] 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.

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

Library	Syntactic 1S-λ	Syntactic 1U-Thunks	Benchmark name	Allocation 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 selected `hackage` libraries.

7.2 Real-world programs

To test our analysis and the cardinality-powered optimisations on some real-world programs, we chose a number of continuation-heavy libraries from the `hackage` repository:[7] `attoparsec`, a fast parser combinator library, `binary`, a lazy binary serialisation library, `bytestring`, a space-efficient implementation of byte-vectors, and `cassava`, a parsing and encoding library for CSV-files.

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 single-entry 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 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.

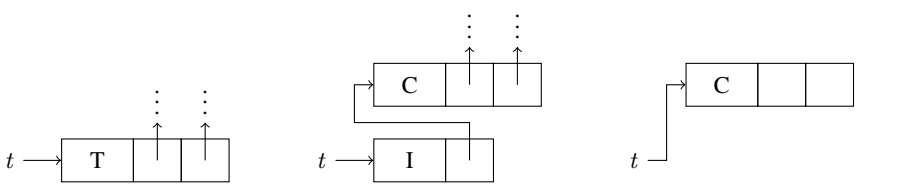
7.3 Precision and missed opportunities

After having formally established that our changes are semantically correct, and empirically that they are beneficial, one might still wonder how *complete* they are: does our analysis find all single-entry thunks and one-shot functions, and if not, what opportunities did it miss? Any static analysis will be approximate, but it would not be surprising if the analysis missed some low-hanging fruit.

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

Program	LOC	GHC Allocation Δ		GHC Runtime 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 of large `nofib` programs with optimised GHC.



(a) Before evaluation **(b)** After evaluation **(c)** After garbage collection

Closure types: T: thunk, I: indirection, C: value

Fig. 10: Heap during evaluation of a thunk t (un-instrumented runtime)

In this section we report on a study in which we use a specially instrumented version of the compiler to make dynamic, runtime measurements to see how often each thunk is entered in an actual program run. Then we compare these runtime figures with the results of the static analysis.

In this study we focus only on single-entry thunks. One could imagine doing a similar study for one-shot lambdas, but we leave that as further work.

7.3.1 Runtime instrumentation

Our goal is this: for every dynamically allocated instance of a thunk, we want to observe how often it is used.

To see why this cannot be observed in an unmodified version of the runtime, let us recall how thunks are evaluated in GHC. At run time, a thunk is represented as a *closure* that is stored in the heap, referencing its program code as well as the values captured by its free variables, as pictured in Fig. 10. Upon its first use, the closure is *entered*, *i.e.*, jumped to. Immediately after that, the thunk code T performs the following actions:

1. First, it replaces the closure by a *black hole*, a special type of closure used to mark values under evaluation,
2. Next, pushes an *update frame*, which will be activated later, onto the stack,

3. Then, it runs the actual code of the closure, which will eventually evaluate to a value C.
4. This value is then returned via the stack to the update frame, which replaces the black hole by an *indirection* I, pointing to the returned value C; see Fig. 10(b).
5. Finally, the value is returned to the code that triggered the evaluation of the thunk T.

Any subsequent use of a pointer to (what used to be) the thunk T enters the indirection I, which simply returns the value C. We might hope to count the number of times T is used by counting the number of times the indirection is entered.

However, the next run of the garbage collector replaces a pointer to the indirection I by a direct pointer to the indirection's target C (Fig. 10(c)). Hence, after garbage collection, only the final value remains in the heap, without any indication that this value came from our original thunk T. Therefore, we have no way to relate any subsequent uses of this value to the original thunk T, whose runtime cardinality we were planning to measure.

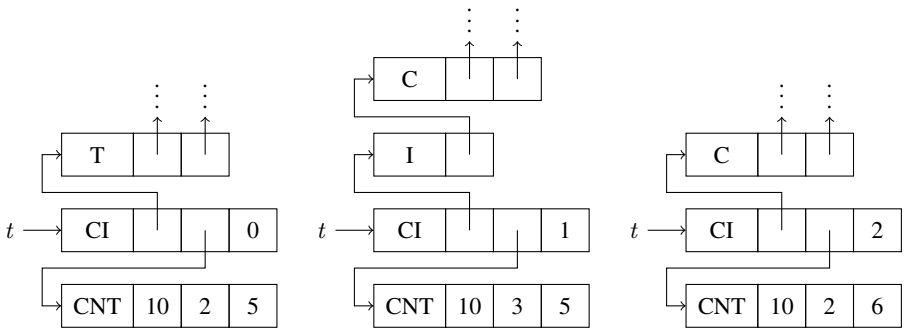
In order to observe *all* uses of a thunk, we implemented a new type of closures in GHC's runtime, dubbed *counting indirection* (CI). When entered, these indirections behave as normal indirections, *i.e.*, they evaluate the closure they are pointing to. The important difference is that the garbage collector does not erase them, but instead copies them like any other closure. More precisely, we do the following:

- When dynamically allocating a thunk in the heap, we allocate *two* heap objects, the thunk itself T, and a counting indirection CI that points to T (Fig. 11(a)).
- As well as pointing to T, the dynamically allocated counting indirection also contains
  - CI.cnt: a pointer to a *static* data structure, CNT.
  - CI.entries: a private count of the number of times the indirection has been entered.
- There is a single, static CNT record for each syntactic closure, or allocation site A. The CNT record contains three fields[8]
  - CNT.allocs: the number of times allocation site A has been executed; that is, how many thunks have been allocated by A.
  - CNT.once: the number of those thunks that have been entered exactly once
  - CNT.multi: the number of those thunks that have been entered more than once.

When the counting indirection is entered the first time (CI.entries = 0), it increments CI.entries, and the CNT.once counter in the static CNT record. If it is entered a second time (CI.entries = 1), it again increments CI.entries, decrements CNT.once and increments CNT.multi. Further uses of the counting indirection simply increase CI.entries.

A particular instance of this modified scenario is depicted in Fig. 11(a), where the counter CNT records indicates that so far 10 closure instances have been allocated, out of which 2 have been used at most once and 5 were used multiple times. After the first evaluation of the newly allocated thunk, the private CI.entries field is incremented, along with CNT.once (Fig 11(b)). After the second entry, CI.entries becomes 2, while the CNT.multi field has gone from 5 to 6, recording that one more instance of this thunk has been entered more than once (Fig 11(c)).

[8] The static CNT record contains additional fields, not relevant to the discussion.



(a) Before evaluation **(b)** After first evaluation **(c)** After second evaluation (and garbage collection)

Closure types: CI: counting indirection, CNT: tick-ticky counter record (static)

Fig. 11: Heap during evaluation of a thunk t (instrumented runtime)

observed	syntactic thunks determined to be single entry	syntactic thunks determined to be multi entry	dynamic thunks determined to be single entry	dynamic thunks determined to be multi entry
never used	19	525	157444	1,608,128
entered once	1,310	3,498	4,893,280	171,101,068
multiple times	0	3,653	0	66,457,533

Table 6: Precision of the analysis: Allocated thunks

7.3.2 Evaluating soundness and completeness

This instrumentation allowed us to check the actual implementation for two things:

- *Soundness*. Does the executing program enter any thunk multiple times that the analysis determined as single-entry? If so, the analysis is wrong.
- *Completeness*. How many thunks are thought to be multiple-entry by the analysis, but are entered only once during execution? Perhaps a more precise analysis could find more single-entry thunks?
  Of course, in a different execution of the same program, the same syntactic thunk might be entered more than once, so the analysis is not necessarily at fault. Moreover the analysis is necessarily approximate. But still, it is worth a manual analysis of these apparently missed opportunities.

We compiled programs from the `nofib` benchmark suite with the instrumentation described above, linked them against an uninstrumented base library and ran each program once. We obtained the results in Table 6. The first pair of columns, “syntactic thunks”,

Extended Syntax

$$
\begin{array}{rcl}
n & ::= & 1 \mid \omega\, \mathscr{P}(r) \\
m & ::= & 0 \mid 1 \mid \omega\, \mathscr{P}(r) \\
r & ::= & \text{datacon} \mid \text{fix} \mid \text{cpe} \mid \text{both} \mid \ldots
\end{array}
$$

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

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

$$
\boxed{n_1 \sqcup n_2 = n_3 \qquad n_1 \,\&\, n_2 = n_3}
$$

$$
\begin{array}{rclrcl}
1 \sqcup 1 & = & 1 & 1 \,\&\, 1 & = & \omega\,\{\text{both}\} \\
1 \sqcup \omega\, r & = & \omega\, r & 1 \,\&\, \omega\, r & = & \omega\,(r \cup \{\text{both}\}) \\
\omega\, r \sqcup 1 & = & \omega\, r & \omega\, r \,\&\, 1 & = & \omega\,(r \cup \{\text{both}\}) \\
\omega\, r_1 \sqcup \omega\, r_2 & = & \omega\,(r_1 \cup r_2) & \omega\, r_1 \,\&\, \omega\, r_2 & = & \omega\,(r_1 \cup r_2 \cup \{\text{both}\})
\end{array}
$$

Fig. 12: Modified syntax and operations to track reasons of precision loss.

gives the results by allocation site. For example, across all the program runs, there were 19 allocation sites that were determined to be single-entry, but were never entered at all.

The second pair of columns, "dynamic thunks", gives the result by dynamically allocated thunk instances. This emphasises those thunks that are evaluated most often; allocation sites with very few instances don't matter much. For example, across all program runs there were 4,893,280 thunks allocated at allocation sites marked single-entry, that were indeed entered exactly once.

On soundness the news is good: the table confirms that every thunk that we determined to be single-entry (the first column of each pair) was indeed used at most once (the zero entries in the third row).

On completeness, the news is not so good. Consider all the syntactic thunks (i.e. allocation sites) whose instances were entered at most once (i.e. the first two rows of the table). These are the candidates that cardinality analysis might determine as single-entry. But only 1,329 (i.e $1,310 + 19$) were so determined, with 4,023 being missed. So we are missing 75% of the plausible opportunities! It get worse when we consider the dynamic-thunk columns: only 2.8% of the thunks that are actually entered at most once are identified as such by the analysis.

So what about these 172,709,196 dynamic thunks that were used once or less, but where our analysis did not predict that? We call them the "plausible opportunity" thunks. The natural question is: could the analysis have done better for these thunks?

7.4 Missed opportunities

To learn more about the missed opportunities, we extended the usage types so that with every ω occurring in a demand on a plausible-opportunity thunk, we could also track the reason for that pessimistic conclusion.

To that end we extended the type for cardinalities (n and m in Fig 1) to keep track of a set of reasons, which are just strings injected at various places in the code; for example the reason datacon is added to the many-used demand put on the arguments of a data constructor application when the incoming demand on its result is non-informative. The operations $\sqcup$ and & combine reasons from both arguments, as shown in Figure 12. When reporting the counters of the instrumented runtime presented in the previous section, all reasons for this particular thunk to not be assumed one-shot are printed along with it.

Using this more detailed analysis, we found that almost all the plausible-opportunity thunks fall into one of four categories:

1. The large majority of missed opportunities (71.7%) are due to thunks that are stored in constructors (*e.g.*, in tuples, lists, arrays). There are two reasons for poor precision:
   - Our analysis can transport the demand on tuples and other product types into the argument of constructors. But this is only helpful if the demand on the product type is known. Since the analysis looks at function definitions *before* their uses, this works in the case of `f(x,y)`, where we can use the nested demand information in the strictness signature of `f` to get information on `x` and `y`. However, if a tuple is returned from a function such as `f x = (x+1, y-1)`, the demand on the result of `f` is not known and we have to assume the thunk `x+1` to be used *multiple* times. Returning a constructor in this way is a very frequent pattern.
   - Currently, our analysis only computes nested demand information for *product* types. Extending it to sum types is possible, but experiments using a prototype[9] showed no relevant improvements. This is not surprising, as data constructors of sum types are routinely returned from functions and thus especially affected by the afore-mentioned problem. Additionally, extending demand analysis to sum-types poses the problem of getting precise results for *recursive* types (which are almost invariably sums), not addressed by this work.
2. The next frequent case, accounting for 22.2% of missed opportunities, arises from when the cardinality analysis has to give up because the use of the thunk occurs inside a recursive function.[10] This is often the result of using `foldr` together with short-cut deforestation (Gill *et al.* 1993), and typically results in code of the following shape:

```
let foo xs = let thunk = f x
             in let go [] = thunk
                    go (x:xs) = g x (go xs)
                in go xs
```

   Clearly, the thunk is called at most once, but the call comes from a recursive function `go`, where the cardinality analysis has to make the conservative assumption that everything used by `go` is used more than once, as discussed in Section 6.5.
   In order for our analysis to detect that `thunk` in `foo` is called at most once, it would have to see that

[9] provided by Ömer Sinan Ağacan
[10] This number is severely inflated by a single static thunk in `fannkuch-redux` accounting for 21.0%.

(a) although it is called from within a recursive function, it is not called *together* with the recursive function, so it lies, in a way, on the exit path from the loop,
(b) the recursion here is *linear*: once it is started, its exit path is executed once, and
(c) the recursion is initially started at most once.

An analysis that is capable of doing such reasoning is Call Arity (Breitner 2015a), which is a separate analysis in GHC. Call Arity is a forward analysis, while our analysis is a backwards analysis, so combining the two to improve the handling of recursive functions is non-trivial and future work.

3. Around 4% of the missed opportunities are thunks created in the last Core-to-Core pass, which transforms the program into A-normal form, in preparation of lowering the program to STG. This involves introducing let-bindings for all non-trivial function arguments. Usually, the pass will use the information found in the function's strictness signature and attach it to the newly created thunks, but if there is no such signature, or the function is not saturated, a conservative assumption is made here. There might be room for improvement here, but 4% is hardly a fat target.
4. Only 1.3% of the missed opportunities are due to uses of the both operator (&). Such a case can arise from a call to the function `maybe d f mb`. The function `maybe` uses either `d` or `f` (depending on `mb`), but never both; the analysis does not see that.

Less than 1% of missed opportunities have other reasons (*e.g.*, arguments to primitive operations); 0.2% of missed opportunities are due to more than one reason.

In short, there does not seem to be a lot of low-hanging fruit here. We are not optimistic for radical improvements in the treatment of data structures. Probably the best opportunity is using Call Arity to improve case (2).

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 & Partain 1994). This question was first studied in the late 80's, when an elegant representation of usage analysis in terms of *projections* (Hinze 1995) was given by Wadler and Hughes (Wadler & 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 & 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 & 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[11] (Turner *et al.* 1995; Gustavsson 1998; Wansbrough & Peyton Jones 1999; Wansbrough 2002).

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 & 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 & 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 brand-new lines of 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 (even though we use the type system vocabulary when defining demand types). 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 the following function?

```
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 take the least upper bound of 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`.

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

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 a *polymorphic effect system* (Hage *et al.* 2007; Holdermans & Hage 2010; Verstoep & Hage 2015). Moreover, the analysis approach deals very naturally with absence, and with product types such as pairs, which are ubiquitous. Most of type-based approaches do not do so well here (except for the type-based analysis by Verstoep & Hage (2015), which handles absence, but has not been implemented and evaluated in practice).

Comparing to polymorphic effect systems, a weakness of our approach is that as soon as a value is stored in a data structure, we entirely lose track of its usage cardinality. Type-based approaches can use usage polymorphism to track usage *within* data structures. Consider, for example, a usage-polymorphic data type `Tree`, defined as follows

```
data Tree c = Leaf (Int ->c Int)
            | Node (Tree c) (Tree c)
```

where "`->c`" is a type of functions called no more than `c` times. So a value of type `(Tree 1)` is a tree of called-once functions. This approach works, but when Wansborough tried it at scale he found that he had to add *thousands* of cardinality variables to some data types (Wansbrough 2002, § 6.4.11). So the approach did not appear to scale well at all.

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. However, ours is the only one that deals with one-shot lambdas; the others are concerned only with single-entry thunks.

One other prominent type-based usage system is Clean's *uniqueness types* (Barendsen & 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). In a related analysis framework by Hage *et al.* (2007), based on a polymorphic type-and-effect system, a similar dichotomy is accounted for by two different subeffecting rules (T-SubUp) and (T-SubDown).

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 & Hage 2010). In particular, applicativeness means that an expression is 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.

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 & Might 2010). This approach also greatly simplifies integration of the counting abstract domain to account for sharing (Might & 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 topic for the future work to build a backwards analysis from AAM.

8.4 Related analyses in GHC

Besides the implementation of the cardinality analysis we present there are two further related analyses employed by the compiler.

The goal of *arity analysis* (Xu & Peyton Jones 2005) is to enable the transformation known as *lambda-floating* by providing an answer to the question "given a function `f`, what is the *minimal* number of arguments `f` will be always given when called?". Taking the number of top-level lambdas is sound, but imprecise. We believe that the information necessary for lambda-floating can be inferred from the results of our cardinality analyser. What makes us sure is the observation that operationally an inferred call demand `C(C(...))` for a function `f` indicates that `f`, whenever used, is applied to *at least* as many arguments as there are `C`s in the demand.

The goal of *Call Arity* analysis (Breitner 2015a) is similar: it also tries to determine a lower bound on the number of arguments a function is given. Motivated by runtime inefficiencies caused by applying list fusion to left folds, the main strength of the call arity analysis is that it is able to determine that a thunk or a function is used once even if the call site lies within a recursive function. In order to do so, it analyses all `let`-bindings downwards and returns *co-call graphs*, indicating which functions and thunks are called together. For this analysis, an Isabelle formalization exists that proves not only that the analysis and transformation preserves the semantics, but also and more notable that it does not degrade the program (Breitner 2015b). A more detailed treatment of the analysis and its formalization can be found in the third-named author's thesis (Breitner 2016).

9 Conclusion

The fourth-named author has been trying to crack this problem for nearly two decades. The tradeoff between precision, information flow, complexity, and implementation payoff, is a complex one. We now have better news. The cardinality analysis described here is

simple to implement (it added 250 lines of code to a 140,000 line compiler), and, even in the presence of the shortcomings and potential precision losses identified in Section 7.3, 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 2014 and JFP reviewers for their substantial, detailed, and constructive feedback. Finally, we are grateful to Matthias Felleisen for his work as our JFP editor.

References

Barendsen, Erik, & Smetsers, Sjaak. (1996). Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical structures in computer science*, **6**(6), 579–612.

Breitner, Joachim. (2015a). Call arity. *Pages 34–50 of: Trends in functional programming*. LNCS, vol. 8843. Springer.

Breitner, Joachim. (2015b). Formally proving a compiler transformation safe. *Pages 35–46 of: Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM.

Breitner, Joachim. (2016). *Lazy evaluation: From natural semantics to a machine-checked compiler transformation*. Ph.D. thesis, Karlsruhe Institute of Technology.

Gill, Andy. (1996). *Cheap deforestation for non-strict functional languages*. Ph.D. thesis, University of Glasgow, Department of Computer Schence.

Gill, Andy, Launchbury, John, & Peyton Jones, Simon L. (1993). A Short Cut to Deforestation. *Pages 223–232 of: Proceedings of the Sixth ACM Conference on Functional Programming Languages and Computer Architecture*. ACM Press.

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

Goldberg, Benjamin. (1987). Detecting sharing of partial applications in functional programs. *Pages 408–425 of: Proceedings of International Conference on Functional Programming Languages and Computer Architecture*. LNCS, vol. 274. Springer-Verlag.

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

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

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

Henglein, Fritz. (1994). Iterative fixed point computation for type-based strictness analysis. *Pages 395–407 of: Proceedings of the First International Static Analysis Symposium (SAS'94)*. LNCS, vol. 864. Springer-Verlag.

Hinze, Ralf. (1995). *Projection-based strictness analysis - theoretical and practical aspects*. Ph.D. thesis, Bonn University.

Holdermans, Stefan, & Hage, Jurriaan. (2010). Making "stricternes" more relevant. *Pages 121–130 of: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM 2010)*. ACM.

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

Jones, Richard. (1992). Tail recursion without space leaks. *J. funct. program.*, **2**(1), 73–79.

Launchbury, John, Gill, Andy, Hughes, John, Marlow, Simon, Peyton Jones, Simon L., & Wadler, Philip. (1993). Avoiding unnecessary updates. *Pages 144–153 of:* Launchbury, John, & Sansom, Patrick M. (eds), *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Workshops in Computing. Springer.

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

Might, Matthew, & Shivers, Olin. (2006). Improving flow analyses via ΓCFA: abstract garbage collection and counting. *Pages 13–25 of: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*. ACM.

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

Partain, Will. (1993). The `nofib` benchmark suite of Haskell programs. *Pages 195–202 of:* Launchbury, John, & Sansom, Patrick M. (eds), *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Workshops in Computing. Springer.

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

Peyton Jones, Simon L., & Partain, Will. (1994). Measuring the effectiveness of a simple strictness analyser. *Pages 201–220 of: Proceedings of the 1993 Glasgow Workshop on Functional Programming*. Springer.

Peyton Jones, Simon L., & Santos, André. (1998). A transformation-based optimiser for Haskell. *Science of computer programming*, **32**(1-3), 3–47.

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

Sabry, Amr, & Felleisen, Matthias. (1992). Reasoning about programs in continuation-passing style. *Pages 288–298 of: Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*. LISP Pointers, Vol. V, No. 1. ACM.

Sergey, Ilya, Vytiniotis, Dimitrios, & Peyton Jones, Simon L. (2014). Modular, higher-order cardinality analysis in theory and practice. *Pages 335–348 of: Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2014)*. ACM.

Sestoft, Peter. (1997). Deriving a lazy abstract machine. *J. funct. program.*, **7**(3), 231–264.

Turner, David N., & Wadler, Philip. (1999). Operational interpretations of linear logic. *Theor. comput. sci.*, **227**(1-2), 231–248.

Turner, David N., Wadler, Philip, & Mossin, Christian. (1995). Once upon a type. *Pages 1–11 of: Proceedings of the seventh ACP Conference on Functional Programming Languages and Computer Architecture*. ACM.

Van Horn, David, & Might, Matthew. (2010). Abstracting abstract machines. *Pages 51–62 of: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*. ACM.

Verstoep, Hidde, & Hage, Jurriaan. (2015). Polyvariant cardinality analysis for non-strict higher-order functional languages: Brief announcement. *Pages 139–142 of: Proceedings of the 2015 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2015)*. ACM.

Wadler, Philip, & Hughes, John. (1987). Projections for strictness analysis. *Pages 385–407 of: Proceedings of International Conference on Functional Programming Languages and Computer Architecture*. LNCS, vol. 274. Springer-Verlag.

Wansbrough, Keith. (2002). *Simple polymorphic usage analysis*. Ph.D. thesis, Computer Laboratory, University of Cambridge.

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

Xu, Dana, & Peyton Jones, Simon L. (2005). *Arity analysis*. Unpublished draft.

Appendix

A Proofs of soundness of the analysis

This appendix provides typing rules for stacks and heaps, omitted from the main paper body, and proves the soundness of the analysis (Section 4).

A.1 Stack and heap typing for analysis safety

Definition A.1 (*Configuration typing*)
We write $P \vdash \langle \mathsf{H}; \mathsf{e}; \mathsf{S} \rangle$ to mean that there exist d, τ, φ_1 and φ_2 such that $P \vdash \mathsf{e} \downarrow d \Rightarrow \langle \tau; \varphi_1 \rangle$ and $P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d, \tau) \Rightarrow \varphi_2$ and $P \vdash H \sim (\varphi_1 \,\&\, \varphi_2)$ according to the heap and stack typing rules of Fig. 13.

Fig. 13 explains how we type stacks and heaps. The judgement $P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d, \tau) \Rightarrow \varphi$ intends to identify the fv-usage environment of the stack S, given that the argument that we intend to place in the hole of the stack has type τ when being imposed with demand d. Rule SHU deals with the case when we impose no demand on the hole of the stack – consequently the stack must be empty! Rule SARR deals with the case when the stack demands the application of the expression in the hole to an argument and hence the shape of the stack has to be $(\bullet\ y) : \mathsf{S}$. The corresponding demand that this particular stack expresses is $C^1(d)$ where d is the demand expressed by the rest of the stack. The following three rules (SUPDUP, SUPDUPABS, and SUPDDN) correspond to the flavors of LETDN that we encountered in the typing rules. If we encounter a stack $\#(x, n) : \mathsf{S}$ then what is the demand that is placed on x? In the continuation S the variable x will be immediately used with some demand d but it might be that the continuation induces further calls to x which

end up pressing an additional $m * d_x$. In total the demand that this stack presses on the hole is $d \,\&\, d_x$ – and it must be the case that the multiplicity n on the stack be higher than the indirect multiplicity in $\mathsf{S}\,(m)$, plus *one*, for the immediate pressure on the top of the stack. This is in-line with our intuition that the only way we can exercise more pressure than just a linear $C^1(d)$ on a function is via the heap: in the continuation we could potentially be immediately calling the function but we might as well be calling it indirectly later on. Rule SUPDUPABS is of similar flavor, only simpler, since the indirect pressure on x is just A.

The SUPDUP and SUPDUPABS rules deal with demand on x being gathered up from the continuation of the execution, but rule SUPDDN is rather different: if x is bound with a transformer in P then we – in effect – treat it as if the expression bound by x is inlined so we only gather the φ from the continuation and check that the multiplicity of x is sufficient.

Rule SCASE is interesting, too. The stack has the shape of a `case` elimination branch. If there exists a demand d, such that the rhs e can be typed with it, giving $\langle \tau\,; \varphi_1 \rangle$ and the stack, when pressed with d, can give φ_2, then we can simply return $\varphi_1 \backslash_{x,y} \,\&\, \varphi_2$. In this case the demand pressed on the hole of the stack can be any $d_p \sqsubseteq U(\varphi_1(x), \varphi_1(y))$.

The heap typing judgement $P \vdash \mathsf{H} \sim \varphi$ ensures that the heap H has enough multiplicity to withstand the pressure that φ will exercise. Rules HPVARABS and HPEMPTY are boring. However HPVARUP ensures that if φ needs to press $m * d$ on x then (i) x must have enough multiplicity in the heap, but also that (ii) the expression or value bound by x can be checked at this demand yielding a new φ_1. Finally, (iii) the remaining heap must have enough multiplicity to withstand the newly unleashed demand from φ_1.

Rule HPVARDN is simpler: it checks that (i) the multiplicity of x in the heap is high enough, (ii) the transformer is well-formed for the bound expression, and (iii) the expression can indeed be typechecked in the demand that φ presses.

With these definitions in place we can prove the generalised safety statement, Lemma 4.2, which is needed for the proof of Theorem 4.1.

A.2 Soundness theorems

The partial order $\sqsubseteq$ and the least upper bound $\sqcup$ are defined for usage types naturally:

$$\tau_1 \sqsubseteq \tau_2 \quad \Leftrightarrow \quad (\tau_1 \sqcup \tau_2) = \tau_2$$

For usage environments φ the partial order is defined as a point-wise lifting of partial order on multi-demands in their codomains (assuming each φ is predetermined with A by default).

Lemma A.1 (*Monotonicity of usage typing*)
If the transformer environment P consists of monotone functions and $P \vdash \mathsf{e} \downarrow d \Rightarrow \langle \tau\,; \varphi \rangle$ and $d' \sqsubseteq d$ then $P \vdash \mathsf{e} \downarrow d' \Rightarrow \langle \tau'\,; \varphi' \rangle$ and $\tau' \sqsubseteq \tau$ and $\varphi \sqsubseteq \varphi'$.

Proof
The intuition is that if we use an expression "less" than how it was originally typed, then the annotations in it are still adequate, and we get smaller types and environments out.

The proof goes by induction on the typing derivation.

- Case TVARDN follows by monotonicity of the transformer and monotonicity of the operations on usage environments.

$$\boxed{P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d,\tau) \Rightarrow \varphi}$$

$$\frac{d \sqsubseteq HU}{P \vDash^{\mathsf{s}} \varepsilon \downarrow (d,\tau) \Rightarrow \varepsilon}\ \textsc{SHU}$$

$$\frac{\tau_h \preceq d_y^{\dagger} \to \tau \quad P \vDash^{*} y \downarrow d_y^{\dagger} \quad P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d,\tau) \Rightarrow \varphi}{P \vDash^{\mathsf{s}} (\bullet\, y) : \mathsf{S} \downarrow (C^1(d), \tau_h) \Rightarrow \varphi}\ \textsc{SArr}$$

$$\frac{\begin{array}{c} x \notin dom(P) \quad m+1 \leq n \\ P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d,\bullet) \Rightarrow \varphi \quad m * d_x = \varphi(x) \end{array}}{P \vDash^{\mathsf{s}} (\#(x,n) : \mathsf{S}) \downarrow (d \,\&\, d_x, \tau) \Rightarrow \varphi \backslash_x}\ \textsc{SUpdUp}$$

$$\frac{\begin{array}{c} x \notin dom(P) \quad 1 \leq n \\ P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d,\bullet) \Rightarrow \varphi \quad \varphi(x) = A \end{array}}{P \vDash^{\mathsf{s}} (\#(x,n) : \mathsf{S}) \downarrow (d, \tau) \Rightarrow \varphi \backslash_x}\ \textsc{SUpdUpAbs}$$

$$\frac{\begin{array}{c} (x{:}\rho) \in P \quad n \geq \mu(\varphi(x)) + 1 \\ P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d, T_\rho^d) \Rightarrow \varphi \end{array}}{P \vDash^{\mathsf{s}} (\#(x,n) : \mathsf{S}) \downarrow (d, \tau) \Rightarrow \varphi \backslash_x}\ \textsc{SUpdDn}$$

$$\frac{\begin{array}{c} d_p \sqsubseteq U(\varphi_1(x), \varphi_1(y)) \\ P \vdash \mathsf{e} \downarrow d \Rightarrow \langle \tau ; \varphi_1 \rangle \quad P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d,\tau) \Rightarrow \varphi_2 \end{array}}{P \vDash^{\mathsf{s}} ((x,y) \to \mathsf{e}) : \mathsf{S} \downarrow (d_p, \bullet) \Rightarrow \varphi_1 \backslash_{x,y} \,\&\, \varphi_2}\ \textsc{SCase}$$

$$\boxed{P \vdash \mathsf{H} \sim \varphi}$$

$$\frac{P \vdash \mathsf{H} \sim \varphi}{P \vdash \mathsf{H} \sim \varphi, (x{:}A)}\ \textsc{HpVarAbs} \qquad \frac{}{P \vdash \mathsf{H} \sim \varepsilon}\ \textsc{HpEmpty}$$

$$\frac{\begin{array}{c} n \geq m \quad x \notin dom(P) \\ P \vdash \mathsf{e}/\mathsf{v} \downarrow d \Rightarrow \langle \tau ; \varphi_1 \rangle \quad P \vdash \mathsf{H} \sim \varphi \,\&\, \varphi_1 \end{array}}{P \vdash \mathsf{H}, [x \overset{n}{\mapsto} \mathtt{Exp}(\mathsf{e})/\mathtt{Val}(\mathsf{v})] \sim \varphi, (x{:}m * d)}\ \textsc{HpVarUp}$$

$$\frac{\begin{array}{c} n \geq m \quad (x{:}\rho) \in P \quad P \vDash^{\mathsf{t}} \mathsf{e}/\mathsf{v} : \rho \\ P \vdash \mathsf{e}/\mathsf{v} \downarrow d \Rightarrow \langle \tau_1 ; \varphi_1 \rangle \quad P \vdash \mathsf{H} \sim \varphi \end{array}}{P \vdash \mathsf{H}, [x \overset{n}{\mapsto} \mathtt{Exp}(\mathsf{e})/\mathtt{Val}(\mathsf{v})] \sim \varphi, (x{:}m * d)}\ \textsc{HpVarDn}$$

Fig. 13: Stack and heap typing

- Case TVarUp is trivial.
- Case TLam is an easy application of the induction hypothesis, and then either TLam or TLamHU. Note that this relies on the non-deterministic choice of return type of TLamHU which lets us choose the same type as the TLam used for typing the λ-abstraction.

- Case TLAMHU is straightforward.
- Case TPAIR and TCASE are easy applications of the induction hypothesis.
- Case TLETDN follows by induction hypothesis for e_2, noting that ρ is monotone by the assumption $P \vdash^{\ddagger} \mathsf{e}_1 : \rho$.
- Case TLETUP follows by induction hypothesis and then applying either TLETUP or TLETABS.
- Case TLETABS follows by induction hypothesis and TLETUP.

□

Lemma A.2 (*Discrete usage signatures are well-formed*)
If

$$\mathsf{e} = \lambda^{n_1} x_1 \ldots \lambda^{n_k} x_k \,.\, \mathsf{e}_1,\ n_1, \ldots, n_k > 0 \tag{1}$$

$$P \vdash \mathsf{e}_1 \downarrow U \Rightarrow \langle \tau_1 \,;\, \varphi_1 \rangle \tag{2}$$

$$\langle \tau_0 \,;\, \varphi_0 \rangle = \langle \varphi_1(\overline{x}) \to \tau_1 \,;\, \varphi_1 \backslash_{\overline{x}} \rangle \tag{3}$$

$$\rho = \lambda\, d \,.\, \mathit{transform}(\langle k \,;\, \tau_0 \,;\, \varphi_0 \rangle, d) \tag{4}$$

then $P \vdash^{\ddagger} \mathsf{e} : \rho$.

Proof
By the typing rule WFTRANS, we need to show that

$$\forall d_1, d_2 \,.\, d_1 \sqsubseteq d_2 \Longrightarrow T_\rho^{d_1} \sqsubseteq T_\rho^{d_2} \wedge \Phi_\rho^{d_1} \sqsubseteq \Phi_\rho^{d_2} \tag{5}$$

$$\forall d, \varphi, \tau \,.\, (P \vdash \mathsf{e} \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle) \Longrightarrow \tau \sqsubseteq T_\rho^{d} \wedge \varphi \sqsubseteq \Phi_\rho^{d} \,. \tag{6}$$

The proof of (5) is straightforward, since ρ is a monotonic *step*-function.

For the second part, let us first define the threshold d_t as $d_t = C^1(\ldots k\text{-fold} \ldots C^1(U) \ldots)$, where k-fold stands for applying the constructor (C^1 in this case) k times. We remark that, by consecutive applications of rule TLAM, we can obtain:

$$P \vdash \mathsf{e} \downarrow d_t \Rightarrow \langle \tau_0 \,;\, \varphi_0 \rangle$$

Let us assume that $P \vdash \mathsf{e} \downarrow e \Rightarrow \langle \tau_d \,;\, \varphi_d \rangle$. We show that $\tau_d \sqsubseteq \tau_0$ and $\varphi_d \sqsubseteq \varphi_0$ by induction on the number of λs k.

- If $k = 0$ then it can only be that $d \sqsubseteq U$ and the result follows by monotonicity (Lemma A.1).
- If $k > 0$ then we have several cases on the shape of d.
  - — $d = U(d_1^\dagger, d_2^\dagger)$. This can *only* happen if $d \sqsubseteq HU$ and rule TLAMHU was used, otherwise the lambda is not typeable at all. But $HU \sqsubseteq d_t$ anyway so this case follows by monotonicity.
  - — $d = HU$. This is similar as above.
  - — $d = C^m(d_1)$. In this case we can invert the TLAM rule used to type $\mathsf{e} = \lambda^n \,.\, \mathsf{e}_b$ with $C^m(d_1)$, $n \geq m$, and apply the induction hypothesis for the body e_b. We get back a pair $\langle \tau_b \,;\, \varphi_b \rangle$. If $m = 1$ then we are easily done by the induction hypothesis. If $m = \omega$

then it is definitely the case that $d \not\sqsubseteq d_t$ and hence we multiply both components of $\langle \tau_b \,;\, \varphi_b \rangle$ by ω and we are done, using the induction hypothesis.[12]

— $d = U$. We observe that $d \sqsubseteq C^{\omega}(U)$ and hence the case follows as the previous one using inversion on TLAM.

□

Lemma A.3 (*Analysis produces well-typed terms* (***Lemma 4.1***))
If $P \blacktriangleright e \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle \rightsquigarrow \mathsf{e}$ then $P \vdash \mathsf{e} \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle$.

Proof
The proof is by induction on the height of the derivation $P \blacktriangleright e \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle \rightsquigarrow \mathsf{e}$. We abuse the notation, considering a demand signature environment P from the perspective of both *discrete* and *generalised* usage signatures. Obviously, any discrete signature $\rho = \langle k \,;\, \tau \,;\, \varphi \rangle$ can be considered as a generalised one, ρ, such that

$$\rho(d) \stackrel{\text{def}}{=} \mathit{transform}(\rho, d),$$

where $\mathit{transform}(\langle k \,;\, \tau \,;\, \varphi \rangle, d)$ is defined in Fig. 1.

- Case VARDN. Corresponds straightforwardly to the application of rule TVARDN, where $\rho(d) = \mathit{transform}(\rho, d)$.
- Case VARUP. Straightforward by the rule TVARUP.
- Case LAM. By the rule TLAM. By induction hypothesis, we have $P \vdash \mathsf{e} \downarrow d_e \Rightarrow \langle \tau \,;\, \varphi \rangle$. Moreover, by the formulation of LAM, $d = C(d_e)$ (exact equality) and $m = n$, so the premises of the rule TLAM are fulfilled.
- Case LAMU. Follows by rule TLAM observing that $U \sqsubseteq C^{\omega}(U)$.
- Case LAMHU. Straightforward by the rule TLAMHU.
- Case APPA. By induction hypothesis and a simple additional statement relating $\blacktriangleright^*$ and $\vdash^*$ (ensuring that variables transformed unde via $\blacktriangleright^*$ are well-typed under $\vdash^*$, the proof is by considering two trivial cases of the corresponding relation), we have

$$P \vdash^* y \downarrow d_2^{\dagger} \Rightarrow \varphi_2 \tag{7}$$

$$P \vdash \mathsf{e}_1 \downarrow C^1(d) \Rightarrow \langle d_2^{\dagger} \to \tau_r \,;\, \varphi_1 \rangle \tag{8}$$

  Now, let us just take $\tau_1 = d_2^{\dagger} \to \tau_r$, so the premises of the rule TAPP are fulfilled.
- Case APPB. By induction, we have

$$P \vdash^* y \downarrow \omega * U \Rightarrow \varphi_2 \tag{9}$$

$$P \vdash \mathsf{e}_1 \downarrow C^1(d) \Rightarrow \langle \omega * U \to \tau_r \,;\, \varphi_1 \rangle \tag{10}$$

  Moreover, by the definition of $\preceq$ (Fig. 1),

$$\bullet \preceq \omega * U \to \bullet,$$

  so we just take $\tau_1 = \bullet$, which fulfils the premise of the rule TAPP.
- Case PAIR. Straightforward by the typing rule TPAIR, taking $d = U(d_1^{\dagger}, d_2^{\dagger})$.

[12] Note that we can guarantee the same result by choosing a different more expressive transform that only infinitizes the *previous* types but not the current one, yielding tighter types, but we have not done that for simplicity.

- Case PAIRU. Straightforward by the typing rule TPAIR, observing that $U \sqsubseteq U(\omega * U, \omega * U)$.
- Case PAIRHU. By the typing rule TPAIR, taking $d = U(A, A)$. Both subderivations for the components of the pair are processed thus via the typing rule TABS, which gives empty environments (ε) in both cases. Finally, $\varepsilon \,\&\, \varepsilon = \varepsilon$, which concludes the proof for this case.
- Case CASE. By induction hypothesis,

$$P \vdash \mathsf{e}_r \downarrow d \Rightarrow \langle \tau \,;\, \varphi_r \rangle \tag{11}$$

$$P \vdash \mathsf{e}_s \downarrow U(\varphi_r(x), \varphi_r(y)) \Rightarrow \langle \_ \,;\, \varphi_s \rangle, \tag{12}$$

  so we can directly apply the typing rule TCASE.
- Case LETUP By induction, we have

$$P \mapsto \mathsf{e}_2 \downarrow d \Rightarrow \langle \tau \,;\, \varphi_2 \rangle \tag{13}$$

$$n * d_x = \varphi_2(x) \tag{14}$$

$$P \mapsto \mathsf{e}_1 \downarrow d_x \Rightarrow \langle \_ \,;\, \varphi_1 \rangle \tag{15}$$

  The proof for this case is completed by applying the typing rule TLETUP with $m = n$.
- Case LETUPABS Straightforward by the rule TLETUPABS.
- Case LETDN. In this case we have that:

$$\begin{aligned} P \mapsto \mathtt{let}\ x = \lambda \overline{y}^{1..k} \,.\, e_1\ \mathtt{in}\ e_2 \downarrow d \Rightarrow \langle \tau \,;\, (\varphi_2 \backslash_x) \rangle \\ \rightsquigarrow \mathtt{let}\ x \overset{n}{=} \lambda^{n_1} x_1 \ldots \lambda^{n_k} x_k \,.\, \mathsf{e}_1\ \mathtt{in}\ \mathsf{e}_2 \end{aligned}$$

  Let us call the resulting RHS term $\mathsf{e} = \lambda^{n_1} x_1 \ldots \lambda^{n_k} x_k \,.\, \mathsf{e}_1$. By inversion we have that:

$$\begin{gathered} P \mapsto e_1 \downarrow U \Rightarrow \langle \tau_1 \,;\, \varphi_1 \rangle \rightsquigarrow \mathsf{e}_1 \\ \tau_x = \varphi_1(\overline{y}) \to \tau_1 \\ P, x{:}\langle k \,;\, \tau_x \,;\, \varphi_1 \backslash_{\overline{y}} \rangle \mapsto e_2 \downarrow d \Rightarrow \langle \tau \,;\, \varphi_2 \rangle \rightsquigarrow \mathsf{e}_2 \\ \varphi_2(x) \sqsubseteq n * C^{n_1}(\ldots(C^{n_k}(\ldots)\ldots)) \end{gathered}$$

  Hence it is easy to show by induction and monotonicity that $P \vdash \mathsf{e} \downarrow (C^{n_1}(\ldots\ldots)) \Rightarrow \langle \_ \,;\, \_ \rangle$. We know that $n \geq \mu(\varphi_2(x))$. Moreover $P \vDash \mathsf{e} : \rho$ for the concrete transform used, by Lemma A.2. Finally the statement for the body follows by induction hypothesis. The case is finished by putting these all together and applying rule TLETDN.
- Case LETDNABS. Similar to the case LETDN.

□

Lemma A.4 (*Value splitting* (***Lemma 4.3***))
If $P \vdash \mathsf{v} \downarrow (d_1 \,\&\, d_2) \Rightarrow \langle \tau \,;\, \varphi \rangle$ then there exists a split $split(\mathsf{v}) = (\mathsf{v}_1, \mathsf{v}_2)$ such that: $P \vdash \mathsf{v}_1 \downarrow d_1 \Rightarrow \langle \tau_1 \,;\, \varphi_1 \rangle$ and $P \vdash \mathsf{v}_2 \downarrow d_2 \Rightarrow \langle \tau_2 \,;\, \varphi_2 \rangle$ and moreover $\tau_1 \sqsubseteq \tau$, $\tau_2 \sqsubseteq \tau$ and $\varphi_1 \,\&\, \varphi_2 \sqsubseteq \varphi$.

Proof
This is an extremely important property. It says that for a *value* (and *only* for values!) the unleashed environment is additive with respect to the placed demands. This allows one to use a variable directly (by dereferencing a variable and using it with a particular continuation) and indirectly in the continuation! Here is the proof, by case analysis on the shape of the value v:

- Case $\mathsf{v} = (x, y)$. In this case without loss of generality assume that: $d_1 = U(d_1^\dagger, d_2^\dagger)$ and $d_2 = U(d_3^\dagger, d_4^\dagger)$. If one of them is a call demand then their $\&$ is not defined, and if one of them is a naked U or HU then that is equivalent to some $U(d_1^\dagger, d_2^\dagger)$ in terms of how the result will be typed. The result then follows by monotonicity of the $\&$ operation and Lemma A.5 (see below).
- Case $\mathsf{v} = \lambda^n x\,.\,\mathsf{e}$. In this case, if one of d_1 or d_2 is less or equal to HU, assume d_1, then the split is by choosing $n_1 = 0$ and $n_2 = n$. The $n_1 = 0$ split uses the TLAMHU rule assigning the same type as the other split assigns. The other split merely uses the typing rule that was originally used to type v. If on the other hand no d_1 nor d_2 is less or equal to HU then they cannot be non-call-demands either (because their $\&$ would not be defined). Assume then without loss of generality that $d_1 = C^{n_1}(d_1')$ and $d_2 = C^{n_2}(d_2')$. (If one of them was U then we simply type it as $C^\omega(U)$). Let us use the split induced by d_1 and d_2, that is $n = n_1 + n_2$. From typing the body e with d_1 we will get $\langle \varphi_1'(x) \to \tau_1' \,;\, n_1 * \varphi_1' \rangle$ and similarly $\langle \varphi_2'(x) \to \tau_2' \,;\, n_2 * \varphi_2' \rangle$, where φ_i' and τ_i' are the results of typing e with d_i' respectively. However we know that the body is typeable with $d_1' \sqcup d_2'$ resulting in $\langle \varphi_\sqcup(x) \to \tau_\sqcup \,;\, (n_1 + n_2) * \varphi_\sqcup \rangle$ for v. By monotonicity we get that for $i \in \{1, 2\}$:

  $$\varphi_i'(x) \to \tau_i' \sqsubseteq \varphi_\sqcup(x) \to \tau_\sqcup$$

  as required. Moreover we need to show that

  $$n_1 * \varphi_1' \,\&\, n_2 * \varphi_2' \sqsubseteq (n_1 + n_2) * \varphi_\sqcup$$

  By monotonicity it suffices to show that:

  $$n_1 * \varphi_\sqcup \,\&\, n_2 * \varphi_\sqcup \sqsubseteq (n_1 + n_2) * \varphi_\sqcup$$

  and the result follows by the easy-to-show fact that $n_1 * d^\dagger \,\&\, n_2 * d^\dagger \sqsubseteq (n_1 + n_2) * d^\dagger$ for any $d^\dagger$.

$\square$

Lemma A.5 (*Variable demand splitting*)
Assume that the transformer environment P is monotone. If $P \vdash^* x \downarrow (d_1^\dagger \,\&\, d_2^\dagger) \Rightarrow \langle \tau \,;\, \varphi \rangle$ then $P \vdash^* x \downarrow d_1^\dagger \Rightarrow \langle \tau_1 \,;\, \varphi_1 \rangle$ and $P \vdash^* x \downarrow d_2^\dagger \Rightarrow \langle \tau_2 \,;\, \varphi_2 \rangle$ and $\varphi_1 \,\&\, \varphi_2 \sqsubseteq \varphi$.

Proof
If $x \notin dom(P)$ then the result is trivial. If $x \in dom(P)$ then there is a transformer $(x{:}\rho) \in P$. First of all let us examine the case where either $d_1^\dagger$ or $d_2^\dagger$ is A. Without loss of generality assume $d_1^\dagger = A$. In this case the result is trivial since $\varphi_2 = \varepsilon$ and $\varphi = \varphi_1$. Assume instead that $d_1^\dagger = n_1 * d_1$ and $d_2^\dagger = n_2 * d_2$. In this case it suffices to show that:

$$n_1 * \Phi_\rho^{d_1} \,\&\, n_2 * \Phi_\rho^{d_2} \sqsubseteq (n_1 + n_2) * \Phi_\rho^{d_1 \,\&\, d_2}$$

However by monotonicity we know that: $n_1 * \Phi_\rho^{d_1} \sqsubseteq n_1 * \Phi_\rho^{d_1 \,\&\, d_2}$ and similarly $n_2 * \Phi_\rho^{d_2} \sqsubseteq n_2 * \Phi_\rho^{d_1 \,\&\, d_2}$ so it suffices to show for every binding in $\Phi_\rho^{d_1 \,\&\, d_2}$, call it $(y{:}d^\dagger)$, that it is the case that:

$$n_1 * d^\dagger \,\&\, n_2 * d^\dagger \sqsubseteq (n_1 + n_2) * d^\dagger$$

This is easy to show using the fact that $\omega * d^\dagger = d^\dagger \,\&\, d^\dagger = d^\dagger \,\&\, \ldots \,\&\, d^\dagger \,\&\, d^\dagger$. $\square$

Lemma A.6 (*Single-step safety* (***Lemma 4.2***))
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 $\langle \mathsf{H}_1 ; \mathsf{e}_1 ; \mathsf{S}_1 \rangle \hookrightarrow \langle \mathsf{H}_2 ; \mathsf{e}_2 ; \mathsf{S}_2 \rangle$ such that $\mathsf{H}_2^\natural = H_2$, $\mathsf{e}_2^\natural = e$ and $\mathsf{S}_2^\natural = S_2$, and moreover $\vdash \langle \mathsf{H}_2 ; \mathsf{e}_2 ; \mathsf{S}_2 \rangle$.

Proof
By induction on the height of the derivation $\vdash \langle H ; \mathsf{e} ; S \rangle$. We proceed by case analysis on the rule used for $\longrightarrow$ in the *uninstrumented* semantics.

- Case ELET. We have three cases to consider, depending on whether rule TLETUP, TLETUPABS, or TLETDN is used.
  - — Case TLETUP. In this case we have that:

$$P \vdash \texttt{let}\ x \stackrel{m}{=} \mathsf{e}_1\ \texttt{in}\ \mathsf{e}_2 \downarrow d \Rightarrow \langle \tau ; \varphi_1 \,\&\, \varphi_2 \rangle \tag{16}$$
$$n \leq m \tag{17}$$
$$P \vdash \mathsf{e}_2 \downarrow d \Rightarrow \langle \tau ; \varphi_2, (x{:}n * d_1) \rangle \tag{18}$$
$$P \vdash \mathsf{e}_1 \downarrow d_1 \Rightarrow \langle \_ ; \varphi_1 \rangle \tag{19}$$

    Moreover:

$$P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d, \tau) \Rightarrow \varphi_S \tag{20}$$
$$P \vdash \mathsf{H} \sim \varphi_1 \,\&\, \varphi_2 \,\&\, \varphi_S \tag{21}$$

    The rule ELET fires in the instrumented semantics as well, giving us a new heap $\mathsf{H}, [x \stackrel{m}{\mapsto} \texttt{Exp}(\mathsf{e}_1)]$. By using HPVARUP we can conclude:

$$P \vdash \mathsf{H}, [x \stackrel{m}{\mapsto} \texttt{Exp}(e_1)] \sim \varphi_2, (x{:}n * d_1) \,\&\, \varphi_S \tag{22}$$

    from (17), (19), (21). Hence, from (18), (22), and (20) we conclude that the resulting configuration is well-typed.
  - — Case TLETUPABS. Similar but simpler than the case for TLETUP.
  - — Case TLETDN. In this case we have that:

$$P \vdash \texttt{let}\ x \stackrel{m}{=} \mathsf{e}_1\ \texttt{in}\ \mathsf{e}_2 \downarrow d \Rightarrow \langle \tau ; \varphi_2 \rangle \tag{23}$$
$$n \leq m \tag{24}$$
$$P \vdash \mathsf{e}_1 \downarrow d_1 \Rightarrow \langle \_ ; \varphi_1 \rangle \tag{25}$$
$$P \vDash^{\mathsf{t}} \mathsf{e}_1 : \rho \tag{26}$$
$$P, (x{:}\rho) \vdash \mathsf{e}_2 \downarrow d \Rightarrow \langle \tau ; \varphi_2, (x{:}d^\dagger) \rangle \tag{27}$$
$$d^\dagger \sqsubseteq n * d_1 \tag{28}$$

    Moreover:

$$P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d, \tau) \Rightarrow \varphi_S \tag{29}$$
$$P \vdash \mathsf{H} \sim \varphi_2 \,\&\, \varphi_S \tag{30}$$

    The rule ELET fires in the instrumented semantics as well, giving us a new heap $\mathsf{H}, [x \stackrel{m}{\mapsto} \texttt{Exp}(\mathsf{e}_1)]$. We need to use HPVARDN to deduce that:

$$P, (x{:}\rho) \vdash \mathsf{H}, [x \stackrel{m}{\mapsto} \texttt{Exp}(\mathsf{e}_1)] \sim \varphi_2, (x{:}d^\dagger) \,\&\, \varphi_S \tag{31}$$

which follows from (30), (26), (24), (25). Moreover, from (29) and the observation that $x \notin fv(S)$ it is easy to deduce that $P, (x{:}\rho) \models^{\mathsf{s}} \mathsf{S} \downarrow (d, \tau) \Rightarrow \varphi_S$ (simple inductive weaking proof). From this, and (27) and (31), we get that the resulting configuration is well typed.

- Case ELKPE. In this case we have two cases depending on how the variable was typed.
  - Case TVARUP. In this case we have:

$$P \vdash x \downarrow d \Rightarrow \langle \bullet\,; (x{:}1 * d) \rangle \tag{32}$$

where $x \notin dom(P)$. Moreover:

$$P \models^{\mathsf{s}} \mathsf{S} \downarrow (d, \bullet) \Rightarrow \varphi_S \tag{33}$$

$$P \vdash \mathsf{H}, [x \overset{n}{\mapsto} \mathtt{Exp(e)}] \sim (\varphi_S)\backslash_x, (x{:}1 * d \,\&\, \varphi_S(x)) \tag{34}$$

We have two cases: if $\varphi_S(x) = A$ then we only press $1 * d$ on x. If $\varphi_S(x) = m * d_x$ then we press $(1 + m) * (d \,\&\, d_x)$ on x. Let us consider the latter case first:

$$P \vdash \mathsf{H}, [x \overset{n}{\mapsto} \mathtt{Exp(e)}] \sim (\varphi_S)\backslash_x, (x{:}(1 + m) * (d \,\&\, d_x)) \tag{35}$$

By inverting HPLETUP, it must be that:

$$n \geq m + 1 \tag{36}$$

$$P \vdash \mathsf{H} \sim (\varphi_S)\backslash_x \,\&\, \varphi_e \tag{37}$$

$$P \vdash e_1 \downarrow (d \,\&\, d_x) \Rightarrow \langle \tau_1\,; \varphi_e \rangle \tag{38}$$

To finish the case by SUPDUP we need to show that:

$$P \models^{\mathsf{s}} (\#(x, n) : \mathsf{S}) \downarrow (d \,\&\, d_x, \tau_1) \Rightarrow \varphi_S\backslash_x$$

which will be the case if we show that:

$$P \models^{\mathsf{s}} \mathsf{S} \downarrow (d, \bullet) \Rightarrow (\varphi_S)\backslash_x, (x{:}m * d_x)$$

and also: $n \geq 1 + m$. The first is exactly (33) and the second is just (36).
If it was the case that $\varphi_S(x) = A$ then we could similarly use SUPDUPABS.

  - Case TVARDN. In this case we have that

$$P \vdash x \downarrow d \Rightarrow \langle T^d_\rho\,; \Phi^d_\rho \,\&\, (x{:}1 * d) \rangle \tag{39}$$

Let us assume that bindings are not recursive so $x \notin dom(\Phi^d_\rho)$. Moreover:

$$P \models^{\mathsf{s}} \mathsf{S} \downarrow (d, T^d_\rho) \Rightarrow \varphi_S \tag{40}$$

$$P \vdash \mathsf{H}, [x \overset{n}{\mapsto} \mathtt{Exp(e)}] \sim \Phi^d_\rho \,\&\, (\varphi_S)\backslash_x, (x{:}1 * d \,\&\, \varphi_S(x)) \tag{41}$$

Let us assume that $\varphi_S(x) = m * d_x$ (the case where $\varphi_S(x) = A$ is easier). By rule HPLETDN this also means that:

$$P \vdash \mathsf{H} \sim \Phi^d_\rho \,\&\, (\varphi_S)\backslash_x$$

and moreover $P \vdash \mathsf{e} \downarrow (d \,\&\, d_x) \Rightarrow \langle \tau_e\,; \varphi_e \rangle$ – hence by monotonicity it is also the case that $P \vdash \mathsf{e} \downarrow d \Rightarrow \langle \tau_d\,; \varphi_d \rangle$, and in fact we also have $\varphi_d \sqsubseteq \Phi^d_\rho$ and $\tau_d \sqsubseteq T^d_\rho$.

Now for the right hand side the environment from the expression is φ_d (we press the demand d). The environment for the stack-typing is the one we get from: $P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d, T^d_\rho) \Rightarrow \Phi_S \backslash_x$. Hence we need to show that: $P \vdash \mathsf{H} \sim \varphi_e \,\&\, (\varphi_S)\backslash_x$ and the result follows from monotonicity.

- Case ELKPV. Again we have two cases depending on how the variable is typed.
  - — Case TVARUP. In this case we have

$$P \vdash x \downarrow d \Rightarrow \langle \bullet\,; (x{:}1 * d)\rangle \tag{42}$$

where $x \notin dom(P)$. Moreover:

$$P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d, \bullet) \Rightarrow \varphi_S \tag{43}$$

$$P \vdash \mathsf{H}, [x \overset{n}{\mapsto} \mathtt{Val}(\mathsf{v})] \sim (\varphi_S)\backslash_x, (x{:}1 * d \,\&\, \varphi_S(x)) \tag{44}$$

Again we have two cases depending on $\varphi_S(x)$.

  - – Case $\varphi_S(x) = m * d_x$. We know that $n \geq 1 + m$ and hence the expression can take a step in the counting semantics. From (44) we get that:

$$P \vdash \mathsf{H} \sim (\varphi_S)\backslash_x \,\&\, \varphi_v \tag{45}$$

where $P \vdash \mathsf{v} \downarrow (d \,\&\, d_x) \Rightarrow \langle \tau\,; \varphi_v\rangle$.
By Lemma A.4 we get that: $P \vdash \mathsf{v}_1 \downarrow d \Rightarrow \langle \tau_1\,; \varphi_1\rangle$ and $P \vdash \mathsf{v}_2 \downarrow d_x \Rightarrow \langle \tau_2\,; \varphi_2\rangle$ such that $\varphi_1 \,\&\, \varphi_2 \sqsubseteq \varphi_v$, $\tau_1 \sqsubseteq \tau$ and $\tau_2 \sqsubseteq \tau$ for some v_1 and v_2 with $split(\mathsf{v}) = (\mathsf{v}_1, \mathsf{v}_2)$.
To finish the case we need to show that:

$$H \sim \varphi_1 \,\&\, \varphi_2 \,\&\, (\varphi_S)\backslash_x$$

which follows from (45) and strengthening (Lemma A.7).

  - – Case $\varphi_S(x) = A$. This case is easy as it induces a trivial split for v_1 and v_2 where v_1 gets a 0 counter if it is a lambda. This reflects the fact that this is never used indirectly in the continuation but only directly in the stack S.
  - — Case TVARDN. In this case we have

$$P \vdash x \downarrow d \Rightarrow \langle T^d_\rho\,; \Phi^d_\rho \,\&\, (x{:}1 * d)\rangle \tag{46}$$

where $(x{:}\rho) \in P$. Moreover:

$$P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d, T^d_\rho) \Rightarrow \varphi_S \tag{47}$$

and

$$P \vdash \mathsf{H}, [x \overset{n}{\mapsto} \mathtt{Val}(\mathsf{v})] \sim \\ \Phi^d_\rho \,\&\, (\varphi_S)\backslash_x, (x{:}1 * d \,\&\, \varphi_S(x))$$

Let us deal with the case when $\varphi_S(x) = m * d_x$ (the case where $\varphi_S(x) = A$ is easier). By inverting rule HPLETDN we get:

$$P \vdash \mathsf{H} \sim (\varphi_S)\backslash_x \,\&\, \Phi^d_\rho \tag{48}$$

and moreover $P \vdash \mathsf{v} \downarrow (d \,\&\, d_x) \Rightarrow \langle \_\,; \varphi_v\rangle$ (i.e. the v is sufficiently annotated). So the configuration can indeed step and for the right-hand side, by Lemma A.4, we must have $P \vdash \mathsf{v}_2 \downarrow d \Rightarrow \langle \tau_2\,; \varphi_2\rangle$ and we must also have $P \vDash^{\mathsf{s}} \mathsf{S} \downarrow (d, \tau_2) \Rightarrow \varphi'_S$. By

the well-formedness of the transformer it must be that $\tau_2 \sqsubseteq T^d_\rho$ and it must also be $\varphi_2 \sqsubseteq \Phi^d_\rho$. Hence by monotonicity $\varphi'_S \sqsubseteq \varphi_S$ as well. To finish the case we need to show that:

$$P \vdash \mathsf{H}, [x \overset{n'}{\mapsto} \mathtt{Val}(\mathsf{v}_1)] \sim (\varphi'_S)\backslash_x \,\&\, \varphi_2, (x{:}\varphi'_S(x))$$

By rule HPLETDN it suffices to show two things: First, that $P \vdash \mathsf{H} \sim (\varphi'_S)\backslash_x \,\&\, \varphi_2$ – this follows by (48) and monotonicity. Second, that if $\varphi'_S(x) = d^\dagger$, v_1 is still typeable under that $d^\dagger$. However by the splitting lemma A.4 we know that: $P \vdash \mathsf{v}_1 \downarrow d_x \Rightarrow \langle \_; \varphi'_1 \rangle$ and the result follows by monotonicity since it must be the case that $d^\dagger \sqsubseteq m * d_x$.

- Case EUPD. Similar to ELKPV case.
- Case EBETA. Using the substitution lemma (Lemma A.8).
- Case EAPP. Trivial.
- Case EPAIR. Trivial.
- Case EPRED. Using the substitution lemma (Lemma A.8).

□

Lemma A.7 (*Heap-typing strengthening*)
If $P \vdash \mathsf{H} \sim \varphi_1$ and $\varphi_2 \sqsubseteq \varphi_1$ then $P \vdash \mathsf{H} \sim \varphi_2$.

Proof
Easy induction, appealing to the monotonicity of the typing Lemma A.1. □

Lemma A.8 (*Substitution*)
Assume that P is monotone and $P \vdash \mathsf{e} \downarrow d \Rightarrow \langle \tau ; \varphi_1 \rangle$ and $x \notin dom(P)$. If $P \vdash^* y \downarrow \varphi_1(x) \Rightarrow \varphi_2$ then $P \vdash \mathsf{e}[y/x] \downarrow d \Rightarrow \langle \tau_e ; \varphi_e \rangle$ such that $\varphi_e \sqsubseteq \varphi_1\backslash_x \,\&\, \varphi_2$ and $\tau_e \sqsubseteq \tau$.

Proof
By induction on the derivation $P \vdash e \downarrow d \Rightarrow \langle \tau ; \varphi_1 \rangle$. First of all, if $y \notin dom(P)$, then the result follows easily by a renaming. So we will only be concerned with the case when $y \in dom(P)$, in particular $(y{:}\rho) \in P$.

- Case TVARDN. In this case we know that the variable we exercise pressure on is *not* x and therefore the result follows trivially (y is absent).
- Case TVARUP. If the variable is not x then the result follows trivially (y is absent). If it is x then we have that the pressure on x is $(x{:}1 * d)$. Then $\varphi_2 = \Phi^d_\rho \,\&\, (y{:}1 * d)$. For the substituted expression we get that: $\varphi_e = \varphi_2$ and $\tau_e = T^d_\rho$. Clearly $T^d_\rho \sqsubseteq \bullet$ and moreover $\varphi_2 \sqsubseteq 1 * \varphi_2$ as required.
- The rest of the cases are straightforward but somewhat tedious applications of the induction hypothesis and monotonicity of typing. They rely on the following property: If $\varphi_1(x) = n_1 * d_1$ and $\varphi_2(x) = n_2 * d_2$ then

$$n_1 * \Phi^{d_1}_\rho \,\&\, n_2 * \Phi^{d_2}_\rho \sqsubseteq (n_1 + n_2) * \Phi^{d_1+d_2}_\rho$$

which follows by the monotonicity of the transformer ρ.

□