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

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

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

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

\[
\begin{align*}
wurble1, wurble2 & : (\text{Int} \to \text{Int}) \to \text{Int} \\
wurble1 \ k & = \text{sum} \ (\text{map} \ k \ [1..10]) \\
wurble2 \ k & = 2 * k \ 0 \\
f1 & : [\text{Int}] \to \text{Int} \\
f1 \ xs & = \text{let} \ ys = \text{map} \ \text{costly} \ xs \\
& \quad \text{in} \ wurble \ (\text{\textbackslash}n. \ \text{sum} \ (\text{map} \ (+ \ n) \ ys)) \\
f2 & : [\text{Int}] \to \text{Int} \\
f2 \ xs & = \text{let} \ ys = \text{map} \ \text{costly} \ xs \\
& \quad \text{in} \ wurble \ (\text{\textbackslash}n. \ \text{sum} \ (\text{map} \ (+ \ n) \ (\text{map} \ \text{costly} \ xs)))
\end{align*}
\]

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:

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

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

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

The reverse is also true. If the programmer writes f2 in the first place, the full laziness transformation (Peyton Jones et al. 1996) will float the sub-expression (map costly xs) out of the \

expression, so that it can be shared. That would be good for wurble1 but bad for wurble2.

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

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

We discuss related work in Section 8. Distinctive features of our work are (a) the notion of call demands, (b) a full implementation measured against a state of the art optimising compiler, and (c) the combination of simplicity with worthwhile performance improvements.

2. What is Cardinality Analysis?

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

- How many times is a particular, syntactic lambda-expression called (Section 2.1)?
- Which components of a data structure are never evaluated; that is, are absent (Section 2.2)?
- How many times is a particular, syntactic thunk evaluated (Section 2.4)?

2.1 Call cardinality

We saw in the introduction an example where it is helpful to know when a function calls its argument at most once. A lambda that is called at most once is called a one-shot lambda, and they are extremely common in functional programming: for example a continuation is usually one-shot. So cardinality analysis can be a big win when optimising continuation-heavy programs.

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

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

You can see that build calls its argument exactly once, and inlining ys in calls like (build (\cn \ldots yzs)) turns out to be crucial to making short-cut deforestation work in practice. Gill devotes a section of his thesis to elucidating this point (Gill 1996 Chapter 4.3). Gill lacked an analysis for one-shot lambdas, so his implementation (still extant in GHC today) relies on a gross hack: he taught GHC’s optimiser to behave specially for build itself, and a couple of other functions. No user-defined function will have this good behaviour. Our analysis subsumes the hack, by providing an analysis that deduces the correct one-shot information for build, as well as many other functions.

2.2 Currying

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

\[
f3 \ a = \text{wurble a } (\lambda x. \text{let t = costly x in } y. \ t+y)
\]

\[
wurble1 \ a \ g = g \ 2 \ a \ + \ g \ 3 \ a
\]

\[
wurble2 \ a \ g = \text{sum } (\text{map } (g \ a) [1\ldots1000])
\]

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

\[
f4 \ a = \text{wurble1 a } (\lambda x. y. \ \text{costly x + y})
\]

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

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

\[
wurble1 :: U \rightarrow C(C'(U)) \rightarrow \bullet
\]

\[
wurble2 :: U \rightarrow C(C'(U)) \rightarrow \bullet
\]

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

2.3 Absence

Consider this function

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

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

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

\[/fw p q = <\text{cbody}>
\]

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

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

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

\[/fw p q = let x = (p,q) in <\text{tfbody}>
\]

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

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

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

\[/fw p = let x = (p, error "urk") in <\text{tfbody}>
\]

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

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

1 We will always use “called” to mean “applied to one argument”. 

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

### 2.4 Thunk cardinality

Consider these definitions:

$$
f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
$$

$$
f x c = \begin{cases} 
  & \text{if } x > 0 \text{ then } c + 1 \text{ else} \\
  & \text{if } x = 0 \text{ then } 0 \text{ else } c - 1
\end{cases}
$$

$$
g y = f y \text{ (costly } y)$$

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

$$
f :: \omega \ast U \rightarrow 1 + U \rightarrow \bullet
$$

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:

$$
f1 \ g = g \ '\text{seq}' \ 1 \quad -- \quad f1 :: \ast U \rightarrow \bullet
$$

$$
f2 \ g = g \ '\text{seq}' \ g \ 2 \quad -- \quad f2 :: \omega \ast C^1(U) \rightarrow \bullet
$$

$$
f3 \ g = g \ 3 \quad -- \quad f3 :: \ast C^1(U) \rightarrow \bullet
$$

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

$$
f \ (\lambda x. \ x + y)
$$

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

### 3. Formalising Cardinality Analysis

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

#### 3.1 Usage demands

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

- $U(d_1, d_2)$ applies to pairs. The pair itself is evaluated and its first component is used as described by $d_1$ and its second by $d_2$.
- $C^n(d)$ applies to functions. The function is called at most $n$ times, and on each call the result is used as described by $d$.
- $\omega \ast U$, 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 :: $U \rightarrow \bullet$.

#### Usage types

- $\tau ::= \bullet \mid d_1 \rightarrow \tau$

#### Usage type expansion

- $d_1 \rightarrow \tau \leq \bullet \leq \omega \ast U \rightarrow \tau$

#### Free-variable usage environments (fv-usage)

- $\varphi ::= (x : d_1) \mid \varphi \mid \epsilon$

#### Auxiliary notation on environments

- $\varphi(x) = d_1$ when $(x : d_1) \in \varphi$

#### Usage signatures and signature environments

- $\rho ::= (k : \tau ; \varphi) \quad k \in \mathbb{Z}_{\geq 0}$
- $P ::= (x : \rho) \mid P \mid \epsilon$

#### Transform

- $\text{transform}((k : \tau ; \varphi), d) = (\tau ; \varphi)$ if $d \subseteq C^1(\ldots \ k \text{-fold} \ldots C^1(U))$
- $= (\omega \ast \tau ; \omega \ast \varphi)$ otherwise

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

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

A usage demand $d$ always uses the root of the value exactly once; it cannot express absence or multiple evaluation. That is done by $d_1$, which is either $A$ (absent), or $n \ast d$ indicating that the value is used at most $n$ times in a way described by $d$. In both $C^n(d)$ and $n \ast d$, the multiplicity $n$ is either $1$ or $\omega$ (meaning “many”). Notice that a call demand $C^n(d)$ has a $d$ inside it, not a $d_1$; if a function is called, its body is evaluated exactly once. This is different from pairs. For example, if we have

- $\text{let } x = (e_1, e_2) \text{ in } \text{fst} \ x + \text{fst} \ x$

then $e_1$ is evaluated twice.

Both $U$ and $HU$ come with some non-syntactic equalities, denoted by $\equiv$ in Figure 1 and necessary for the proof of well-typedness (Section 4). For example, $U$ is equivalent to a pair demand whose components are used many times, or a many-call-demand where the result is used in an arbitrary way. Similarly, for pairs $HU$ is
The analysis itself is shown in Figures 3 and 4. The main judgement

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

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

- \( \mu(\omega) = m \)
- \( \mu(A) = 0 \quad \mu(n \bullet d) = n \)
- \( d_1 \downarrow d_2 \Rightarrow d_1 \downarrow d_2 = d_3 \)
- \( d_1 \& U = U \quad U \& d = U \quad d \& \text{HU} = d \quad \text{HU} \& d = d \)
- \( C^n(d_1) \& C^n(d_2) = C^n(d_1 \cup d_2) \)
- \( U(d_1 \downarrow d_2 \cup U(d_1, d_2) = U(d_1 \downarrow d_1 \cup d_2 \cup d_1) \)
- \( \varphi_1 \& \varphi_2 = \varphi_3 \quad \varphi_1 \cup \varphi_2 = \varphi_3 \)
- \( (d_1 \downarrow d_2) \cup (d_1 \downarrow d_2) \Rightarrow (\tau_1 \& \tau_2) \)
- \( n \bullet d = \{x : n \bullet d(x) = d\} \)
- \( n \bullet \varphi = \{x : n \bullet \varphi(x) = d\} \)
- \( \text{PAIR} \quad \text{PAIRHU} \quad \text{CASE} \)
- \( \text{APP} \quad \text{APPB} \quad \text{MULTI} \)

Figure 3: Algorithmic cardinality analysis specification, part 1.
The signature environment $P$ maps some of free variables of $e$ to their usage signatures, $\rho$ (Section 3.5). Any free variables outside the domain of $P$ have an uninformative signature.

The usage demand, $d$, describes the degree to which $e$ is evaluated, including how many times its sub-components are evaluated or called.

Using $P$, the judgement transforms the incoming demand $d$ into the demands $(\tau;\phi)$ that $e$ places on its arguments and free variables respectively:

- The usage that $e$ places on its argument is given by $\tau$, which gives a demand $d^1$ for each argument.
- The usage that $e$ places on its free variables is given by its free-variable usage (fv-usage), $\phi$, 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

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

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

$$\vdash e : C^i(U) \Rightarrow (1 \times U(\omega \times U, A) \Rightarrow \cdot ; \{f \mapsto 1 \times C^i(U)) \}

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

In short, we think of the analysis as describing a demand transformer, transforming a demand on the result of $e$ into demands on its arguments and free variables.

### 3.3 Pairs and case expressions

With these definitions in mind, we can look at some of the analysis rules in Figure 3. Rule PAIR explains how to analyse a pair under a demand $U(d^1_k, d^2_k)$. We simply analyse the two components, under $d^1_k$ or $d^2_k$ respectively, and combine the results with “$\&$”. The auxiliary judgement $\triangleright$ (Figure 3) deals with the multiplicity of the argument demands $d^1_k$.

The “$\triangleright$” operator, pronounced “both”, is defined in Figure 2 and combines the free-variable usages $\phi_1$ and $\phi_2$. For the most part the definition is straightforward, but there is a very important wrinkle for call demands:

$$C^{\omega_1}(d^1_k) \& C^{\omega_2}(d^2_k) = C^{\omega}(d^1_k \sqcup d^2_k)

The “$\sqcup$" part is easy, since $n_1$ and $n_2$ are both at least 1. But note the switch from $\land$ 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 $1 \times C^i(U)$, and if we use $\&$ to combine that demand with itself we get $\omega \times C^i(U)$). The inner “$1$” is a consequence of the switch to $\sqcup$, and rightly expresses the fact that no partial application of $f$ is called more than once.

The other rules for pairs PAIRU, PAIRHU, and case expressions CASE should now be readily comprehensible ($\phi_{r^{1/2}}$ stands for the removal of $(x, y)$ from the domain of $\phi_r$).

### 3.4 Lambda and application

Rule LAM for lambdas expects the incoming demand to be a call demand $C^{\omega}(d^1_k)$. Then it analyses the body $e$ with demand $d^1_k$ to give $(\tau;\phi)$. If $n = 1$ the lambda is called at most once, so we can return $(\tau;\phi)$; 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 $n \times \phi$ operation on the bottom line accounts for this multiplicity, and is defined in Figure 2. Rule LAMHU 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 $\tau$ can be any type, since the abstraction is not going to be applied, but is only head-used. Dually, given an application $(e \ y)$, rule APPU analyses $e$ with demand $C^i(d)$, reflecting that $e$ is here called once. This returns the demand $(d^1_k \to \tau_1; \phi_1)$ on the context. Then we can analyse the argument under demand $d^1_k$, using $\triangleright^*$, yielding $\phi_1$; and combine $\phi_1$ and $\phi_2$. Rule APP applies when analysing $e_1$ yields the less-informative usage type $\bullet$.

### 3.5 Usage signatures

Suppose we have the term

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

We would like to determine the correct demands on $p$ and $q$, namely $\lambda (\text{SG}(U))$ and $\lambda (\text{AR})$ respectively. The gold standard would be to analyse $f$’s right-hand side at each 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 fresh at each call site; and it does not work at all for recursive functions. Instead, we want to analyse $f$, summarise its behaviour, and then use that summary at each call site. This summary is called $f$’s usage signature. Remember that the main judgement describes how a term transforms a demand for the value into demands on its context. So a usage signature must be a (conservative approximation of this) demand transformer.

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

- Look at $f$’s right hand side $\lambda y_1 \ldots y_k. e_1$, where $e_1$ is not a lambda-expression.
- Analyse $e_1$ in demand $U$, giving $(\tau_1; \phi_1)$.
- Record the triple $(k; \phi(\pi) \to \tau_1; \phi_1)$ 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 transform($\rho$, $d$) to use the recorded usage signature $\rho$ to transform the demand $d$ for this occurrence of $f$.

What does transform($k; \tau; \phi$, $d$) do (Figure 5)? If the demand $d$ on $f$ is stronger than $C^i(\ldots C^i(U))$, where the call demands are nested $k$ deep, we can safely unleash $(\tau; \phi)$ at the call site. If not, we simply treat the function as if it were called many times, by unleashing $(\omega \times \tau; \omega \times \phi)$, multiplying both the demand type $\tau$ and the usage environment $\phi$ (Figure 2). Rule LETDNABS handles the case when the variable is not used in the body.

### 3.6 Thunks

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

```plaintext
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

```plaintext
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:
The body of the `let` places usage demand \( 1 \vdash U(A, A) \) on \( x \), and if we analysed \( x \)'s right-hand side in that demand we would see that \( q \) was unused. So we get more information if we wait until we know the aggregated demand on \( x \), and use it to analyse its right-hand side.

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

### 3.7 Elaboration

How are we to take advantage of our analysis? We do so by elaborating the term during analysis, with annotations of two kinds (see the syntax of annotated expressions in Figure 4):

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

Figure 4 shows the elaborated terms after the `"↓"`. The operational semantics (Section 3.5) gets stuck if we use a thunk or lambda more often than its claimed usage; and the optimising transformations (Section 3.6) are guided by the same annotations.

### 3.8 A more realistic language

The implementation of Figure 1 is stripped to its bare essentials. Our implementation handles all of Haskell, or rather the Core language to which Haskell is translated by GHC. In particular:

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

### 4. Soundness of the Analysis

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

- We present an operational semantics, written \( \rightarrow \), for the annotated language that counts how many times thunks have been evaluated and \( \lambda \)-abstractions have been applied. The semantics simply gets stuck when these counters reach zero, which will happen only if the claims of the analysis are false (Section 4.1).
- Our goal is to prove that if an expression \( e \) is elaborated to \( e' \) by the analysis, then \( e' \) in the instrumented semantics behaves identically to \( e \) in a standard un-instrumented call-by-need semantics (Section 4.2). For reasons of space we omit the rules for the un-instrumented call-by-need semantics which are completely standard (Sestoft 1997), and are identical to the rules of Figure 5 if one simply ignores all the annotations and the multiplicity side-conditions. We refer to this semantics as \( \rightarrow^\ast \).
- We prove soundness by giving a type system for the annotated terms, and showing that for well-typed terms, the instrumented semantics \( \rightarrow^\ast \) simulates \( \rightarrow \), in a type-preserving way.

### 4.1 Counting operational semantics

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

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

Rule `ELKPV` de-references a binding for an already-evaluated expression \( [x \mapsto \text{Val}(v)] \), 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-referencing of $x$. We express this by non-deterministically splitting the value $v$, returning two values $v_1$ and $v_2$ whose top-level $\lambda$-annotations sum up to the original (see split in Figure 5). Our proof needs only ensure that among the non-deterministic choices there exists a choice that simulates $\Rightarrow$. Rule EUPO 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 $\Rightarrow$, then execution of $e$ does not get stuck in the instrumented semantics $\Rightarrow$. To do this we need to prove preservation and progress lemmas, showing that each step takes a well-annotated term to a well-annotated term, and that well-annotated terms do not get stuck.

Figure 5 says what it means to be “well-annotated”, using notation from Figures 1 and 2. The rules look very similar to the analysis rules of Figures 5 and 6, except that we check an annotated term, rather than producing one. For example, rule TLAM checks that the annotation on a $\lambda$-abstraction $(m)$ is at least as large as the call cardinality we press on this $\lambda$-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 $\rho$ as the analysis does (Figure 4), the environment $P$ holds generalised demand transformers $\rho$. A generalised demand transformer is simply a monotone function from a demand to a pair $(\tau; \phi)$ of a type and a usage environment (Figure 4). In the TLETDN rule, we clairvoyantly choose any such transformer $\rho$, which is sound for the RHS expression – denoted with $P \vdash e \vdash \rho$. 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 $(\phi_2(x))$. In rule TVARDN we simply apply the transformer $\rho$ to get a type and fv-usage environment.

Rule WFTTRAN 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 intentional representation used in the analysis satisfies both properties for the $\lambda$-expressions bound with LETDN.

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

### 4.3 Soundness of the analysis

The first result is almost trivial.

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

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

**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 $e^e = e$ to mean that the erasure of all annotations from $e$ is $e$, and we define $S^e = S$ and $H^e = H$ analogously.

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

**Theorem 4.2** (Safety for annotated terms). If $e \vdash e_1 \downarrow HU \Rightarrow (\tau; e)$ and $e_1 = e_2^e$ and $e \vdash e_1 \Rightarrow (\tau; e_2^e)$ then there exist $e_2$ and $S$, such that $e \vdash e_1 \Rightarrow (\tau; e_2)$ and $H^e = H$, $S^e = S$ and $e_2^e = e_2$.

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 $\vdash (H; e; S)$, that extends the well-annotation invariant of Figure 6 to configurations.

For reasons of space, we only give the statement of the theorem below, and defer the details of the well-annotation relation to Appendix A.

**Lemma 4.3** (Single-step safety). Assume that $\vdash (H_1; e_1; S_1)$. If $\langle H_1; e_1^e; S_1^e \rangle \Rightarrow (H_2; e_2; S_2)$ in the un-instrumented semantics, then there exist $H_2$, $e_2$ and $S_2$, such that $\vdash (H_1; e_1; S_1) \Rightarrow (H_2; e_2; S_2)$. $H_2^e = H_2$, $e_2^e = e$ and $S_2^e = S_2$, and moreover $\vdash (H_2; e_2; S_2)$.

Notice that the counting semantics is non-deterministic, so Lemma 4.3 simply ensures that there exists a possible transition in the counting semantics that always results in a well-typed configuration. Lemma 4.3 crucially relies on yet another property, below.
The demand exercised on \( x \) from the body of the \( \text{let} \)-binding will be \( C^1(U) \land C^1(U) = C^2(U) \) and hence the value \( v \) will be checked against this demand (using the \( \text{LETup} \) rule), unleashing an environment \( \phi \). 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 \( \phi_1 \) and \( \phi_2 \) in each call site. Lemma \( \ref{lemma:split} \) 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 \( \ref{lemma:split} \) that requires demand transformers to be monotone in the demand arguments, ensured by \( \text{WfTrans} \).

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

The proof is just a combination of Lemma \( \ref{lemma:split} \) and Theorem \( \ref{thm:optimisation} \).

**5. Optimisations**

We discuss next the two optimisations enabled by our analysis.

**5.1 Optimised allocation for thunks**

We show here that for 0-annotated bindings there is no need to allocate an entry in the heap, and for 1-annotated ones we don’t have to emit an update frame on the stack. Within the chosen operational model, this optimisation is of dynamic flavour so we express this by providing a new, optimising small-step machine for the annotated expressions. The new semantics is defined in Figure \( \ref{fig:trans} \). We will show that programs that can be evaluated via the counting semantics (Figure \( \ref{fig:trans} \)) 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 ∞-relations). We write $e_1 \equiv e_2$ iff $e_1$ and $e_2$ differ only on the $\lambda$-annotations. $H_1 \prec H_2$ and $S_1 \prec S_2$ are defined in Figure 7.

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

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

The relation between stacks is given with $S \sim S_{opt}$. Rule SSIM2 ensures that there are no frames $\#(x, 0)$ in the optimised stack. In fact during evaluation it is easy to observe that there are not going to be any update frames $\#(x, 0)$ in the original or optimised stack.

We can now state the optimisation simulation theorem.

**Theorem 5.1** (Optimised semantics). If \( \langle H_1; e_1; S_1 \rangle \prec \langle H_2; e_2; S_2 \rangle \) and \( \langle H_1; e_1; S_1 \rangle \rightarrow^* \langle H'_1; e'_1; S'_1 \rangle \) then there exists $k \in \{0, 1\}$ s.t. \( \langle H_2; e_2; S_2 \rangle \rightarrow^k \langle H'_2; e'_2; S'_2 \rangle \) and \( \langle H'_1; e'_1; S'_1 \rangle \prec \langle H'_2; e'_2; S'_2 \rangle \).

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

**Theorem 5.2** (Analysis is safe for optimised semantics). If $\star e_1 \downarrow$ \( HU \Rightarrow \langle \tau; e \rangle \leftarrow e_1 \) and $(\epsilon; e_1; e) \rightarrow^n \langle H; e_2; S \rangle$ then $(\epsilon; e_1; e) \rightarrow^m \langle H; e_2; S \rangle$, $n \leq m$, and there exist $H_2$ and $S_2$ s.t. \( H_2 \rightarrow^* H \) and \( S_2 \rightarrow^* S \).

Theorem 5.2 says that if a program $e_1$ evaluates in $n$ steps to $e_2$ in the reference semantics, then it also evaluates to the same $e_2$ (modulo annotation) in the optimised semantics in $n$ steps or fewer; and the heaps and stacks are consistent. Moreover, the theorem has informative content on infinite sequences. For example it says that for any point in the evaluation in the reference configuration we will have earlier reached a corresponding intermediate configuration in the optimised semantics with consistent heaps and stacks.

### 5.2 Let-in floating into one-shot lambdas

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

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

**Definition 5.2** (let-in floating for one-shot lambdas).

\[
\begin{align*}
\text{let } z \overset{m}{=} e_1 \text{ in } (\text{let } f \overset{n}{=} \lambda \cdot x . e_2) & \quad \Rightarrow \quad \text{let } f \overset{n}{=} \lambda \cdot x . (\text{let } z \overset{m}{=} e_1 \text{ in } e_2),
\end{align*}
\]

for any $m$, $n$ and $z \not\in FV(e_2)$.

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

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

\[
\begin{align*}
\langle H; e; S \rangle \downarrow^N & \overset{N}{=} \exists H', v, N. \langle H'; e; S \rangle \downarrow^N\langle H'; v; e \rangle,
\end{align*}
\]

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

\[
\begin{align*}
\langle H; \text{let } z \overset{m}{=} e_1 \text{ in } (\text{let } f \overset{n}{=} \lambda \cdot x . e_2) \rangle \downarrow^N, \quad z \not\in FV(e_2), \quad \text{then}
\end{align*}
\]

\[
\begin{align*}
\langle H; \text{let } f \overset{n}{=} \lambda \cdot x . (\text{let } z \overset{m}{=} e_1 \text{ in } e_2) \rangle \downarrow^N.
\end{align*}
\]

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

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

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

6.1 Analysis

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

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

6.2 Absence

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

6.3 One-shot lambdas

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

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

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

- It may be inlined at \(x\)'s use site, perhaps entirely eliminating the allocation of a thunk for \(x\).
- It may enable a rewrite rule (eg foldr/build fusion) to fire.
- It may allow two lambdas to be replaced by one. For example
  \[
  f = \lambda v. \text{let } x = \text{costly } v \text{ in } \lambda y \ldots x \ldots
  \implies f = \lambda v. \lambda y \ldots \text{(costly } v\ldots)
  \]

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

6.4 Single-entry thunks

The code that GHC compiles for a thunk begins by pushing an update frame on the stack, which includes a pointer to the thunk. Then the code for the thunk is executed. When evaluation is complete, the value is returned, and the update frame overwrites the thunk with an indirection to the values (Peyton Jones 1992). It is easy to modify this mechanism to take advantage of single-entry thunks: we do not generate the push-update-frame code for single-entry thunks. There is a modest code size saving (fewer instructions generated) and a modest runtime saving (a few store instructions saved on thunk entry, and a few more when evaluation is complete).

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

\[
\text{let } y \equiv e \text{ in } \text{let } x = y + 0 \text{ in } x * x
\]

Identity of + \(\implies\) let \(y \equiv e \text{ in } \text{let } x = y + 0 \text{ in } x * x\)

Inline \(x \implies y \equiv e \text{ in } y + y \) Wrong!

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

7. Evaluation

To measure the accuracy of the analysis, we counted the proportion of (a) one-shot lambdas and (b) single-entry thunks. In both cases, these percentages are of the syntactically occurring lambdas or thunks respectively, measured over the code of the benchmark program only, not library code. Table 1 shows the results reported by our analysis for programs from the nofib benchmark suite (Purjan 1993). The numbers are quite encouraging. One-shot lambdas account for 0-30% of all lambdas, while single-entry thunks are 0-23% of all thunks.

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

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

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

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Allocs</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No hack</td>
<td>Hack</td>
</tr>
<tr>
<td>anna</td>
<td>-2.1%</td>
<td>-0.2%</td>
</tr>
<tr>
<td>bept</td>
<td>-2.2%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>cacheprof</td>
<td>-7.9%</td>
<td>-0.6%</td>
</tr>
<tr>
<td>calendar</td>
<td>-9.2%</td>
<td>+0.2%</td>
</tr>
<tr>
<td>constraints</td>
<td>-0.9%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>cryptarithm2</td>
<td>-0.3%</td>
<td>-0.3%</td>
</tr>
<tr>
<td>gcd</td>
<td>-15.5%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>gen_regexps</td>
<td>-1.0%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>hpg</td>
<td>-2.0%</td>
<td>-1.0%</td>
</tr>
<tr>
<td>integer</td>
<td>-0.0%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>life</td>
<td>-0.8%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>mkhprog</td>
<td>-11.9%</td>
<td>+0.1%</td>
</tr>
<tr>
<td>nucleic2</td>
<td>-14.1%</td>
<td>-10.9%</td>
</tr>
<tr>
<td>partstof</td>
<td>-95.5%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>sphere</td>
<td>-1.5%</td>
<td>-1.5%</td>
</tr>
</tbody>
</table>

... and 72 more programs

| Min      | -95.5%  | -10.9%  | -28.2%  | -12.1% |
| Max      | +4.5%   | +0.5%   | +1.8%   | +2.8%  |
| Geometric mean | -6.0%  | -0.3%   | -2.2%   | -1.4%  |

any performance benefits directly; rather, they remove potential obstacles from other compile-time transformations. Single-entry thunks, on the other hand give an immediate performance benefit, by omitting the push-update-frame code, but it is a small one.

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

In Section 7.1 we mentioned a hack, used by Gill in GHC, in which he hard-coded the call-cardinality information for three particular functions: build, foldr and runST. Our analysis renders this hack redundant, as now the same results can be soundly inferred. We therefore report two sets of results: relative to an un-hacked baseline, and relative to a hacked baseline. In both cases binary size of the (statically) linked binaries falls slightly but consistently (2.0% average), which is welcome. This may be due to less push-update-frame code being generated.

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

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

Table 3. Optimisation of the programs from Benchmarks Game

<table>
<thead>
<tr>
<th>Library</th>
<th>λ1</th>
<th>Thnk1</th>
<th>Benchmarks</th>
<th>Alloc Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>attoparsec</td>
<td>32.8%</td>
<td>19.3%</td>
<td>benchmarks</td>
<td>-7.1%</td>
</tr>
<tr>
<td>binary</td>
<td>16.8%</td>
<td>0.9%</td>
<td>builder</td>
<td>-0.2%</td>
</tr>
<tr>
<td>bytestring</td>
<td>5.3%</td>
<td>4.3%</td>
<td>boundcheck</td>
<td>-0.5%</td>
</tr>
<tr>
<td>cassava</td>
<td>26.4%</td>
<td>9.8%</td>
<td>benchmarks</td>
<td>-0.7%</td>
</tr>
</tbody>
</table>

Table 4. Analysis and optimisation results for hackage libraries

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>GHC Alloc Δ</th>
<th>GHC RT Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>anna</td>
<td>5740</td>
<td>-1.6%</td>
<td>-1.5%</td>
</tr>
<tr>
<td>cacheprof</td>
<td>1600</td>
<td>-1.7%</td>
<td>-0.4%</td>
</tr>
<tr>
<td>fluid</td>
<td>1579</td>
<td>-1.9%</td>
<td>-0.4%</td>
</tr>
<tr>
<td>gamteb</td>
<td>1933</td>
<td>-0.5%</td>
<td>-0.5%</td>
</tr>
<tr>
<td>parser</td>
<td>2379</td>
<td>-0.7%</td>
<td>-0.2%</td>
</tr>
<tr>
<td>veritas</td>
<td>4674</td>
<td>-1.4%</td>
<td>-0.3%</td>
</tr>
</tbody>
</table>

Table 5. Compilation with optimised GHC

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

7.2 Real-world programs

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

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

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

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

8. Related Work

8.1 Abstract interpretation for usage and absence

The notion of “single-entry” thunks and “one-shot” lambdas is reminiscent of linear types (Girard 1995; Turner and Wadler 1999), a similarity that was noticed very early (Launchbury et al. 1993). Linear types per se are far too restrictive (see, for example, Wansbrough and Peyton Jones (1999)). Then usage polymorphism proved essential to express usage information inspired a series of “once upon a type” papers (Gustavsson 1998; Turner et al. 1995; Wansbrough 2002; Wansbrough and Peyton Jones 1999).

Alas, a promising idea turned out to lead, step by step, into a deep swamp. Firstly, subtyping proved to be essential, so that a function that used its argument once could have a type like $\text{Int}\!^1 \rightarrow \text{Int}$, but still be applied to an argument $\text{x}$ that was used many times and had type $\text{Int}\!^\omega$ (Wansbrough and Peyton Jones 1999). Then usage polymorphism proved essential to cope with currying: “[Using the monomorphic system] in the entirety of the standard libraries, just two thunks were annotated as used-once” (Wansbrough 2002). Gustavsson advocated bounded polymorphism to gain greater precision (Gustavsson and Sveningsson 2001), while Wansbrough extended usage polymorphism to data types, sometimes resulting in data types with many tens of usage parameters. The interaction of ordinary type polymorphism with all these usage-type features was far from straightforward. The inference algorithm for a polymorphic type system with bounds and subtyping is extremely complex. And so on. Burdened with these intellectual and implementation complexities, Wansbrough’s heroic prototype in GHC (around 2,580 LOC of brand-new code; plus changes to thousands of lines of code elsewhere) turned out to be unsustainable, and never made it into the main trunk.

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

$$f \times g = g \times f$$

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

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

Another intriguing difference is in the handling of free variables:

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

How many times is the free variable $y$ evaluated in this expression? Obviously just once, and LETD discovers this, because we un-leash the demand on $y$ at $f$’s call site, and lub the two branches of the if. But type systems behave like LETU: compute the demand on $f$ (namely, called once) and from that compute the demand on $y$. Then combine the demand on $y$ from the body of the let (used at most once), and from $f$’s right hand side (used at most once), yielding the result that $y$ is used many times. We have lost the fact that the two uses come from different branches of the conditional.

The fact that our usage signatures include the $\varphi$ component makes them more expressive than mere types — unless we extend the type system yet further with an polymorphic effect system (Hage et al. 2007; Holdermans and Hage 2010). Moreover, the analysis approach deals very naturally with absence, and with product types such as pairs, which are ubiquitous. Type-based approaches do not do so well here.

In short, an analysis-based approach has proved much simpler intellectually than the type-based one, and far easier to implement. One might wonder if a clever type system might give better results in practice, but Wansbrough’s results (mostly zero change to allocation; one program allocated 15% more, one 14% less (Wansbrough 2002)) were no more compelling than those we report. Our proof technique does however share much in common with Wansbrough and Gustavsson’s work, all three being based on an operational semantics with an explicit heap.

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

8.3 Other related work

Call demands, introduced in this paper, appear to be related to the notion of applicativeness, employed in the recent work on relevance typing (Holdermans and Hage 2010). In particular, applicativeness means that an expression either “guaranteed to be applied to an argument” (S), or “may not be applied to an argument” (L). In this terminology S corresponds to a “strong” version of our demands $C^d(d)$, which requires $d \subseteq U$, 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 and Migh [2010]]. This approach also greatly simplifies integration of the counting abstract domain to account for sharing [Might and Shivers [2006]]. However, the abstract interpreters obtained this way are whole-program forward analysers, which makes them non-modular. It would be, however, an interesting future work to build a backwards analysis from AAM.

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

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

References
Optional Appendices

In the optional appendices we provide detailed proofs of key lemmas and theorems for the soundness of the cardinality analysis and optimisations it enables. Appendix A provides typing rules for stacks and heaps and proves the soundness of the analysis (Section 4). Appendix B contains a proof of Theorem 5.1 which connects the counting semantics \(\xi\) with the optimised semantics \(\xi^+\) (Figure 5) with the optimised semantics \(\Rightarrow\) (Figure 6). It also provides a proof sketch of the let-in float improvement (Theorem 5.3).

A. Soundness (Section 4)

A.1 Stack and heap typing for analysis safety

Definition A.1 (Configuration typing). We write \(P \vdash (H : e : S)\) to mean that there exist \(d, \tau, \phi_1\) and \(\phi_2\) such that \(P \vdash e \downarrow d \Rightarrow (\tau : \phi_1)\) and \(P \vdash S \downarrow (d, \tau) \Rightarrow \phi_2\) and \(P \vdash H \sim (\phi_1 \& \phi_2)\) according to the heap and stack typing rules of Figure 2.

Figure 2 explains how we type stacks and heaps. The judgement \(P \vdash S \downarrow (d, \tau) \Rightarrow \phi\) identifies the \(\xi\)-usage environment of the stack \(S\), given that the argument that we intend to place in the hole of the stack has type \(\tau\) when being pressed with demand \(d\). Rule SHU deals with the case when we press 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 \((\bullet : y) : S\). 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 LETD\(N\) that we encountered in the typing rules. If we encounter a stack \(#(x, n) : S\) 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 \(m \times d_1\). In total the demand that this stack presses on the hole is \(d \& d_1\) – and it must be the case that the multiplicity on the stack \(n\) be higher that the indirect multiplicity in \(S(m)\), 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 \(\phi\) 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 \((\tau : \phi_1)\) and the stack, when pressed with \(d\), can give \(\phi_2\), then we can simply return \(\phi_1 \& \phi_2\). In this case the demand pressed on the hole of the stack can be any \(d_0 \subseteq U(\phi_1(x), \phi_2(y))\).

The heap typing judgement \(P \vdash H \sim \phi\) ensures that the heap \(H\) has enough multiplicity to withstand the pressure that \(\phi\) will exercise. Rules HPVarABS and HPEmpty are boring. However HPVarUP ensures that if \(\phi\) needs to press \(m \times d\) 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 \(\phi_1\). Finally, (iii) the remaining heap must have enough multiplicity to withstand the newly unleashed demand from \(\phi_1\).

Rule HPVarDN is simpler: 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 \(\phi\) presses.

With these definitions in place we can prove the generalised safety statement, Lemma 4.3 which is needed for the proof of Theorem 4.2.

A.2 Soundness theorems

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

\[\tau_1 \sqsubseteq \tau_2 \Leftrightarrow (\tau_1 \sqcup \tau_2) = \tau_2\]

For usage environments \(\phi\) the partial order is defined as a pointwise lifting of partial order on multi-derivatives in their codomains (assuming each \(\phi\) is predetermined with \(A\) by default).

Lemma A.1 (Monotonicity of usage typing). If the transformer environment \(P\) consists of monotone functions and \(P \vdash e \downarrow d \Rightarrow P \vdash e \downarrow d \Rightarrow P \vdash e \downarrow d \Rightarrow P \vdash e \downarrow d \Rightarrow (\tau_1 \sqcup \tau_2) = \tau_2\)
Lemma A.2 (Discrete usage signatures are well-formed). If
\[
e = \lambda x_1^{n_1} \ldots \lambda x_k^{n_k}.e_1. n_1, \ldots, n_k > 0
\]
\[
P \vdash e \downarrow U \Rightarrow \langle \tau_1 ; \varphi_1 \rangle
\]
\[
\langle \tau_0 ; \varphi_0 \rangle = \langle \varphi_0 (\Psi) \rightarrow \tau_1 \ \varphi_1 \setminus \tau \rangle
\]
\[
g = \lambda \tilde{d}. transform((k ; \tau_0 ; \varphi_0), d)
\]
then \(P \not\vdash e : g\).

Proof: By the typing rule WFTRANS, we need to show that
\[
\forall d_1, d_2, d_3 \subseteq d_4 \Rightarrow T^p_{d_4} \subseteq T^q_{d_4} \land \Phi^p_{d_4} \subseteq \Phi^q_{d_4}
\]
\[
\forall d, \varphi, \tau. (P \vdash e \downarrow d \Rightarrow \langle \tau ; \varphi \rangle) \Rightarrow \tau \subseteq T^d_{\varphi} \land \varphi \subseteq \Phi^d_{\varphi}.
\]
The proof of (5) is straightforward, since \(g\) is a monotonic step-function.

For the second part, let us first define the threshold \(d_k\) as \(d_k = C^k(U)\). We remark that, by consecutive applications of rule TLAM, we can obtain:
\[
P \vdash e \downarrow d_k \Rightarrow \langle \tau_0 ; \varphi_0 \rangle
\]
Let us assume that \(P \vdash e \downarrow e \Rightarrow \langle \tau_d ; \varphi_d \rangle\). We show that \(\tau_d \subseteq \tau_0\) and \(\varphi_d \subseteq \varphi_0\) by induction on the number of \(\lambda\)s.

- If \(k = 0\) then it can only be that \(d \subseteq U\) and the result follows by monotonicity (Lemma A.1).
- If \(k = l + 1\) then we have several cases on the shape of \(d\).
  - \(d = U(d'_1, d'_2)\). This can only happen if \(d \subseteq HU\) and rule TLAMHU was used, otherwise the lambda is not typeable at all. But \(HU \subseteq d\) 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 \(e = \lambda_0.e_0\) with \(C_m(d_1)\), \(n \geq m\), and apply the induction hypothesis for the body \(e_0\). We get back a pair \(\langle \tau_1 ; \varphi_1 \rangle\). If \(m = 1\) then we are easily done by the induction hypothesis. If \(m = \omega\) then it is definitely the case that \(d \not\subseteq d\) and hence we multiply both transform results by \(\omega\) and we are done, using the induction hypothesis\[4\].
- \(d = U\). We observe that \(d \subseteq C^*(U)\) and hence the case follows as the previous one using inversion on TLAM.

\[\square\]

\[\text{Lemma A.3} (\text{Analysis produces well-typed terms (Lemma 4.1))}.\]

If \(P \vdash e \downarrow d \Rightarrow \langle \tau ; \varphi \rangle\) then \(P \vdash e \downarrow d \Rightarrow \langle \tau ; \varphi \rangle\).

Proof. The proof is by induction on the height of the derivation \(P \vdash e \downarrow d \Rightarrow \langle \tau ; \varphi \rangle\). 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, 
\[
g(d) \triangleq \text{transform}(\rho, d),
\]
where \(\text{transform}(\langle k ; \tau ; \varphi \rangle, d)\) is defined in Figure[1].

- Case VarDN. Corresponds straightforward to the application of rule TVARDN, where \(g(d) \triangleq \text{transform}(\rho, d)\).
- Case VarUP. Straightforward by the rule TVARUP.
- Case Lam. By the rule TLAM. By induction hypothesis, we have \(P \vdash e_1 \downarrow d_1 \Rightarrow \langle \tau_1 ; \varphi_1 \rangle\). Moreover, by the formulation of Lam, \(d = C(d_1)\) (exact equality) and \(m = n\), so the premises of the rule TLAM are fulfilled.
- Case LAMU. Follows by rule TLAM observing that \(U \subseteq C^*(U)\).
- Case LAMHU. Straightforward by the rule TLAMHU.
- Case APPA. By induction hypothesis an simple additional statement relating \(\vdash^1\) and \(\vdash^2\), we have
\[
P \vdash y \downarrow d_1^1 \Rightarrow \varphi_2\]
\[
P \vdash e_1 \downarrow C^1(d) \Rightarrow \langle \tau_1 \rightarrow \tau_1 ; \varphi_1 \rangle\]
And let us just take \(\tau_1 = d_1^2 \Rightarrow \tau_1\), so the premises of the rule TAPP are fulfilled.
- Case APPB. By induction, we have
\[
P \vdash y \downarrow \omega \ast U \Rightarrow \varphi_2\]
\[
P \vdash e_1 \downarrow C^1(d) \Rightarrow \langle \omega \ast U \rightarrow \tau_1 ; \varphi_1 \rangle\]
Moreover, by the definition of \(\preceq\) (Figure[1]),
\[
\bullet \preceq \omega \ast U \Rightarrow \bullet,
\]
so we just take \(\tau_1 = \bullet\), which fulfills the premise of the rule TAPP.

- Case PAIR. Straightforward by the typing rule TPAIR, taking \(d = U(d_1^1, d_1^2)\).
- Case PAIRU. Straightforward by the typing rule TPAIR, observing that \(\omega \ast U \subseteq U(\omega \ast U, \omega \ast U)\).
- 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 (\(e\)) in both cases. Finally, \(e \& e = e\), which concludes the proof for this case.

\[\text{Note that we can guarantee the same result by choosing a different more expressive transform that only infinitizes the previous types but not the current one, yielding tighter types, but we have not done that for simplicity.}\]
• Case CASE. By induction hypothesis,
\[ P \vdash e_1 \downarrow d \Rightarrow (\tau_1; \varphi_1), \]
\[ P \vdash e_2 \downarrow U.(\varphi_2(x), \varphi_2(y)) \Rightarrow (\tau_2; \varphi_2), \]
so we can directly apply the typing rule TCASE.

• Case LETUP By the induction hypothesis, we have
\[ P \vdash e_1 \downarrow d \Rightarrow (\tau_1; \varphi_1) \]
\[ n \times d_1 = \varphi_2(x) \]
\[ P \vdash e_1 \downarrow d \Rightarrow (\tau_1; \varphi_1) \]

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:
\[ P \vdash \text{let } x = \lambda y^=k : e_1 \text{ in } e_2 \downarrow d \Rightarrow (\tau_2; \varphi_2) \]
\[ \text{let } x \equiv \lambda^n x_1 \ldots \lambda^n x_n : e_1 \text{ in } e_2 \]

Let us call the resulting RHS term \( e = \lambda^n x_1 \ldots \lambda^n x_n : e_1 \). By inversion we have that:
\[ P \vdash e \downarrow (C^n(\ldots)) \Rightarrow (\tau_2; \varphi_2) \]
\[ \text{let } \langle x \equiv \lambda^m x_1 \ldots \lambda^m x_m : e_1 \rangle \text{ in } e_2 \]

Hence it is easy to show by induction and monotonicity that \( P \vdash e \downarrow (C^n(\ldots)) \Rightarrow (\tau_2; \varphi_2) \) if \( n \geq \mu(\varphi_2(x)) \). Moreover we know that \( n \geq \mu(\varphi_2(x)) \).

Finally the statement for the body follows by induction hypothesis. The case is finished by putting these results together and applying rule TLETDN.

• Case LETDNABS. Similar to the case LETDN.

\[ \Box \]

Lemma A.4 (Value splitting (Lemma 4.4)). If \( P \vdash \nu \downarrow (d_1 \& d_2) \Rightarrow (\tau_2; \varphi_2) \) then there exists a split \( (\nu, \varphi) \) such that:
\[ P \vdash \nu_1 \downarrow d_1 \Rightarrow (\tau_1; \varphi_1) \]
\[ P \vdash \nu_2 \downarrow d_2 \Rightarrow (\tau_2; \varphi_2) \]
and moreover \( \nu_1 \preceq \tau_1 \), \( \nu_2 \preceq \tau_2 \), \( \varphi_1 \preceq \varphi \).

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 \( \nu \).

• Case \( \nu = \lambda x.e \). In this case without loss of generality assume that:
\[ d_1 = U(d_1^1, d_1^2) \]
\[ d_2 = U(d_2^1, d_2^2) \]
If one of them is a call demand then its \& is not defined, and if one of them is a naked \( U \) or \( \text{H} \) then that is equivalent to some \( U(d_1^1, d_2^1) \) in terms of how the result will be typed. The result can follow by monotonicity of the \& operation and Lemma A.5 (see below).

• Case \( \nu = \lambda^n x.e \). In this case, if one of \( d_1 \) or \( d_2 \) is less or equal to \( \text{H} \), assume \( d_1 \) then the split is by choosing \( n_1 = 0 \) and \( n_2 = n \). If \( n_1 = 0 \) then split uses the TLAMRU rule assigning the same type as the other split assigns. The other split merely uses the typing rule that was originally used to type \( \nu \). If on the other hand no \( d_1 \) nor \( d_2 \) is less or equal to \( \text{H} \) then they cannot be used-demands either (because their \& would not be defined). Assume then without loss of generality that:
\[ d_1 = C^m(n_1^1, d_1^2) \]
\[ d_2 = C^m(n_2^1, d_2^2) \]
If one of them was \( U \) then we simply type it as \( C^m(U) \). 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:\[ (\varphi_1(x) \Rightarrow \tau_1^1 ; n_1 \times \varphi_1^1) \] and similarly:
\[ (\varphi_2^2(x) \Rightarrow \tau_1^2 \times n_2 \times \varphi_2^1) \]
where \( \varphi_1 \) and \( \tau_1 \) are the results of typing \( e \) with \( d_1 \) respectively. However we know that the body is typeable with \( d_1 \) or \( d_2 \) resulting in:\[ \varphi_1(x) \Rightarrow \tau_1^1 ; (n_1 + n_2 \times \varphi_1) \]
for \( \nu \). By monotonicity we get that:
\[ \varphi_1(x) \Rightarrow \tau_1 \cup \varphi_1(x) \Rightarrow \tau_1 \]
as required. Moreover we need to show that:
\[ n_1 \times \varphi_1^1 \cup \varphi_2^1 \cup (n_1 + n_2) \times \varphi_1 \]

By monotonicity it suffices to show that:
\[ n_1 \times \varphi_1 \cup \varphi_2 \cup (n_1 + n_2) \times \varphi_1 \]
and the result follows by the easy-to-show fact that:
\[ n_1 \times d_1 \cup n_2 \times d_2 \cup (n_1 + n_2) \times d_1 \]
for any \( d_1 \).

\[ \Box \]

Lemma A.5 (Variable demand splitting). Assume that the transformer environment \( P \) is monotone. If \( P \vdash x \downarrow (d_1 \& d_2) \Rightarrow (\tau; \varphi) \) then \( P \vdash x \downarrow d_1 \Rightarrow (\tau_1; \varphi_1) \) and \( P \vdash x \downarrow d_2 \Rightarrow (\tau_2; \varphi_2) \) and \( \varphi_1 \cup \varphi_2 \subseteq \varphi \).

Proof. If \( x \notin \text{dom}(P) \) then the result is trivial. If \( x \in \text{dom}(P) \) then there is a transformer \( (x; \varphi) \in P \). First of all let us examine the case where either \( d_1 \) or \( d_2 \) is \( A \). Without loss of generality assume \( d_1 = A \). Since this case the result is trivial since \( \varphi_2 = \epsilon \) and \( \varphi = \varphi_1 \). Assume instead that:
\[ d_1 = n_1 \times d_1 \]
\[ d_2 = n_2 \times d_2 \]
In this case it suffices to show that:
\[ n_1 \times \varphi_1 \subseteq \varphi_1 \text{ and } d_1\&d_2 \subseteq \varphi_1 \]
However by monotonicity we know that:
\[ n_1 \times \varphi_1 \subseteq \varphi_1 \text{ and } d_1\&d_2 \subseteq \varphi_1 \]
so it suffices to show for every binding in \( \varphi_1 \& d_1 \& d_2 \), call it \( (y; d_1) \) that it is the case that:
\[ n_1 \times d_1 \cup n_2 \times d_2 \subseteq (n_1 + n_2) \times d_1 \]
This is easy to show using the fact that \( \omega \times d_1 = d_1 \& d_1 \text{ and } d_1 \& d_1 \& d_1 \& d_1 \) for all \( d_1 \).

\[ \Box \]

Lemma A.6 (Single-step safety (Lemma 4.3)). Assume that:
\[ \vdash \langle H_1 ; e_1 ; S_1 \rangle \]
\[ \vdash \langle H_2 ; e_2 ; S_2 \rangle \]
in the uninstrumented semantics, then \( \vdash \langle H_1 ; e_1 ; S_1 \rangle \rightarrow \langle H_2 ; e_2 ; S_2 \rangle \)
such that:
\[ H_2^1 = H_2, e_2^1 = e \text{ and } S_2^1 = S_2 \text{ and moreover } \vdash \langle H_2 ; e_2 ; S_2 \rangle \]

Proof. By induction on the height of the derivation \( \vdash \langle H ; e ; S \rangle \). We proceed by case analysis on the rule used for \( \rightarrow \) 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:
\[ P \vdash \text{let } x \equiv e_1 \text{ in } e_2 \downarrow d \Rightarrow (\tau; \varphi_1 \& \varphi_2) \]
\[ n \leq m \]
\[ P \vdash e_2 \downarrow d \Rightarrow (\tau; \varphi_2, (x; n \times d_1)) \]
\[ P \vdash e_1 \downarrow d \Rightarrow (\tau; \varphi_1) \]

Moreover:
\[ P \vdash S \downarrow (d, \tau) \Rightarrow \varphi S \]
\[ P \vdash H \Rightarrow \varphi_1 \& \varphi_2 \& \varphi S \]
The rule ELET fires in the instrumented semantics as well, giving us a new heap \( H, [x \Rightarrow_{\text{Exp}} e_1] \). By using HP-VARU we can conclude:
\[ P \vdash H, [x \Rightarrow_{\text{Exp}} e_1] \sim \varphi_2, (x; n \times d_1) \& \varphi S \]
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:
  
  \[ P \vdash \textit{let } x = e_1; e_2 \downarrow d \Rightarrow (\tau; \varphi_2) \]  
  
  \[ n \leq m \]  
  
  \[ P \vdash e_1 \downarrow \tau_1 \Rightarrow (\sim; \varphi_1) \]  
  
  \[ P \vdash e_1 : \tau : \varphi_1 \]  
  
  \[ P, (x : \tau_2) \vdash e_2 \downarrow d \Rightarrow (\tau; \varphi_2; (x : d_1)) \]  
  
  \[ d_1 \subseteq n + d_1 \]  
  
  Moreover:

  \[ P \nvdash S \downarrow (d, \tau) \Rightarrow \varphi_S \]  
  
  \[ P \vdash H \sim \varphi_2 \& \varphi_S \]  
  
  The rule ELET fires in the instrumented semantics as well, giving us a new heap \( H, [x \rightarrow \text{Exp}(e_1)] \). We need to use HPVarDN to deduce that:

  \[ P, (x : \tau_2) \vdash H, [x \rightarrow \text{Exp}(e_1)] \sim (\varphi_2; (x : d_1) \& \varphi_S) \]  

  which follows from (30), (26), (24), (25). Moreover, from (29) and the observation that \( x \notin \text{fv}(S) \) it is easy to deduce that

  \[ P, (x : \tau_2) \nvdash S \downarrow (d, \tau) \Rightarrow \varphi_S \]  

  (simple inductive weakening proof). From this, and (27) and (31) we get that the resulting configuration is well typed.

- Case ELKPV. Again we have two cases depending on how the variable was typed.

- Case TVARUp. In this case we have:

  \[ P \vdash x \downarrow d \Rightarrow (\bullet; (x : 1 + d)) \]  

  where \( x \notin \text{dom}(P) \). Moreover:

  \[ P \nvdash S \downarrow (d, \bullet) \Rightarrow \varphi_S \]  

  \[ P \vdash H, [x \rightarrow \text{Exp}(e)] \sim (\varphi_S); (x : 1 + d) \& \varphi_S(x) \]  

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

  \[ P \vdash H, [x \rightarrow \text{Exp}(e)] \sim (\varphi_S); (x : (1 + m) * (d \& d_e)) \]  

  By inverting HPLETUp, it must be then:

  \[ n \geq m + 1 \]  

  \[ P \vdash H \sim (\varphi_S); \& \varphi_e \]  

  \[ P \vdash e_1 \downarrow (d \& d_e) \Rightarrow (\tau_1; \varphi_e) \]  

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

  \[ P \nvdash (\#(x, n) : S \downarrow (d \& \tau_1) \Rightarrow \varphi_S) \]  

  which will be the case if we show that:

  \[ P \vdash S \downarrow (d, \bullet) \Rightarrow (\varphi_S); (x : m + d_e) \]  

  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:

  \[ P \vdash x \downarrow d \Rightarrow (T_d^x; \Phi_\delta \& (x : 1 + d)) \]  

  Let us assume that bindings are not recursive so \( x \notin \text{dom}(\Phi_\delta) \). Moreover:

  \[ P \nvdash S \downarrow (d, T_d^x) \Rightarrow \varphi_S \]  

  \[ P \vdash H, [x \rightarrow \text{Exp}(e)] \sim (\Phi_\delta \& (\varphi_S); (x : 1 + d \& \varphi_S(x))) \]  

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

  \[ P \vdash H \Rightarrow (\Phi_\delta \& (\varphi_S); \tau_d \subseteq T_d^x) \]  

  Now for the right hand side the environment from the expression is \( \varphi_d \) (we press the demand \( d \)). The environment for the stack-typing is the one we get from: \( P \nvdash S \downarrow (d, T_d^x) \Rightarrow \Phi_\delta \& \varphi_d \) Hence we need to show that: \( P \vdash H \sim (\varphi_d \& (\varphi_S); \tau_d \subseteq T_d^x) \).

- Case ELKPV. Again we have two cases depending on how the variable is typed.

  - Case TVARUp. In this case we have:

    \[ P \vdash x \downarrow d \Rightarrow (\bullet; (x : 1 + d)) \]  

    where \( x \notin \text{dom}(P) \). Moreover:

    \[ P \vdash S \downarrow (d, \bullet) \Rightarrow \varphi_S \]  

    \[ P \vdash H, [x \rightarrow \text{Val}(v)] \sim (\varphi_S); (x : 1 + d \& \varphi_S(x)) \]  

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

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

      \[ P \vdash H \sim (\varphi_S); \& \varphi_e \]  

      where \( P \vdash v \downarrow (d \& d_e) \Rightarrow (\tau_1; \varphi_e) \). Hence it must be that

      \[ P \vdash H \sim (\varphi_S); \& \varphi_e \]  

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

      \[ H \sim \tau_1 \& \varphi_2 \& (\varphi_S); \tau_1 \subseteq \tau ; \tau_2 \subseteq \tau \]  

      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:

  \[ P \vdash x \downarrow d \Rightarrow (T_d^x; \Phi_\delta \& (x : 1 + d)) \]  

  where \( (x : \tau) \in P \). Moreover:

  \[ P \vdash S \downarrow (d, T_d^x) \Rightarrow \varphi_S \]  

  and

  \[ P \vdash H, [x \rightarrow \text{Val}(v)] \sim (\Phi_\delta \& (\varphi_S); (x : 1 + d \& \varphi_S(x)) \]  

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

\[ P \vdash H \bowtie (\varphi_\beta) \downarrow \land \Phi^d_{\beta} \]  

(48)

and moreover \( P \vdash \nu \downarrow (d \land d_{\beta}) \Rightarrow (\cdot \land \varphi_\nu) \) (i.e. the \( \nu \) is sufficiently annotated). So the configuration can indeed step and for the right-hand side, by the splitting lemma, we must have: \( P \vdash v_2 \downarrow d \Rightarrow (\tau_2 \land \varphi_\nu) \) and we must also have \( P \vdash S \downarrow (d; \tau_2) \Rightarrow \varphi_\nu \). By the well-formedness of the transformer it must be \( \tau_2 \land \tau \downarrow \Phi^d_{\beta} \) and it must also be \( \varphi_\nu \land \varphi_\nu \). Hence by monotonicity \( \varphi_\nu \land \varphi_\nu \) as well. To finish the case we need to show that:

\[ P \vdash H, [x \mapsto \nu_\beta, \text{Val}(v_1)] \sim (\varphi_\nu) \downarrow \land \varphi_\nu, (x; \varphi_\nu(x)) \]

By rule HPLETDN it suffices to show two things: First, that \( P \vdash H \sim (\varphi_\nu) \downarrow \land \varphi_\nu - \) this follows by (48) and monotonicity. Second, that if \( \varphi_\nu(x) = d \downarrow \), \( v_1 \) is still typeable under that \( d \downarrow \). However by the splitting lemma we know that: \( P \vdash v_1, d \downarrow \Rightarrow (\cdot \land \varphi_\nu) \) and the result follows by monotonicity since it must be the case that \( d \downarrow \subseteq m \times d_{\beta} \).

- Case EUFD. Similar to ELKPV case.
- Case EBETA. Using the substitution lemma (Lemma A.8).
- Case EAAPP. Trivial.
- Case EPS. Trivial.
- Case EPRED. Using the substitution lemma (Lemma A.8).

\[ \downarrow \]

Lemma A.7 (Heap-typing strengthening). If \( P \vdash H \sim \varphi_1 \) and \( \varphi_2 \subseteq \varphi_1 \) then \( P \vdash H \sim \varphi_2 \).

\[ \downarrow \]

Proof. Easy induction, appealing to the monotonicity of typing lemma.

\[ \downarrow \]

Lemma A.8 (Substitution). Assume that \( P \) is monotone and \( P \vdash e \downarrow d \Rightarrow (\cdot \land \varphi_1) \) and \( x \notin \text{dom}(P) \). If \( P \vdash y \downarrow \varphi_1(z) \Rightarrow \varphi_2 \) then \( P \vdash e[y/x] \downarrow d \Rightarrow (\cdot \land \varphi_2) \) such that \( \varphi_2 \subseteq \varphi_1 \) and \( \varphi_2 \land \varphi_2 \) as required.

\[ \downarrow \]

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

- Case TVARDN. In this case we know that 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 \( x \) is then we have that the pressure on \( x \) is \( x(1 \times d) \). Then \( \varphi_2 = \Phi^d_\beta \land (y(1 \times d)) \). For the substituted expression we get that: \( \varphi_\nu = \varphi_2 \land \tau_\epsilon = T^d_\beta \). Clearly \( T^d_\beta \subseteq \cdot \) and moreover \( \varphi_\nu \subseteq 1 \times \varphi_\nu \) 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 \( \varphi_\nu(x) = n_1 \times d_1 \) and \( \varphi_\nu(x) = n_2 \times d_2 \) then

\[ n_1 \times \Phi^d_{\beta} \land n_2 \times \Phi^d_{\beta} \subseteq (n_1 + n_2) \times \Phi^d_{\beta} \]

which follows by the monotonicity of the transformer \( \varphi_\nu \).

\[ \downarrow \]

B. Optimisation (Section 5).

Theorem B.1 (Optimised semantics (Theorem 5.1)). If \( (H_1; e_1; S_1) \sim (H_2; e_2; S_2) \) and \( (H_1; e_1; S_1) \Rightarrow (H_2; e_1; S_1') \) then there exists \( k \in \{0,1\} \times t \). \( (H_2; e_2; S_2) \Rightarrow (H_2; e_2; S_2') \) and \( (H_1; e_1; S_1) \Rightarrow (H_1; e_2; S_2') \).

Proof. The proof is by case analysis on the \( \Rightarrow \) relation:

- Case ELET. We have two cases to consider. If \( n \geq 1 \) then obvious. If \( n = 0 \) then \( H_2' = H_1, (x \mapsto \text{Val}(e_1)) \) and \( H_2' = H_2 \) and \( H_1 \prec H_2 \) as required.
- Case ELKPE. In this case we have that:

\[ (H_1, (x \mapsto \text{Val}(e_1)); x; S_1) \Rightarrow (H_1; e; \#(x, n) \times S_1) \]

given that \( n \geq 1 \). Then either OPT-ELKPE or OPT-ELKPEO will fire:

- If \( n = \omega \) the result follows trivially.
- If \( n = 1 \) then \( S_1' = \#(x, 1) \times S_1 \) and \( S_2' = S_2 \) and by rule SSIM2 we are done.

- Case ELKPV. By the side condition \( m = m + 1 \) it can only be that \( n = 1 \) or \( n = \omega \). By the heap invariant for \( H_1 \) and an easy induction it has to be that \( n = \omega \). The corresponding rule that can fire in the optimised semantics is OPT-ELKPV and the result is trivial.

- Case EUFD. We have that:

\[ (H_1; v; \#(x, n) \times S_1) \Rightarrow (H_1, [x \mapsto \text{Val}(v_1)]; v_2; S_1) \]

where \( n = m + 1 \) and \( \text{split}(\nu) = (v_1, v_2) \). By this last condition it has to be the case that \( n = \omega \) or \( n = 1 \).

- If \( n = \omega \) then rule OPT-EUFD gives the result.
- If \( n = 1 \). Then, assume that \( H_1; v; \#(x, n) \times S_1 \sim (H_2; v; S_2) \) which will happen if \( S_1 \sim S_2 \). However, in this case, \( m = 0 \), which means that it also must be the case that \( H_1, [x \mapsto \text{Val}(v_1)]; v_2; S_1 \sim (H_2; v_2; S_2) \) so we are done in 0 steps (hence we have \( \Rightarrow \) and not just \( \Rightarrow \) in the statement of the theorem).

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

\[ \downarrow \]

Theorem B.2 (Let-in float improvement (Theorem 5.3)). For any \( H \) and \( S \), if

\[ (H; \text{let } z \equiv e_1 \text{ in } (\text{let } f \equiv \lambda x. e \text{ in } e_2); S) \not\subseteq^N \]

and \( z \notin \text{FV}(e_2) \), then

\[ (H; \text{let } f \equiv \lambda x. \text{ (let } z \equiv e_1 \text{ in } e \text{ ) in } e_2); S) \not\subseteq^N. \]

Proof sketch: Let us refer to the first configuration as \( q \) and the second as \( q' \). The proof goes in four stages.

1. It is the case that \( q \) evaluates in two steps to some \( q_1 = (H_1; e_2; S) \) and \( q' \) steps to some \( q_2 = (H_2; e_2; S) \) such that \( H_1 \sim H_2 \) and \( H_2 \). We define \( H_1 \sim H_2 \) if

\[ H_1 = H_0, [z \mapsto e_1], [f_1 \equiv \lambda x. e_1], \ldots, [f_k \equiv \lambda x. e_1] \]

\[ H_2 = H_0, [f_1 \equiv \lambda x. e_2], \ldots, [f_k \equiv \lambda x. e_2] \]

for some \( H_0 \) and \( k \), where \( e_1 = (\text{let } z \equiv e_1 \text{ in } e) \) and \( e_1 \) and \( z \) are from the statement of the theorem, and
\[ \sum_{i=1}^{k} n_i = 1 \] Now we need to show that \( q_2 \) will make at most one step more than \( q_1 \) before they both terminate.

2. Taking \((H_1 \simeq H_2)\) 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 \( S = \epsilon \) the case is done. (And so is if \( S \) contains a case alternative because both computations will be stuck.) If \( S = \#(x, n) : S' \) then we update the heap in both branches in a \( \simeq \)-preserving way, so we are back to stage (2). If \( S = (\bullet y) : S' \) 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.

\[ \square \]

\[ ^9 \text{The Val}(\cdot)/\text{Exp}(\cdot) \text{ distinction does not affect the core of the proof.} \]