Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

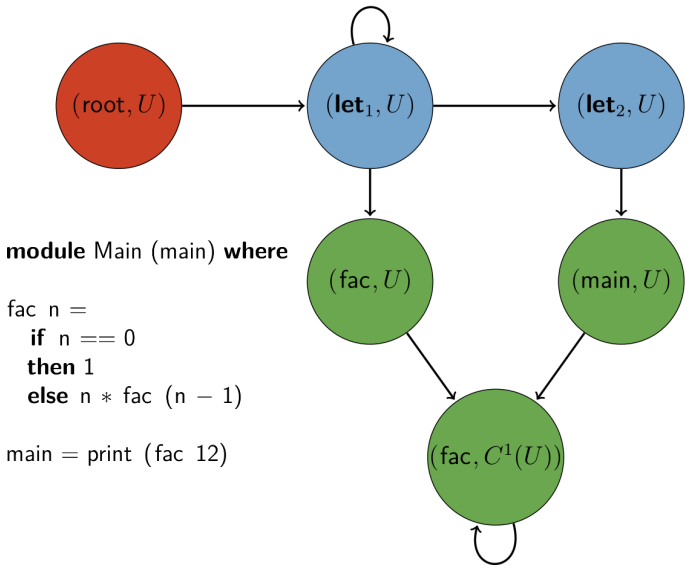
Lehrstuhl Prof. Dr.-Ing. Snelting

Call Arity vs. Demand Analysis

Masterarbeit von

Sebastian Graf

an der Fakultät für Informatik



Erstgutachter: Prof. Dr.-Ing. Gregor Snelting

Zweitgutachter: Prof. Dr. rer. nat. Bernhard Beckert

Betreuende Mitarbeiter: Dr. rer. nat. Joachim Breitner, Dipl.-Inform. Denis Lohner

Bearbeitungszeit: 6. März 2017 – 4. August 2017

Zusammenfassung

Optimierungen in Compilern für funktionale Programmiersprachen verlassen sich auf gute Schätzungen dafür, wie oft syntaktische Konstrukte relativ zum umgebenden Kontext ausgewertet werden. Eine *Kardinalitätsanalyse* berechnet untere und obere Schranken für diese Auswertungsmultiplizitäten. Während untere Schranken den Ergebnissen einer *Strictness Analyse* entsprechen, berechnen *Usage Analysen* obere Grenzen.

Der Glasgow Haskell Compiler (GHC) integriert zwei Analysen, Call Arity und Demand Analysis, die beide unabhängig voneinander Usage Resultate berechnen.

Diese Arbeit stellt eine Usage Analyse vor, die ihr Gegenstück in beiden Analysen verallgemeinert. Eine komplexe Analysereihenfolge hat zu einem neuartigen, graph-basierten Ansatz zur Fixpunktiteration geführt, losgelöst vom Syntaxbaum. Saubere Trennung von Analyselogik und Aufbau des Datenflussnetzwerks wird durch eine eingebettete domänenspezifische Sprache gewährleistet.

Optimisation passes in compilers for functional languages rely on good estimates for how often some syntactic construct is evaluated relative to its enclosing context. A *cardinality analysis* computes lower and upper bounds to these relative evaluation multiplicities. While the lower bounds correspond to the results of a *strictness analysis*, the upper bounds are computed by *usage analyses*.

The Glasgow Haskell Compiler (GHC) integrates two analyses, Call Arity and Demand Analysis, that compute usage results independently of another.

This thesis introduces a usage analysis that generalises its counterpart in both analyses. Challenges in analysis order necessitated a novel graph-based approach to fixed-point iteration, detached from the syntax tree. An embedded domain specific language cleanly separates construction of the data-flow framework from analysis logic.

Contents

1 Introduction

High-level programming languages abstract from operational details so that the programmer can focus on solving problems instead of fighting with the concrete hardware the solution is supposed to run on.

Haskell, being a non-strict purely functional programming language, is an extreme example of this, going well beyond established language features such as garbage collection. Rather than encoding a program as an imperative sequence of commands to execute, programs are specified declaratively as a pure expression to evaluate.

While the programmer benefits from laziness and purity in various ways, the compiler must be equipped to optimise away entire layers of abstraction to produce programs with reasonable performance. On the other hand, high-level constructs such as laziness and purity enable better reasoning about programs also for compilers, which in turn can do a much better job at optimising.

The Glasgow Haskell Compiler (GHC) is a particularly miraculous implementation of Haskell, producing artifacts which perform within the same order of magnitude as hand-tuned C code.

As a compiler becomes more mature, it aggregates a growing number of complicated analyses and transformations, some of which overlap and interact in non-trivial ways. Although GHC started out as an academic project to push the boundaries of what 'high-level' means, after more than 25 years of development and more than 100.000 lines of code it has also become a challenge in terms of software engineering.

Hence, principles such as separation of concerns and 'Don't repeat yourself' (DRY) also apply to writing compilers. Also, as the number of users and language extensions grows, compiler performance has been an increasing woe.

GHC, as of version 8.2, ships with two analyses that share interesting ideas: There's Call Arity [3], the result of Joachim Breitner's dissertation, and a new Higher-order Cardinality Analysis [17], which integrates with the existing Demand Analyser. Call Arity tries to η-expand bindings based on usage information in order to make `foldl` a good consumer for list fusion, whereas Cardinality Analysis (referring to the contributions of Sergey et al. [17]) is concerned with identifying single-entry thunks and one-shot lambdas.

It is not safe in general to η-expand thunks, as it possibly duplicates otherwise shared work. All is not lost however: It is always safe to η-expand single-entry thunks to the minimum arity of all possible calls. Thus, in order for Call Arity to justify η-expansion of thunks, it has to interleave the arity analysis with its own sharing analysis based on co-call graphs [3] to identify said single-entry thunks. Edges in co-call graphs correspond to a 'is possibly evaluated with' relation between variables. Therefore, absence of a self-edge implies that a variable is evaluated at most once,

e.g. is single-entry.

In a perfect world, we would just integrate Call Arity's co-call graphs into Cardinality Analysis and have Call Arity access the analysis results, even if it only was for concentrating the analysis logic in one place (adhering to DRY). Yet there are quite a number of problems that need to be solved over the course of this thesis before we provide a unified analysis in Chapter 3:

Complexity Call Arity currently interleaves the co-call analysis with its own arity analysis. Information flows both ways, so it seems that the Demand Analyser will additionally have to integrate an arity analysis. Considering the complexity of the current implementation at the time of this writing (GHC 8.2) this is hardly possible. Needless to say that GHC already integrates an arity analysis in the form of its CoreArity module.

Let bindings Cardinality Analysis employs different analysis orders for **let** bindings, depending on whether the bound expression is a thunk (LetUp) or not (LetDn). The arity analysis within Call Arity relies on the fact that all calls to a binding are analysed before the bound expression is analysed to find the minimum arity of all calls. This corresponds to the LetUp rule in Cardinality Analysis and it is not immediate if at all and how these analysis orders can be combined.

Data structures Although there is some overlap in language when talking about η-expansion of single-entry thunks, it is not obvious how information from Cardinality Analysis is to be translated into a format suitable for Call Arity and vice versa.

1.1 Contributions

We present a usage analysis that generalises both Cardinality Analysis [17] and Call Arity [3].

After giving a broad overview over the different analyses involved in Section 2.1, specifically Call Arity (Section 2.3) and the Demand Analyser (Section 2.4), we make our point for extracting the usage analysis from the Demand Analyser into a separate analysis in 2.4.2.

A formal specification of the analysis in terms of a simplified language is given in Chapter 3. The analysis works by abstract interpretation over a domain quite similar to that of Sergey et al. [17], outlined in Section 3.2. In Section 3.3 follows the definition of the transfer function, while we discuss how to recover Call Arity (e.g. η-expansion based on usage) and Cardinality Analysis in Section 3.4.

Chapter 4 discusses challenges solved while implementing the analysis within GHC 8.2.1. Of special note are our insights on solving the graph-based data-flow problem in Section 4.8 and our thoughts on approximating usage transformers in Section 4.6.

We discuss benchmark results in Chapter 5. As the goal of this thesis was to merge both analyses, improvements in benchmarks are quite secondary and serve more as a safety check for the analysis. More critical, we discuss regressions in compiler performance and how to resolve them.

Chapter 6 discusses related and future work, focussing on how our approach compares to papers on usage analyses published after Sergey et al. [17], who gave a brilliant treatment. We also discuss our novel graph-based solver for data-flow problems and advocate future work in the form of refactorings within GHC, just before wrapping up in Chapter 7.

Regarding the aforementioned list of problems:

Complexity It turned out that through adopting the language of Cardinality Analysis to describe analysis results of Call Arity (cf. Section 3.4), we could leverage the pre-existing arity analysis to handle (the bulk of) η-expansion for us. We also integrated co-call graphs into the analysis and measured their impact on analysis results and performance in Chapter 5. Integrating the analysis directly with the Demand Analyser was considered at first, but quickly rejected, because of the far reaching consequences of what would have happened to be a rewrite. We also advocate that this was for the better from a perspective of separation of concerns as outlined in Section 2.4.2.

Let bindings Current analyses within GHC perform fixed-point iteration directly along the syntactic structure of the expression to analyse. Effectively, every analysis defines its own data-flow iteration algorithm, interleaved with actual analysis logic. Driven by the problems described in Section 4.8 we propose a novel, graph-based iteration method that separates denoting the transfer function to syntactic elements from actually computing the fixed-point of the modelled data-flow problem. In doing so, we can improve precision of analysis information unleashed at call sites, approximating a polyvariadic analysis. We also elaborate on the Pros and Cons of LetUp vs. LetDn in Sergey et al. [17] and prove that, if it wasn't for losing shared work, LetDn is strictly more precise than LetUp even in the presence of co-call graphs at the end of Section 3.3.6.

Data structures We realised that the analysis information Call Arity provides can be recovered by doing arity expansion based on usage information as explained in Section 3.4. This allowed us to get rid of the interleaving of arity und sharing analysis within Call Arity, which finally led to full generalisation of both analyses.

2 Preliminaries

This chapter will start out by laying out some ground work for later sections.

There seems to be no ubiquitous domain language on the various cardinality analyses floating around in papers over the years [9, 20, 18, 7, 12, 3, 22, 14, 17], so Section 2.1 will provide a glossary for that.

We'll give abridgements of the two analyses we aim to generalise in Section 2.3 and Section 2.4.

2.1 Analysis Zoo

Prior work disagrees in what meaning they assign to different concepts related to analyses that track evaluation counts in some way.

Just to name an example, the concept of *demand* within GHC refers to a pair of strictness and usage information, while Wansbrough [22, Appendix C.2] defined demand as evaluating a binding to weak head normal form (WHNF), as opposed to applying the bound expression to one argument (*use*).

2.1.1 Cardinality Analysis

A *cardinality analysis* answers questions regarding how often some syntactic thing is used with respect to a single evalutation of the outer context.

Let us understand this by examining the following example:

```
main = do
  let a = ...
      b = ...
      c = ...
      d = ...
      e = ...
  print (a + if b then a*d else c*c)
```

A sophisticated compiler for a lazy language can find out the following facts, always assuming a single execution of main:

- The binding for a is evaluated at least once.
- The binding for b is evaluated exactly once.
- The binding for c is either evaluated twice or not at all.

- The binding for d is either evaluated once or not at all.
- The binding for e is absent, e.g. not used even once.

Based on these facts, the compiler can apply a number of optimisations:

Call-by-value Since a and b are evaluated *at least once*, the compiler is free to apply a call-by-value evaluation strategy for them. Recovering strictness in this way makes a huge difference, as subsequent transformations such as a worker/wrapper transformation [5] may exploit this information to great effect.

Call-by-name Because b and d are evaluated *at most once*, the compiler can employ a call-by-name strategy instead of call-by-need. Operationally, this omits unnecessary thunk updates for these so-called *single-entry* thunks, because the computed value doesn't need to be memoised.

Absence The example doesn't mention any use of e. Such absence can be exploited by not generating code for the binding at all, or replace the binding by an error message in case of analysis failure. Absence is also important for the worker/wrapper transformation [5], in that it conjures custom calling conventions that won't need to mention (partially) absent arguments at all.

Of course, call-by-value and call-by-name are mutually exclusive, which means that for b the compiler must choose between the two. In practice, that is an easy choice: Call-by-value enables much more effective optimisations than call-by-name. Nonetheless, the share of single-entry thunks is dominating (70% according to same dated results of Marlow [12]) and deemed worth optimising.

Note that we only care for the three cardinalities $\{0, 1, \omega\}$ in our examples: Everything beyond the second evaluation carries no usable information, thus we denote the cases of 'evaluated multiple times' with ω.

As in the above example, possible cardinality may also depend on runtime information, so that information is better reflected as a subset of $\{0, 1, \omega\}$. Most interesting, however, are the over-approximating (e.g. maximum) and under-approximating (e.g. minimum) cardinalities, which act as proofs for the compiler to justify said optimisations.

Thus, a cardinality analysis assigns each binding an interval of its maximum und minimum evaluation cardinality, relative to a single evaluation of the binding site.

In the example above, a would be annotated with $[1, \omega]$ (evaluated at least once, possibly many times), whereas the absent e would be annotated with $[0, 0]$ (evaluated at most never).

This is exactly the notion of usage-interval analysis in Sestoft [18, Chapter 5], which defines 'usage count' as what we call cardinality.

2.1.2 Strictness Analysis

Analogous to the distinction of alias analyses between MayAlias and MustAlias in typical imperative languages, cardinality analysis can be split in two separate passes,

MinCard and MaxCard. Looking at the MinCard problem, the only information that is exploited by compilers so far is that of *strictness*.

An expression ... **let** x = e **in** body ... is strict in the binding x if the whole expression diverges whenever e does. Put another way: If an expression is *strict* in some binding x, the expression will certainly evaluate x on all code paths, e.g. x is evaluated at least once.

Finding out whether or not a binding is evaluated at least once, relative to a single evaluation of the binding expression, is the goal of *strictness analysis*. Strictness analysis enables the call-by-value optimisations explained above and caters for the worker/wrapper transformation [5], sketched out in Section 2.2.

Note that from a MinCard perspective, there is at least one more bit of information that would also be of interest, namely if a binding is evaluated *at least multiple times* (e.g. annotation $[\omega, \omega]$). There is no real gain for compilers in knowing that information! This leads to delightful simplicity in the implementation of strictness analysis compared to MinCard or MaxCard in the case of thunks (cf. Section 2.4.2).

2.1.3 Usage Analysis

Strictness analysis captures all necessary information on MinCard, e.g. if α in the annotation $[\alpha, \beta]$ is 0 or 1.

We refer to the analogue of MaxCard as a *usage analysis*. A usage analysis provides an over-estimate to cardinality. For a given binding, it finds out *at most* how often the binding is evaluated in a single evaluation of the binding expression.

Both *absence analysis* (e.g., is β in the cardinality annotation $[\alpha, \beta]$ at most 0?) and *sharing analysis* (e.g., is β in the cardinality annotation $[\alpha, \beta]$ at most 1?) are generalised by usage analysis.

The results of a sharing analysis support the call-by-name optimisation from above, while absence information is needed together with strictness information for the worker/wrapper transform.

As Verstoep and Hage [21, Section 2.4] point out, a sharing analysis finds out similar results as a static analysis based on *uniqueness types*. They serve different purposes, however; Uniqueness information is propagated during type-checking and may affect which programs are rejected, while sharing analysis is an enabling analysis for other optimisations in the middleend.

That also means that the benefits of uniqueness types might carry over to thunks that a sharing analysis finds to be single-entry, just by changing their type after it passed type-checking. Hage et al. [7] give an overview over commonalities and differences of sharing analyses and uniqueness types and provide an analysis generalising both.

Less far-fetched is the benefit of identifying *one-shot* lambdas. Roughly speaking, a lambda is one-shot if a single evaluation of the expression that reduces to the lambda leads to at most one call of the lambda.

This is best understood by an example.

Example. Consider the function f in the following expression:

```
let f x y = m*x + y
in f 1 2 + f 3 4
```

The outer lambda, binding x, is not one-shot: As the work of evaluating the expression bound to f to WHNF (which it trivially has) is shared between the two calls to the lambda.

It is different for the inner lambda, binding y, however. In each of the two calls, the result of applying f to one argument is immediately applied to another argument. The redexes f 1 and f 2 represent the relative evaluations here and in each case the resulting lambda is called once.

Thus, the lambda which binds y is one-shot.

Recognising one-shot lambdas opens up opportunities for a number of further optimisations [22, Section 6.6.2]:

Floating Normally, floating a **let** binding inside a lambda risks duplicating shared work. One-shot lambdas guarantee that the body will not be evaluated more often than the containing expression, so floating bindings inside is safe.

Note that in the above example, m could not be floated inside the body of f. Although the inner lambda (y) is one-shot, the outer isn't.

η-expansion Instead of floating **let** bindings (and other syntactic things) inside a one-shot lambda, we can go the other way and float inner one-shot lambdas *out*. This process is called *η-expansion*, as opposed to η-reduction. η-expansion is not generally safe for ordinary lambdas for the same reasons as floating **let** bindings in is not, e.g. possible duplication of work.

η-expansion based on usage information is the key idea behind the efforts of Breitner [3] of making foldl a good consumer for list fusion.

Inlining Inlining bindings under a lambda risks duplication of shared work, which is why the inliner needs additional confirmation that the chain of lambdas under which to inline is one-shot. There is large overlap with the float in case, but remember that inlining may be beneficial in cases where a binding cannot be floated in.

All these opportunities boil down to the compiler being cautious not to duplicate work when shoving something under a lambda. It seems reasonable to annotate one-shot lambdas while performing usage analysis as the information falls off as a by-product anyway.

2.1.4 Arity Analysis

We close by characterising *arity analyses*, a concept unrelated to cardinality on first sight.

An arity analysis annotates bindings with the number of value arguments they can be applied to before doing any non-negligible work.

The simplest possible arity analysis would just count the leading chain of lambdas in the bound expression to ascribe bindings with this *manifest arity*.

```
let f b =
      if b
      then id
      else (*2)
in f True 2
```

Here, f has manifest arity 1. While this starts out as a rather manageable analysis, GHC's arity analysis is much more involved, having to deal with coercions, instrumentation and cost models.

In the above example, GHC would consider the **if** expression matching on a variable cheap to duplicate and thus *expand* f's *arity* (by η-expanding its bound expression) to two.

2.2 Worker/Wrapper Transformation

Discussing a transformation in a chapter that is all about analyses seems out of place, but we refer to the worker/wrapper transformation quite often and this seems like the best place to sketch out its idea. A detailed treatment can be found in Gill and Hutton [5].

Consider the following function and its usage:

```
f (a, b) =
  <large body mentioning b>

main = print (f (4, 5))
```

There is some pointless packing involved in calling f with the pair literal: After all, f immediately unpacks the pair when called! Also, since a is not used by f in the first place, it would be enough to pass b.

If f can be inlined, this packing/unpacking immediately gets optimised away by the simplifier.

If f cannot be inlined (it may be recursive or its body too big), then the compiler has to leave the call site untouched. That's a shame, since the repeated boxing and unboxing performs needless allocation, which especially hurts in hot loops.

The common solution to this problem is to apply a *worker/wrapper split* to the function:

```
f (a, b) =
  $wf b

$wf b =
```

```
    <large body mentioning b>

main = print (f (4, 5))
```

This makes f only a thin *wrapper* over the actual *worker* function $wf, which only takes those components of the pair it really uses (b, that is). Now, f is really cheap and will always be inlined at call sites, so the needless boxing and unboxing vanishes. Effectively, the inlinable wrapper exposes a specialised *calling convention* for the worker function.

This works even in the face of recursion, where the wrapper can be inlined within the body of the worker to reduce allocations.

The worker/wrapper transformation is paramount to performance in a non-strict language like Haskell. In order to justify unboxing primitives like machine integers (which can be treated like 1-tuples), a strictness analysis must approve the strictness in the (generally lifted) argument to unbox. An absence analysis can identify which parts of a record (like a above) are not used at all by the function, so that the number of arguments passed to the worker is kept minimal. Without this transformation and its interplay with strictness, GHC could never reach performance within the same magnitude as C.

2.3 Call Arity

Call Arity [3] is a carefully crafted analysis within GHC, motivated by making foldl take part in list fusion without compromising in terms of allocations.

It does so by η-expanding bindings based on usage information. This is best demonstrated by an example:

```
let f x =
      if expensive
      then id
      else (∗2)
in f 1 2 + f 4 5
```

Call Arity recognises f as always being called with two arguments and justifies η-expansion of the expression bound to f to arity two. The justification for why this doesn't lose shared work of evaluating the arbitrarily expensive expression (cf. Section 2.1.3) is that there is no call with arity less than two, so that the sharing would never be observed.

This would be all there is to arity expansion (e.g. η-expansion of the bound expression) based on usage, if it weren't for thunks:

```
let f =
      if expensive
      then λx y → y
      else λx y → y∗2
in f 1 2 + f 4 5
```

This computes the same expression as the previous example, but has different operational behavior. Since f is now a thunk (e.g. binds an expression that is not in WHNF), expanding f to *call arity* – the minimum arity of each call, as opposed to Call Arity, the analysis – duplicates the work associated with evaluating expensive. This work would otherwise be shared between the two calls and unsharing it may in general degrade performance by an arbirary amount.

It is safe to η-expand thunks to call arity if they are called just once, though:

```
let f =
      if expensive
      then λx y → y
      else λx y → y*2
in if b then f 1 2 else f 4 5
```

This also demonstrates that f can't be inlined due to imminent explosion in code size.

Thus, Call Arity tracks in its abstract domain for each binder whether it was called only once and the call arity (as in minimum arity of each call). The 'called once' part is tracked by a sharing analysis, while call arity is computed by a simple arity analysis.

As the examples demonstrate, the arity analysis leans on the sharing analysis. The next example shows that information also flows in the reverse direction:

```
let f x = expensive
in f 1
```

A (rather simple) sharing analysis would analyse the binding of f and would have to assume that expensive is evaluated possibly multiple times, because it is hidden under a lambda. Under the assumption that f is only called once, with one argument (!), the sharing analysis can conclude that expensive is only used once.

Thus, the interleaving of sharing and arity analysis within Call Arity makes sense, although as we see later in Section 3.4, call arity is just the result of exploiting one-shot and single-entry information.

In order for Call Arity to have available the minimum call arity of an identifier when analysing its bound expression, it has to analyse **let** from the bottom up, e.g. analyse the body before the bound expression.

This has limitations in cases like this:

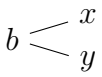
```
let x = ...
in let y = 2*x
   in if b
      then x
      else y
```

Here, the bottom-up sharing analysis will not find out that x is only evaluated once. At the point where y is analysed, it is already clear that x was evaluated, and because y was also evaluated, it seems that x was evaluated twice. That is too

conservative, however: The bottom-up scheme forgot that y and x were evaluated on different code paths.

Thus, Call Arity employs a novel sharing analysis based on *co-call graphs* instead. Co-call graphs track which identifiers are possibly evaluated with each other by an edge. The absence of a co-call edge proves that the unrelated identifiers are never evaluated on the same code path.

For the **if** expression in the above example, the co-call graph looks like this:



It properly reflects that x is never called together with y. The analysis then makes use of that information when handling the **let** expression binding y, by effectively performing substitution of the co-call graph of its bound expression (which contains the single node x without any edges) for y in the above co-call graph of the body.

After this substitution step, there will be no loop on x, because there was no edge between y and x. This represents the fact that x is not called with itself, or only called once, plainly speaking.

Co-call graphs and this rather involved substitution procedure are illuminated in detail in Section 3.2.3 and Section 3.3.6. Section 5.1 discusses the impact of co-call graphs on analysis precision and performance.

2.4 Demand Analyser

GHC's approach to strictness and usage analysis is that of *demand analysis*.

It is what we referred to as a cardinality analysis (Section 2.1.1), integrating both MinCard and MaxCard analyses. To make matters more confusing, the part concerning usage analysis is introduced in Sergey et al. [17], titled 'Modular, Higher-order Cardinality Analysis in Theory and Practice'. Later chapters refer to the usage analysis as implemented in GHC's Demand Analyser as Cardinality Analysis, in title case, to disambiguate it from the notion of cardinality anaysis in Section 2.1.1.

On top of producing cardinality results, the Demand Analyser also performs *constructed product result analysis* (CPR) [1], the benefits of which are reaped in the worker/wrapper transformation.

2.4.1 Implementation

Like Call Arity, the Demand Analyser works by abstract interpretation. The abstract domain is however quite different from that of CallArity: It is a product lattice of strictness and usage information (e.g. lower and upper bounds of cardinalities).

We focus on the usage analysis (Cardinality Analysis) here, which aims to identify single-entry thunks, one-shot lambdas and absent bindings [17, Section 2] for the reasons we explained in Section 2.1.1. They achieve remarkable precision by a number of interesting features, which describe not only the maximum cardinality of a binding,

but also *how* a syntactic thing was used. We borrow the same language for our usage analysis, so the curious reader can refer to Section 3.2 for a formal definition.

Call Uses

Identifying one-shot lambdas requires to know about how the expression containing the lambda was used and to track that information somehow.

The analysis captures this information in *call uses* it tracks in addition to evaluation cardinality.

Consider the following expression, similar to earlier examples:

```
let f x y = ...
in f 1 2 + f 1 3
```

Here, f is called twice and while the outer lambda (which binds x) is not one-shot, the inner lambda is. Cardinality Analysis expresses that as a call usage of $\omega * C^{\omega}(C^{1}(U))$ on f, where the outer ω is the evaluation cardinality and the ω in the superscript of the first C indicates that the outer lambda is not one-shot. The inner lambda is identified as one-shot, as evident by the superscript 1 on the inner C.

Call uses enable the analysis to differentiate the converse case, where the outer lambda is one-shot, but the inner is not:

```
let f x y = ...
in let g = f 1
   in g 2 + g 3
```

This results in a call usage of $1 * C^{1}(C^{\omega}(U))$, reflecting that f was only evaluated once, because the partial application bound to g was shared.

Product Uses

Call uses would be enough to identify one-shot lambdas. Inspired by the preceding absence analysis, Sergey et al. [17] however also introduced *product uses* to track absence of parts of a structure. This is so that the worker/wrapper transformation can expose specialised calling conventions as an inlinable wrapper function (see Section 2.2 for details), while the worker itself might not be inlinable for different reasons.

Consider the following funny function:

```
funny p@(a, b) = sum [1..a]
```

Obviously, funny doesn't use the second component of its pair argument when called. In terms of the abstract domain, p is exposed to usage $1 * U(1 * U, A)$, e.g. first evaluated to WHNF and then its first component is used once according to $1 * U$, but its second component is absent, A.

A neat side-effect of borrowing this from the prior absence analysis is that this is able to capture call uses on type class methods.

Usage Signatures

There is another important ingredient to Cardinality Analysis.

Languages like Haskell make it easy to define many small functions, which effortlessly compose to model more complex logic. This becomes a burden for the compiler, as every analysis heavily relies on the inliner for good interprocedural results. Even if the inliner does a good job, there are cases where inlining isn't possible for reasons of code size or recursion.

In these cases, an analysis must provide its own interprocedural mechanism to achieve good results. Cardinality Analysis is no different, so it approximates usage behavior of functions through *usage signatures*.

As always, an example helps to get the point:

```
let const a b = a
in const True (fac 1000)
```

In this snippet, const has a usage signature of $1 * U \to A \to \bullet$, meaning that when called with two arguments, const will use its first argument once, but its second argument not at all.

This is valuable information at the call site of const, where the expression fac 1000 can be regarded as absent.

In order to have the usage signature for functions available at call sites, Cardinality Analysis analyses the function bound to const before it analyses the body of the **let**. This LetDn rule [17] is in contrast to Call Arity, where arity information dictates an information flow from the bottom up.

Thunks

The LetDn approach works quite well for functions. Things look different for thunks, though:

```
let x = ...
in let y = 2*x
   in y + y
```

Unleashing y's *usage type*, comprised of free variable usages and usage signature, at the evalation sites LetDn-style suggests that x is evaluated twice. That is not the case, because the reduction to WHNF of y's bound expression will be shared, thus x is actually single-entry.

In order not to lose precision, thunks are analysed after the usage they are exposed to is collected from analysing the body of the binding **let** expression. This bottom-up approach is backwards compared to how functions are treated (e.g. bound expression before body) and is embodied in the LetUp rule. Call Arity uses a LetUp-style approach for all bindings, because it isn't concerned with unleashing usage signatures at call sites.

2.4.2 Untangling Analyses

Demand Analysis, as implemented in GHC, integrates three interdependent analyses. We will attempt to provide insight into how information flows between strictness, usage and constructed product result analysis, as well as which other parts of the compiler depend on the produced information.

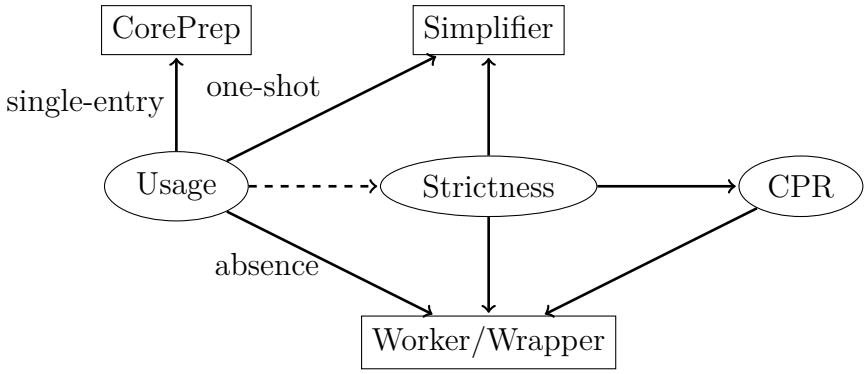


Figure 2.1: Information flow between usage, strictness and constructed product result analysis and how their results are used by different transformations within GHC.

Figure 2.1 references the three analyses in question, as well as the transformations which access the analysis results.

Usage analysis, as this thesis will prove, is pretty much a stand-alone analysis; we only rely on arity results from a prior arity analysis (part of GHC's simplifier) to be present to distinguish thunks (which have arity zero) from functions.

The dashed line between usage analysis and strictness analysis indicates that there probably is some kind of information flow, like in the form of absence information. It is hard to tell, because the interactions between usage and strictness lattice in the Demand module are quite convoluted.

Other than that, constructed product result analysis relies on strictness results to be present, as can be reproduced by following the GHC Note[1] 'CPR Example' in the DmdAnal module.

Absence, strictness and constructed product results are the key ingredients for the worker/wrapper transformation [5]. GHC's simplifier uses one-shot annotations on lambdas for a multitude of optimisations outlined earlier in Section 2.1.3. Single-entry annotations are only exploited by the backend, after the Core-to-Core-pipeline, which is represented by the CorePrep pass in Figure 2.1.

We initially tried to integrate Call Arity into the Demand Analyser, but quickly gave up on the endeavor. Changing the Demand Analyser has far-reaching consequences through the whole compiler and we were afraid to spend more time fixing regressions than to actually solve the core problems of this thesis outlined in the introduction.

[1]Notes are a documentation idiom in widespread use within GHC, to be able to refer to explanations by a title and reduce duplicate inline documentation.

In fact, we advocate splitting up the Demand Analyser in its three sub-analyses for the following reasons:

Complexity Stricter separation of concerns would definitely get rid of some of the complexity in demand analysis. Over the years, many hacks accumulated within the implementation and the intertwining means that it is not always obvious which sub-analysis is affected by them. The constant context switching is also very distracting. Also, the Demand, which models the analysis domain, has grown quite big and complex over time.

Incompatibilities There are a number of hacks to make the analyses work together.

For example, there is an extra 'virgin' iteration only to have stable strictness results available to do CPR analysis (For further details see the Note 'Optimistic CPR in the "virgin" case' in the DmdAnal module).

Also the LetUp rule is a misfit for strictness analysis. Strictness does not try to separate 'evaluated at least once' from 'evaluated multiple times' (e.g. $[1, \omega]$ vs. $[\omega, \omega]$), hence accounting for thunk sharing is not needed. As we will see later in Section 3.3.6, LetDn provides strictly better precision in some cases (Note 'Aggregated demand for cardinality'), so this is unfortunate.

There's also the issue of perfomance: The Demand Analyser always computes all analysis results, even if only results of one sub-analysis might be needed. On the other hand, the repeated AST traversals if the analysis would be split up probably incur a greater performance hit than what can be gained by being able to analyse more fine-grained. Of course, this is all just hand-waving as long as there are no measurements.

The combination of these problems led us to carve out a stand-alone usage analysis that also generalises the results of Call Arity.

3 Formal Specification

As we are heading towards implementation considerations in Chapter 4, we provide a formal specification of the usage analysis in this section.

Despite being massively more simple than Haskells surface syntax, GHC Core still captures many details inessential to the analysis. In order to keep the mathematical formulation as concise as possible, we define the transfer function in terms of a simplified language in Section 3.1.

Section 3.2 introduces the lattice of *usage transformers* the analysis will operate upon.

Finally, in Section 3.3 we will see how each language construct can be denoted by a usage transformer through a *transfer function*.

3.1 Object Language

$$
\begin{aligned}
x, y, z, (,), \mathsf{True}, \mathsf{False} &\in \mathsf{Var} \\
e \in \mathsf{Exp} &::= x \\
&\mid (x_1, x_2) \\
&\mid \lambda x.\, e \\
&\mid e\, x \\
&\mid \textbf{if } e_s \textbf{ then } e_t \textbf{ else } e_f \\
&\mid \textbf{case } e_s \textbf{ of } (x_1, x_2) \to e_r \\
&\mid \textbf{let } x_1 = e_1 \textbf{ in } e \\
&\mid \textbf{letrec } \overline{x_i = e_i} \textbf{ in } e
\end{aligned}
$$

Figure 3.1: A simple untyped lambda calculus

The formalization of the usage analysis operates on a simply untyped lambda calculus, defined in Figure 3.1.

We extend it in interesting ways to carve out particular details of our analysis. There are (possibly recursive) **let** bindings to illustrate sharing of values. We provide pair constructors, complemented with **case** expressions to destruct them.

An **if**/**then**/**else** construct will highlight how the analysis will cope with alternative branches of execution.

We draw variable names from an abstract set Var. Of particular note is the identifier (,), which always refers to the pair constructor as a function of two arguments and may not be rebound by a **let** expression. For the sake of simplicity, we mention boolean literals True and False as variables. The transfer function is unconcerned with them and can handle them like any other imported identifier.

As is customary in both Sergey et al. [17] and Breitner [3], we assume A-normal form [16], so that arguments to applications can only mention identifiers. Applications to non-trivial arguments like $e_1\, e_2$ must be rewritten as **let** $x_2 = e_2$ **in** $e_1\, x_2$, hence issues concerning sharing only surface while handling **let**-expressions.

Most code examples will be written in a Haskell-like language anyway, rather than in this stripped-down language. If there are code samples in object language (distinguishable by the typesetting of binders, e.g. x vs. x), they should implicitly assumed to be rewritten according to these rules.

3.2 Analysis Domain

The analysis, defined by the transfer function in Section 3.3, will denote each syntactic expression with a *usage transformer*. This section introduces them from the ground up by defining the relevant concepts.

3.2.1 Expression Use and Identifier Usage

A usage analysis (in the sense of Section 2.1.3) approximates how a use of an expression translates into a use of its subexpressions. Important analysis information includes [17]:

1. How many times is the body of a lambda expression evaluated, with respect to its defining scope?
2. How many times is a particular thunk evaluated, with respect to its defining scope?
3. Which components of a syntactic expression are never used, that is, absent?

As we saw in Section 2.1.1, a usage analysis computes upper bounds on these cardinality questions. The answer to the first question above then consists of identification of *one-shot* lambdas, e.g. the cases where the body of the lambda expression is called at most once, relative to its defining scope. Similarly, the second question is answered by marking thunks which have the *single-entry* property, e.g. thunks that are evaluated at most once relative to an evaluation of its binding expression. Absence is a matter of answering the question if a binding is used at all, e.g. at most zero times.

Cardinality Analysis [17] – the usage analysis that GHC's Demand Analyser integrates – provides answers to these questions. Our analysis builds on this approach in that the central lattices have been borrowed, but have been enriched and distinguished with further semantic meaning.

Figure 3.2 depicts the Use and the Usage lattice, giving the language we used in Section 2.4 proper meaning. Typical for a sharing analysis, Multi captures usage multiplicity, of which the only interesting values in our setting are *at most once* (1) or *possibly multiple times* (ω).

Semantically, a Use describes how the value of an expression is used, after evaluating it to weak head normal form (WHNF) *exactly once*.

- The pair use $U(u_1^*, u_2^*)$ captures the Usage of pair components when the expression evaluates to a pair.

- A call use $C^n(u)$ suggests that the expression evaluates to a lambda expression. Additionally, the resulting value was called at most n times *relative to the single reduction to WHNF* and the use on the result of each call was not worse than u.

- If the value of an expression wasn't used beyond reduction to WHNF, we can attest the expression a head-use HU, the bottom of the lattice defined by $\sqcup$. Through the first non-syntactic equality in Figure 3.2, HU is identical to a pair use of $U(A, A)$.

  This arises naturally when, beyond reduction to the pair constructor, no components of the value were used. Other than that, HU can only be unleashed on the first argument of a call to the binary primitive seq. Since seq can be applied to arguments of any type, expressions of function type can also be head-used.

  When a function expression is in head-use, evaluation will stop immediately after uncovering the outer lambda, the body will not be used. This corresponds to a hypothetical ill-typed use $C^0(\_)$, meaning the lambda is called 0 times relative to the single reduction of the function expression to WHNF.

- The most conservative analysis result would be the top of the lattice U, representing an unknown use beyond reduction to WHNF. Where this differs from head-use is best understood in terms of the attached equalities in Figure 3.2.

  For an expression that evaluates to a pair, unknown use would correspond to the pair use $U(\omega * U, \omega * U)$, e.g. the pair components have the worst possible usage. An unknown use on a function expression can be interpreted as a use of $C^\omega(U)$ by the last equality: We don't know how the resulting function will be used, so we have to assume multiple calls, each with unknown use.

The syntax for call uses $C^n(u)$ allows to model *multiple* calls of a function expression relative to a *single* reduction to WHNF. Yet it is still unclear how expressions can

$$
\begin{aligned}
u \in \mathsf{Use} &::= HU \mid U \mid C^n(u) \mid U(u_1^*, u_2^*) \\
u^* \in \mathsf{Usage} &::= A \mid n * u \\
n \in \mathsf{Multi} &::= 1 \mid \omega
\end{aligned}
$$

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

$$
\begin{aligned}
\boxed{u_1 \sqcup u_2 = u_3} \\
U \sqcup u &= U \\
u \sqcup U &= U \\
HU \sqcup u &= u \\
u \sqcup HU &= u \\
C^{n_1}(u_1) \sqcup C^{n_2}(u_2) &= C^{n_1 \sqcup n_2}(u_1 \sqcup u_2) \\
U(u_1^*, u_2^*) \sqcup U(u_3^*, u_4^*) &= U(u_1^* \sqcup u_3^*, u_2^* \sqcup u_4^*)
\end{aligned}
$$

$$
\begin{aligned}
\boxed{u_1^* \sqcup u_2^* = u_3^*} \\
A \sqcup u^* &= u^* \\
u^* \sqcup A &= u^* \\
n_1 * u_1^* \sqcup n_2 * u_2^* &= (n_1 \sqcup n_2) * (u_1^* \sqcup u_2^*)
\end{aligned}
$$

$$
\begin{aligned}
\boxed{n_1 \sqcup n_2 = n_3} \\
\omega \sqcup n &= \omega \\
n \sqcup \omega &= \omega \\
1 \sqcup 1 &= 1
\end{aligned}
$$

Figure 3.2: Syntax of expression Use and identifier Usage with non-syntactic equalities and the definition of the least upper bound operator $\sqcup$.

be used in such a way – we would need to *share* the work done by reducing the a function expression to WHNF, so that we can call the resulting value at least twice. Such sharing can be introduced by binding an expression to an identifier. By referring to the bound expression through the identifier, we can call the same expression multiple times, while the reduction of the bound expression to WHNF only happens (at most) once. Generally, there are various ways to bind an expression to an identifier. However, our simple object language was carefully crafted so that sharing is always introduced through **let** bindings. Other binding mechanisms like lambdas and **case** just alias another binding.

It is inappropriate to model usage of an identifier with Use, as identifiers can syntactically occur more than once, or even be absent. Yet every use site, or *usage*, puts the identifier's bound expression under a Use. Thus, every identifier has an associated Usage, which describes how often and how, if at all, the identifier was used.

An identifier Usage of A represents *absence* of any usage. Right-hand sides of absent bindings are dead code and never used. A Usage of $n * u$ represents that the associated identifier was used at most n times, putting its bound expression under combined use u.

A few examples are instructive in understanding the definitions so far.

Example. Consider the following Haskell expression:

```
let a = f 0
in let b = f 1
   in case (a, b) of
        (x, y) → y
```

If we put the expression under some use U, we find out the following facts by backtracing evaluation:

- The pattern match variable y has usage $1 * U$, x is absent (A).
- The usage on pattern match variables translate to a pair use on (a,b) of $U(A, 1 * U)$.
- The pair constructor forwards its component usages to identifier usages for a (A) and b ($1 * U$).
- Since we recorded a usage of $1 * U$ for b, we put the right-hand side of the binding under use U. One call with a single argument results in a usage of $1 * C^1(U)$ on f.
- We recognize that the binding for a is dead, so we don't need to look at its right-hand side at all.

We just saw a single call usage $1 * C^1(U)$ on a free variable f. The following expressions hint at the fact that, in general, right-hand sides of bindings can be

put under call uses $C^{n_1}(C^{n_2}(\ldots C^{n_k}(U)\ldots))$ for any $n_i \in \mathsf{Multi}$. This is through exploiting currying and sharing through bindings.

Example. Consider the multi-call case first:

```
f 0 0 + f 1 1
```

This demonstrates how multi-call uses can be introduced. A use of U on the expression manifests in two sequential usages $1 * C^1(C^1(U))$ of `f`, each corresponding to a single call with two arguments. These combine to an identifier usage of $\omega * C^\omega(C^1(U))$: After all, the lambda expression, which the function bound to `f` reduces to, is called twice, resulting in two (non-shared!) values. In each case, these function values are immediately called with one argument.

The following snippet illustrates how multi-calls can be shifted further down the chain of call uses:

```
let g = f 0
in g 0 0 + g 1 0
```

When this expression is put under use U, the subexpression `g 1 0 + g 1 0` will expose `g` to a usage of $\omega * C^\omega(C^1(U))$, similar to the previous situation. The expression bound to `g` is thus used with $C^\omega(C^1(U))$. Itself being a partial application, the bound right-hand side will subject `f` to a usage of $1 * C^1(C^\omega(C^1(U)))$.

The Usage multiplicity $n * \_$ represents how often a given thunk is evaluated. Of course, since thunks promise that the bound expression will be reduced to WHNF at most once, after the first evaluation the value will be *memoised* and returned for each subsequent evaluation of the thunk. As Sergey et al. [17, Section 2.4] points out, the ritual around memoisation is superfluous if the thunk is not evaluated more than once anyway! In these cases of *single-entry* thunks, GHC exploits that call-by-need coincides with a leaner call-by-name evaluation strategy.

As the remarks in Sergey et al. [17, Section 2.5] point out, in a non-strict language with the `seq` primitive, it is even possible for functions to be *evaluated* multiple times, yet *called* only once. If an expression like `seq f (f 3)` is put under use U, this exposes `f` to a usage of $\omega * C^1(U)$: `f` is evaluated multiple times, yet called only once. This is the same argument that Wansbrough [22, Appendix C.2] makes with his language of *use* vs. *demand*.

3.2.2 Usage signatures

In the last section we developed vocabulary for describing how an expression is used and to which usage a used expression exposes its free variables. Things don't look so well across function boundaries, though:

```
let f k = 2 * k 5
in let g x = x + 3
   in f g
```

If this is put under the typical use U, closer look reveals that `k` is used as a continuation inside `f`, e.g. has the one-shot usage $1 * C^1(U)$. By beta reduction, this is also the usage that `f g` exposes `g` to. How do we digest that information in a way that it is available at the call site of `f`?

We follow the approach from Sergey et al. [17] and ascribe *usage signatures* to function expressions. Their definition is as follows:

$$\sigma \in \mathsf{Sig} ::= \bot \mid \top \mid u^* \to \sigma$$

Equipped with the right language, we would assign the following usage signatures to `f` and `g`:

$$\begin{aligned} \mathsf{f} &:: 1 * C^1(U) \to \top \\ \mathsf{g} &:: 1 * U \to \top \end{aligned}$$

Usage signatures also come with two non-syntactic identities:

$$\begin{aligned} \omega * U \to \top &\equiv \top \\ A \to \bot &\equiv \bot \end{aligned}$$

These hint at the fact that usage signatures can be expanded to include arbitrary many arguments (although these are usually prohibited through the type system). This is helpful in situations where we only have a usage signature to a call with less arguments available.

The usage signatures $\top$ and $\bot$ hint at the fact that there is some lattice structure, so here comes the definition for $\sqcup$:

$$\begin{aligned} \boxed{\sigma_1 \sqcup \sigma_2 = \sigma_3} \\ \top \sqcup \sigma &= \top \\ \sigma \sqcup \top &= \top \\ \bot \sqcup \sigma &= \sigma \\ \sigma \sqcup \bot &= \sigma \\ (u_1^* \to \sigma_1) \sqcup (u_2^* \to \sigma_2) &= u_1^* \sqcup u_2^* \to \sigma_1 \sqcup \sigma_2 \end{aligned}$$

In any case, $\top$ as a usage signature is always a safe conservative approximation. The opposite applies to $\bot$, which expands to an infinite sequence of absent usages. Where is $\bot$ useful? Apart from giving the lattice a least element and thus an identity to $\sqcup$, it arises in error conditions and non-termination. Specifically, GHC's primitive $\mathsf{raise}\#$ has the usage signature $\omega * U \to \bot$.

Similary, usage signatures elegantly abstract from the myriad of primitive operators in GHC. For another noteworthy example, the `seq` operator would have a usage signature of $1 * HU \to 1 * U \to \top$.

3.2.3 Free-variable graph

We saw in Section 2.3 that central to GHC's *Call Arity* analysis [3] is an undirected, non-transitive, graph datastructure, representing a called-with relationship between free variables. Such a co-call graph would contain an edge between two free variables, if it is at all possible that the variables could be evaluated *simultaneously*, in the same evaluation of their containing expression, that is. Self-edges in co-call graphs correspond to a usage multiplicity of ω, e.g. multiple evaluations.

We define co-call graphs as a symmetric binary relation on Var and syntax for talking about edges within them:

$$\gamma \in \mathsf{Graph} = \{\gamma \subseteq \mathsf{Var} \times \mathsf{Var} \mid (x,y) \in \gamma \Leftrightarrow (y,x) \in \gamma\}$$
$$x_1 — x_2 \in \gamma \Leftrightarrow (x_1, x_2) \in \gamma$$
$$N_x(\gamma) = \{y \mid x — y \in \gamma\}$$

We also use the notation $\gamma\backslash_x$ to mean the graph γ purged by any edges to x.

As teased in 2.3, the cases where co-call graphs yield better results than the usual usage analyses can be boiled down to the following example:

```
let x = ...
in let y = 2*x
   in if b
      then x
      else y
```

The co-call graph for the **if** expression crucially *does not* contain an edge x—y, since usages of both variables can never happen in the same evaluation of the expression. Knowing that x is not forced together with y[1], we avoid a self-edge on x after analysing the right-hand side of y.

Consequently, x is recognized of only being evaluated once. Other usage analyses like that of GHC's Cardinality Analysis [17] only model self-loops in the underlying co-call structure (e.g. usage multiplicity), so they would fail to recognize the absent edge between x and y and conservatively assume x to be called more than once[2].

We enrich the *usage types* of GHC's Cardinality Analysis [17] with co-call graphs in order to generalise both (c.f. Section 3.4).

3.2.4 Free-variable use environment

Another key ingredient of the analysis domain is an environment in which we track usages of free variables. While Sergey et al. [17] encode these directly with a mapping

[1] Prior to looking at the right-hand side of y is, of course.

[2] It is worth pointing out that this is only the case when y is a thunk (e.g. not in WHNF), as this imprecision pertains to the LetUp rule of Sergey et al. [17]. We discuss this in more detail in Section 3.3.6.

from free variables to usages (*multi-demands* in their language), we track usage multiplicity separately in a co-call graph (Section 3.2.3).

Thus, we define use environments as partial functions[3] with finite domain from free variables Var to the Use their bound expression is put under:

$$\varphi \in \mathsf{UseEnv} = \mathsf{Var} \rightharpoonup \mathsf{Use}$$

We write $\mathsf{dom}\,\varphi$ for the domain of the use environment (or any partial function for that matter). As with graphs, we also use the notation $\varphi\backslash_x$ to mean φ where the domain no longer contains x. Similarly, $\varphi\,|_V$ is the restriction of φ to the domain $V \subseteq \mathsf{Var}$.

To see how we can recover free-variable Usage from a combination of co-call graph and use environment, see Section 3.2.5.

3.2.5 Usage types and lookup of free-variable usage

The usage signature mechanism captures all relevant argument usage information for imported global functions. For local identifiers, there's more relevant information, however. Consider the following expression:

```
let f x = x + y
in f y
```

If put under use U, this evaluates `y` twice, e.g. exposes it to usage $\omega * U$. That is because `y` also has a use site within the function expression bound to `f`.

So, in addition to a usage signature, a *usage type* should also include what usage a function exposes its free variables to[4]. We're quite lucky, just having developed the appropriate structures to capture this!

We define usage types as triples of a co-call graph γ (Section 3.2.3) and use environment φ (Section 3.2.4) for free variables, as well as a usage signature σ to capture usage of possible arguments:

$$\theta \in \mathsf{UType} ::= \langle \gamma, \varphi, \sigma \rangle$$

Additionally, with the overloaded notation $\theta\backslash_x$ we mean θ, purged by any mentions of x in its co-call graph and use environment.

As hinted at in Section 3.2.4, we can recover free variable usage in the interplay of co-call graph and use environment, denoted by the following lookup syntax:

$$\langle \gamma, \varphi, \_ \rangle (x) = \begin{cases} A, & \text{when } x \notin \mathsf{dom}\,\varphi \\ 1 * \varphi(x), & \text{when } x — x \notin \gamma \\ \omega * \varphi(x), & \text{otherwise} \end{cases}$$

[3] Indicated by $\rightharpoonup$, following wide-spread syntax

[4] We diverge a little from Sergey et al. [17] here, which define usage types as synonymous to a usage signature.

In our above example, f has the usage type $\langle \emptyset, [y \mapsto U], 1 * U \to \top \rangle$. Notably, the co-call graph is empty, while the use environment attests y a use of U. By the rules of usage lookup, this corresponds to usage of $1 * U$ exposed on y.

In fact, a usage type describes how a use on an *expression* translates into usages on arguments and free variables. So while it is convenient to think of *identifiers* like f having a usage type, it neglects that work is shared between multiple usages:

```
let x = 5*y
in x + x
```

If put under use U, this expression exposes x to usage $\omega * U$. The usage type of x (or, rather its bound expression) exposes y to usage $1 * U$. This suggests that y will be exposed to usage $\omega * U$ in the whole expression because of the two uses of x, which is too conservative: The binding of x will share the reduction to WHNF of its right-hand side, exposing y to a single usage of $1 * U$.

We will revisit this when discussing analysis of **let** bindings.

3.2.6 Usage transformers

Talking about how an expression uses its free variables makes only sense when presuming a certain use the expression is put under.

To understand why, the following example is instructive:

```
let x = expensive 0
in let y = expensive 1
   in (x, y)
```

If put under head use HU, this would not expose x or y to any usage, thus there are no calls to expensive. When put under use U however, both x and y are exposed to usage $\omega * U$, in turn calling expensive twice. That can make a huge difference in computational work, so it is important to always include the use which an expression is put under.

So, the appropriate way to denote a syntactic expression in terms of usage information is in the form of *usage transformers*[5]:

$$\tau \in \mathsf{UTrans} = \mathsf{Use} \to \mathsf{UType}$$

When denoting expressions, usage transformers map a Use the expression is put under to a usage type that describes how the expression uses its arguments and free variables.

Usage transformers of expressions are monotone maps, in line with the intuition that the stronger the use you put an expression under, the stronger is the usage it exposes arguments and free variables to.

[5]Sergey et al. [17] define these as *generalised* demand transformers, while their notion of ordinary demand transformers best corresponds to an approximation of our usage transformers

3.2.7 Lattice Structure

Previously, we introduced constructs to which we loosely attributed the properties of a lattice, specifying how to build least upper bounds in some cases. This section is dedicated to (re-)defining join operators, so that the join-semilattice structure we rely upon in Section 3.3 is made explicit. Also, all structures are equipped with a least element referred to by the overloaded notation $\bot$. Readers familiar with Sergey et al. [17] may skip this section and Section 3.2.8 after having understood the interplay of co-call graphs and use environment for these operations.

The principal example where joins occur in real languages are when combining the analysis results of alternatives of a **case**-expression. However, since our object language from Section 3.1 only supports very simple **case** expressions with the sole purpose of deconstructing pairs, we won't see joins just there. Instead, we introduced **if** expressions for exactly that purpose: Analysis of **if**s needs to join the results of the **then** and **else** branches, so that examples can introduce joins for interesting (e.g. non-complete) co-call graphs.

Figure 3.3 lists all definitions of $\sqcup$ for the various algebras involved. The overloaded least upper bound operators $\sqcup$ for Use, Usage, Multi and Sig are mentioned again for completeness.

Definitions for bottom elements are not explicitly given, but should be obvious given the definition of $\sqcup$. For UType in particular, it is $\bot = \langle \bot, \bot, \bot \rangle = \langle \emptyset, [], \bot \rangle$.

We ommitted how UseEnv, TransEnv and UTrans are join-semilattices for brevity. The first two are partial functions ranging over join-semilattices, giving rise to a join-semilattice with $\bot = []$ themselves. For usage transformers, the join-semilattice structure is the pointwise lattice on UType, with the bottom element $\_ \mapsto \bot$.

The notation $a \sqsubseteq b$ is short for $a \sqcup b = b$ and induces the typical partial order on the join-semilattices.

Of special interest in Section 3.3 are *monotone* usage transformers, e.g. where a stronger incoming Use maps to a stronger resulting UType, and the approximations thereof.

3.2.8 Sequential Composition

Until now, we treated sequential composition rather informally. For cases like $\mathsf{x} + \mathsf{x}$ it is simple enough to see that x is exposed to usage $\omega * U$. That's less obvious for function calls: For example, the usage f is exposed to in $\mathsf{f\ 0\ 0} + \mathsf{f\ 0\ 0}$ is $\omega * C^\omega(C^1(U))$.

How can we formalize the process that combines single usages in such a manner? The answer to that question bears & (pronounced 'both') in Figure 3.4.

Both calls to f in $\mathsf{f\ 0\ 0} + \mathsf{f\ 0\ 0}$, for example, would result in a usage type of $\langle \emptyset, [f \mapsto C^1(C^1(U))], \top \rangle$. Sequential composition with itself would then yield the expected $\langle \{(f, f)\}, [f \mapsto C^\omega(C^1(U))], \top \rangle$.

Sequential composition of usage types deserves special mention with respect to co-call graphs. The sequential composition of co-call graphs is basically the union

$$\boxed{u_1 \sqcup u_2 = u_3}$$
$$U \sqcup u = U$$
$$u \sqcup U = U$$
$$HU \sqcup u = u$$
$$u \sqcup HU = u$$
$$C^{n_1}(u_1) \sqcup C^{n_2}(u_2) = C^{n_1 \sqcup n_2}(u_1 \sqcup u_2)$$
$$U(u_1^*, u_2^*) \sqcup U(u_3^*, u_4^*) = U(u_1^* \sqcup u_3^*, u_2^* \sqcup u_4^*)$$

$$\boxed{u_1^* \sqcup u_2^* = u_3^*}$$
$$A \sqcup u^* = u^*$$
$$u^* \sqcup A = u^*$$
$$n_1 * u_1^* \sqcup n_2 * u_2^* = (n_1 \sqcup n_2) * (u_1^* \sqcup u_2^*)$$

$$\boxed{n_1 \sqcup n_2 = n_3}$$
$$\omega \sqcup n = \omega$$
$$n \sqcup \omega = \omega$$
$$1 \sqcup 1 = 1$$

$$\boxed{\sigma_1 \sqcup \sigma_2 = \sigma_3}$$
$$\top \sqcup \sigma = \top$$
$$\sigma \sqcup \top = \top$$
$$\bot \sqcup \sigma = \sigma$$
$$\sigma \sqcup \bot = \sigma$$
$$(u_1^* \to \sigma_1) \sqcup (u_2^* \to \sigma_2) = u_1^* \sqcup u_2^* \to \sigma_1 \sqcup \sigma_2$$

$$\boxed{\gamma_1 \sqcup \gamma_2 = \gamma_3}$$
$$\gamma_1 \sqcup \gamma_2 = \gamma_1 \cup \gamma_2$$

$$\boxed{\theta_1 \sqcup \theta_2 = \theta_3}$$
$$\langle \gamma_1, \varphi_1, \sigma_1 \rangle \sqcup \langle \gamma_2, \varphi_2, \sigma_2 \rangle = \langle \gamma_1 \sqcup \gamma_2, \varphi_1 \sqcup \varphi_2, \sigma_1 \sqcup \sigma_2 \rangle$$

Figure 3.3: Least upper bound definitions

$$\boxed{u_1 \,\&\, u_2 = u_3}$$
$$U \,\&\, u = U$$
$$u \,\&\, U = U$$
$$HU \,\&\, u = u$$
$$u \,\&\, HU = u$$
$$C^{n_1}(u_1) \,\&\, C^{n_2}(u_2) = C^{\omega}(u_1 \sqcup u_2)$$
$$U(u_1^*, u_2^*) \,\&\, U(u_3^*, u_4^*) = U(u_1^* \,\&\, u_3^*, u_2^* \,\&\, u_4^*)$$

$$\boxed{u_1^* \,\&\, u_2^* = u_3^*}$$
$$A \,\&\, u^* = u^*$$
$$u^* \,\&\, A = u^*$$
$$n_1 * u_1^* \,\&\, n_2 * u_2^* = \omega * (u_1^* \,\&\, u_2^*)$$

$$\boxed{\varphi_1 \,\&\, \varphi_2 = \varphi_3}$$
$$(\varphi_1 \,\&\, \varphi_2)\, x = \begin{cases} \varphi_1(x) \,\&\, \varphi_2(x), & \text{when } x \in \mathsf{dom}\, \varphi_1 \cap \mathsf{dom}\, \varphi_2 \\ \varphi_1(x), & \text{when } x \in \mathsf{dom}\, \varphi_1 \\ \varphi_2(x), & \text{otherwise} \end{cases}$$

$$\boxed{\theta_1 \,\&\, \theta_2 = \theta_3}$$
$$\langle \gamma_1, \varphi_1, \sigma_1 \rangle \;\&\; \langle \gamma_2, \varphi_2, \sigma_2 \rangle = \langle \gamma_1 \sqcup \gamma_2 \sqcup (\mathsf{dom}\, \varphi_1 \times \mathsf{dom}\, \varphi_2), \varphi_1 \,\&\, \varphi_2, \sigma_1 \rangle$$

Figure 3.4: Sequential composition operator &

of both graphs, plus the complete bipartite graph between their domains. Using proper graph theoretic language, this is exactly the *graph join* of the two graphs, but we will refrain from calling it that way in order not to confuse it with the order theoretic least upper bound operator $\sqcup$ from Section 3.2.7 which corresponds to the *graph union*.

We also extend the multiplicity notation of usages to usage types to mean exactly this notion of sequential composition with itself:

$$1 * \theta = \theta$$
$$\omega * \theta = \theta \,\&\, \theta$$

We don't mention usage transformers in Figure 3.4, but assume sequential composition on them to be pointwise.

It is worth pointing out that it makes only sense to talk about sequential composition with respect to free variable usage.

When combining usage types, this means we have to decide which usage signature to return, as combining them has no justifiable meaning. We arbitrarily bias the left argument in this regard, which is important to keep in mind for the analysis.

3.3 Transfer Function

Equipped with the proper language to talk about usage, in this section we associate a denoting usage transformer to a syntactic expression of our object language Exp.

The typical way to do so is via a *transfer function* from expressions to the abstract domain the analysis operates on. *Abstract interpretation* is the underlying scheme here: We interpret an expression in the approximate semantics of usage transformers. Usage analysis has also been approached via *type inference* in the past [22], [21].

The following subsections will introduce the different cases of the central transfer function

$$\mathcal{T}[\![\_]\!]\_ : \mathsf{Exp} \to \mathsf{TransEnv} \to \mathsf{UTrans}$$

Where TransEnv is an environment mapping local **let** binders to their denoting usage transformer to be unleashed on use sites:

$$\rho \in \mathsf{TransEnv} = \mathsf{Var} \rightharpoonup \mathsf{UTrans}$$

3.3.1 Lambda Abstraction

Let's start by discussing analysis of lambda expressions:

$$\mathcal{T}[\![\lambda x.\, e]\!]_\rho\, u = \begin{cases} \langle \emptyset, [], \bot \rangle\,, & \text{when } u = HU \\ n * \langle \gamma \backslash_x, \varphi \backslash_x, \theta(x) \to \sigma \rangle\,, & \text{where } \begin{array}{l} u \sqsubseteq C^n(u_b), \\ \langle \gamma, \varphi, \sigma \rangle = \theta = \mathcal{T}[\![e]\!]_\rho\, u_b \end{array} \end{cases}$$

If the expression is only put under head use, no call happens and there is no need to analyze the lambda body. Consequently, in that case we can return the $\bot$ element of the UType lattice, reflecting no free variable usage and no use of arguments. Note this only happens in the wild in conjunction with the seq primitive. For example in seq (λx → y) 3, the lambda will be exposed to a head use, while 3 will be fully used.

In the other case, we try to approximate the incoming use u by a call use $C^n(u_b)$. We'd like that u is some kind of call use, but if you give a programmer enough rope to hang himself (e.g. unsafeCoerce) to bypass the type system, it is possible for u to be incomparable to a call use, i.e. $U(1 * U)$. In this case, the pattern match in the definition would be incomplete. Thus the slightly less precise, yet still sound notation using $\sqsubseteq$: In these cases, u always compares to the bottom element $U = C^\omega(U)$. With some more notation we could have made clear that we want the least upper bound of u that is a call use.

Having figured out u_b, the use the body is put under, we can transform this use by analysing the lambda body. The resulting usage type $\theta = \langle \gamma, \varphi, \sigma \rangle$ needs to be post-processed. For one, we need to remove the free variable x from the type, that is captured by lambda binder. We also need to prepend the usage on the captured identifier to the usage signature which we forward from the body. At this point, an implementation of the analysis would want to annotate the lambda binder with its usage $\theta(x)$. Finally, we have to multiply the usage type by the relative body usage n.

3.3.2 Application and Pairs

The application case $e\,x$ will have to handle the called expression e first to find out the usage signature and then expose x to the argument usage:

$$\begin{aligned}\mathcal{T}[\![e\,x]\!]_\rho\,u = \langle \gamma_e, \varphi_e, \sigma \rangle \;\&\; [\mathcal{T}[\![x]\!]_\rho]^*\,u^* \\ \text{where } \langle \gamma_e, \varphi_e, u^* \to \sigma \rangle \;= \mathcal{T}[\![e]\!]_\rho\,C^1(u)\end{aligned}$$

For an incoming use u we put e under a single call use of $C^1(u)$. In the resulting usage type, we match on the usage signature to get out the argument usage u^* we expose x to. Since the domain of usage transformers is concerned with expression Use, we have to provide a function lifting the domain of usage transformers to Usages. This is done by $[\_]^*\_$:

$$\begin{aligned}[\_]^*\;A &= \langle \emptyset, [], \bot \rangle \\ [\tau]^*\;(n * u) &= n * (\tau\;u)\end{aligned}$$

The syntax here is reminiscent of the Kleene star, multiplying the result of τ by the usage multiplicity. Notably, if e doesn't use its argument, then $u^* = A$ and x will be exposed to no usage at all.

Finally, we sequentially compose the result of using e with the usage type that the lifted usage transformer of x yielded for an exposure to u^*. After all, both usage types are unleashed in order to satisfy the use on $e\,x$. Note that $\&$ is left-biased with respect to the usage signature. Therefore the signature of the whole application expression is that of e, relieved by the usage of x.

The seemingly unrelated case of pair literals is handled by the same application mechanism:

$$\mathcal{T}[\![(x_1, x_2)]\!]_\rho = \mathcal{T}[\![(,)\,x_1\,x_2]\!]_\rho$$

Representative for any product type, we handle pairs as applications to a special identifier for the pair constructor, (,). As mentioned in Section 3.1, (,) is special in that it cannot be rebound (e.g. shadowed) by **let** bindings.

Although desugaring pairs in such a way is a nice way to keep the definition short, it is worth pointing out that the complexity is just delegated to the usage transformer for the (,) identifier, which will have to provide an appropriate usage signature. We will handle this in the variable case.

3.3.3 case Expressions

The limited **case**-expressions of our object language serve the sole purpose of desctructuring pairs.

We denote it with a usage transformer that makes it evident that we are dealing with a *backwards analysis*:

$$\begin{aligned}\mathcal{T}[\![\textbf{case}\ e_s\ \textbf{of}\ (x,y) \to e_r]\!]_\rho\, u &= \theta_r \setminus_{x,y}\ \&\, \mathcal{T}[\![e_s]\!]_\rho\, U(\theta_r(x), \theta_r(y)) \\ &\text{where} \\ \theta_r &= \mathcal{T}[\![e_r]\!]_\rho\, u\end{aligned}$$

Observe from the flow of information that in order to analyze the scrutinee e_s, we first have to analyze the (in general: every) case branch e_r. Case statements are Haskell's basic construct to specify evaluation order: A typical operational semantics would first analyze the scrutinee, then choose the appropriate case branch.

The denoting usage transformer will apply the incoming use to the case branch. In the resulting usage type θ_r we can look up the usage the pair component binders x and y are exposed to. Based on that, we can formulate the use on the scrutinee as $U(\theta_r(x), \theta_r(y))$.

Finally, we sequentially compose the usage type of the case alternative (purged of the case binders) with the usage type of the scrutinee under the calculated use. The left argument to $\&$ is θ_r since the resulting usage signature should be that of the case alternative, rather than that of the scrutinee.

3.3.4 if Expressions

In a more elaborate core language like that of GHC, there would be a single **case** expression handling algebraic data types with any number of cases.

However, to keep the transfer function simple and concerns orthogonal, it is favorable to separate out **case** statements with multiple branches into an **if**/**then**/**else** construct.

$$\mathcal{T}[\![\textbf{if } e_s \textbf{ then } e_t \textbf{ else } e_f]\!]_\rho\, u = (\mathcal{T}[\![e_t]\!]_\rho\, u \sqcup \mathcal{T}[\![e_f]\!]_\rho\, u) \,\&\, \mathcal{T}[\![e_s]\!]_\rho U$$

This looks like a simpler version of the transfer function on **case** expressions. Yet the usage type of **then** and **else** branches are joined together before sequentially combining with the usage type of the scrutinee. For a general **case** expression, we would join analysis results of all branches together in much the same way.

We primarily need **if** expressions for the sake of illustrative examples, with non-complete co-call graphs introduced by joining the two branches. For instance, in the object language expression **if** b **then** t **else** f there will be no edge t—f in the co-call graph.

3.3.5 Variables

For the variable case, we want the following things to happen:

- Note the single evaluation of x. We need this information for annotating identifiers, as well as analysing bound expressions later on.
- For any free variables, track to what usage the use on x exposes them to. This is important for **let**-bound identifiers.
- Tack a proper usage signature onto the usage type in the case of call uses.

We begin with an auxiliary transformer that captures the first point. This is the definition for τ_x^1, the transformer putting the variable x once under the incoming use u:

$$\tau_x^1\ u = \langle \emptyset, [x \mapsto u]\,, \top \rangle$$

Co-call graph and use environment convey that x is exposed to usage $1 * u$, while $\top$ is always a sound approximation to argument usage without knowing more about x.

With that in mind, we can have a look at the transfer function for variables:

$$\mathcal{T}[\![x]\!]_\rho = \begin{cases} \tau_{(,)}, & \text{when } x = (,) \\ \rho(x) \,\&\, \tau_x^1, & \text{when } x \in \mathsf{dom}\, \rho \\ \tau_x^1, & \text{otherwise} \end{cases}$$

Ignoring the first two cases for a moment, the default case denotes the single use of x with τ_x^1.

However for **let**-bound identifiers, which are handled by the second case, we can do better than that.

For every **let** binder, we have a usage transformer denoting its bound expression available in ρ. The (potential) call to x is approximated by unleashing that usage transformer at the call site. This is analogous to the LetDn rule in Sergey et al. [17] and comes with the same trade-off, discussed in Section 3.3.6. We still have to remember the call to x and do so by sequentially composing with τ_x^1. Recall that the sequential composition operator & is left-biased with respect to the usage signature, so the result will have the signature of $\rho(x)$ instead of $\top$.

The remaining first case handles calls to the pair constructor (,) by delegating to another auxiliary usage transformer $\tau_{(,)}$:

$$\tau_{(,)}\ u = \begin{cases} \langle \emptyset, [], \bot \rangle\,, & \text{when } u \sqsubset C^1(C^1(\_)) \\ \langle \emptyset, [], u_1^* \to u_2^* \to \top \rangle\,, & \text{when } u = C^1(C^1(U(u_1^*, u_2^*))) \\ \langle \emptyset, [], \top \rangle\,, & \text{otherwise} \end{cases}$$

As mentioned in Section 3.3.2, we desugar pair literals as binary applications to the pair constructor (,).

Information on free variable usage is uninteresting for constructors, in contrast to their usage signatures.

When the incoming use is less than $C^1(C^1(U))$ (e.g. $C^1(HU)$), we are dealing with a partial constructor application. In that case, the components of the pair are unused and we propagate back a usage signature of $\bot$.

For the second case it is illustrative to consider an application to the pair constructor like (,) x y.

If put under use $U(1 * U, A)$, we want the component usages to propagate to x and y through a usage signature announced by (,). What is the use that (,) is put under? The use on the whole application, wrapped in two single shot calls, $C^1(C^1(U(1 * U, A))$! For this specific incoming use, we want the transformer denoting (,) to have a usage signature of $1 * U \to A \to \bot$. In general, for saturated single calls with arity at least as high as the data constructor's (that of pairs is two), we can propagate the usage of the pair components in the usage signature.

The last case handles all other incoming uses with a conservative usage signature of $\top$.

3.3.6 Non-recursive Let

Resolution of **let** bindings often bears interesting design decisions of an analysis.

This analysis is no exception to that rule: It combines the ideas from co-call graphs [3] with the general approach of Sergey et al. [17].

As pointed out in Sergey et al. [17, Section 3.5], on one hand, we want to unleash calls right at their use site, as if it were inlined, to have a usage signature available:

```
let b t f = t
in b 0 (expensive 0)
```

The usage signature for b would reveal that the expensive computation is never needed. Thus, we ideally want to analyse **let** bindings in a *downward* manner, analyzing bound expressions before **let**-bodies. In Sergey et al. [17] this is accounted for by the LetDn rule.

As they suggest in section 3.6 and we alluded to in Section 3.2.5 however, this approach loses the shared work of bringing the bound expression to WHNF:

```
let x = 5*y
in x + x
```

Unleashing the usage type of x at each use site would cause the analysis to report y to be exposed to usage $\omega * U$, where in reality evaluation of the right hand side of x is shared, hence y is only exposed to usage $1 * U$.

In these cases analysis should proceed in an *upward* manner: Knowing that the **let**-body exposes x to usage $\omega * U$, we can analyze the bound expression once in use U. Remarkably, this means just one evalution to WHNF, exposing y to usage $1 * U$.

This is embodied in the LetUp rule of Sergey et al. [17].

At the point where upward analysis unleashes usage types of the bound expression, precise co-call information is no longer available. As discussed in Section 3.2.3, that is why co-call graphs are valuable. Recall the example from Section 3.2.3:

```
let x = expensive 0
in let y = 2*x
   in if b
      then x
      else y
```

The LetUp rule would assign x a usage of $\omega * U$. A co-call graph captures the mutual exclusive call relationship between x and y, so evaluation of y does not need to be sequentially composed with the other usage of x, resulting in x being exposed to the expected usage $1 * U$.

Note that the LetDn approach would not suffer from this imprecision: If the usage type of x's bound expression's evaluation was immediately unleashed at its use sites, usages would be on different branches and analysis results would be fine.

As a result, analysis of thunks (e.g. bindings which are not in WHNF) should respect sharing in a similar way to LetUp, while function calls should be treated in LetDn-style, as if calls where inlined. Therefore, we flavor the approach of Sergey et al. [17] with co-call graphs from Breitner [3] to mitigate the discussed imprecision of LetUp.

That said, here is the transfer function on non-recursive **let** bindings:

$$\begin{aligned}\mathcal{T}[\![\textbf{let } x_1 = e_1 \textbf{ in } e]\!]_\rho\, u &= (\theta \ltimes [x_1 \mapsto \theta_1])\backslash_{x_1} \\ &\text{where} \\ \tau_1 &= \mathcal{T}[\![e_1]\!]_\rho \\ \rho' &= [x_1 \mapsto \tau_1{\downarrow_{e_1}}]\,\rho \\ \theta &= \mathcal{T}[\![e]\!]_{\rho'}\, u \\ \theta_1 &= [\tau_1{\uparrow_{e_1}}]^?\theta(x_1)\end{aligned}$$

Let us start by understanding the auxiliary definitions.

The usage transformer τ_1 is the denotation of the bound expression e_1. We add this new usage transformer to the transformer environment ρ under the identifier x_1, yielding a new environment ρ' which we need to analyse e.

There is a small catch though: Actually we don't extend ρ with τ_1 directly, but a version modified by the LetDn operator $\_{\downarrow}\_ : \mathsf{UTrans} \to \mathsf{Exp} \to \mathsf{UTrans}$, which we define shortly. This makes sense: The way the transformers in ρ are unleashed in the variable case (cf. Section 3.3.5) corresponds to the LetDn rule in Sergey et al. [17].

Equipped with the extended transformer environment ρ', we can proceed to analyse the body e. We put its denotation under the incoming use u and observe a usage type θ.

This usage type has information about which usage x_1 was exposed to in the body, expressed as $\theta(x_1)$. Broadly speaking, θ_1 is the result of putting the bound expression, denoted by τ_1, under this usage for a LetUp-like treatment.

For this, τ_1 is again modified, this time by the LetUp operator $\_{\uparrow}\_ : \mathsf{UTrans} \to \mathsf{Exp} \to \mathsf{UTrans}$.

The resulting usage transformer is then lifted to the domain of Usages with $[\_]^?\_$. We already saw a similar operator, $[\_]^*\_$ in Section 3.3.2. This one lifts with a question mark, reminiscent of 'zero or one times'.

$$\begin{aligned}[\_]^?\ A &= \langle \emptyset, [], \bot \rangle \\ [\tau]^?\ (\_ * u) &= \tau\ u\end{aligned}$$

Now is a good time to introduce the LetDn and LetUp operators:

$$\begin{aligned}&\mathsf{zap} : \mathsf{UType} \to \mathsf{UType} \\ &\mathsf{zap}\ \langle \_, \_, \sigma \rangle = \langle \emptyset, [], \sigma \rangle\end{aligned}$$

$$\_\uparrow\_ \;:\; \mathsf{UTrans} \to \mathsf{Exp} \to \mathsf{UTrans}$$
$$\tau\uparrow_e = \begin{cases} \mathsf{zap} \circ \tau, & \text{when } e \text{ is in WHNF} \\ \tau, & \text{otherwise} \end{cases}$$

$$\_\downarrow\_ \;:\; \mathsf{UTrans} \to \mathsf{Exp} \to \mathsf{UTrans}$$
$$\tau\downarrow_e = \begin{cases} \mathsf{zap}, & \text{when } e \text{ is in WHNF} \\ \mathsf{zap} \circ \tau, & \text{otherwise} \end{cases}$$

The zap function deletes any information about free variables from a usage type. The LetDn operator zaps free variable information exactly when the LetUp operator does not, so we can be sure that for every **let**-bound identifier, we either unleash free variable information in LetUp or LetDn fashion, but never both.

When do we unleash LetUp-style? For those identifiers whose bound expression e is not a value. Conversely, if the bound expression e is already a value, usage information on free variables will be unleashed at use sites, LetDn-style.

With the ancillary definitions explained, we visit the final expression defining the denoting transformer. The expression $\theta \ltimes [x_1 \mapsto \theta_1]$ sequentially composes θ with θ_1. However it relates only a subset of the free variables of θ to those of θ_1 with co-call edges. This composition step is critical to the precision of Call Arity [3], where resolution with co-call graphs is first described. Only the neighbors of x_1 in θ can be co-called with everything from θ_1, other co-call edges are ommitted. The $\_ \ltimes \_$ operator can be thought of as a substitution operator for variable graphs. In $\theta \ltimes [x_1 \mapsto \theta_1]$, the operator will substitute every mention of x_1 by its associated usage type θ_1 and inherit all relevant co-call edges in the process.

We see in a minute an example that clears things up, but let us first have a look at the definition of this substitution operator:

$$\begin{aligned}
_ \ltimes _ &: \mathsf{UType} \to (\mathsf{Var} \rightharpoonup \mathsf{UType}) \to \mathsf{UType} \\
\theta \ltimes [] &= \theta \\
\theta \ltimes [x_i \mapsto \langle \gamma_i, \varphi_i, _ \rangle]\, \mu &= \langle \gamma, \varphi, \sigma \rangle \\
&\quad \text{where} \\
\langle \gamma', \varphi', \sigma \rangle &= \theta \ltimes \mu \\
N &= N_{x_i}(\gamma') \\
\gamma &= \gamma' \sqcup \gamma_i \sqcup (N \times \mathsf{dom}\, \varphi_i) \\
\varphi &= \varphi' \sqcup \varphi_i \sqcup (\varphi' \upharpoonright_N \;\&\, \varphi_i)
\end{aligned}$$

Apart from recursing over the entire finite map in the second parameter (which makes it a well-defined function), it works very much like the sequential composition operator $\&$ on usage types from Section 3.2.8.

The substitution operator is more precise than plain sequential composition by selecting *only a subset* N of variables from $\mathsf{dom}\, \varphi'$ which are to be sequentially composed with $\mathsf{dom}\, \varphi_i$.

As Breitner [3] proves, it is safe to choose only the neighbors of x_i in the co-call graph for this: All other variables are not actually evaluated together with x_i, so we should not need to sequentially compose them.

Mind that the substitution operator will not actually delete x_i from the usage type, so the transfer function must do so after substitution. The reason for this is anticipation of recursive binding groups, where the fixed-point iteration relies on binder usage from the last iteration.

The substitution operator is strongly related to *composition graphs* $G_0[G_1, \ldots, G_n]$ as introduced in Golumbic [6, pp. 109], where the n vertices in the *outer factor* G_0 are substituted by the *inner factors* $G_1, \ldots, G_n$.

To wrap this up, we visit some examples contrasting LetUp style analysis with LetDn, discussing strengths and weaknesses.

Example. Consider the following non-closed Haskell expression:

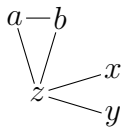
```
let x = 5*a + 3*b
in  if z
    then x
    else y
```

We focus on how the **let** binding will be resolved. For the sake of the example we can assume that all mentioned identifiers have bound non-function expressions which are not in WHNF, thus usage types are unleashed LetUp style.

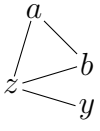
For an incoming use of U, all particular uses in this example are U and all usage signatures are $\top$. Only the co-call graphs are interesting.

Analysis of the **if** expressions yields the usage type θ. It reveals that x is exposed to usage $1 * U$, so its bound expression is put under use U. This results in a usage type θ_1 for the bound expression. The associated co-call graphs γ and γ_1 are the following:

The substitution step will substitute γ_1 into x and immediately flatten the resulting hypergraph:



Finally, the transfer function will delete all mentions of x from the usage type, resulting in the final co-call graph



Note that a and b are not co-called with y, as expected.

Example. For another example that focuses on the LetDn rule instead of the LetUp rule, consider the following slightly modified variant from a moment ago:

```
let f x = 5*a + 3*b
in if z
   then f y
   else y
```

This is the nearly same example as before, only the work of the bound expression is now hidden behind a lambda. So the expression is in WHNF and f will be unleashed according to LetDn.

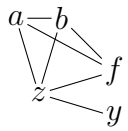
When analysing that **let** binding, we extend the transformer environment with the usage transformer τ_1, denoting the expression bound to f, to yield ρ'.

Analysis proceeds with the **let** body. The call site of f entails a lookup in ρ' and will unleash the usage type (cf. the variable case in Section 3.3.5)

$$\rho'(f)\, C^1(U) \,\&\, \tau_f^1\, C^1(U) = \left\langle f \overset{a}{\underset{b}{<}}, \left[f \mapsto C^1(U), a \mapsto U, b \mapsto U \right], A \to \top \right\rangle$$

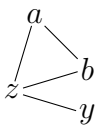
The usage signature conveys that f does not use its argument, so the call does not use y at all.

That means the whole **if** expression will have the co-call graph



This differs from the prior example's graph only in that there are co-call edges between f and the free variables of its bound expression. These are irrelevant, however, as for non-recursive **let** these can never introduce loops on f. After all, how would f's bound expression call itself?

The final co-call graph after deleting f is exactly the same as in the preceding example:



This exemplifies that in the case of functions, LetDn is as precise as LetUp with co-call graphs.

Example. We even found a minimal counter-example where LetUp with co-call graphs yields worse results than an approach based on LetDn.

Let's investigate the following definitions:

```
let f =
      if expensive
```

```
        then id
        else (*2)
in  let x = f 0
    in  if b
        then f `seq` x
        else f 0
```

The expression bound to x is a thunk, so the binding is unleashed LetUp-style.

Knowing that the usage signature of seq is $1 * HU \to 1 * U \to \top$, the usage type of the inner-most **if** expression is

$$\left\langle f\text{—}x, \left[f \mapsto C^1(U), x \mapsto U\right], \top \right\rangle$$

Notice how there is a co-call edge from f to x due to the **then** branch, but also that f exposed to usage $1 * C^1(U)$ because of the **else** branch. Substituting the usage type of its bound expression for x will then expose f to usage $\omega * C^\omega(U) = \omega * U$ because of the existing co-call edge.

That's clearly imprecise: f is only ever called once, even if in the **then** branch it is brought to WHNF (e.g. put under head use HU) and then called by evaluation of x, resulting in a usage of $\omega * C^1(U)$.

Indeed, that is what we would find out if we unleashed x LetDn-style. As should be clear by now, we don't do this because in general there might be multiple calls to x and the work of bringing down its bound expression to WHNF is shared.

A similar scenario can be elicited by using pair uses instead of call uses.

3.3.7 Recursive Let

Section 3.3.6 was dedicated to explaining analysis of **let** bindings in great detail. In particular, the distinction between functions and thunks and how substitution in co-call graphs works are delicate technicalities.

This section discusses how to extend the analysis to recursive **let** bindings.

As usual, this will involve fixed-point iteration over the analysis lattice of usage transformers. However, showing that a fixed-point exists is not trivial, as the domain formed by usage transformers has infinite height.

Repressing all thoughts on the termination problem ahead, we postulate existence of a fixed-point combinator **fix**, satisfying the following typical equation for all f we possibly pass to it:

$$\mathbf{fix} f = f(\mathbf{fix} f)$$

Where exactly do we need to apply fixed-point iteration?

For one, in order to recurse, the bound expression must have access to its own usage transformer τ_1 through the transformer environment ρ', so some mutual entanglement caused by LetDn-style analysis can be expected.

Moreover, the usage type θ from Section 3.3.6, which is used to look up usage of the bound identifiers within their scope for LetUp-style resolution, only captures

usage from the body. That was fine as long as only the body represented the entire scope for the newly bound identifiers, but now that bound expressions may refer to their own binding group. That's why the substitution operator does not delete substituted identifiers from usage types: The iterated usage type will represent the usage in the scope of the binding group.

Without further ado, the transfer function on recursive bindings is defined as

$$
\begin{aligned}
\mathcal{T}[\![\textbf{let } \overline{x_i = e_i} \textbf{ in } e]\!]_\rho\, u &= (\textbf{fix}\,\mathsf{up})\backslash_{\overline{x_i}} \\
&\quad \text{where} \\
\mathsf{up}\,\theta &= (\mathcal{T}[\![e]\!]_{\rho'}\, u \sqcup \theta) \ltimes \left[\overline{x_i \mapsto \theta_i}\right] \\
\overline{\theta_i} &= \overline{[\mathcal{T}[\![e_i]\!]_{\rho'} \uparrow_{e_i}]^?(\theta(x_i)\,\&\,\theta(x_i))} \\
\rho' &= \textbf{fix}\,\mathsf{down} \\
\mathsf{down}\,\rho' &= \left[\overline{x_i \mapsto \mathcal{T}[\![e_i]\!]_{\rho\sqcup\rho'} \downarrow_{e_i}}\right] \rho
\end{aligned}
$$

Where the dependency chain of auxiliary definitions flows upwards, e.g. down can be defined without looking at any other auxiliary definition.

Let's comprehend it in exactly that order.

We perform the fixed-point iteration that yields the extended transformer environment ρ' first, by iterating with down. Note that compared to Section 3.3.6, we had to abandon τ_1 (τ_i, hypothetically), the transformer denoting the bound expression. Instead, the definition of τ_1 was unfolded at its two use sites within down and θ_i. Also, since fixed-point iteration potentially begins with $\bot$, we have to make sure to pass the least upper bound $\rho \sqcup \rho'$ to the bound expression's transfer function within down.

Given a stable transformer environment ρ', we can compute the outer fixed-point of the up function.

Actual LetUp-style substitution is done in up. The substitution step changed only in that we substitute into the usage type of the last iteration θ, joined with the usage type of the body for the first iteration.

As mentioned above, θ represents the usage type of the entire scope of the binding group. Thus, the bound expressions can be analysed according to the usage θ reports. Looking at the definitions for θ_i, identifier usage of x_i within its scope is sequentially composed (&) with itself immediately.

That is a conservative approximation, but a tractable one: As Breitner [3, pp. 102–104] points out, one-shot recursive bindings are rare enough to neglect this case and co-call information for the recursive thunk case is flawed anyway.

When iteration of up finally reached a fixed-point, all that remains is to strip off the identifiers from the resulting usage type.

Existence of the Fixed-point

As noted at the begin of Section 3.3.7, proofs for the existence of fixed-points in the two occurences of **fix** are non-trivial. At the same time, a rigorous proof is out of scope for this thesis (with respect to both extent and time), so this section will just provide a sketch. This is so that we motivate and foreshadow implementation challenges in Chapter 4.

As the Kleene fixed-point theorem states, iterating a Scott-continuous function $f\colon D \to D$ over a directed-complete partial order $(D, \sqsubseteq)$ starting at a bottom element $\bot$ always converges towards the least fixed-point:

$$\mu\, f = \bigsqcup \{f^n \bot \mid n \in \mathbb{N}\}$$

By giving usage transformers the structure of a directed-complete partial order and proving the iterated functions up and down Scott-continuous, we can identify **fix** with μ and recover well-definedness.

As noted in Section 3.2.7, usage transformers form a join-semilattice with least element $\_ \mapsto \langle \emptyset, [], \bot \rangle$. For usage transformers to form a directed-complete partial order, it is enough to show completeness of the semilattice, which transitively reduces to completeness of the UType join-semilattice and all other mentioned substructures.

However, pair uses don't even form a directed-complete partial order, as the following arbitrarily ascending chain proves:

$$A \sqsubset 1 * U(1 * U, A) \sqsubset 1 * U(1 * U, 1 * (1 * U, A)) \sqsubset \ldots \sqsubset (1 * U(1 * U, \_))^n\, A$$

In practice, and that's also what the implementation of Sergey et al. [17] within GHC does, we have to artificially bound the depth of pair uses to reach a fixed-point.

We will revisit this in Section 4.5, where we discuss implementation details regarding efficient approximation of usage transformers.

Postulating a directed-complete partial order on usage transformers (e.g. by bounding the depth of pair uses), what remains to be proven is the Scott-continuity of the iterated functions up and down. Scott-continuity implies monotonicity, but it is not obvious how up and down are monotone.

Because the up function looks up identifier usage from the usage type, monotonicity requires that a stronger usage leads to a stronger usage type unleashed by the bound expression.

In other words: Monotonicity of up requires that the transfer function ranges only over *monotone* usage transformers. In fact, termination bugs regarding monotonicity actually occured while realising the implementation due to the chosen data structures (cf. Section 4.9).

The proof for Scott-continuity of up would follow a similar argument, so we also need to prove continuity of all usage transformers in the image of the transfer function.

Of course, just showing existence of the fixed-point does not imply termination, let alone any guarantees with respect to runtime complexity. Section 4.6 discusses efficient approximation of usage transformers in more detail.

Instead of a proper proof of safety in the style of Sergey et al. [17] or even Breitner [3], we trust in benchmark results (cf. Chapter 5) to have uncovered possible problems.

3.4 Relationship to Call Arity and Demand Analysis

As we jumped through quite some hoops in this chapter to incorporate co-call graphs into an approach resembling usage analysis in GHC's Demand Analyser, we now look at how the combined usage analysis in Section 3.3 relates to Call Arity and the Demand Analyser.

We will begin by looking at how call arity (the analysis result of the Call Arity analysis) can be recovered by a combination of arity analysis and η-expansion based on Use. Subsequently, we show how Call Arity arises as a special case of our usage analysis.

For the part of GHC's Demand Analyser that is concerned with usage information [17] the connection is more obvious, since there is no need to translate between analysis domains.

3.4.1 Call Arity and η-Expansion

Call Arity [3] presents itself as an arity analysis interleaved with a sharing analysis based on co-call graphs.

Per-identifier information is a pair of an approximation of the number of calls (e.g. at most one vs. many) and the minimum arity of any such calls. For single-entry identifiers, it is always safe to η-expand bound expressions, until their arity matches the number of arguments of the call. In the case of multiple calls, η-expansion might hide the otherwise shared evaluation of the bound expression behind a lambda. Hence, the conservative assumption in the thunk case is to discard all analysis information and assume arity 0.

Call Arity annotates every **let**-binder with this *call arity*, the arity to which the bound expression can be η-expanded without losing any sharing.

A pair (n, α) of call multiplicity n and minimum arity α is accurately represented as a Usage of $n * C^n(\underbrace{C^1(\ldots C^1(U)\ldots)}_{\alpha-1 \text{ times}})$.

Call arity – or, expanded arity based on usage – can be recovered from identifier usage and the results of an arity analysis via the following pair of mutual recursive

functions:

$$
\begin{array}{lcl}
\mathsf{expand}_{\mathsf{Usage}} & : & \mathsf{Usage} \to \mathbb{N} \to \mathbb{N} \\
\mathsf{expand}_{\mathsf{Usage}}\, A\, \alpha & = & \alpha \\
\mathsf{expand}_{\mathsf{Usage}}\, (\omega * \_)\, 0 & = & 0 \\
\mathsf{expand}_{\mathsf{Usage}}\, (\_ * u)\, \alpha & = & \mathsf{expand}_{\mathsf{Use}}\, u\, \alpha \\
 & & \\
\mathsf{expand}_{\mathsf{Use}} & : & \mathsf{Use} \to \mathbb{N} \to \mathbb{N} \\
\mathsf{expand}_{\mathsf{Use}}\, HU\, \alpha & = & \alpha \\
\mathsf{expand}_{\mathsf{Use}}\, C^{\omega}(u)\, 0 & = & 0 \\
\mathsf{expand}_{\mathsf{Use}}\, C^{\text{-}}(u)\, \alpha & = & 1 + \mathsf{expand}_{\mathsf{Usage}}\, u\, (\max\, 0\, (\alpha - 1)) \\
\mathsf{expand}_{\mathsf{Use}}\, \_\, \alpha & = & \alpha
\end{array}
$$

In a nutshell, we can expand beyond arity α as long as the remaining call uses are still one-shot.

Absent identifiers could be expanded to arbitrary arity (e.g. the bottom of the lattice), but there is no reason to do so. For identifier usage, we have to take care of the case where there are multiple use sites and the bound expression is not in WHNF (implied by an arity of 0). As stated above, expanding thunks with multiple uses loses sharing, so we don't do that. In the remaining cases, the bound expression is a function or with a single-entry thunk, which we can expand based on call use.

A head use can be treated similarly to the absent case, with the exception that for zero arity we should not expand at all, to avoid surprising interplay with `seq`. In case of a multi call on an expression of zero arity, arity expansion must stop in order not to lose sharing. In the remaining cases of call uses, the bound expression is a function (hence non-zero arity) or the call use is one-shot. Either way, we won't lose shared work by lifting one more lambda to the top-level, so we assume just that and arity expand the hypothetical lambda body with the peeled call use u.

Knowing how to derive call arity from a combination of the results of usage and arity analysis, it stands to reason that Call Arity is generalised by our usage analysis.

However, it is not immediately obvious how the analysis strategy for **let** bindings is subsumed. Because Call Arity does arity expansion based on use sites, it employs a LetUp strategy for all bindings. We talked about the trade-offs of LetUp vs. LetDn in Section 3.2.3 and Section 3.3.6, which are adequately summarised by saying that our analysis never yields worse results than always analysing LetUp-style would.

Other than that, we improve on Call Arity in a number of ways:

Usage signatures The mixed LetDn/LetUp style allows us to unleash usage signatures at call sites for higher-order information, where Call Arity effectively assumes a usage signature of $\top$.

Pair uses Propagating pair use allows better usage information for functions wrapped in the pair constructor. This is a common scenario for type class dictionaries.

Call uses Call uses capture more precisely the analysis information of Call Arity without having to interleave an arity analysis with co-call analysis. Also, there is less of a split between handling thunks and functions.

As witnessed by the benchmark results in Chapter 5, our usage analysis yields no results worse than Call Arity, which is rather the point of this thesis.

3.4.2 Recovering Demand Analysis

Our usage analysis bears quite some resemblance to that described in Sergey et al. [17] (henceforth 'Cardinality Analysis'), currently implemented in GHC's Demand Analyser.

Since, at the core, we employ the same analysis lattice, comparing the two is much easier than it was for Call Arity. In a way, we generalised Cardinality Analysis as much as needed to derive the information produced by Call Arity. When we realised that further generalisation would make for a more uniform formulation, we arrived at the transfer function in Section 3.3.

This results in an analysis that is strictly more precise than both Call Arity and Cardinality Analysis and that precision can be traded for compiler performance on a broad spectrum (cf. Chapter 4).

That said, we generalise Cardinality Analysis in several aspects:

Unified LetUp/LetDn The strictly separated LetUp and LetDn rules have been superseded by an approach that always unleashes usage signatures at call sites LetDn-style (also for thunks). Also, in the LetDn case we put the bound expression under the precise incoming use instead of digesting the bound expression for the case of a single call with manifest arity (the number of leading lambdas, that is).

Co-call graphs We gain precision for the LetUp case by tracking the co-call relationship (cf. Section 3.2.3 and Section 3.3.6).

Use vs. Usage We treat the *usage* an identifier is exposed to separate from the *use* an expression is put under. This results in a better understanding of what a *one-shot* lambda is and when to utilise which domain in general.

Ideal specification The transfer function serving as a specification captures the essence of the analysis without leaking any implementation details, such as how approximate the usage transformers unleashed through LetDn are.

We will revisit the issues of how approximate we want usage transformers to be in Section 4.6.

Specifically, we recover Cardinality Analysis by forgetting information and being more approximate in the LetDn case, so that a single analysis pass over the bound expression is enough.

In short, we would need to

- Modify the $\_\downarrow\_$ combinator (cf. Section 3.3.6) to replace usage signatures by $\bot$ for thunks and to distinguish only the three cases of
  1. No call having happened ($u \sqsubset \underbrace{C^1(C^1(\ldots C^1(\_)\ldots))}_{n \text{ times}}$ for manifest arity n), in which case a usage type of $\bot$ is unleashed.
  2. A single call with arity higher than manifest arity n has happened ($u \sqsubseteq \underbrace{C^1(C^1(\ldots C^1(\_)\ldots))}_{n \text{ times}}$). Actually analyse the bound expression through τ in that case.
  3. Something stronger than a single call happened. Multiply the extracted usage type by ω in that case.
- Track only self-edges in the co-call graph and conservatively assume all other edges exist. This corresponds to tracking per-identifier usage multiplicity separately instead of in a co-call graph, see Section 3.2.3.

4 Implementation

This chapter is concerned with the implementation of a usage analysis as specified in Chapter 3 within the Glasgow Haskell Compiler (GHC). Changes are scattered over a huge number of files, so the implementation is only available online at `https://github.com/sgraf812/ghc` under the branch `cocall-full`[1]. The main analysis file can be found in `compiler/simplCore/UsageAnal/Analysis.hs`, which is a fork of the subsumed Call Arity analysis.

A detailed walkthrough of the Haskell code is out of scope and would not be particularly interesting, so we will instead discuss design decisions and interesting problems we encountered.

4.1 Object Language

After a Haskell program passes through GHC's frontend, it is compiled down to an explicitly typed core calculus called GHC Core. Core is the first of a number of intermediate languages the program is translated to before an executable artifact is produced. Already being vastly simpler than Haskell's huge surface syntax, as can be seen in Figure 4.1, Core is still quite complex compared to the object language introduced in Section 3.1.

While Core is still unconcerned with operational details, most optimisations within GHC are realised as Core-to-Core passes. A non-strict, high-level language like Haskell provides ample opportunities for optimisation even in this macroscopic context. The representation as a lambda calculus allows simplification based on term rewriting, which GHC makes great use of in its simplifier. Functional languages encourage composition of concise definitions, so GHC supports its optimisations with an aggressive inliner.

Apart from the simplifier, GHC employs other transformations which rely on precise information made available by analyses like GHC's Demand Analyser and Call Arity. While the Demand Analyser combines strictness analysis [14] with a usage analysis [17] and constructed product result analysis [1], Call Arity is an arity analysis interleaved with a sharing analysis based on co-call graphs, to find out which bindings can be η-expanded without losing any shared work.

As is the case for Demand Analysis and Call Arity, our usage analysis will operate on and annotate GHC Core expressions.

[1] And of course in the eventual submission.

```
data Expr b
  = Var       Id
  | Lit       Literal
  | App       (Expr b) (Expr b)
  | Lam       b (Expr b)
  | Let       (Bind b) (Expr b)
  | Case      (Expr b) b Type [Alt b]
  | Cast      (Expr b) Coercion
  | Tick      (Tickish Id) (Expr b)
  | Type      Type
  | Coercion  Coercion

type Alt b = (AltCon, [b], Expr b)

data AltCon
  = DataAlt DataCon
  | LitAlt  Literal
  | DEFAULT

data Bind b
  = NonRec b (Expr b)
  | Rec [(b, Expr b)]
```

Figure 4.1: Part of the data types representing the syntax of GHC Core

4.2 Top-level Bindings

The code of a module in GHC Core is represented as a list of top-level definitions, some of which are exported.

To avoid duplication with the treatment of **let** bindings, we translate the list of definitions into an expression of nested **let**s before the analysis and back after the analysis.

Which usage are top-level bindings exposed to? For exported bindings, we don't oversee all potential use sites, so we have to be conservative and assume $\omega * U$. Exported bindings are similar to garbage collection roots: All non-absent bindings must be reachable through an exported binding. This is because the whole scope of non-exported top-level bindings is statically known.

Based on this observation, it is also clear what the expression within the innermost **let** should be: A tuple of the exported identifiers. This encoding of modules is common-place in languages like JavaScript (by the name of *Revealing module pattern*) that lack(-ed) a proper module system.

Considering exported identifiers as roots is necessary, but, as it turned out, not sufficient. GHC's rewrite rules and vectorisation declarations possibly mention

identifiers which are neither exported, nor otherwise reachable. These must be included in the root set!

A similar problem occurs for *unfoldings*. Unfoldings enable inlining across module boundaries by serialising the *unoptimised* bound expression into the module's interface file, a Haskell-specific compilation artifact like object files. These unfoldings play a crucial role in revealing opportunities for custom rewrite rules.

Because unfoldings consist of the unoptimized bound expressions, they potentially reference bindings which are already optimised away or replaced by an optimised variant in the actual object code. As for rewrite rules and vectorisation declarations, ignoring unfoldings can result in surprising behavior and unforeseen crashes due to execution of supposedly absent code.

Correct handling of unfoldings would require to treat them as alternative right-hand sides of the binding they decorate. Experimental support for unfoldings in the style of `if True then rhs else unfolding` resulted in scoping issues of inner bindings, as well as distortions of analysis results. As we didn't observe any crashes related to unfoldings when compiling and running the entire compiler, test suite and benchmark suite, we postponed proper handling of the problem.

Of course, the simplest sufficient root set would be to include all top-level definitions, regardless if exported or not. However, that leads to severe performance regressions, as GHC aggressively floats out local bindings to the top-level if possible. Call Arity, in particular, relies on the assumption that only exported identifiers are externally visible to achieve its good results. The Demand Analyser, in contrast, goes with the conservative assumption that all top-level bindings are used.

The problems we faced are closely related to the problem GHC's Occurence Analyser tries to solve, but we refrained from mirroring even more unrelated logic into an already quite complex usage analysis.

4.3 On 'Interesting' Identifiers

Call Arity utilises co-call graphs for its sharing analysis, which can be quite expensive, because of the inherent quadratic complexity. Although the graph data structure used for co-call graphs allows for efficient insertion, constructing the adjacency set of a node is quite costly.

That is why Call Arity tracks only 'interesting' identifiers in its data structures, assuming conservative results for all other identifiers [3]. Identifiers which are deemed interesting have a function type and are locally **let**-bound. This is good enough for the very specific purpose that Call Arity set out to optimize: Formulating the commonly used `foldl` as a right fold without causing unnecessary allocation.

Except, we can't make the same assumptions when we also want to generalise the usage analysis within the Demand Analyser. From a usage perspective, all bindings carry important usage information.

Because of the same challenges regarding huge constructor applications outlined in Breitner [3, Section 3.4.1], co-call graphs are the time and space bottleneck of our

analysis.

While the previous co-call graph data structure is well-suited for small graphs, the representation as an unreduced union of complete and complete-bipartite graphs makes edge tests require time linear in the size of the union in the worst case. Let alone the unpredictable space usage, possibly exceeding quadratic complexity for the same reason. Paired with the requirement imposed by fixed-point iteration to efficiently check co-call graphs for equality, we chose to revise the graph data structure to be represented in a more predictable reduced form, as explained in Section 4.4.

4.4 Graph Representation

Section 4.3 brought up performance issues regarding the data structure used to model graphs.

Call Arity necessitated an efficient way to handle either sparse or dense graphs. Breitner [3] chose to represent graphs as a simple, unreduced union of complete and complete bipartite graphs, simply because it provided the right tradeoffs for small- to medium-sized graphs.

However, since we got rid of the notion 'interesting' variables (cf. Section 4.3), the performance issues resurfaced.

The unreduced union representation has problems when the union consists of many, small graphs: Computing the adjacency set of a node, or even simply testing for an edge in a graph of constant size, may take time linear in the length of the union. Space complexity is unpredictable in the same way: Even for represented graphs of constant size, the size of the representation scales linearly in the length of the union. Uniting a graph itself results in a graph of twice the size. Such an operation is quite common in fixpointing, so exponential blowup is imminent.

More concretely, at one point through development, space usage exceeded sixteen gigabytes for some input files, bogging down the whole development system.

Hence, in order to have better guarantees about space and runtime complexity, we took inspiration in representing the common cases of sparse and dense graphs efficiently. We made sure that edge tests were still efficient by storing the represented graph's node-indexed adjacency sets (witnessed by an isomorphism $\mathsf{Graph} \simeq \mathsf{Var} \to \mathcal{P}(\mathsf{Var})$ modulo symmetry) directly in a **newtype** over an IntMap IntSet, leaning on unique integer keys GHC assigns to variables.

The representation either stores the edge set of the complement graph or the graph's edge set directly, depending on the number of edges in the graph. Obviously, the former exhibits better performance characteristics for dense graphs, while the latter should be favored for sparse graphs.

When to flip the representation is determined based on the density of the graph: If the number of edges in representation exceeds a constant threshold greater than $1/2$ times the number of potential edges, the graph representation is to be flipped.

Edge tests have become very cheap in the new data structure, while the complexity

hides in implementing the $\sqcup$ and & operators, which will need to flip between representations as appropriate.

While this doesn't get rid of the potential quadratic space complexity, this tremendously helped in bringing down memory usage to more predictable figures.

4.5 Bounding Product Uses

As pointed out in Section 3.3.7, we need to make sure to bound the depth of product uses in order for the domain of monotone usage transformers to satisfy the ascending chain condition, giving some guarantee of termination.

Otherwise, the mentioned infinitely ascending chain actually occurs for usage signatures of coinductive definitions. Such definitions are permitted in a lazy functional language like Haskell and can also be emulated in strict languages through explicitly delayed computations. Consider this snippet on lazy streams:

```
data IntStream = MkStream Int IntStream

triple :: IntStream → IntStream
triple (MkStream x xs) = MkStream (3 ∗ x) (triple xs)
```

Approximation of the usage transformer of `triple` will begin with $\bot$. If put under use U, the usage on the first argument will ascend to $1 * U(1 * U, A)$, then to $1 * U(1 * U, 1 * U(1 * U, A))$ and so on.

We currently bound the depth of product uses to 10, which is quite arbitrary. Termination time, however, is affected exponentially by the cut-off depth.

Foreshadowing the graph-based data-flow iteration approach in Section 4.8, another approach worth considering would be a cut-off based on iteration state. Instead of bounding product uses, we could track the number of different uses under which the same expression was analysed (e.g. the number of stable points in which we already approximate the usage transformer) and return conservative results for when a threshold is exceeded.

4.6 Approximating Usage Transformers

As we saw in Section 3.3.7, all denoting usage transformers are monotone, which is a necessary condition for termination of the analysis.

However, we can't just naively approximate a function. At least we would need an appropriate data structure for monotone maps between lattices. Then, we would also need some way of predicting the points where actual steps in the ascending chain happen, otherwise we would still need to compute every point.

However, we don't need to approximate every point of a usage transformer. We can do much better by recognising that only ever finitely many (most of the time only one) points of a transformer are accessed!

After all, for the outermost **let** expression representing the module, the only usage type that we are interested in is that under use U. To compute that usage type, we only ever access single points of a usage transformer, etc.

So, instead of approximating usage transformers at *every possible point*, we employ a more demand-driven approach and approximate only those points we transitively access from the root expression under use U.

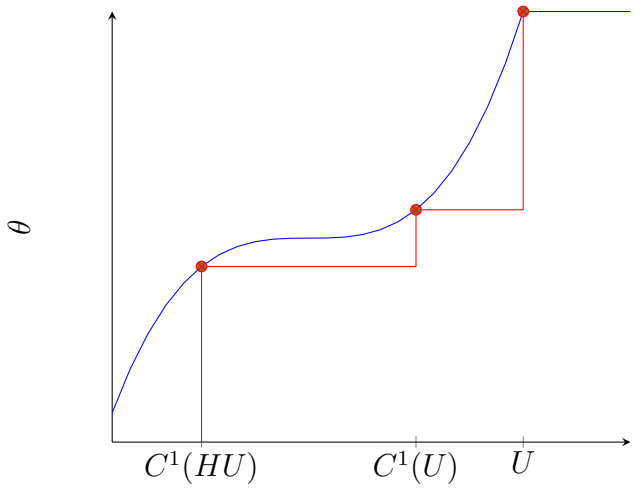


Figure 4.2: Sketch of a monotone usage transformer and a monotone approximation in four points. Maps a chain in Use to one in UType.

Figure 4.2 shows a (for illustrative purposes continuous) usage transformer that is approximated in finitely many points. Note that the approximation is stable only in four points, but possibly unstable in all others. That doesn't matter much, assuming that only the four stable points are ever accessed.

This argument is comparable to the difference in evaluation order between dynamic programming and memoisation: Memoisation follows a demand-driven top-down scheme, whereas dynamic programming computes the solution from the bottom up. When the solutions to all subproblems are really needed, both approaches perform the same work.

However consider the following artificial recurrence of a function for fixed naturals c and p:

$$f(n,i) = \begin{cases} n+i, & \text{when } n = 0 \\ i * f(n-1, i*c \pmod p), & \text{otherwise} \end{cases}$$

Aside from solving this recurrence in some smart way, let's compare how the different approaches would calculate an arbitrary point $f(n,i)$.

A reasonably efficient dynamic programming strategy would need to fill a tableau with $p*n$ entries, starting with the column $f(0,i)$ for all i. In contrast, memoisation will just need to follow a single thread of n points through the recurrence to compute

$f(n, i)$[2].

To make matters worse, if we removed the modulo operator, we could not solve f with a tableau at all! This is the same situation with monotone usage transformers: Without knowing at which points the steps in the strictly ascending chain are made, we cannot compute the monotone map in finite time. Also, a memoisation-based (or demand-driven, lazy, top-down, ...) approach computes only those points we are interested in.

Of course, a recurrence like that of Section 3.3 generalises on problems suited to be solved by dynamic programming and memoisation, in that the definition of a point may directly or indirectly refer to itself (e.g., introducing cycles to the dependency graphs we solve). In terms of a solution strategy, for a typical memoisation problem, it is sufficient to memoise already computed results in some kind of map. For a recurrence however, we need to propagate updates of unstable points, leading to the usual data-flow frameworks solved through fixed-point iteration.

4.7 Annotations

The abstract specification in Section 3.3 leaves out many details a real implementation must account for.

The most glaring simplification is that of returning analysis results. The specification describes how to interpret an expression abstractly with respect to usage, but forgets to actually announce its findings!

A real implementation would thus thread annotations as an additional output. In GHC it is common to thread the annotated expressions directly, without collecting analysis results in some kind of map first. This is a little unfortunate, as a map would allow to efficiently check for changed annotations. However, intermediate passes within GHC guarantee only that the unique keys of identifiers are unique *within their scope*, so there might be clashes when merging the results of two different closed expressions.

Thus, our usage analysis annotates expressions directly (this has impliciations on change detection in Section 4.8) and in fact we annotate the same information as Sergey et al. [17] does:

1. When analysing lambda expressions, we mark the lambda as *one-shot* if the incoming use is of the form $C^1(\_)$.

2. We annotate any binder with the usage recorded in the usage type relative to incoming use. This applies to lambda binders (after we multiply body usage, but before we delete the binder from the usage type), **case** binders, data constructor fields and of course **let** binders (looked up after the fixed-point of `up` is reached, before we delete the binders of the group).

[2]Granted, there are no overlapping sub-problems, so no caching is needed at all, but that could be changed by just adding one additional case to the recurrence.

Note that the annotations depend on the incoming use. This means annotations only make sense when the use is a conservative approximation to every possible use. We see in Section 4.8 an example where use on a call site produces too optimistic annotations, which of course may not be used.

If at one point an expression is absent (which happens only for **let** bound expression or arguments), we mark all binders in that expression as absent.

In order to enable more precise results across module boundaries, Peyton Jones et al. [14] provided *demand signatures* for each exported function in the module's interface file. This was extended by Sergey et al. [17] to usage signatures, and as such we also annotate each exported function with its usage signature.

Note that by the time we import a function, its free variable usage is uninteresting: The environments would only mention free variables which were already compiled and whose usage was conservatively approximated because of unforeseeable use sites. Hence the usage signature captures all important information.

Now, the usage signature of an expression alone has no semantic meaning without the use it was produced under. The annotated usage signature is actually a digest of the full usage transformer, approximated at three prominent points. From a usage signature for an incoming call use corresponding to the arity α (as computed by an arity analysis) of the expression, we can derive the following usage transformer:

$$\tau_x\ u = \begin{cases} \bot, & \text{when } u \sqsubset \underbrace{C^1(C^1(\ldots C^1(\_)\ldots))}_{\alpha \text{ times}} \\ \langle \emptyset, [], \sigma \rangle, & \text{when } u \sqsubseteq \underbrace{C^1(C^1(\ldots C^1(U)\ldots))}_{\alpha \text{ times}} \\ \langle \emptyset, [], \omega * \sigma \rangle, & \text{otherwise} \end{cases}$$

This is really similar to the treatment of product constructors (cf. $\tau_{(,)}$ in Section 3.3.5), except that there is no polymorphism in regard to the product use of the call.

Figure 4.3 shows a sketch mapping a chain in Use to one in UType in the spirit of Figure 4.2. In contrast to the situation for fixed-point iteration, where we start with optimistic approximations, the digested usage transformer is conservative, e.g. approximates from above.

It also shows how our usage analysis generalises on the LetDn rule in Sergey et al. [17]: Their approach is to always interpolate the usage transformer in those three points, even for local bindings. This guarantees at most one pass over an expression, disregarding fixed-point iteration. We can recover the same results by modifying the $\_\downarrow\_$ combinator to simplify all incoming call uses to one of the three cases.

Since we wanted to generalise on Call Arity, at least arbitrary call demands should be permitted (e.g. truncating products mentioned in incoming call uses to U). We will see the implications on compiler performance and performance of the produced artifact in Chapter 5.

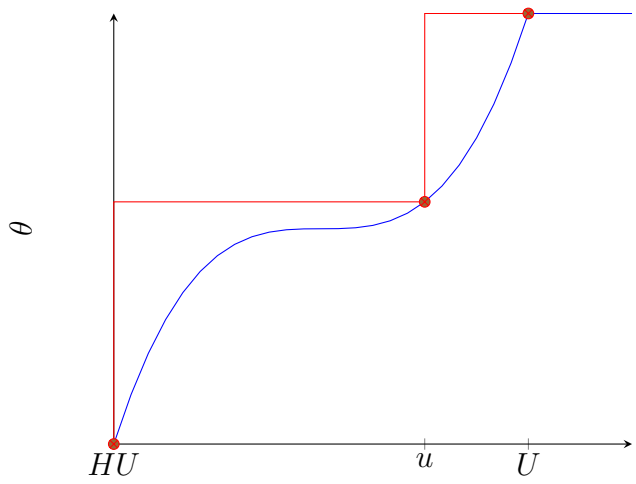


Figure 4.3: Sketch of a usage transformer derived from a usage signature for a single incoming call use u.

4.8 Solving the Data-flow Problem

Analyses within GHC are commonly guided by the structure of the syntax tree. This is attractive from the point of view of simplicity: An analysis is just a fold over the expression, returning the annotated expression.

However, for our analysis and with the approach from the last section, we encounter a problem at bindings like **let** f x = x **in** e: How can we know at which uses we need to calculate the usage transformer of f without looking at e? We know for sure at the call sites of f, so we can just approximate its usage transformer on demand *while analysing* e and memoise it through laziness.

This gets complicated really quickly if we add recursive binding groups:

```
let fac n =
      if n == 1
      then 1
      else n * fac (n−1)
in fac 12
```

When we hit the call to fac in the body, we have to query the approximated usage transformer for a stable value at the incoming call use. For this, we have to perform fixed-point iteration, assuming a very optimistic $\bot$ usage transformer (or a prior approximation) for fac and iterate until the usage type for the requested use doesn't change any more. This means we have to flag any reference to unstable points of the usage transformer.

Hinting at mutual recursion and recursive calls in different uses, it should have become clear that going down this route quickly becomes too complex to follow and implement. That is why analyses like the Demand Analyser deliberately settled for analysing bound expressions before the body and vice versa, embodied in the

LetDn and LetUp rules, respectively. These were discussed in Sergey et al. [17, Section 3.5–3.6] and Section 3.3.6.

Another drawback is that techniques which are concerned with computing fixed-points, but are otherwise orthogonal to the analysis implemented, are intertwined with the analysis logic. Analysis order depending on the syntactic structure is just a minor example of this. A less simple technique is that of caching of analysis results described in Peyton Jones et al. [14, Section 9.2].

All these approaches are present in every analysis within GHC (or they would benefit from them, at least) and every analysis for itself may get it wrong in a particular way, let alone the increase in cognitive complexity.

Looking at compilers for imperative programming languages, data-flow problems are solved by fixed-point iteration over the control flow graph (or over a graph-based intermediate representation [2], [11]) and is broken down into two parts.

An *iteration strategy* determines which nodes in the data-flow graph are still unstable and need to be recomputed, and the order in which to do so. The chosen strategy is completely opaque to the data-flow framework to solve, but severely impacts the time it takes to arrive at a stable solution. Commonly, the nodes which need to be updated are tracked within a *worklist* in order not to update currently stable nodes. The order in which the worklist processes unstable nodes also affects performance [4]. The caching of analysis results between iterations [14] is a nice side-effect of the explicit graph abstraction.

Additionally, associated with each node is a *transfer function* which has the single responsibility of recomputing analysis information in terms of the current state of its dependencies.

Driven by the looming complexity in analysis order, we decided to break down our analysis in a similar manner. In an effort to specify transfer functions separate from the iteration strategy, we arrived at the (slightly simplified) interface in Figure 4.4.

A DataFlowFramework assigns to each abstract node a TransferFunction and a ChangeDetector that compares the old value with the updated value to detect when the node has become stable. For reasons becoming clear later on, the ChangeDetector is also supplied the set of referenced nodes that changed since the last iteration.

Although we purposefully named the type variable denoting the analysis domain lattice , we don't actually require it to model any kind of algebraic structure. This is something we come back to in Section 4.9.

The first two type parameters of TransferFunction specify in what kind of data-flow framework the function operates, while the last parameter corresponds to the return value of that TransferFunction. Talking about what a transfer function 'returns' might not make much sense just yet, but the derived Monad instance hints at how this will turn out. As always, a Monad instance is useful to weave effects into otherwise pure computations. We can see that TransferFunction is just an opaque wrapper around a stateful computation, but have no way of conjuring such a side-effect apart from cheating our way in with pure.

The single means for introducing a 'side-effect' is through dependOn, which announces an edge in the data-flow graph by referencing the value of another node.

```
runFramework
  :: Ord node
  → DataFlowFramework node lattice
  → Set node
  → Map node lattice
runFramework = ...

data DataFlowFramework node lattice = DFF
       (node → TransferFunction node lattice  lattice )
       (node → ChangeDetector node lattice)

type ChangeDetector node lattice
  = Set node → lattice → lattice → Bool

data TransferFunction node lattice a
  = TFM (State (WorklistState node lattice) a) -- not exported!
  deriving (Functor, Applicative, Monad)

dependOn
  :: Ord node
  ⇒ node
  → TransferFunction node lattice (Maybe lattice)
dependOn
```

Figure 4.4: The essence of the Worklist module.

Depending on whether the iteration strategy can provide a value (at least cycles in the graph have to be broken at some point), it may or may not return a value, in which case the TransferFunction is obliged to continue with an optimistic approximation (e.g. $\bot$).

Note that just by executing the TransferFunction in another WorklistState, we can iterate a transfer function in a completely isolated manner. The iteration strategy decides if the value of a node is immediately to be recomputed in a call to dependOn, or if a value from a prior iteration is to be returned, or none at all.

The Ord instance on node, apart from being necessary for use as key in a Map, serves as a priority on the nodes in the worklist. By choosing a total order that mirrors how an analysis would proceed along the syntax tree, we are promised fast convergence by GHC's Occurence Analyser which arranged bindings in a suitable order.

In practice, we modeled each single point of a usage transformer as a separate node. This resulted in a specialisation of node to (FrameworkNode, Use)[3], where the total or-

[3]Of course, the total order on Use is not actually compatible with the join-semilattice we usually

der on FrameworkNode mirrors the syntax tree, and lattice to (UsageType, CoreExpr), where the returned CoreExpr is annotated with the findings of the analysis. As it turned out, we lost important structure in forgetting about monotonicity (cf. Section 4.9).

Allocating FrameworkNodes for every syntactic element enables caching of intermediate results and avoids whole chains of nodes to be recomputed when no change is detected. On the other hand, the bookkeeping in the iteration algorithm might outweigh any performance benefits of caching. Also, memory usage becomes an issue for big graphs. This is why we decided to only allocate FrameworkNodes where we had to break cycles in the data-flow graph. Cycles arise exactly where we used the **fix** operator to tie knots in Section 3.3.7, thus at least we need to allocate nodes for the right-hand sides of bindings (referred to as LetDn nodes) and for whole **let** bindings (LetUp nodes).

Of course, just by allocating nodes the feedback cycles aren't broken yet: We need to detect when an iteration of a node does not change anymore. For usage types, detecting change is pretty standard by delegating to the derived Eq and can be sped up considerably by exploiting monotonicity. As an example, we can avoid potentially quadratic time comparison of co-call graphs by just checking if the number of total edges changed.

Detecting changes in the annotated syntax tree isn't so cheap. In fact, traversing entire expressions is infeasible for huge modules from a performance perspective. We can rely on another convenient fact, though: Annotated expressions only depend on subexpressions and themselves don't introduce dependency cycles. This means that when the only referenced node that changed was the current node itself, there was no change to the annotated expression.

That is why ChangeDetectors are supplied the set of changed references: Apart from checking usage types for changes, it checks if the only reference that changed was the node itself.

Other than that, we have to allocate a node (root in the example that follows) for the module expression to have a way to refer to it and then kick off iteration with a call to runFramework like the following:

```
result :: Map (FrameworkNode, Use) (UsageType, CoreExpr)
result = runFramework framework (Set.singleton (root, U))
```

By passing the singleton set as the second argument, we express that the module expression is put under top use U. From there, the iteration algorithm begins to explore the data-flow graph in a depth-first fashion. As we already pointed out at the begin of this section, this is crucial for termination: The graph itself is infinite, while the set of nodes reachable from (root, U) is finite.

This is best understood by a simple example which already exhibits quite a complex iteration order.

Example. Consider the following example program, printing the factorial of 12:

refer to and doesn't have any semantic meaning.

```
module Main (main) where

fac n =
  if n == 0
  then 1
  else n * fac (n - 1)

main = print (fac 12)
```

This will be translated to the following module expression:

```
let fac n =
      if n == 0
      then 1
      else n * fac (n - 1)
in let main = print (fac 12)
   in (main) -- This is a 1-tuple, e.g. introduces a box
```

Figure 4.5 depicts the resulting data-flow framework, or rather the finite part reachable from node (root, U) (again, root represents the usage transformer denoting the module expression).

During the depth-first discovery of nodes, forward edge labels correspond to discovering ('calling') a node and backward edge labels go in the reverse direction of the edge, e.g. finishing the target node. After the initial depth-first phase, the worklist algorithm takes over, iterating unstable nodes and propagating changes in reverse direction of data dependencies, thus after step 12 there are only backward labels.

The iteration algorithm will start with iterating the TransferFunction of the module expression, represented by the red node, under use U and then proceed in the following order:[4]

1. The transfer function associated with (root, U) forwards to the LetUp node of the **let** expression binding fac in the same use, $(\mathbf{let}_1, U)$ in blue.

2. The depth-first strategy immediately descends into said LetUp node. Since the binding for fac is recursive, the LetUp node dependsOn itself, for usage of fac in the last iteration. This is the first iteration, so dependOn returns Nothing and the usage in the body serves as a first approximation.

3. The body of the outer **let** is the inner **let** binding for main, represented by another blue node $(\mathbf{let}_2, U)$. Since main is non-recursive, looking at usages in the body is sufficient.

[4]Note that we effectively uncurry our transfer function from Section 3.3, turning something of type Exp → Use → UType into something of type Exp × Use → UType. Thus, TransferFunction transfers expressions into the domain of usage types instead of usage transformers (see Section 4.9 on impliciations for monotonicity).

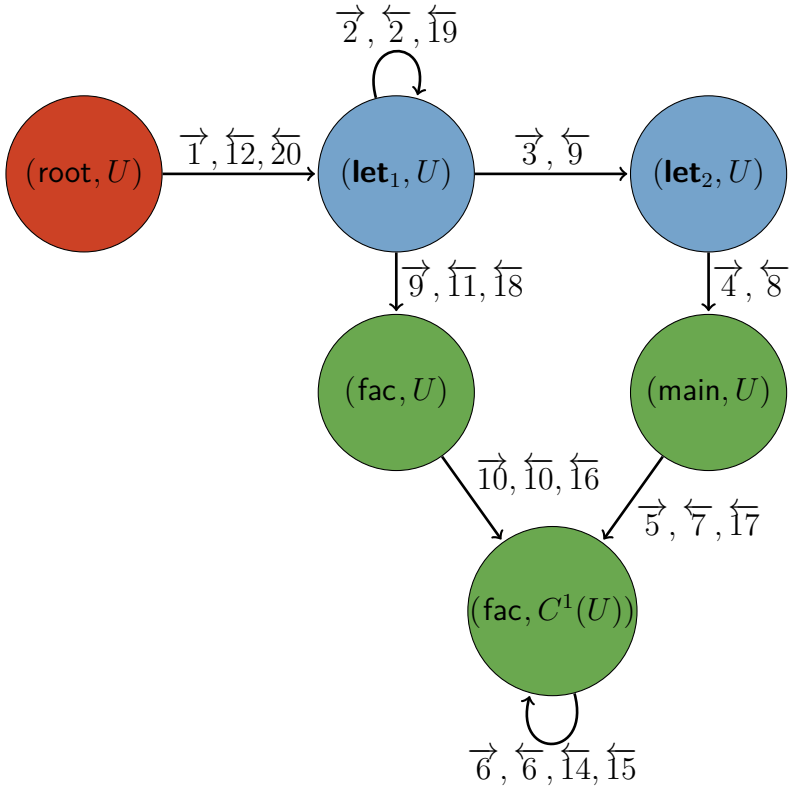


Figure 4.5: Relevant part of the data-flow framework for the fac example. The module node is drawn in red, LetUp nodes are blue and LetDn nodes are green. Edge labels correpond to which step in the list below the edge is traversed during analysis.

4. The incoming use U translates into a use of $U \equiv U(\omega * U)$ on the implied 1-tuple main, which causes a dependency on the LetDn node of main in use U (LetDn nodes are green).

5. We immediately descend into said LetDn node, where the use U, through print, puts fac under a call use $C^1(U)$.

6. After descending into ('calling') the LetDn node fac, the analysis tries to recurse into $(\mathsf{fac}, C^1(U))$. This cycle is broken by returning $\mathsf{Nothing}$ from $\mathsf{dependOn}$; the analysis will compensate for that by unleashing a usage type of $\bot$ at the call site. The result is a (first, approximate) usage type of $\langle \emptyset, [\mathsf{fac} \mapsto C^1(U)], 1 * U \to \top \rangle$ for $(\mathsf{fac}, C^1(U))$. (An irrelevant fact, because we don't use the annotated expression: The argument to the bound expression is annotated too optimistically as being single-entry and the lambda as one-shot.)

7. Analysis of the expression bound to main continues. The call use on fac gets sequentially combined with the call use from the unleashed LetDn node, for a total call use of $C^\omega(U) \equiv U$. Nothing interesting happens to the literal 12, thus the usage type of the LetDn node for (main, U) is $\langle \{(\mathsf{fac}, \mathsf{fac})\}, [\mathsf{fac} \mapsto U], \top \rangle$. However, because main's bound expression is not in WHNF, only the uninteresting usage signature is propagated to the call site within $(\mathbf{let}_2, U)$. Note that the annotated expression would be calculated at this point, too. We don't need it at the call site, but when we handle the **let** binding in the next step.

8. Unwinding the call stack once more, the inner **let** binding for main is now resolved as part of the LetUp node $(\mathbf{let}_2, U)$. Within the body, main is exposed to usage $\omega * U$. Thus, there is a dependency on the LetDn node (main, U), at least for the annotated expression. The algorithm just iterated that node in the last step, so its result is reused. Since the expression bound to main is not in WHNF, we also unleash the associated usage type, resulting in a usage type of $\langle \{(\mathsf{fac}, \mathsf{fac})\}, [\mathsf{fac} \mapsto U], \top \rangle$ for the whole inner **let** expression.

9. Another unwind resumes analysis in the LetUp node $(\mathbf{let}_1, U)$, the binding for fac. The body exposes fac to a usage of $\omega * U$, even before sequentially composition with itself induced by the recursive **let** case. Although the right-hand side of fac is in WHNF (so usage types have been unleashed at call sites), we still need the annotated expression under use U. This introduces a dependency on the LetDn node (fac, U) for which there is no value yet, causing the algorithm to do a 'call'.

10. Analysis of fac under use U yields the same usage type as under use $C^1(U)$, but that is quite irrelevant. More important, the lambda is not one-shot and the argument binder for n is not single-entry, contrary to the optimistic situation under use $C^1(U)$. Also, (fac, U) is not 'recursive', rather it depends on $(\mathsf{fac}, C^1(U))$.

11. Unwinding to $(\mathbf{let}_1, U)$ again, this results in an uninteresting usage type $\langle \emptyset, [], \top \rangle$ for the annotated expression.

12. Finally, analysis proceeds in the top-level (root, U) node, which just forwards the results from $(\mathbf{let}_1, U)$.

13. Now the actual worklist algorithm takes over: While computing the current approximation, we were using unstable results (e.g. where dependOn returned Nothing) for $(\mathsf{fac}, C^1(U))$ and $(\mathbf{let}_1, U)$.

14. $(\mathsf{fac}, C^1(U))$ has the higher priority, so it is iterated first. This results in a more precise usage type of $\langle \{\mathsf{fac}, \mathsf{fac}\}, [\mathsf{fac} \mapsto U], \top \rangle$. The change marks the referrers $(\mathsf{fac}, C^1(U))$, (fac, U) and (main, U) unstable.

15. After one more iteration, $(\mathsf{fac}, C^1(U))$ is deemed stable. Although irrelevant, the annotated expression stayed the same.

16. Next highest priority node is (fac, U), where the change on $(\mathsf{fac}, C^1(U))$ yields the same usage type (which was not used) and the same annotations. The change in usage type marks its referrer $(\mathbf{let}_1, U)$ as unstable once more.

17. In (main, U), the changes in $(\mathsf{fac}, C^1(U))$ did not make a difference at all, so the node is still stable. Hence, no need to reiterate $(\mathbf{let}_2, U)$.

18. The only unstable node left is $(\mathbf{let}_1, U)$. The usage type from the last iteration is unchanged, but the annotations in fac changed. So its referrers $(\mathbf{let}_1, U)$ and (root, U) are marked as unstable.

19. Another iteration on $(\mathbf{let}_1, U)$ reveals no further change.

20. Finally, the (root, U) node is iterated and marked as changed because of annotations in sub-expressions. That however does not mark any referrer as unstable, since there are none.

21. The algorithm returns the current stable graph as a map from nodes to results.

The infiniteness of the graph surfaces at the two different nodes for fac: Actually, there are many more of these nodes, but only the two points for $C^1(U)$ and U are reachable from (root, U).

Breitner [3, Section 3.6.6] anticipates the connection of the ad-hoc analyses in GHC to data-flow problems and the possibility of a uniform solution procedure through iterating data-flow frameworks. This work, in particular the Worklist module, solidifies these findings and provides a rather elegant embedded domain-specific language for constructing such frameworks. Eventually, we plan to release this module as a separate package on Hackage, after it has gone through some polishing.

We could see this kind of graph-based data-flow iteration becoming more feasible by associating a FrameworkNode with each CoreExpr constructor (or rather each subexpression). This would get rid of the extra book-keeping required to assign good priorities to nodes through the FrameworkBuilder code, which is actually inessential to the analysis and mostly contained in buildAnalFramework in UsageAnal.Analysis.

An optimisation worth implementing would be to take advantage of the boundedness of allocated FrameworkNodes, through which the graph data-structure admits a mapping to a plain STVector. Using lazy functional state threads [10], this would achieve constant time lookup and update at least for whole usage transformers.

4.9 Monotonicity

As we outlined in Section 3.3.7, monotonicity of all involved usage transformers is essential in proving existence of the fixed-point. Section 4.6 pointed out that approximating usage transformers in finite time is still impossible without restricting interest to a finite set of points.

This led to a data-flow framework in Section 4.8 where we modeled each single point of a usage transformer as a separate node. We argued that this is similar to uncurrying the transfer function in Section 3.3 from $\mathsf{Exp} \to \mathsf{UTrans} \equiv \mathsf{Exp} \to \mathsf{Use} \to \mathsf{UType}$ to $\mathsf{Exp} \times \mathsf{Use} \to \mathsf{UType}$. The problem with doing so is that it doesn't preserve the monotonicity of denoting usage transformers.

To be more precise, the transfer function is more accurately described by the type $\mathsf{Exp} \to (\mathsf{Use} \to_+ \mathsf{UTrans})$, where $\to_+$ denotes a monotone map.

Where does this bite? Well, we relied on monotonicity in our argument for proving existence of the fixed-point. Of course, uncurrying didn't change the actual semantics, but modeling each point separately means that prior to reaching the fixed-point, there are unstable intermediate approximations which might not be monotone. It turns out that convergence depends even on these unstable approximations to be monotone.

This is easy to reproduce by removing all mentions of the monotonize function from the main analysis module and then running the test `joao-circular` with a `devel2` flavored build of the compiler. The file `Visfun_Lazy.hs` will violate a monotonicity check while updating the value of a framework node. Unfortunately, that file is rather huge and convoluted, so a detailed analysis is out of question.

4.10 Hacking on GHC

Working on a central part of a 'real-world' compiler such as GHC was challenging in ways beyond thinking about combining two analyses on a drawing table.

Usage information is critical to many other Core-to-Core passes within GHC. More subtle bugs require entire days of tracing through tests and thinking hard for minimal reproductions in order to better understand the problem.

Many bugs hid behind module boundaries, because the code concerned with serialising usage signatures is quite scattered. Some identifiers, such as dictionary selectors, primitive operators and runtime errors, get special treatment by GHC, the places at which this happens were discovered in a number of successive debugging sessions.

Not-so-absent thunks which the analysis identified as absent led to crashes at runtime, only more informative than a segmentation fault in that the message mentions absence as a reason.

So it came that the most gnarly class of bugs manifested themselves as absent errors which only popped up across module boundaries, involving type class instances referencing absent thunks. It took quite some time and head-scratching to nail down rewrite rules as the culprit, which had to be regarded as reachability roots as explained in Section 4.2.

GHC's build system was another thing that took some time to get accustomed to. Especially figuring out which things needed to be rebuilt after some change and what various build settings did, took a lot of trial and error. Sometimes even a `make clean` wouldn't get rid of some clearly build system-related issues, leaving no choice but to do a complete fresh checkout. Experiences like the latter don't exactly strengthen confidence in the build system, so we look forward to seeing `hadrian`[5] succeed.

Lastly, the test and benchmark suites of GHC are quite essential in crushing occasional hopes after having fixed a complicated bug, by immediately confronting the developer with another regression. Of course, this is a good thing for both the maturity of GHC as well as a humbling experience for the soul.

[5] `https://github.com/snowleopard/hadrian`

5 Evaluation

With implementation considerations sorted out in Chapter 4, we turn to assessing the analysis by means of the performance of the generated code in Section 5.1. We also discuss compiler performance in Section 5.1.3 and see ways to trade artifact performance for analysis performance.

5.1 Benchmarks

Enough with the mathy chit chat, give me numbers!

In this section we will look at how the analysis influences benchmark results in GHC's nofib [13] benchmark suite.

The (at the time of writing most recent) GHC 8.2.1 release – revision `0cee252`, tagged as `ghc-8.2.1-release` – serves as a baseline. We compare to three variants of our fork[1]:

(1) The most precise form of our analysis. Replaces Call Arity entirely and is run two additional times, immediately after the Demand Analyser. The usage information the Demand Analyser produces is only accessed to check for regressions.

(2) This variant discards any product use at call sites. E.g., $C^1(U(1*U,A))$ will be truncated to $C^1(U)$ in a call like **let** f x = ... **in** fst (f 42). This is so that less nodes in the data-flow framework are accessed, to save space and time. Results should still show an improvement compared to the baseline.

(3) In addition to the changes of **(2)**, remove the co-call graph from the usage type. This is equivalent to a co-call graph that only tracks self-edges and conservatively assumes the existence of all other co-call edges. The intention is to measure if the space and time impact of co-call graphs matters in practice. Note that regressions in some cases are expected, as the baseline employs Call Arity with accurate modelling of edges in co-call graphs at least between 'interesting' variables [3, Section 3.4.1].

Table 5.1 shows the interesting results of running nofib. We will break them down by discussing allocations and instructions executed separately.

[1] Available at https://github.com/sgraf812/ghc under the branches `cocall-full`, `cocall-approximate-calls` and `cocall-complete-graphs`.

	Bytes allocated			Instructions executed		
Program	**(1)**	**(2)**	**(3)**	**(1)**	**(2)**	**(3)**
`ansi`	-0.0%	-0.0%	-0.0%	-0.4%	-0.4%	-0.4%
`awards`	-0.0%	-0.0%	-0.0%	-0.5%	-0.5%	-0.5%
`cryptarithm2`	0.0%	0.0%	0.0%	-0.9%	-0.9%	-0.9%
`eliza`	0.0%	0.0%	0.0%	+0.1%	+0.1%	+0.1%
`expert`	-0.0%	-0.0%	-0.0%	+0.1%	+0.1%	+0.1%
`fannkuch-redux`	-0.0%	-0.0%	-0.0%	+11.4%	+11.4%	+11.4%
`fft2`	-0.9%	-0.9%	-0.9%	-0.7%	-0.7%	-0.7%
`fish`	0.0%	0.0%	0.0%	-0.5%	-0.5%	-0.5%
`fluid`	-1.5%	-1.5%	-1.5%	-1.4%	-1.4%	-1.4%
`gen_regexps`	0.0%	0.0%	0.0%	-28.1%	-28.1%	-28.1%
`hidden`	-0.0%	-0.0%	-0.0%	-1.1%	-1.1%	-1.1%
`infer`	+0.0%	+0.0%	+0.0%	-0.2%	-0.2%	-0.2%
`last-piece`	0.0%	0.0%	0.0%	-0.9%	-0.9%	-0.9%
`listcompr`	0.0%	0.0%	0.0%	-0.7%	-0.7%	-0.7%
`listcopy`	0.0%	0.0%	0.0%	-0.7%	-0.7%	-0.7%
`maillist`	-0.0%	+0.0%	-0.0%	-0.2%	+2.1%	+3.4%
`mandel`	-0.0%	-0.0%	-0.0%	-1.0%	-1.0%	-1.0%
`mkhprog`	0.0%	0.0%	0.0%	-0.8%	-0.8%	-0.8%
`paraffins`	-0.0%	-0.0%	-0.0%	-3.4%	-3.4%	-3.4%
`parstof`	-0.0%	-0.0%	-0.0%	-0.5%	-0.5%	-0.5%
`prolog`	-0.0%	-0.0%	-0.0%	-0.6%	-0.6%	-0.6%
`puzzle`	0.0%	0.0%	0.0%	-15.4%	-15.4%	-15.4%
`queens`	-0.0%	-0.0%	-0.0%	-1.2%	-1.2%	-1.2%
`reptile`	-0.1%	-0.1%	-0.1%	-0.5%	-0.5%	-0.5%
`sphere`	-0.0%	-0.0%	-0.0%	-4.5%	-4.4%	-4.4%
`tak`	-0.4%	-0.4%	-0.4%	-0.0%	-0.0%	-0.0%
... and 77 more						
Min	-1.5%	-1.5%	-1.5%	-28.1%	-28.1%	-28.1%
Max	+0.0%	+0.0%	+0.0%	+11.4%	+11.4%	+11.4%
Geometric Mean	-0.0%	-0.0%	-0.0%	-0.6%	-0.6%	-0.6%

Table 5.1: Benchmark results of running nofib where the number improved by more than 0.3% or regressed by more than 0.0%. The GHC 8.2.1 release on which all work is based was used as a baseline. Variants **(1)** to **(3)** are increasingly approximate, but theoretically speaking, only **(3)** may possibly yield worse results than the baseline's combination of Call Arity and Demand Analysis.

	Bytes allocated		
Program	**(1)**	**(2)**	**(3)**
`fft2`	-0.9%	-0.9%	-0.9%
`fluid`	-1.5%	-1.5%	-1.5%
`infer`	+0.0%	+0.0%	+0.0%
`n-body`	-0.2%	-0.2%	-0.2%
`reptile`	-0.1%	-0.1%	-0.1%
`rfib`	-0.1%	-0.1%	-0.1%
`spectral-norm`	-0.1%	-0.1%	-0.1%
`tak`	-0.4%	-0.4%	-0.4%
... and 95 more			
Min	-1.5%	-1.5%	-1.5%
Max	+0.0%	+0.0%	+0.0%
Geometric Mean	-0.0%	-0.0%	-0.0%

Table 5.2: Interesting allocation results from the same run as Table 5.1. The only runs that were excluded were those with improvements of less than 0.1%.

5.1.1 Allocations

The total impact on allocations of our rather complex analysis is rather meager, with a reduction that doesn't even exceed the 0.1% margin in the geometric mean over all benchmarks. But recalling the goals of this thesis, namely unifying both Call Arity [3] and the work of Sergey et al. [17] into a single analysis, this is good news and asserts our claims in Section 3.4!

Allocation results are summarised in Table 5.2. There is one tiny regression to allocations due to heuristics: A partial application in `infer` is regarded cheap to duplicate by GHC. Therefore, our analysis spots a new opportunity for η-expansion and the binding gets inlined subsequently. Although the resulting program allocates more than before, the number of executed instructions (cf. Table 5.1) went slightly down over all.

The most significant improvement happened to the allocations of `fluid`, with a 1.5% reduction due to arity expansion of some parsing function.

It is more instructive to look at `fft2`. The 0.9% improvement in allocations comes from an expression within `dfth` similar to the following program:

```
module Main (main) where

fac :: Int → Int
fac n =
  if n == 0
  then 1
  else n * fac (n−1)
```

```
double :: [Int] → ([Int] → [Int]) → [Int] → [Int]
double xs k =
  case xs of
    [] → k
    x:xs' → λys → x*2 : double xs' k ys

f :: Int → [Int] → [Int]
f n =
  if fac n < 10
  then λxs → 10:xs
  else λxs → double xs (f (n−1)) xs

main =
  print (f 1 [2])
```

Our analysis finds out that f can be η-expanded to arity 2. Neither Call Arity nor the Demand Analyser recognises this.

Call Arity doesn't have any mechanism similar to usage signatures, so it assumes a conservative call arity of 1 for f because of the call double xs (f (n−1)) xs.

The Demand Analyser, on the other hand, has available only the usage signature of double for manifest arity 2, which is too conservative and unleashes a usage of $1 * U$ on f (n−1) instead of $1 * C^1(U)$.

Our analysis infers from the call to double with incoming arity 3 a call use of $C^1(U)$ for f (n−1), exposing f to a total usage of $C^\omega(C^1(U))$, which we can leverage by expanding the arity to 2.

In expanding the analogue of f within `dfth`, the analysis enables additional inlining and other transformations.

Apart from the very modest extreme cases, it is worth pointing out that allocation remained the same throughout all variants! This is extremely important for practicality of the analysis, as we will see in Section 5.1.3.

Although the implementation of foldl contains a oneShot annotation as a hint for the compiler, the same example motivating Call Arity in Breitner [3, Section 3.5.1] is still η-expanded based on the results of variant **(3)**:

```
let go x =
      let r =
            if x == 2016
            then id
            else go (x + 1)
      in if f x
         then λa → r (a + x)
         else r
in go 42 0
```

GHC produces this code when compiling the expression `sum ( filter f [42..2016])`, where `sum = foldl (+) 0` and `foldl` is implemented in terms of `foldr`. For reasons outlined by Breitner [3], it is crucial for performance to η-expand `go`, so that no closure is allocated for `r`.

Analysing the above expression under use U with our variant **(3)** will proceed in the following order:

1. Since `go` is exposed to an optimistic usage of $1 * C^1(C^1(U))$, it will analyse `go`'s bound expression under use $C^1(C^1(U))$.

2. This translates into usage of $1*C^1(U)$ on `r`, in both branches of the **if** expression. Notably, the **then** branch will first peel off one layer of the one-shot call use, then analyse `r (a + x)` under use U to find out the usage of $1 * C^1(U)$ on `r`. This alone would be enough evidence to η-expand `r`.

3. Analysing the bound expression of `r` in use $C^1(U)$ will expose `go` to a usage of $1 * C^1(C^1(U))$.

4. After some fixpointing, `r` gets its single-entry $1 * C^1(U)$ annotation and `go` will have an annotation of $\omega * C^\omega(C^1(U))$.

According to our definition of $\mathsf{expand}_{\mathsf{Usage}}$ in Section 3.4, `go`'s bound expression can be η-expanded, enabling vital further optimisations.

5.1.2 Instructions Executed

Regarding instructions executed (measured with `cachegrind`), there are more significant improvements, although that might not reflect in performance on an actual machine. In the geometric mean we achieve a total improvement of 0.6%, which still is rather modest, but as said earlier not the primary goal of this thesis.

Let's focus instead on the most significant outliers in Table 5.3. With 11.4% more instructions executed, `fannkuch-redux` from the benchmarks game regresses significantly.

Looking at the Core output, it is not obvious to see where that regression comes from. There were no subsequent transformations kicked off by the added annotations, and the annotations seem to be correct, judging from sprinkling a few `Debug.Trace.trace` calls. However when considering the decrease in allocations, it seems that GHC's heuristic cost model could be responsible for this, if not in some imported `base` module.

Conversely, the cases where performance improves are more frequent. The largest boost receives `gen_regexps`, with 28.1% less executed instructions. Yet again, it is not appearent where those improvements come from. All annotations in the Core output seem proper and beyond that no further transformation happened. This suggests that either the backend (the implementation of which the author is blissfully unaware) or some optimisation within referenced `base` modules is responsible.

	Instructions executed		
Program	**(1)**	**(2)**	**(3)**
`fannkuch-redux`	+11.4%	+11.4%	+11.4%
`gen_regexps`	-28.1%	-28.1%	-28.1%
`maillist`	-0.2%	+2.1%	+3.4%
`paraffins`	-3.4%	-3.4%	-3.4%
`puzzle`	-15.4%	-15.4%	-15.4%
`sphere`	-4.5%	-4.4%	-4.4%
... and 97 more			
Min	-28.1%	-28.1%	-28.1%
Max	+11.4%	+11.4%	+11.4%
Geometric Mean	-0.6%	-0.6%	-0.6%

Table 5.3: Interesting programs with respect to instructions executed, from the same run as Table 5.1. Excluded were those runs with improvements of less than 3% and regressions of less than 1%.

The same arguments apply to the 15.4% improvement in `puzzle`: A number of new annotations that didn't enable any further Core-to-Core transformation. The set of functions referenced from `base` is quite minimal, so it is unlikely that optimisations hide there.

5.1.3 Compiler Performance

The graph data structure necessary for modelling co-call graphs is quite complicated, optimised for sparse and dense graphs to offer even remotely manageable compiler performance.

Yet for some input files, time and – worse – space complexity goes through the roof: The space needed to compile some innocent file like `GHC/Types.hs` from `ghc-prim` goes up from a few hundred megabytes to multiple gigabytes. Clearly, this is unacceptable for a mature and widely used compiler like GHC.

Just before finalising this thesis, we also realised that variant **(1)** is not even able to compile an optimised version of the stage 2 compiler! Memory usage while compiling the LlvmCodeGen module exceeded the 32 GB of our build machine, probably because of the involvement of DynFlags, a huge tuple. That is why we use GHC's `bench` build flavour for our benchmarks, which still generate completely optimised artifacts with an unoptimised stage 2 compiler.

Heap profiling revealed that the explosion is indeed related to co-call graphs, peaks in memory needed to store IntMaps, in particular. Practically all of the overhead is attributed to UsageAnal.Types.bothUsageType, which computes the union of three graphs on each invocation. It is being used quite often, for example as part of the expensive graph substitution procedure.

	Bytes allocated			Instructions executed		
Program	**(1)**	**(2)**	**(3)**	**(1)**	**(2)**	**(3)**
nofib	+29.8%	+26.8%	+10.9%	+27.1%	+24.6%	+11.1%
ghc	+2.4%	+2.3%	+1.5%	+2.6%	+2.0%	+2.3%

Table 5.4: Performance figures recorded while compiling stage 2 GHC (unoptimised, but with optimised libraries) and nofib (completely optimised) with the three variants, relative to GHC 8.2.1. Resource usage of **(1)** and **(2)** grows beyond limits in specific cases, accounting for a major blowup overall. The most approximate variant, **(3)**, shows an acceptable increase, knowing that there are opportunities left for making it more efficient.

Even in a version of **(3)** where the co-call graphs still were present in a dumb fashion (e.g. just tracking self-edges), UnVarGraph.completeBipartiteGraph seemed to be responsible for the major blowup, even though the quadratic space complexity should have been eliminated.

With that out of the way, we may risk a look at Table 5.4, which shows the performance figures recorded while compiling itself and nofib.

Compiling and optimising nofib seems to have caused a greater regression (between 10 to 20 percent) than compiling the stage 2 compiler (only 1.5 to 2.6 percent). That is likely the case because we used the `bench` build flavour, which optimises the dependent libraries but not the stage 2 compiler. This means our optimisations are not run on GHC itself, which is entirely due to the above mentioned space explosion in variant **(1)**; the other variants compiled just fine. Again, the performance figures regarding nofib are unaffected by this.

Other than that, every variant consumes more resources than the baseline. That is not surprising: We replaced a single run of Call Arity by three runs of our analysis, of which a single run can hardly be cheaper. However, the Demand Analyser still performs Cardinality Analysis on every of its two runs, which could be left out.

There is still plenty of opportunity left to reach acceptable performance. Our variant **(3)**, for example, doesn't exhibit the same problematic space complexity as the other variants, because it gets rid of co-call graphs altogether. It turns out that call arity aware LetDn-style analysis (as in **(3)**) is precise enough to optimise all important cases!

The whole graph-based data-flow iteration story doesn't yet run as performant as it could. Currently, a Map is used to map nodes to transfer functions. If all syntax nodes would provide sensible FrameworkNodes, modeling the graph as a fixed-size array of monotone maps is possible. Also, the whole FrameworkBuilder incantation would be obsolete if CoreExpr carried appropriate FrameworkNodes.

All in all, it is too early to judge the replacement for Cardinality Analysis and Call Arity by compiler performance.

6 Related and Future Work

This section is dedicated to comparison of our analysis with prior approaches in Section 6.1 and what loose ends can be tied up in the future in Section 6.2.

6.1 Related Work

Abstract Interpretation

Within the framework of *abstract interpretation*, questions of cardinality have been successfully answered by backwards analyses based on *projections* [8] in the past.

Being an extension to the approach of Sergey et al. [17], with all the bells and whistles such as call and product uses, our usage analysis is no different: What we called usage transformers came into the world as the strictness-specific concept of *projection transformers* [9], describing how a use on an expression translates into a use on its free variables and arguments. We showed how the overlap in Call Arity [3] and Cardinality Analysis (as in [17]) can be leveraged by giving a usage analysis that generalises the results of both analyses, in order to possibly replace both in the long run.

Key was the observation that call arity (the minimum number of arguments a binding is applied to) can be computed independently of whether η-expanding the binding to that arity is possible, considering sharing. In other words: By enriching the domain of discourse to model more precise usage information, we could break the interleaving of sharing and arity analysis out of Call Arity [3]. Usage information is now computed by our usage analysis, while the arity analysis is done by GHC's regular arity analysis, specifically feeding on one-shot annotations[1].

Type Systems

For comparing our analysis to approaches based on type systems, we kindly refer to Sergey et al. [17, Section 8], who provide a great overview over recent advances. We provide a summary of their excellent writing for completenes.

While linear type systems can express single-entry and one-shot annotations quite naturally (modulo the caller vs. callee distiction [17]), they proved to be too restrictive [23]. Nonetheless, this led to a series of papers on type systems, specifically tailored to do usage analysis [19]. Wansbrough and Peyton Jones [23]

[1] Provided a little incentive from our side to always η-expand cheap (according to GHC's cost model) expressions, so that the behavior of Call Arity is matched.

identified polymorphism and subtyping to be essential for good results, which in conjunction with an exponentially growing number of annotations needed [22] elicited unacceptable complexity, both in terms of performance and in the implementation.

The main disadvantage of the approach by abstract interpretation is worse approximations across function boundaries. Sergey et al. [17] (and therefore our solution) recovers a good deed of that opportunity by computing usage signatures to appropriately analyse first- and second-order functions. Also, the very aggressive inliner of GHC reduces the cases where interprocedural information flow is important to recursive functions.

Sergey et al. [17] also compare the analysis strategy for **let** bindings to that of type system. They conclude that type systems operate similar to LetUp, which has several drawbacks compared to LetDn (corresponding to the more operational view) as outlined in Section 3.3.6. Dealing with free variables in a precise manner requires polymorphic effect systems as in Hage et al. [7], which try to alleviate some of the pain regarding subtyping.

A recent type-based approach by Verstoep and Hage [21] seemingly originating from his master's thesis [20]. It offers a similar take at the problem as the Demand Analyser, computing all relevant cardinality information (absence, sharing, strictness, uniqueness) in one run. Strongly inspired by Wansbrough [22], they differentiate *use* from *demand*, resp. *call use* and *evaluation* in our language (cf. Section 2.1), in order to handle seq appropriately. They plan to integrate their work into the Utrecht Haskell Compiler (UHC), the inliner of which doesn't seem to be as aggressive as GHC's. According to own claims [20], the approach enriches the work of Wansbrough [22] with a combination of polymorphism and polyvariance, also adding an option for uniqueness typing. Although the ensemble of polymorphism, subtyping and annotating data types was identified as problematic by Sergey et al. [17], it is argued that running the analysis plays out in terms of compiler performance, because it provides a wealth of information [21]. There are no numbers yet to substantiate that claim, appearently because integration into UHC isn't finished yet.

Data-flow Frameworks

For the implementation of imperative languages, data-flow analysis on control flow graphs or directly on graph-based intermediate representations [2] [11] are the norm.

There's even a Haskell library for analysing and transforming control flow graphs, `hoopl`, introduced in Ramsey et al. [15] and used in the C– backend of GHC. On first sight, our Worklist module seems to be a direct contender to `hoopl` and in fact we considered to use `hoopl` for our purposes. However, as we discovered the API, it became evident quickly that expressing our rather complex analysis in terms of the traditional gen/kill set terminology was quite impossible.

Thus we created our own solution to the problem, which worked out quite well for us: We got to abstract data-flow iteration logic behind a State monad, for which we exposed a single impure primitive, leading to analysis logic that is completely decoupled from fixed-point iteration and change detection logic.

6.2 Future Work

It was quite time consuming to develop the deep understanding of Call Arity and Cardinality Analysis necessary to merge the two. Implementing a few unmentioned iterations of the analysis, combined with fixing all the repercussions of hacking on a central part of GHC, took even more time. Also considering the effort invested in fleshing out the writing, it is clear that some things were out of scope for this thesis.

For reasons conjectured in Section 5.1, the full analysis can't cope with some input programs, space-wise. Interestingly, the problem wasn't so much the topology of graphs (which were almost exclusively sparse or dense), so it remains to be seen if the painful resource usage explosion can be remedied. The obvious solution is to employ the much better performing third variant, which still provides a benefit over the separate Call Arity and Cardinality Analysis at hardly any cost in precision.

It would be interesting to see if some more explicitly operational model of sharing would alleviate the need for the LetUp analysis order, as it is responsible for some imprecision even when refined with co-call graphs (cf. the examples in Section 3.3.6). All our attempts to split off shared evaluation into a heap-like model failed, because of the complex interaction with sequential composition (e.g. $\&$). Also, the grain of improvement to precision probably wouldn't matter much, as we already are at the far end of diminishing returns, it seems. But it would be interesting to see the comparison to variant **(3)** exactly for that reason.

The requirement for an iteration order decoupled from the syntax tree came from better arity-aware LetDn-style analysis. In Section 4.8, we quickly came to the conclusion that in order to keep our sanity, we had to separate iteration logic from analysis logic.

This turned out to be an interesting path that no-one seems to have taken before: We abstract in the Worklist module a (currently rather limited) approach to iterating data-flow frameworks defined by mutually recursive TransferFunctions. Our formulation still misses a lot of potential abstraction and performance tuning.

Thinking in broader strokes, we could easily see this graph-based fixed-point iteration scheme become more widely used throughout GHC's analyses. In order for that to happen, assignment of FrameworkNodes to syntactic elements should be done once in a central place after transformation passes, that this needless redundancy won't bleed into analysis logic.

There's also the issue of monotonicity, discussed in Section 4.9. This is a consequence of how we encoded usage transformers: The lack of an appropriate data structure to model monotone maps over join-semilattices forced us to use ordinary maps instead. The total ordering used for the balanced search tree doesn't coincide with the join-semilattice structure expressed by $\sqcup$ in any way, so the potentially helpful functions Map.lookupLT were of no use at all. Let alone that they have the wrong return type to express multiple lower bounds, which are possible in a lattice. Thus, an easy improvement over the current situation would be to look out for an algorithmic solution to lookup values in a data structure that is keyed by a (join semi-)lattice. The underlying data structure would have to maintain multiple sources

of a directed acylclic graph to be remotely efficient, but considering that only very few points of a usage transformer are ever requested, this seems like no big deal.

A data structure for monotone maps would completely get rid of the monotonicity hacks in Section 4.9 and conceivably lead to better performance of the analysis overall.

Strictness analysis possibly benefits from the same improvements to LetDn analysis strategy, which would be interesting to pursue. Shattering the Demand Analyser in its three parts (usage, strictness, constructed product result analysis) sketched in Section 2.4.2 seems worthwhile from a perspective of software engineering and could also lay the ground work for integrating the graph-based data-flow iteration approach.

Lastly, we feel like there needs to be held a discussion about what the arity analyser should compute. Some GHC comments contradict each other in the details of what idArity is really supposed to mean: There's the notion of 'does essentially no work until applied to idArity arguments' (in the haddock of ArityInfo) vs. 'We want expressions returning one-shot lambdas to have arity one' (Note 'One-shot lambdas' in CoreArity). Either approach seems fine to us, but documentation should be clear about it, and possibly integrate usage information directly instead of one-shot annotations only. We anticipate some advantages in looking at the usage of a binder: The example in Note 'Arity analysis' in CoreArity would not need any fixed-pointing to figure out that f has arity 2; this property seems entirely specific to how many arguments f is applied to within its scope, a typical usage property. E.g., a usage analysis would compute the dreaded fixed-point to find out a usage of $\omega * C^{\omega}(C^{1}(U))$, based on which the arity analyser could compute the arity based on the arity expansion procedure in Section 3.4.1. Of course, even without messing about with usage information, the arity analyser aggregates more than enough heuristic concern (e.g. if an exprIsCheap enough to arity expand over) to justify its existence.

Hence, cutting out any logic in the arity analyser that computes usage information seems like a good idea to reduce intellectual and runtime cost of the analysis (though it's arguably rather cheap anyway).

7 Conclusion

Demand Analysis and Call Arity share subtle commonalities. Our work made these very explicit by providing a usage analysis that subsumes Call Arity and the usage analysis within Demand Analysis, referred to as Cardinality Analysis.

We highlighted problematic differences between the two (Chapter 1), like the LetUp vs. LetDn analysis order, and showed how to systematically solve them.

In Section 2.4.2, we advocated a clear separation of concerns and a split of the Demand Analyser in its sub analyses, consequently. We substantiated our claim by pointing out a number of tradeoffs associated with performing all analyses in lockstep, without any real data dependency between them which justifies doing so.

Although the abstract domain in which we interpret programs is not fundamentally different to that of Sergey et al. [17], we gave a more precise meaning to each element of the domain. In particular our treatment of usage transformers was more explicit than in Sergey et al. [17], which was needed because we denote our programs with them via our transfer function in Section 3.3.

As Chapter 4 reveals, implementing the analysis was challenging, as we needed to provide quite some infrastructure before writing the analysis in the final form was even feasible. We are particularly proud of our take on solving data-flow problems with a graph-based approach, decoupled from the syntax tree in Section 4.8 and expect interesting further discussions. There were quite some hacks involved in making the analysis realisable, some of which evoked further problems, like the lack of a data structure for maps indexed by join-semilattices in Section 4.9.

Although according to Chapter 5 the full generalisation has performance issues due to ubiquotious use of co-call graphs (which might be entirely related to the implementation), a variant without co-call graphs seems to compromise little on precision but still reaches performance comparable to prior approaches.

Opportunities for future work were discussed in Section 6.2 and left the author excited to pursue some open problems and to contribute to GHC in the future.

Bibliography

[1] Clem Baker-Finch, Kevin Glynn, and Simon Peyton Jones. "Constructed Product Result Analysis for Haskell". In: *J. Funct. Program.* 14.2 (Mar. 2004), pp. 211–245. ISSN: 0956-7968. DOI: 10.1017/S0956796803004751. URL: http://dx.doi.org/10.1017/S0956796803004751.

[2] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. *Firm - A graph-based intermediate representation*. KIT, Fakultät für Informatik, 2011.

[3] Joachim Breitner. "Lazy Evaluation: From natural semantics to a machine-checked compiler transformation". PhD thesis. Karlsruhe Institute of Technology, 2016.

[4] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. "Iterative dataflow analysis, revisited". In: (2002).

[5] Andy Gill and Graham Hutton. "The Worker/Wrapper Transformation". In: *J. Funct. Program.* 19.2 (Mar. 2009), pp. 227–251. ISSN: 0956-7968. DOI: 10.1017/S0956796809007175. URL: http://dx.doi.org/10.1017/S0956796809007175.

[6] Martin Charles Golumbic. "Comparability graphs". In: *Annals of Discrete Mathematics*. Vol. 57. 2004, pp. 105–148. ISBN: 9780444515308. DOI: 10.1016/S0167-5060(04)80053-0.

[7] Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. "A Generic Usage Analysis with Subeffect Qualifiers". In: *SIGPLAN Not.* 42.9 (Oct. 2007), pp. 235–246. ISSN: 0362-1340. DOI: 10.1145/1291220.1291189. URL: http://doi.acm.org/10.1145/1291220.1291189.

[8] Ralph Hinze. "Projection-based strictness analysis - theoretical and practical aspects." PhD thesis. Universität Bonn, 1995, p. 237.

[9] Ryszard Kubiak, John Hughes, and John Launchbury. "Implementing Projection-based Strictness Analysis". In: *Functional Programming, Glasgow 1991: Proceedings of the 1991 Glasgow Workshop on Functional Programming, Portree, Isle of Skye, 12–14 August 1991*. Ed. by Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler. London: Springer London, 1992, pp. 207–224. ISBN: 978-1-4471-3196-0. DOI: 10.1007/978-1-4471-3196-0_17. URL: https://doi.org/10.1007/978-1-4471-3196-0_17.

[10] John Launchbury and Simon L. Peyton Jones. "Lazy Functional State Threads". In: *SIGPLAN Not.* 29.6 (June 1994), pp. 24–35. ISSN: 0362-1340. DOI: 10.1145/773473.178246. URL: http://doi.acm.org/10.1145/773473.178246.

[11] Roland Leißa, Marcel Köster, and Sebastian Hack. "A Graph-based Higher-order Intermediate Representation". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '15. San Francisco, California: IEEE Computer Society, 2015, pp. 202–212. ISBN: 978-1-4799-8161-8. URL: http://dl.acm.org/citation.cfm?id=2738600.2738626.

[12] Simon Marlow. "Update Avoidance Analysis by Abstract Interpretation". In: *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, 5–7 July 1993*. Ed. by John T. O'Donnell and Kevin Hammond. London: Springer London, 1994, pp. 170–184. ISBN: 978-1-4471-3236-3. DOI: 10.1007/978-1-4471-3236-3_14. URL: https://doi.org/10.1007/978-1-4471-3236-3_14.

[13] Will Partain. "The nofib Benchmark Suite of Haskell Programs". In: *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992*. Ed. by John Launchbury and Patrick Sansom. London: Springer London, 1993, pp. 195–202. ISBN: 978-1-4471-3215-8. DOI: 10.1007/978-1-4471-3215-8_17. URL: https://doi.org/10.1007/978-1-4471-3215-8_17.

[14] Simon Peyton Jones, Peter Sestoft, and John Hughes. *Demand Analysis*. 2006. URL: https://www.microsoft.com/en-us/research/publication/demand-analysis/.

[15] Norman Ramsey, João Dias, and Simon Peyton Jones. "Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation". In: *SIGPLAN Not.* 45.11 (Sept. 2010), pp. 121–134. ISSN: 0362-1340. DOI: 10.1145/2088456.1863539. URL: http://doi.acm.org/10.1145/2088456.1863539.

[16] Amr Sabry and Matthias Felleisen. "Reasoning About Programs in Continuation-passing Style." In: *SIGPLAN Lisp Pointers* V.1 (Jan. 1992), pp. 288–298. ISSN: 1045-3563. DOI: 10.1145/141478.141563. URL: http://doi.acm.org/10.1145/141478.141563.

[17] Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. "Modular, Higher-order Cardinality Analysis in Theory and Practice". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 335–347. ISSN: 0362-1340. DOI: 10.1145/2578855.2535861. URL: http://doi.acm.org/10.1145/2578855.2535861.

[18] Peter Sestoft. "Analysis and Efficient Implementation of Functional Programs". PhD thesis. 1991.

[19] David N. Turner, Philip Wadler, and Christian Mossin. "Once Upon a Type". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: ACM, 1995, pp. 1–11. ISBN: 0-89791-719-7. DOI: 10.1145/224164.224168. URL: http://doi.acm.org/10.1145/224164.224168.

[20] Hidde Verstoep. "Counting Analyses". MA thesis. Utrecht University, 2013.

[21] Hidde Verstoep and Jurriaan Hage. "Polyvariant Cardinality Analysis for Non-strict Higher-order Functional Languages: Brief Announcement". In: *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*. PEPM '15. Mumbai, India: ACM, 2015, pp. 139–142. ISBN: 978-1-4503-3297-2. DOI: 10.1145/2678015.2682536. URL: http://doi.acm.org/10.1145/2678015.2682536.

[22] Keith Wansbrough. "Simple polymorphic usage analysis". PhD thesis. University of Cambridge, 2005.

[23] Keith Wansbrough and Simon Peyton Jones. "Once Upon a Polymorphic Type". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: ACM, 1999, pp. 15–28. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292545. URL: http://doi.acm.org/10.1145/292540.292545.

Erklärung

Hiermit erkläre ich, Sebastian Graf, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

Danke

Ich möchte mich bei Denis Lohner für seine außerordentlich hilfreiche und herzliche Betreuung bedanken. Dass er sich bereit erklärt hat, sich in ein für ihn weitgehend fremdes Thema einzuarbeiten, verdient meinen größten Respekt. Dein steter Rat hat mir aus so mancher Bredouille während des Aufschriebs geholfen!

Außerdem möchte ich diese Chance nutzen, meinem Mentor Joachim Breitner zu danken. Danke für Deine Doktorarbeit, ohne die diese Arbeit nicht zustande gekommen wäre. Danke auch für Deine fachlichen Hilfestellungen, mit denen Du mir aus den regelmäßigen Sackgassen geholfen hast. Nicht zuletzt möchte ich mich für Deine stets wohlwollend angenommenen Schubser ins kalte Wasser bedanken :).

Zum Schluss sei noch der gesamten Haskell Community mein Dank ausgesprochen. Es ist ein Privileg und eine Übung in Bescheidenheit zugleich, wenn man so mühelos über das Internet Kontakt zu unglaublich netten und klugen Leuten herstellen kann, die für die gleichen Themen brennen.